How Good is the Distributed Databases in Supporting Transaction Processing

Luyi Qu[1†], Huidong Zhang[1†], Rong Zhang[1†], Peng Cai[1†],
Quanqing Xu[2‡], Zhifeng Yang[2‡], ChuanHui Yang[2‡]
[1]{luyiqu@stu, hdzhang@stu, rzhang@dase, pcai@dase}.ecnu.edu.cn, [†]East China Normal University
[2]{xuquanqing.xqq,zhuweng.yzf,rizhao.ych}@oceanbase.com, [‡]OceanBase

ABSTRACT

Distributed transactional databases (OLTP DBMSs) have recently gained popularity for their remarkable capability in *Scalability*, *Availability* and *Schedulability*. This paper argues that quantitative control is essential in benchmarking these new properties, which has not been schemed well by the state-of-the-art OLTP benchmarks. Then they may mislead the comparing, designing and implementing of the modern distributed OLTP DBMSs. To address this problem, we present *Dike*, a new benchmark suite extended from the popular TPC-C benchmark, which aims to launch a fair and in-depth evaluation to distributed OLTP DBMSs in 1) scalability by a *quantitative distribution control* of workload and data; 2) availability by constructing a general *exception control* to simulate exceptions of either hardware or software; 3) schedulability by quantifying *imbalance* for both data and workload; 4) unified metrics $w.r.t$ the core designs for distributed transaction processing. *Dike* is the first benchmark suite specified to benchmark distributed OLTP DBMSs. Our evaluations justify the designs of *Dike* and expose interesting observations for the mainstream distributed transaction processing architectures. The source code is available on github [1].

ACM Reference Format:
Luyi Qu[1†], Huidong Zhang[1†], Rong Zhang[1†], Peng Cai[1†],, Quanqing Xu[2‡], Zhifeng Yang[2‡], ChuanHui Yang[2‡]. 2022. How Good is the Distributed Databases in Supporting Transaction Processing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With continuing business expansion, an urgent requirement for database scalabilility arises, which inspires a fast proliferation of distributed database management systems (DDBMS). By leveraging on data sharding or horizontal data partition among multiple nodes, DDBMS can launch distributed parallel processing for high *scalability*. Since more nodes are involved in a database service, it has a higher probability to meet a fault [6, 17] caused by either the software or the hardware. To guarantee a high performance, data segmentation or replication (redundancy) among nodes further gives a way for uninterruptable services, i.e., high *availability*. Furthermore, distributed multiple nodes also provide a good chance to resolve the problem in load imbalance by workload or data migration, i.e., *schedulability*, to improve overall performance. However, providing all those characteristics in DDBMS trades off the performance of databases for the expensive remote interactions or coordinating among nodes [27, 33].

Most of the state-of-the-pratice OLTP benchmarks have been abstracting from classic applications to evaluate the overall performance of an OLTP databases by providing quantitative metrics, a general methodology and even tools. But as summarized in Table 2, most of these OLTP benchmarks are constructed tens of years ago, which fail to characterize the properties in the newly designed DDBMSs, i.e., scalability, availability and schedulability, and then it is incommensurable or incomprehensive to benchmark DDBMS for these new technical designs. It is then prerequisite to justify what the critical factors are for benchmarking the distributed transactional DBMS $w.r.t$ these new properties.

Transactions deployed on distributed nodes to mitigate the pressure on a single node is essential to support the keep growing of applications. Ideally, each transaction involves a single partition of data and is deployed on an individual node, processed like in a centralized system. Performance is then linearly scalable with the number of nodes. However, the complex application logic (behind the workload) may not be consistent with the data partition rule and then one transaction may access data on multiple nodes, i.e., distributed transactions, which then lowers down performance. To deal with distributed transactions to upgrade the overall performance of a database has always been a hot topic [19, 38, 44, 49, 54].
Remarks: To fairly compare the ability of DDBMSs in handling distributed transactions, it is important to quantitatively control transaction distributions. However, none of the existing transactional benchmarks [3–5, 8, 15, 72] is designed for this purpose.

Although we have obtained significant progresses in improving scalability of DDBMS, frequent access to highly contended data still bottleneck modern DBMSs. To guarantee ACID is the main reason to restrict high performance, for concurrent transactions accessing the same data usually take a blocking-way to execute to guarantee isolation, i.e., be serialized in order. Contended workload gets in the way of a linear scalability and then DDBMS continues to be plagued by performance problems on high contention workloads [52, 70].
Remarks: In order to fairly compare the ability dealing with resource contentions, it is necessary to generate workloads with the same intensity of contentions. Although benchmarks, such as TPC-C and YCSB, have given coarse ways to adjust contentions by changing either the data size or workload access distribution, it cannot be adapted well for elaborate database comparison.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Scheduling transactions across nodes to keep all nodes occupied with balanced workload to maximize concurrency is ideal, i.e., a thorough concurrency. For example, if we distribute all hot data to one node, this single node will meet a severer contention on resources compared to the other nodes. This kind of imbalance of workloads leads to a drop of throughput, which is usually resolved by scheduling transactions or data in a balance way. It has been shown that scheduling transactions using the statistics in history can achieve a high throughput for partitionable workloads [55, 73].
Remarks: Based on the evidence in history or current workload status to schedule workloads or data across nodes has been a kernel function to a distributed transactional database for upgrading its transaction processing ability. However, none of the existing classic transactional benchmarks is designed on purpose to trigger scheduling capabilities.

Availability is a liveness guarantee and it is a warranty for the continuity of correct service [10, 26]. Confronting communication failures or data corruptions, unavailable distributed software is barely useful. Dividing the database horizontally into fragments and deploying each fragment to a distributed node essentially make some nodes available to provide services during failures; it will be better to create multiple remote replicas for each local segement which inherently reduces the downtime footprint by the sharding factor. However, the elaborate designs on distributed consistency protocols among replicas or nodes to guarantee the correctness of transactions is at the cost of throughput.
Remarks: Guaranteeing high consistency among replicas for availability trades off performance of transaction processing. Previous work cares more about the recovery time of database services instead of the ability of providing stable services.

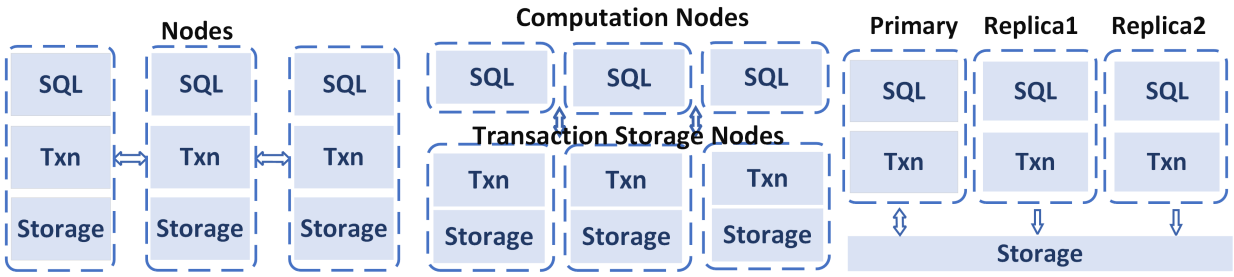
With respect to these new properties, we propose *Dike*, a new benchmark suite, to help users perform comprehensive and in-depth comparisons among distributed transactional DBMSs or evaluate the technical designs independently. *Dike* is adapted from the widely-used industrial-standard TPC-C benchmark, which preserves the practical application scenario defined in TPC-C, but enriches TPC-C in the following aspects:

- Control of transaction distributions and contentions: To expose the scalability $w.r.t$ distributed transaction processing and contention intensities, it controls workload generation in a quantitative way to facilitate fair comparisons.
- Configurable workload and data imbalance: To benchmark the schedulability $w.r.t$ imbalance, we expose the parameters to define and make application scenario with different imbalance degrees.
- Agile failure simulation: To expose the availability $w.r.t$ failures, we provide a convenient way to simulate brokens of hardwares or corruption of data to demonstrate the service quality.
- Explicit evaluation metrics: For a comprehensive evaluation to the good properties of DDBMS, we define new metrics to demonstrate its distributed ability $w.r.t$ the above three characteristics.

So we summarize the main contributions of *Dike* as follows.

(1) We analyze and summarize the main technique designs $w.r.t$ the three representative properties of **scalability**, **availability** and **schedulability** in distributed transactional DBMSs. For each property, we generalize the technique, the challenges introduced and classic optimization methods.
(2) We design and implement an extensible benchmark suite *Dike*. Compared against the previous transactional benchmarks, *Dike* has a distinguished property of distribution controls. Specifically,
    - As the result of data partition, transactions are issued with various parameter bindings. By instantiating parameters in an individual transaction from a number of distributed nodes, we successful control distribution ratio and degree among workloads in probability.
    - Imbalance among workloads and data inspires the load balance processing. To evaluate the adaptability of DDBMSs, both data generation and parameter instantiation are constructed following a specific cumulative distribution function (CDF).
    - Quantifiable metrics are designed $w.r.t$ scalability, availability and schedulability of DDBMSs. We provide comprehensive guidance on how to comprehensively and thoroughly compare the trade-offs for performance and scalability, availability or schedulability.
(3) Extensive experiments are conducted on three mainstream distributed transactional databases, i.e., OceanBase [69], TiDB [32] and CockroachDB [61]. We have observed the following insights. Remote communication should be avoided as much as possible, which has the most terrible impact on performance, so an adaptive routing mechanism can be introduced to reduce extra RPC; automatic scheduling mechanism can indeed improve the performance of DDBMSs, but currently scheduling mechanism is only designed to tackle with relatively simple application scenarios, and more elaborate scheduling methods are to be explored. Compared OceanBase (with *Shared-Nothing* architecture) with TiDB and CockroachDB (with *Shared-Nothing with separation*), *Shared-Nothing* architecture has an obvious higher performance but also is sensitive to transaction distribution; and *Shared-Nothing* architecture is more suitable for well partitioned application scenarios with an dominant advantage; *Shared-Nothing with separation* is more transparent to the business which reduces burden on application side and its low latency makes it more insensitive to transaction distribution; all of them are lack of an upstanding schedulability.

Our paper is organized as followings. In Sec. 2 we discuss three representative architectures of transactional DDBMS. In Sec. 3 we explore three representative properties and their technique, potential challenges and existing optimization methods, and then point out the supportability of classic transactional benchmarks. Sec. 4 presents the design of Dike concerning the quantitative control techniques in distribution, imbalance control mechanism and metric definitions. In Sec. 6, we show how well Dike works and present the scalability, availability, and schedulability on three classic distributed transactional databases. Related work is summarized in Sec. 7 and we conclude in Sec. 8.



(a) Shared Nothing (b) Shared Nothing with Separation (c) Shared Storage

Figure 1: Representative Architectures for Distributed Transaction Processing

2 PRELIMINARIES

In this section, we classify distributed transactional databases into three types of design architectures as in Fig. 1 and summarize their representative characteristics *w.r.t* distributed transaction processing in Table 1.

2.1 Architecutures of Distributed Transactional Databases

Distributed transactional DBMS is consistent with NoSQL databases [14, 20, 25] in scalability, availability, and resilience while simultaneously meeting the rigorous consistency and requirements of transaction processing [48]. DBMS is made up of three layers, i.e., **Query Engine**, **Transaction Engine** and **Storage Engine** as shown in Fig. 1. Setting a cut-off line between different engines and distributing the down level engines yield three typical distributed transactional database architectures.

- If the cut-off line is above *Query Engine*, it is a **Shared-Nothing** architecture and each node has its own three layers of functional modules as in Fig. 1 (a). The representative examples are H-Store [34], Citus [18], Spanner [16] and OceanBase [69]. Specifically, Citus is a distributed database plugins, which organizes multiple PostgreSQLs into a distributed database cluster. So a compute task in each node has its local transaction manager and storage engine. But each node has the opportunity to coordinate with any other remote node to complete a transaction by messaging, it then introduces distributed transactions. However, query engine is tied with transaction manager and storage engine, which lowers down the computing elasticity.
- If the cut-off line moves one level down to *Transaction Engine*, it is a **Share-Nothing with Separation** model as in Fig. 1 (b), and the representative databases are CockroachDB [61], TiDB [32] and FoundationDB [74]. Query engines are distributed on compuation nodes, which are stateless and uncoordinated to issue queries, so we can restart a query (if fail) without any data restoring or recovering. Transaction manager and storage engine are tied together to compose a stateful storage node, which are distributed. Instead of having a predefined data partition rule, it divides data into sub-blocks distributed (randomly) to each storage node. It uses a centralized management controller to balance the storage and workload among all storage nodes. Each computation node can access any storage node with enough flexibility and distributed transactions are unavoidable.
- If the cut-off line moves down to *Storage Engine*, **Shared-Storage** model is constructed by putting query engine and transaction manager on computation node as in Fig. 1 (c). Aurora [65], PolarDB [13], Socrates [9], and Taurus [21] are examples of such an architecture. The computation node supports transaction processing, called *Primary Instance*. It has a distributed and extensible storage layer, which can be accessed as a whole by any computation node. So this architecture has a full scalability for storage. To improve performance, it allows to deploy more distributed SQL-Transaction engine, called *Replica Instance*, for read purpose instead of transaction processing. Then the single write node has a high probability to bottleneck DBMS. Specifically, in order to make data of replica instances (read nodes) consistent with the primary instance (write node), read nodes synchronize redo logs from write node to update their caches. Since only one node can process transactions, it meets no distributed transactions.

The main characteristics related to distributed processing are summarized in Table 1. Each database has its own concurrency control protocols, e.g., MVCC+2PL for Citus, to guarantee its IL. All distributed transactions are committed by 2PC. To tolerant errors (for high availability), each storage has multiple copies, which are synchronized by different protocols, e.g., K-Safety [34], Paxos [31, 37] or master-slave replication [30, 56]. For different distributed model, if they have a global timestamp management mechanism, they can have a consistent snapshot among nodes. Among the example databases in Table 1, only Citus has no timestamp synchronization among nodes and cannot support consistent distributed queries. **Either *Shared-Nothing* or *Shared-Nothing with Separation* can coordinate to complete transactions, while *Shared-Storage* model has no distributed transactions which owns only one single write node on a shared storage.** So we will not discuss or compare the performance of databases of *Shared-Storage* model any more.

3 WHAT TO BENCHMARK FOR A DISTRIBUTED TRANSACTIONAL DBMS

In this section, we first present the main techniques designed to achieve scalability, availability and schedulability in distributed transactional DBMSs. For each property, we generalize its introduction from four aspects, which are the main idea, the general technique, the challenges introduced and the classic optimization methods (summarized in Fig. 2). Then we present the coverage of classic benchmarks to these properties. Finally, we conclude the

Luyi Qu[1†], Huidong Zhang[1†], Rong Zhang[1†], Peng Cai[1†],
Quanqing Xu[2‡], Zhifeng Yang[2‡], ChuanHui Yang[2‡]

	Fixed Part	Distributed Part	Database	Storage	Transaction	Query
				Replication Technique	Commit Protocol	Global Snapshot
Shared-Nothing	/	Query, Transaction, Storage	H-Store/VoltDB	K-Safety	2PC	T
			Citus	Master-Slave	2PC	F
			Spanner	Paxos	2PC	T
			OceanBase	Paxos	2PC	T
Shared-Nothing with Separation	Query	Transcation, Storage	CockroachDB	Raft	Percolator	T
			FoundationDB	K-Safety	2PC	T
			TiDB	Raft	Percolator	T
Shared-Storage	Query, Transaction	Storage	Aurora	Quorum	/	T
			PolarDB	Raft	/	T

Table 1: Architectures and Comparison of Distributed Transactional Databases

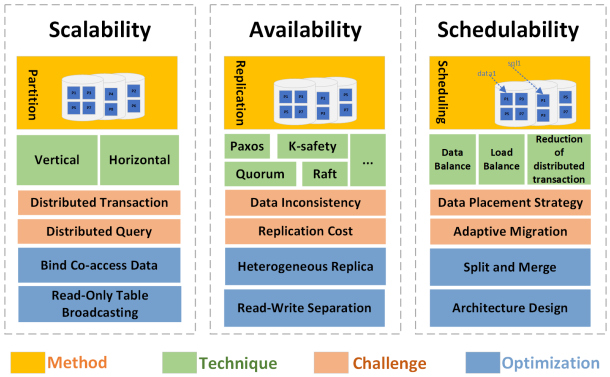


Figure 2: Representative Techniques for Distributed Transaction Processing

design requirements for benchmarking distributed transactional databases.

3.1 Techniques for Distributed Transaction Processing

3.1.1 Partition for Scalability. In face of severe workload or a large data volume, scalability can be achieved by either *scale-up* or *scale-out*. Scale-up upgrades hardware, if possible, for high performance. Unfortunately, its performance is bottlenecked by hardware, which is not flexible enough *w.r.t* the business expansion. Scale-out attracts more attention for its full scalability by inserting new nodes to support applications of any scale. In this way, data is usually partitioned to multiple nodes either vertically or horizontally. Though it provides full scalability for business, we have to tackle with two challenges for this partition-based method.

- One challenge is how to guarantee performance with distributed transactions, which access data across multiple nodes. To ensure atomicity and durability, operations in a transaction must be committed or rollbacked simultaneously. *Two Phase Commit* (2PC) protocol [43] and *Three Phase Commit* (3PC) protocol [58] are then proposed for distributed transaction processing [16, 32, 61, 69, 74]. To ensure isolation level, an efficient concurrency control is necessary, among which the commonly used ones are *Two-phase Locking* (2PL) [11], *Optimistic Concurrency Control* (OCC) [36] and *Multiversion Concurrency Control* (MVCC) [51]. However, coordinating and synchronizing participants of a distributed transaction over network cost expensive I/Os, which degrades performance extremely.
- Query optimization is challenged by collecting the statistics from distributed nodes in a dynamic environment. Statistics of data are prerequisite for generating an optimal query plan. It is still a hot research topic to collect statistics in a centralized DBMS and deal with the trade-off between effectiveness and performance *w.r.t* dynamicity. As a distributed query, optimization meets more challenge from data transferring and summarizing, which is highly related to the distribution of data among nodes. Even though we have the data partition rule, since we may not know the exact mapping between data and physical nodes, especially the existence of slave nodes for read (if any), it can not tell which node will be taken to resolve the query, and it is seriously deteriorated if it takes an adaptive or random data partition mechanism.

Optimization Remarks: Distributed processing introduces too much overhead on network, and optimization focuses more on reducing distributed transactions and queries [19, 49, 54, 60, 64, 71]. There are two optimizations commonly used. One is to bind the co-access data together by analyzing the workload [16, 69] and the other one is to broadcast small tables to all nodes to avoid remote access, e.g., *Item* table in TPC-C.

3.1.2 Replication for Availability. High availability of DDBMS is prerequisite for mission critical tasks, e.g., banking service. Though DDBMS provides a good chance for uninterrupted services by backing up remote nodes (replications), expensive consensus algorithms, e.g., Paxos [37], Raft [46], K-safety [34], and Quorum [2], have to be initiated to guarantee the consistency among replications. It will have a negative impact on performance, for a client cannot receive a response until a certain number of replicas have achieved successfully.

Optimization Remarks: To reduce replication cost, it proposes to create heterogeneous replicas. For instance, TiDB introduces *learner* to Raft by detecting and synchronizing logs, if needed. OceanBase creates four types of replicas for different purposes, i.e., full-featured replicas, log-only replicas, encrypted voting replicas and read-only replicas. The first three replicas join in a Paxos group and participate in voting but only the full-featured replicas can be elected as a leader; log-only replicas and encrypted voting replicas only contain incremental logs instead of the whole data source. In addition, with the explosive increase of client requests, read/write separation mode is proposed, that is, the slave replicas are used to handle read requests to alleviate the pressure of the master node with primary replicas, which respond to write transactions only. Some distributed transactional databases provide a weak consistent read on slave replicas, which improves read performance by sacrificing data freshness; if it requires a strong consistent read on replicas, e.g., TiDB, we have to synchronize master and slaves, which causes lower performance.

3.1.3 Scheduling for Efficient Resource Usage. In DDBMS, highly overloaded node usually decides the performance of DBMS. Overload is caused by an imbalance distribution of workload and data. Scheduling is vital to balance data/workload and reduce distributed transactions. It is a common way to analyze the execution traces of workloads to plot the access distribution and collect the load of each node to achieve an optimal data placement and to alleviate imbalance [16, 19, 45, 49, 50, 57, 69] or distributed transactions. Unfortunately, current off-online methods are not adaptive to dynamic characteristics of data or workload. Online scheduling needs to collect the statistics of data/workload from all nodes and launch adaptive migration. For the requirement of high online performance, adaptive scheduling is challenged by expensive communication or data migration.

Optimization Remarks: To reduce the number of distributed transactions, we can merge the co-access partition to one node [53, 54, 60]; to balance workload, we can split the hot region to different nodes [32]. Some work tries to minimize the number of physical nodes allocated while ensuring sufficient capacity for the workload to avoid distributed processing [59].

	Scalability		Availablity	Schedulability	
	Control Distribution		Inject Failure	Dynamic	
	Transaction	Query		Contention	Workload
TPC-C	P	×	P	P	×
TPC-E	P	P	P	P	×
TATP	×	×	×	P	×
YCSB+T	P	P	×	P	×
SmallBank	P	×	×	P	×
PeakBench	P	P	×	P	✓

Table 2: Classic Benchmarks for OLTP DBMSs: ×, *P* and ✓ represent no, partial and full support respectively.

3.2 Related Work

Distributed transaction processing (OLTP) attracts a lot efforts [9, 13, 16, 18, 21, 32, 34, 61, 65, 69, 74] for the outstanding properties in scalability, availability and schedulability. To promote the development of DDBMSs, guarantee the healthy competition in database markets, and stimulate engineering and innovations, it becomes increasingly important over the last years to find a way to benchmark the distributed OLTP DBMSs.

Previous OLTP benchmarks are mainly designed for centralized DBMSs and most of them are constructed to simulate the characteristics of the specific applications, such as TATP [3], TPC-C [4], TPC-E [5], SmallBank [8] and so on. YCSB [15] provides CRUD operations on key-value stores. Based on it, YCSB+T [22] is proposed to determine whether the consistency constraint is violated during transaction execution. These benchmarks are not designed specifically to evaluate distributed OLTP DBMSs, which have not taken into account the representative characteristics of distributed OLTP DBMSs seriously. The benchmarking capability of classic benchmarks in terms of scalability, availability and schedulability are summarized in Table 2.

To benchmark scalability, besides the quantity of workload, quantitative distributed transactions and queries are required for a comprehensive comparison, but none of them has been fully considered in current benchmarks. Most benchmarks do have distributed

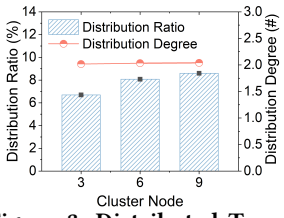


Figure 3: Distributed Transactions

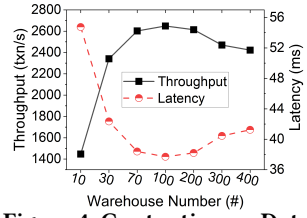


Figure 4: Contention *vs.* Data Size

transactions or queries, which however cannot be controlled quantitatively, including the number of distributed transactions and the number of cross-nodes in a transaction. For example, TPC-C is the most popular OLTP benchmark, but it is partition-friendly, i.e., most transactions can be well mapped to an individual partition located on a single node. Only transaction *Payment* and *NewOrder* have the chance to generate distributed transactions. Specifically, in *Payment*, it has 15% probability to generate a distributed transaction with cross nodes <2. In *NewOrder*, any item bought has a probability of 1% from a remote node. Since *Payment* has a fixed range of distributed transactions with a fixed cross-nodes as explained above, we then run *NewOrder* on OceanBase by setting the warehouse number equals to the concurrent threads to demostrate the distributions of transactions. We deploy 100 warehouses on a 3-node cluster, 200 on a 6-node cluster and 300 on a 9-node cluster in Fig. 14. On different sizes of clusters, the distributions are stable and low. OLxPBench [35] proposes to explore the scalability of TiDB and OceanBase, but it focuses on the performance isolation of these databases rather than transaction processing capacity when scaling out. However, in real world, we may have more distributed transactions on a big cluster since it is not easy to partition data well according to the workload [19, 45, 49, 54].

In terms of availability, stable services from databases are expected $w.r.t$ hardware or software exceptions [7, 42, 47, 66]. Failure injection is required to evaluate the performance of a DDBMS under exceptions. Only TPC-C and TPC-E take node failures into consideration. However, exceptions are more than node failures, it may be network jitter, CPU failure, etc. Current failures in benchmarks are far less than that in production. More over, there is no consistent definition to evaluate the services of distributed OLTP under different environmental anomalies.

To benchmark the schedulability of a DDBMS, it is vital to simulate intensive contentions [28, 63, 68], skewed and dynamic workload accesses [54, 60]. HATtrick [41] and OLxPBench [35] propose to launch resource scheduling, e.g., CPU or Thread, that is to reassign resources between transaction processing and analytical processing to look into the change of overall performance. It can not be used to provoke an active action from the schedule module in DDBMSs, if any. Only PeakBench [72] provides dynamic changes in both workload quality and type distribution. Generally there are two ways to control contentions which are to visit different sizes of data or to change the skewness of data access. Though all benchmarks have contention simulation, none of them can be precisely controlled. For example, TPC-C changes contention by resizing the number of warehouses; YCSB controls contention by adjusting the Zipfian parameters of data distribution. Specifically, we run TPC-C on OceanBase under different numbers of warehouses. Enlarging database does boost throughput at the beginning, but the

Luyi Qu[1†], Huidong Zhang[1†], Rong Zhang[1†], Peng Cai[1†],
Quanqing Xu[2‡], Zhifeng Yang[2‡], ChuanHui Yang[2‡]

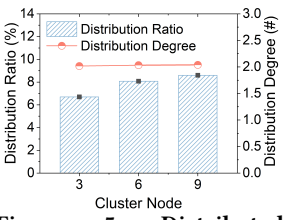


Figure 5: Distributed Transactions

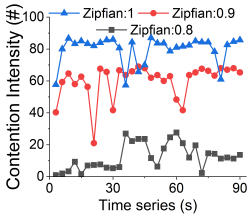


Figure 6: Contention *vs.* Skewed Data

performance decreases from 200 warehouses as shown in Fig. 15. Though PeakBench has proposed a way to generate contentions quantitatively, it is not scalable based on a thread blocking and waiting algorithm.

3.3 Benchmarking Requirement

The fundamental gap between the evaluation principles of classic benchmarks and the advanced design purposes of current distributed transactional databases inspires us to propose a new benchmark, which is expected to be effective to evaluate the advanced characteristics in scalability, availability and schedulability. As we have analyzed above, some benchmarks can touch some of the evaluation points, but it is not fair to launch comparisons among databases, for the key characteristics can not be quantitatively controlled. More over, since application-oriented designs may not demand all these advanced characteristics, it is preferable to provide a pluggable service by the new benchmark. Last but not least, in real scenarios, workload has obvious dynamicity, i.e., changing over time. Dynamic workload is required to be simulated. We then summarize the following five core requirements for benchmarking the advanced distributed transactional databases:

1. **Quantitive:** Workload should be quantitatively controlled with respect to distibution ratio, distribution degree and contention.
2. **Dynamical:** Workload should be dynamically configured in distribution of types and volumes as well as contentions over time.
3. **Unpartitioned:** Data should not be well partioned with respect to workload; data distribution should be imbalance by partiton keys.
4. **Metrizable:** New metrics should be designed for evaluating the new properties in distributed transactional databases.
5. **Pluggable:** It should be possible for users to customize the testing scenario for database selection or POC testing.

4 DESIGNS IN DIKE

We now present the detailed design in **Dike**, which enables the systematic evaluation of distributed transactional databases. We first give a concise summary to TPC-C *w.r.t* its capacity of benchmarking distributed transactional databases (Sec. 4.1). We then present the technique design in workload generation, contention control, data generation and evaluation metrics definition (Sec. 4.2-Sec. 4.4).

4.1 TPC-C Benchmark

TPC-C is one of the most popular on-line transaction processing benchmarks. However, the technical design of distributed transactional databases challenges TPC-C in its evaluation ability. Firstly, it is **partitionable**. TPC-C can be well-partitioned using a partition rule on *WarehouseID* w.r.t its workload, and only a few distributed transactions exist, i.e., about 10% in *NewOrder* and 15% in *Payment* statistically. For fair comparison, quantitive control of transaction distributions and contentions are imperative. Secondly, it is **balanceable** in both data volume and workload. Though TPC-C has different scale factors (SF) based on warehouses, data partitioned by *WarehouseID* has little imbalance which is not good for evaluating the design of schedule. Thirdly, metrics are **general**. Its metrics care more about the overall performance, e.g., *TPS* and *Latency*, instead of highlighting the ability of distributed processing.

4.2 Quantitative Control for Workload Generation

4.2.1 Distribution Control. Usually, the more distribution transactions and the more number of nodes involving in a transaction, the more degradation to performance. We propose to specify distributed transactions from two dimensions. The first one is **distribution ratio** (*disRatio*) which is the percentage of distributed transactions among all transactions. and the second is **distribution degree** (*disDegree*), which is the number of cross nodes for a transaction. Both of them have a remarkable impact on network consumption.

```
dNewOrder
Begin
//Single partition operation;
Loop
O1: select s_information from stock where  s_w_id = s1 and s_i_id = s2 for update;
O2: update s_information=p3 from stock where s_w_id = s1 and   s_i_id = s2;
End Loop
Commit
```

dUpdateStock

Figure 7: Transaction Template of *dNewOrder* and *dUpdateStock*

In TPC-C, both *Payment* and *NewOrder* involve distributed transactions. But *Payment* has fixed the cross-node of 2 by its application logic; *NewOrder* has a loop structure to order any number of items from remote warehouses, which gives a good chance to generate the number of cross-node transactions. We then select to extend *NewOrder* to *dNewOrder* by controlling the generation of distributed transactions and the transaction template is shown in Fig. 7. Inside the loop structure, $O1$ selects $s\_information$ from table *Stock* by the composite primary key s_w_id and s_i_id, and then $O2$ updates the information of the selected stock. In TPC-C, the partition key is s_w_id which decides the nodes $O1$ and $O2$ distributed to; besides, all other SQL operations in *NewOrder* access the same single partition, i.e., with the same s_w_id. So distributed transaction can be controlled by parameters in $O1$ and $O2$.

However, instantiating parameters s_w_id in $dNewOrder$ to guarantee the distributions on a DDBMS is challenged by the uncertainty of data distribution across nodes. Suppose the expected distributed transaction ratio is $disRatio{=}\lambda$ ($0 \le \lambda \le 1$) and cross-node number is $disDegree{=}n$ ($1 \le n$). $disRatio$ can be obtained by controlling the probability of populating different partition keys inside the loop structure of $dNewOrder$; $disDegree$ requires to determine the number of different partition keys c used in the loop structure with $c \le n$. For it has the uncertainty mapping relationship between logic partitions and physical partitions by its partition rule Π on a cluster of N nodes ($n \le N$), if we simply select n distinct

keys for *s_w_id*, a larger deviation of distribution degree will occur with a larger n as declared by Theorem. 4.1. .

We propose to find the number of distinct keys in *dNewOrder*, i.e., c, to guarantee *disDegree=n* using a probabilistic model. Based on the partition function Π, the ratio of data on node i is p_i having $\sum_{i=1}^{N}(p_i) = 1$. Supposing the probability that the transaction does not visit and visit i^{th} node is $P(x_i = 0)$ and $P(x_i = 1)$ respectively, the expectation the transaction visits i^{th} node is $E(x_i)$ calculated as

$$P(x_i = 0) = (1 - p_i)^c; \quad E(x_i) = P(x_i = 1) = 1 - (1 - p_i)^c \tag{1}$$

Then we formulate the expectation of the number of distinct nodes visited by *dNewOrder* as

$$E(x) = \sum_{i=1}^{N} E(x_i) = N - \sum_{i=1}^{N}(1 - p_i)^c = n \tag{2}$$

For $E(x) = n$, c is achieved by *Bisection Method* [12].

Algorithm 1 Generate *dNewOrder* based on *NewOrder*

Input: Table T, $disRatio$=λ, $disDegree$=n, loop number l in *dNewOrder*, static key k_0.
Output: key set Ω to populate transaction template.

```
1: Initialization: count ← 0, Ω ← ∅.
2: obtain c by Equation 2;
3: pr = random(0, 1);
4: if pr ≤ λ then
5:    while count < c − 1 do
6:       repeat
7:          k_new = random(T);
8:       until k_new ∉ Ω
9:       put k_new in Ω;
10:      count ← count + 1;
11:   end while
12: end if
13: while count < l do
14:    put k_0 in Ω;
15:    count++;
16: end while
17: return Ω;
```

The detailed generation process for achieving $disRatio$=λ and $disDegree$=n based on *dNewOrder* is provided in Algo. 1. We first calculate the expected distinct partition keys (c) according to $disDegree = n$ (line 2). For each transaction template, we first flip to decide whether it is a distributed transaction or not. If it is not a distributed transaction, i.e., $pr > \lambda$, we take k_0 to fill all the l-round of partition keys in the transaction template, i.e., *s_w_id*=k_0. If it is a distributed transaction (line 4), we keep selecting c distinct keys into Ω (line 5-11) until $count = c$. For the other $l - count$ number (round) of keys, we put k_0 into Ω (line 12-15). Finally, Ω has c distinct keys selected to fill the parameters in the l-round of loop structure. In Algo. 1, both the computation storage complexity is O(c).

Theorem 4.1. *On a cluster sized N, if we take n distinct keys to instantiate n parameters for SQLs inside a transaction for disDegree=n ($2 \le n \le N$), under a uniform distribution, a smaller N leads to a worse deviation dev between the expected disDegree = n and the runtime distribution degree, i.e., E(x).*

Proof. If Π follows a uniform distribution, i.e., p_i=$1/N$, based on Equation 2, the deviation of distribution degree $dev_u(N)$ is:

$dev_u(N) = n - E(x) = n - N + N \cdot (1 - \frac{1}{N})^n$.

We have $dev'_u(N) = (1 - \frac{1}{N})^{n-1} \cdot (1 + \frac{n-1}{N}) - 1$.

Let $x = \frac{1}{N} (\in (0, \frac{1}{2}])$ and $g(x) = (1 - x)^{n-1}(1 + (n - 1) \cdot x) - 1$.

Then $g'(x) = -n \cdot (n - 1) \cdot (1 - x)^{n-2} \cdot x$.

Since $g'(x) < 0$ with $0 < x \le 1/2$, $g(x) < g(0) = 0$, i.e., $dev'_u(N) < 0$. So $dev_u(N)$ is monotonically decreasing. That is the deviation of *disDegree* decreases for a larger cluster if we assign n distinct partition values for SQLs in a transaction under a uniform distribution. □

Corollary 4.2. *If Π follows a non-uniform distribution, directly assigning n distinct partition values to n parameters for SQLs in a transaction, the deviation dev_{nu} between the expected disDegree =n and the runtime disDegree, i.e., $E(x)$, is larger than the deviation of the uniform distribution under the same sized cluster, i.e., $dev_{nu} > dev_u$.*

Proof. For a non-uniform distribution Π, the deviation $dev_{nu}(\Pi)$ is $dev_{nu}(\Pi) = n - N + \sum_{i=1}^{N}(1 - p_i)^n$.

According to *Jensen Inequality* [40], we have

$dev_{nu}(\Pi) \ge n - N + N(1 - \frac{\sum_{i=1}^{N} p_i}{N})^n = n - N + N(1 - \frac{1}{N})^n$

For a uniform distribution, we have p_i=$1/N$. Then $dev_{nu}(\Pi)$ is minimized under a uniform distribution. So for a non-uniform distribution, the deviation of *disDegree* will be further enlarged. □

4.2.2 Contention Generation. Contention has an obvious negative impact on performance [28, 63, 68]. We propose **Contention Intensity** (CI in Definition 1) to represent the read/write intensity on data x among transactions at a time point. Since the brute force *blocking and waiting* way to organize threads and generate contentions has been proved not scalable [72], we design to create quantitative contentions on-the-fly agilely.

Definition 1. ***Contention Intensity*(CI)** *is the number of transactions (threads) competing for data x at a specific timepoint, one of which must be a write.*

To simplify the explanation, we describe contention generation to an individual data x. Our design is based on two observations:

- Transactions assigned to the same thread are executed in order.
- Only one of the contended transactions on data x can be processed at the same timepoint.

For example, in Fig. 8, there are five threads, three of which (T_{1-3}) compete for data x and two of which (T_{4-5}) perform non-contention transactions on data y and z respectively. All transactions contending for x or managed by one single thread have a partial order relationship $\ll$ on the execution time. In Fig. 8, if we have $\ll$ relationship among T_{1-3} as $T_2 \ll T_1 \ll T_3$, T_2 acquires the exclusive lock on x first, then T_1 and T_3 serially.

A majority of distributed transactional databases take pessimistic locking to implement concurrency control protocols [29, 48]. In such a design, for a specified type of contention on a single data x, it usually has a stable throughput, i.e., TPS_c, which is processed in

Luyi Qu[1†], Huidong Zhang[1†], Rong Zhang[1†], Peng Cai[1†],
Quanqing Xu[2‡], Zhifeng Yang[2‡], ChuanHui Yang[2‡]

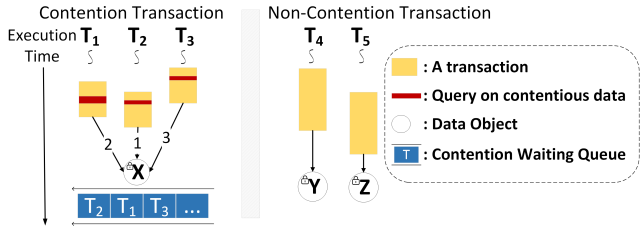


Figure 8: Example Data Accessing on Data x, y and z

order; the throughput of each type of non-contention transactions $w.r.t$ a single thread is also stable, i.e., TPS_{nc}. Considering the number of threads scheduled for contentions, i.e., CI, its throughput for non-contention transactions is formulated as $TPS_n = TPS_{nc} * (M - CI)$. Among all M threads, we can formulate the probability of each thread to be selected to launch contention on data x as $p_c = \frac{TPS_c}{TPS_c + TPS_n}$. Given a specified CI with a total size of threads M, we take the probability p_c to apply threads to write data x, and we can guarantee the specified CI in probability. It provides us a non-blocking way to simulate contentions and this approach can be extended to quantitatively generate contentions on multiple items with different contention intensities.

4.3 Imbalance Control

Imbalance of data or workload among nodes may cause performance bottleneck, which is expected to be tackled with scheduling.

4.3.1 Data Generator. TPC-C consists of nine tables and the size of each table, except *Item*, is dependent on the number of *Warehouses* and scales out according to predefined factors labeled in Fig. 9. For example, each warehouse provides services for 10 districts and each district has $3 \cdot 10^3$ customers. All tables except *Item* can partition their data according to warehouses, i.e., *w_id*. It then has a good property to be partitioned and distributed among nodes uniformly, and workload evenly accesses data in tables related to an individual warehouse. Balance of data and workload can not be used to evaluate the schedulability of a distributed transactional database. So we propose to generate skewed scale factors according to *Zipfian Distribution* instead of a uniform one for different tables.

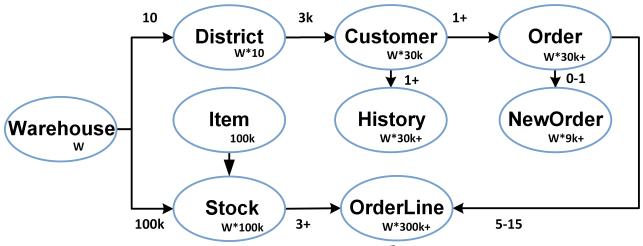


Figure 9: TPC-C Schema

Let's take *District* for example. We order warehouses and assign districts to warehouses following a *Zipfian Distribution*. Specifically, the order is inversely proportional to its assigned data. Given a skew factor s, the proportion of districts for the k^{th} warehouse is formulated as $f(k; s, N) = \frac{\frac{1}{k^s}}{\sum_{n=1}^{N} \frac{1}{n^s}}$. A larger scale factor s means a more concentrated data distribution in the highly ranked warehouse.

4.3.2 Workload Generator. Two skewed accesses are introduced to break the balance of workload distributions in TPC-C, including **Skew Partition Access** and **Skew Co-access**. Data is usually partitioned to nodes by different partition rules, which can be obtained in schema. Skew Partition Access controls workload to access single partitions following an skew distribution. For example, in Fig. 10, data (d_i) is organized into 4 partitions (P_{1-4}) distributed to 4 nodes. The probabilities of data access are skewed, that is 80%, 10%, 5% and 5% respectively. Skew Co-access expects data in different partitions accessed together in a controllable probability. As in Fig. 10, two items d_{11} (in P_1) and d_{23} (in P_2) have 80% probability to co-exist in the same transaction, which form a co-access group, i.e., $G1$. We expect DBMS to schedule the skewed access data into different nodes to balance workload and move the co-access data into a single parition to reduce network I/O.

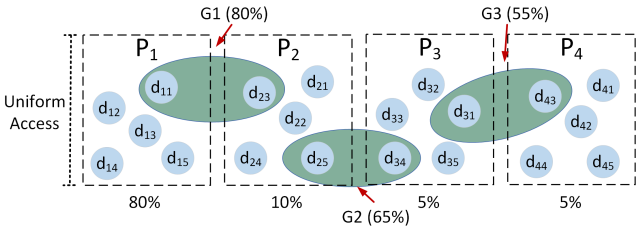


Figure 10: Example Data Distribution

4.4 Metrics Definition

Besides the traditional evaluation metrics, such as *TPS* and *Latency*, we define three new metrics for benchmarking database ability in scalability, availability, and schedulability.

(1) *Scalability*: It is evaluated by the deviation of throughput per node ($\frac{TPS}{N}$) under different cluster sizes N, e.g., from N_1 to N_2 in Equation 3. A better scalability expects to have a stable throughput per node, i.e., $\frac{TPS}{N2} - \frac{TPS}{N1} \rightarrow 0$, but with a high performance.

$$scalability = 1 - \frac{|\frac{TPS}{N2} - \frac{TPS}{N1}|}{\frac{TPS}{N1}}, N_1 < N_2 \tag{3}$$

(2) *Availability*: DDBMS expects to provide uninterrupted services by replications, when meeting an exception event e. We propose to represent *availability* by two dimensions, i.e., the stability of throughput (R_e) as well as the recovery efficiency (T_e), defined in Equation 4. R_e refers to the degree of performance degradation when failures occur, in which TPS_e and TPS_n stand for the throughputs w and w/o exception respectively; T_e refers to the efficiency to restore a stable performance TPS_e with t as the recovery time duration.

$$R_e = 1 - \frac{|TPS_e - TPS_n|}{TPS_n}; \qquad T_e = \frac{1}{t+1} \tag{4}$$

(3) *Schedulability*: Schedule expects to adaptively balance data distribution and transaction processing. We propose to show the data size per node and operation per second(*ops*) per node in the cluster by plotting boxplots of data size and *ops* on each node.

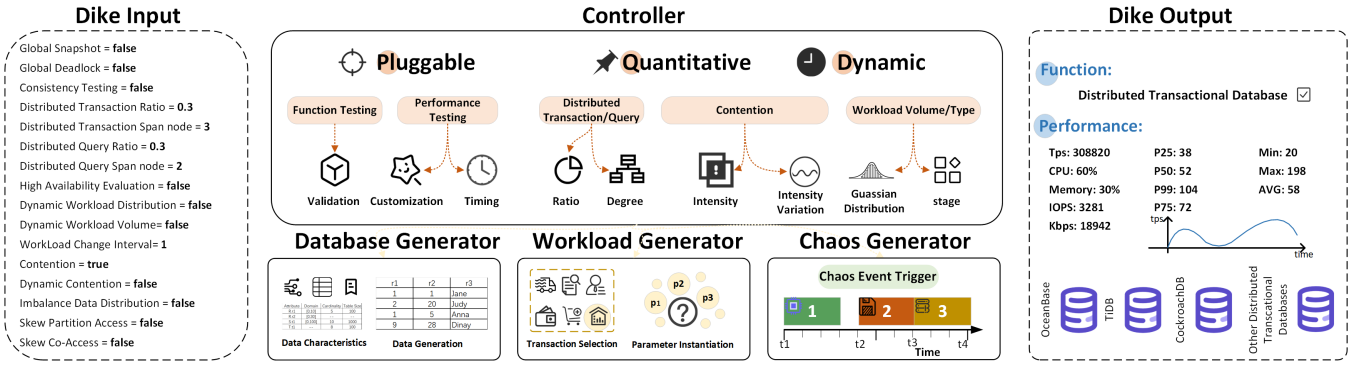


Figure 11: Dike Architecture

5 IMPLEMENTATION OF DIKE

Dike is proposed to benchmark performance of distributed transactional databases. Since databases employ different isolation levels (ILs) to achieve diverse guarantees of execution correctness. A qualified distributed transactional database should at first guarantee ACID properties. *Dike* provides functional testing first before launches performance evaluation which is extended from TPC-C by introducing new transactions to detect global deadlocks and check the consistency of global snapshots in a distribution DBMS. In Fig. 11, we show the overview architecture of *Dike*. It consists of six components, which are **Input**, **Controller**, **Database Generator**, **Workload Generator**, **Chaos Generator** and **Output**. We summarize the workload in Table 3 by comparing it with TPC-C.

		Dike	TPC-C
Scalability	**Distributed Transaction**	dNewOrder, Payment	NewOrder, Payment
	Ditributed Query	dStockLevel	×
Availability	**Failure Injection**	CPU failure, Node shutdown, Network jitter, IO failure	Node shutdown
Schedulability	**Contention**	dNewOrder	Add more warehouses
	Data/Workload Imbalance	Zipfian distribution / dUpdateStock	Uniform distribution
	Dynamicity	Workload volumn and type	Stable

Table 3: Transaction Templates for Both Dike and TPC-C

Input It provides 16 optional parameters to specify the evaluation scenario, i.e., for either functional verification or performance benchmarking. The detailed parameter explanation and example configurations for our following experiments are demonstrated in Appendix File A.

Controller It analyzes the user-defined configuration and provides instructions to *Database Generator*, *Workload Generator*, and *Chaos Generator*. It can customize characteristics for data and workload, which is pluggable to evaluate a database with specific designs in scalability, availability and schedulability. Specifically, for fair and comprehensive comparison:

- It quantifies transaction distribution by *disRatio* and *disDegree*.
- It specifies data contention by contention intensity *CI*.
- It simulates workload dynamics by changing quantity and accessing distribution of workload.
- It controls data imbalance by configuring skewness of data distribution.

dStockLevel

```
select count(*) as low_stock from (
    select s_w_id, s_i_id, s_quantity from bmsql_stock where s_w_id in (?,?,..,?) and
    s_quantity < ? and s_i_id in (
        select ol_i_id from bmsql_district join bmsql_order_line on ol_w_id = d_w_id and
        ol_d_id = d_id and ol_o_id >= d_next_o_id - 20 and ol_o_id < d_next_o_id where
        d_w_id = s_w_id and d_id = ?));
```

Figure 12: dStockLevel Transaction Template

Database Generator It populates database according to the required data characteristics from *Controller*. To break the well partition property of TPC-C, i.e., all tables except *Item* can be partitioned uniformly by *w_id*, we provide a skewed distribution for the scale factors of tables.

Workload Generator It receives instructions from *Controller*, selects the transaction template and instantiates its parameters to meet the specified requirement of workload characteristics. We control to create workload for benchmarking performance (1)-(3) and workload to test the correctness of transanctions(4).

(1) Distributed Transaction/Query: We control the parameter in the loop structure of *dNewOrder* (in Fig. **??**) to guarantee *disRatio* and *disDegree* *w.r.t* distributed writes. To create distributed queries, we extend *StockLevel* to *dStockLevel* by selecting its stock statistics from multiple warehouses according to user-defined *disDegree*. Sepecifically, *StockLevel* counts the items whose level of stock available at a single home warehouse is below the threshold. *dStockLevel* accesses multiple warehouses instead of one warehouse as shown in Fig. 12.

(2) Skew Access: Data are partitioned by *w_id* with the partition rule from the database schema. We extract *dUpdateStock* from *dNewOrder* as shown in Fig. 7 and based on the number of loops, i.e. *l*, we generate different skew accesses as:

- By setting $l = 1$, according to a selected skewed access function, e.g., exponential distribution, we select warehouses to fill *s_w_id* parameters, and achieve Skew Partition Access.
- By setting $l > 1$, we select a fixed set of warehouses of different partitions to instantiate parameter *s_w_id* for different round of the loop to achieve Co-access Partition in a single transaction.

(3) Dynamics: In real scenarios, workload is dynamic along the timeline, which also challenges database for its schedulability [39, 59, 62]. We control to generate three types of dynamics [23], including 1) shifting its 'hot spot' and 'hot degree', i.e., CI, over time. 2) adjusting the distribution of different types of transactions. 3) changing the quantity of transactions by changing the mean and variance of *Guassian Distribution*.
(4) Function Testing: Since Weak ILs will result in substantial vulnerability to concurrency-based attacks [67], we require databases to provide Repeatable Read (RR) IL at least, which is also required by TPC-C [24]. We provide functional testing extended from TPC-C to its transaction processing for fair comparison.

- Global Deadlocks caused by distributed transactions challenge the correctness of DDBMS, thus we propose to create deadlocks across nodes to test the capability to deal with global deadlocks. However, TPC-C avoids deadlocks by carefully control the sequence of lock Acquiring. We define a new transaction named *dUpdateBalance* as shown in Fig. 13 to create global deadlocks by randomly generating warehouses according to a Zipfian distribution in the loop structure and updating *w_ytd* (the year to date balance) of each warehouse. Under multiple concurrent threads, a distributed deadlock is caused by the different order of updating to *w_ytd*.
- Global Snapshot determines whether a DDBMS can support distributed Repeatable Read and stricter isolation levels (IL). A strict IL guarantees less anomalies for business, which is imperative for critical applications. For this purpose, we specify a consistency constraint to *w_ytd* in *dUpdateBalance* in Fig. 13, which exists in several records and should be equal. After the frequent update to *w_ytd*, we run *dSelectBalance* in Fig. 13 to judge whether the consistency constraint is broken for running a period of *dUpdateBalance*. If inconsistency happens, it does not pass the global snapshot test.

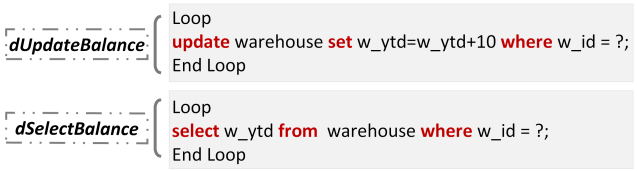


Figure 13: Two Function Testing Transaction Templates

Chaos Generator To create possible severe situation by making exceptions orchestrated along with the normal workload periodically is crucial for evaluating availability of a distributed database service. Currently, Dike supports failures or errors of CPU, Node, Network and Disk.

Evaluator It collects the runtime execution performance and outputs to files. *Dike* currently supports all mysql-compatible DDBMS including TiDB [32], OceanBase [69] and PolarDB [13], and postgresql-compatible DDBMS such as ,CockroachDB [61] and Citus [18]. It can be easily adapted to other databases.

6 EXPERIMENTAL EVALUATION

In this section, we demonstrate the specific design characteristics of Dike, which will be well exposed by its ability in evaluating scalability, availability, schedulability in the mainstream distributed transactional DBMSs. We run experiments to answer the following questions:

- Performance of current benchmarks in quantifying control of distributed transactions and contentions. (§ 6.1)
- Can *Dike* control to generate workload quantitively for testing scalability, availability and schedulability? (§ 6.2)
- Can *Dike* distinguish the scalability of different databases? (§ 6.3)
- Can *Dike* expose the availability of different databases? (§6.4)
- Can *Dike* tell the schedulability of different databases? (§6.5)

Cluster Environment: We build a 12-node cluster on UCloud. Specifically, 9 servers are used to deploy distributed database management systems (DDBMS), configured with 8 Intel Cascadelake 6248R @ 3.0GHz CPU, 32GB memory, and 550GB SSD as data disk; 3 servers act as clients, configured with 16GB memory and 100GB SSD as data disk. Servers are connected using Gigabit Ethernet.
Databases: Since *Shared-Storage* model does not have distributed transactions, e.g., Aurora or PolarDB, we deploy three distributed transactional databases, which are of a *Shared-Nothing* model, i.e., OceanBase (v3.1.3) [69], and of a *Shared-Nothing with Separation* model, i.e., TiDB (v6.0.0) [32] and CockroachDB (v22.1.1) [61]. Each node of OceanBase covers the functionality modules of *Query*, *Transaction* and *Storage* engines, called OBServer. OBProxy is a stateless connector to clients, which routes SQL statements by the proxy table. Specifically, it can take a predefined rule to partition data into multiple (distributed) nodes. We deploy 9 OBServers on 9 machines respectively and 1 OBProxy on each client node. TiDB includes a distributed *Storage Layer*, a *Computation Layer* and a *Placement Driver* (PD). The storage layer consists of a row-based store called tikv and the data stored is organized and partitioned into many regions. The compute layer is consisted of stateless tidb SQL engine instances. PD launches scheduling by collecting statistic information from servers, monitoring the workloads of each server and migrating hot regions to different servers. We deploy 9 tikv and 9 tidb on 9 machines respectively, 1 PD on one of them, and 3 HAProxies on three client nodes. CockroachDB includes *SQL Layer*, *Transaction Layer*, *Distribution Layer*, *Replication Layer* and *Storage Layer*. Specifically, the distribution layer is in charge of dividing the data into 'ranges' and arranging the placement of ranges, and the replication layer replicates data among nodes and ensures consistency among these copies by a consensus algorithm. We deploy one CockroachDB instance on each node and 3 HAProxies on three client nodes.
Dike Settings: Since *Dike* is extended from TPC-C, we follow the default database isolation level (IL) requirement of TPC-C, i.e., *Repeatable Read*. However, CockroachDB only has the IL of *serializable*, which is used by default. The detailed configurations for experiments are listed in Appendix **??-??**. In default, the number of warehouses is 20 per node, which is the same as the thread number. During performance evaluation, we run each experiment for the 300-second with the first 60-second as the warm-up time, and all results are the averages of five rounds of runs.
Functional Testing: We evaluate the ability of three databases in dealing with global deadlocks and global snapshot. All of them pass the testing, and then we launch the performance testing in the following section.

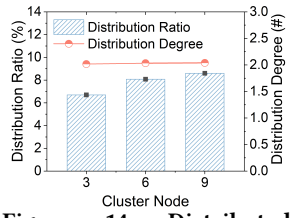


Figure 14: Distributed Transactions

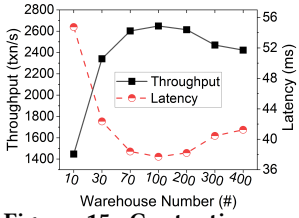


Figure 15: Contention *vs.* Data Size

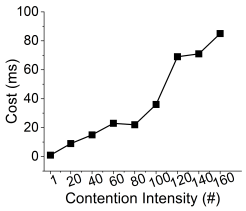


Figure 16: Cost *vs.* Contention

6.1 Quantitative Control Capability on Current Benchmarks

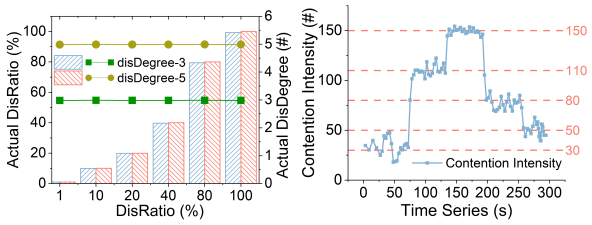
In this section, based on OceanBase, we demonstrate results to analyze: (1) the capability of TPC-C to control distributed transaction and contention quantitatively, and (2) the cost brought by PeakBench when it controls the contention intensity.

6.1.1 Capability of TPC-C on Quantitative Control. TPC-C is the most popular OLTP benchmark, but it is partition-friendly, i.e., most transactions can be well mapped to an individual partition located on a single node.

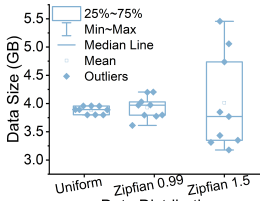
As for distributed transaction, only transaction *Payment* and *NewOrder* have the chance to generate distributed transactions. Specifically, in *Payment*, it has 15% probability to generate a distributed transaction with cross nodes <2. In *NewOrder*, any item bought has a probability of 1% from a remote node. Since *Payment* has a fixed range of distributed transactions with a fixed cross-nodes as explained above, we then run *NewOrder* on OceanBase by setting the warehouse number equals to the concurrent threads to demonstrate the distributions of transactions. We deploy 100 warehouses on a 3-node cluster, 200 on a 6-node cluster and 300 on a 9-node cluster in Fig. 14. On different sizes of clusters, the distributions are stable and low.

As for contention, TPC-C changes contention by resizing the number of warehouses; Specifically, we run TPC-C on OceanBase under different numbers of warehouses with 100 threads. Enlarging database does boost throughput at the beginning, but the performance decreases from 200 warehouses as shown in Fig. 15. This is because a large data size has a side-effect to database performance.

6.1.2 Cost of PeakBench on Quantitatively Controlling Contention. Though PeakBench has proposed a way to generate contentions quantitatively. We would like to explore its cost when quantitatively generating a given contention intensity. In Fig. 16, we deploy 180 warehouses on a 3-node cluster under 180 threads and count how long it takes to gather the expected contention threads on the client side as *Cost*. By increasing the contention intensity, the cost is also linearly increasing. Therefore, it is not scalable based on the thread blocking and waiting algorithm.

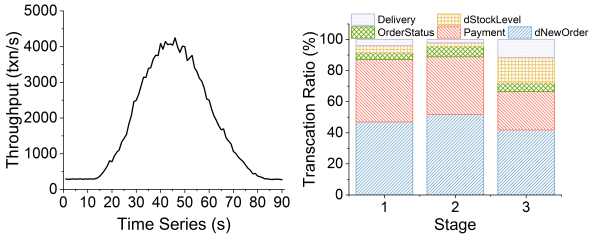


(a) Distributed Transaction Control (b) Contention Control



(c) Data Distribution Control

Figure 17: Validation of Quantitative Control of Dike



(a) Dynamic Workload Volume (b) Dynamic Workload Type

Figure 18: Validation of Dynamicity Control

6.2 Key Design Features in Dike

In this section, based on OceanBase, we demonstrate results to tell: (1) Whether the distribution control algorithm is effective, (2) how well contention control methods act for contention manipulation, and (3) whether the data distribution control method is effective.

6.2.1 Effectiveness of the quantitatively distributed transaction simulation. We run *dNewOrder* by instantiating its parameters according to the expected *disRatio* and *disDegree*. During running the workload, we count the actual *disRatio* and *disDegree* by *gv$partition* in OceanBase, which are then compared with our configured ones. In Fig. 17(a), we adjust *DisRatio* from 1% to 100%, by setting *disDegree*=3 and 5 respectively. The result shows that both the actual *disRatio* and *disDegree* are highly close to our the designated ones.

6.2.2 Effectiveness of the quantitatively contention control. We deploy OceanBase with 180 warehouses, each of which is coupled with a thread for the optimal performance. In Fig. 17(b), we run *dNewOrder* for 300 seconds, which are divided into five phases with five different contention intensity expections, i.e., *CI*=30, 110, 150, 80 and 50 respectively. We count the actual contention intensity on client sides. The results show the actual contention intensity is slightly fluctuated around our settings, with the maximum CI deviation less than 10%.

6.2.3 Effectiveness of the data distribution control. To demonstrate the ability of generating an imbalance data distribution across cluster nodes, Dike loads data specified with different distributions, i.e., uniform distribution, imbalance distribution with *Zipfian 0.99* and *Zipfian 1.5*, into OceanBase. We then count the data size among the 9 nodes. The results show the degree of data distribution across nodes conforms to our settings as shown in Fig. 17(c).

6.2.4 Effectiveness of the dynamicity. To demonstrate the dynamicity of Dike, we vary workload volume according a Gaussian Distribution over time and generate three different distribution among transaction types. Specifically, in Fig. 18(a), we execute *Dike* in 90 seconds. The throughput follows a Gaussian Distribution as expectations. Besides, in Fig. 18(b), there are three distributions among transaction types, i.e., *dNewOrder*, *Payment*, *OrderStatus*, *dStockLevel* and *Delivery* into 3 stages, that is, 45%, 43%, 4%, 4%, 4%; 50%, 40%, 6%, 2%, 2% and 40%, 25%, 5%, 15%, 15%. This experimental effect well follow our expectation.

6.3 Scalability

We present end-to-end performance for the three baseline distributed transactional databases, i.e., OceanBase, TiDB and CockroachDB, in its scalability *w.r.t* the distributed transactions. Based on the results, we answer the following questions: (1) How much do distributed transactions deteriorate database performance? (2) How effective are the optimizations *w.r.t* distributed transaction processing designed in each database? (3) What is the impact of cluster size to the scalability of distributed transactional databases?

6.3.1 The impact of distributed transaction. We vary *disRatio* under different *disDegree*, i.e., 3 and 5, and plot the change of performance in Fig. 19(a) and 19(b). Firstly, the throughput of OceanBase is 2× more higher than that of the other two databases. From the view of design architectures, OceanBase is a *Shared-Nothing* architecture, while TiDB and CockroachDB are of a *Shared-Nothing with Separation* architecture. So TiDB and CockroachDB have longer transaction processing paths, i.e., more messages in the network, which lead to worse performance. More over, from the view of distributed transaction processing, when a transaction accesses different regions of the same node, it will act as visit a single partition processing in OceanBase by its default *tablegroup* mechanism, but the other two databases still execute 2PC among partitions.

When we increase *disRatio* from 1% to 100%, TPS of OceanBase decreases by 22.5% and 39% for *disDegree*=3 and *disDegree*=5 respectively, while TiDB and CockroachDB decreases less, which are 16% and 9% respectively for *disDegree*=3, and 22% and 15% respectively for *disDegree*=5. The reason is that TiDB and CockroachDB are executed as distributed transactions, which are highly related to the number of visiting partitions instead of nodes. So increasing *disRatio* just makes their distributed transaction degree increase, which has a slight impact on performance degradation. In OceanBase, however, increasing *disRatio* means increasing distributed transactions in practice, so the processing cost will increase obviously.

6.3.2 The impact of distributed query. We vary *disDegree* to 1 (local), 2, 3, 4 and 5 for *dStockLevel* by setting *disRatio*=100%. To avoid the bottleneck of CPU (may be caused by range queries), we

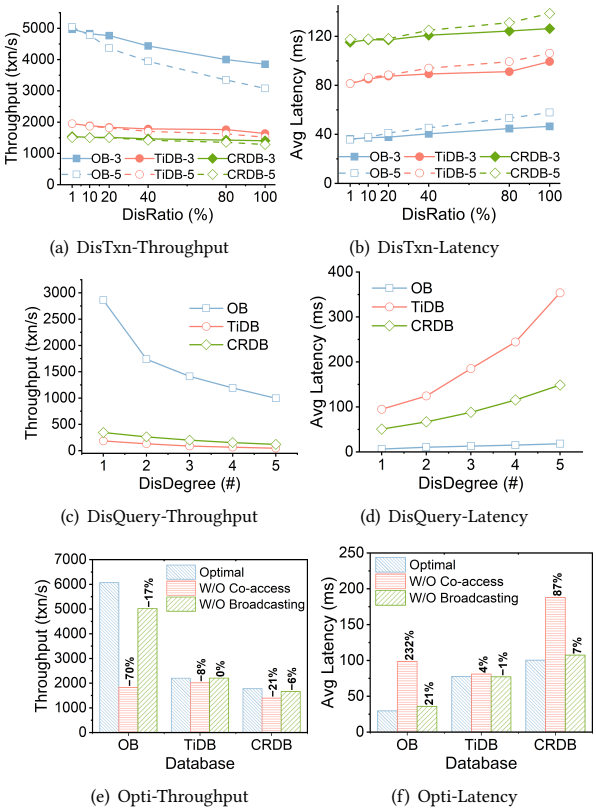


Figure 19: Performance vs. Distributed Transaction, Distributed Query, Optimizations

set two threads on each node, i.e., totally 18 threads on 9 nodes. The performance is shown in Fig. 19(c) and 19(d). We can see the throughput of OceanBase is about 9× (resp. 6×) higher than that of the other two databases for the local (resp. distributed) queries. Firstly, in OceanBase, the local query can be routed directly to the desired node; TiDB and CockroachDB randomly route the query to a compute node and have a high probability to pull substantial amounts of data from a remote storage node, so they have much low performance. Secondly, the distributed query can be executed in parallel across nodes and merged at the last step in OceanBase, which accelerates query processing; in the other two databases, however, compute nodes must pull all the data they need from remote nodes, which results in significant network overhead.

6.3.3 The impact of optimization means. In order to be scalable to distributed transactions, DDBMS provides optimization designs to improve performance. There are two common optimization means designed for distributed transactional databases, i.e., small-table broadcasting and co-access data binding, which are both adopted by default. Fig. 19(e) and 19(f) show performance changes by switching off either of the optimization ways, that is w/o binding co-access data or table broadcasting. We execute the whole Dike workload on databases. In Fig. 19(e) (resp. Fig. 19(f)), w/o co-access data binding, the throughput (resp. latency) of OceanBase drops (resp. increases) drastically by 70% (resp. 232%). The throughputs of TiDB and CockroachDB drop slightly by 8% and 21% respectively. The

reason is that switching off the optimization introduces more distributed transactions in OceanBase, which have severer side effect to OceanBase than to the other two databases. Besides, we explore the latency of each type of transaction and observe that *dStockLevel* costs 65× longer time in process, i.e., growing from 20 ms to 1300 ms, for the introduction of the distributed join in OceanBase. We can arrive at a conclusion that the co-access data binding is critical to improve performance, especially for *Shared-Nothing* architecture. But it is not easy to achieve this goal for the complex workload dependency, which may even evolve dynamically. As for the small-table broadcasting, it has more obvious impact on the performance of OceanBase than that of CockroachDB and TiDB. Specifically, even though TiDB replicates the small table on each tikv, only one leader node can access data and launch computing, which can not make full use of the replication of the small table.

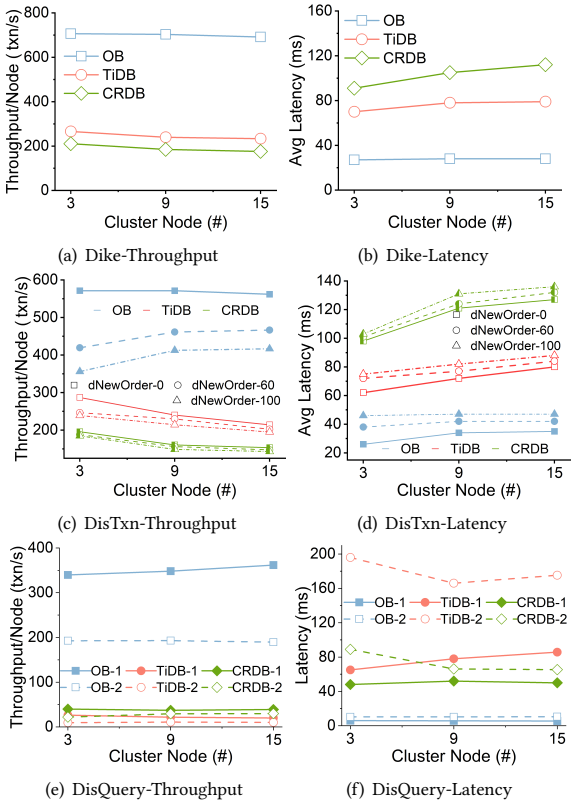


Figure 20: Performance of *Dike*, Distributed Transaction, Distributed Query vs. Cluster Size

6.3.4 Scalability over various cluster sizes. We expose the impact of cluster sizes to database performance *w.r.t* different workloads, i.e., *Dike* for the whole application scenario, *dNewOrder* for distributed transactions, and *dStockLevel* for distributed queries. For *Dike* and *dNewOrder*, we couple each thread with each warehouse; for *dStockLevel*, since it is a computation intensive workload, to avoid CPU bottleneck, we set 2 threads on each node.

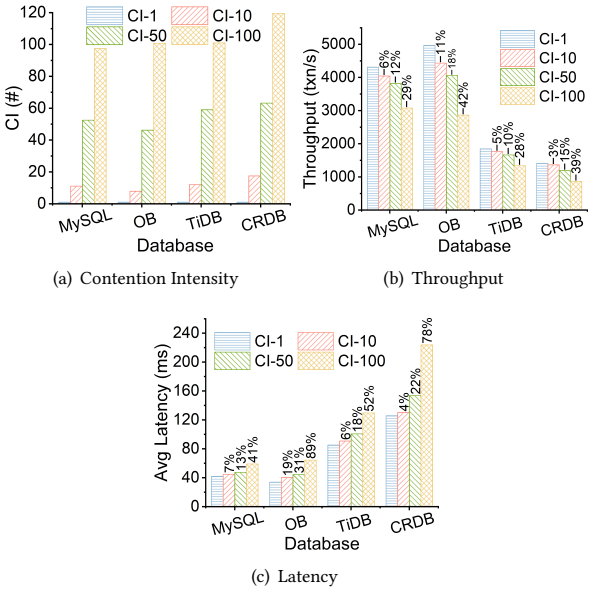
To avoid the disturbance from contention and distributed processing, we run *Dike* under the default configuration without contention and distributed transaction. In Fig. 20(a) and 20(b), OceanBase has the most excellent scalability *w.r.t* varying cluster sizes, i.e., from 3 nodes to 15 nodes (a little drop about 3% for its performance). Compared to OceanBase, both TiDB and CockroachDB have one additional hop from HAProxy randomly to the computation node during transaction processing, i.e., introducing an extra RPC. It leads to a worse performance degradation when increasing cluster sizes, for it lowers down the probability of routing transactions to computation node which locates on the same physical node as data partitions. We can conclude that *Shared-Nothing* architecture has a better scalability *w.r.t* cluster sizes for its less network communication for well partitioned workload.

In Fig. 20(c) and 20(d), we run only distributed write transactions, i.e., *dNewOrder* by setting *disDegree*=3 and varying *disRatio* from 0% to 100%. Note that *dNewOrder-k* means k% *dNewOrder* are distributed transactions. TiDB and CockroachDB have a sharp drop in performance under different *disRatio* when increasing cluster sizes because of more extra RPC. Notice that OceanBase is stable in performance from 9 nodes to 15 nodes, but has an increasing from 3 nodes to 9. This is due to the fact that our designed distribution control algorithm in Algo. 1 samples more warehouses than expected ones when we set *disDegree*=3 on a small cluster, which then introduces more overhead to manage the internal latches. So to make an effective distribution simulation, we are required to run our algorithm in a relative a cluster at least larger than *disDegree*. So we can see that OceanBase has a much better scalability in distributed transaction processing *w.r.t* different cluster sizes, compared to TiDB and CockroachDB.

In Fig. 20(e) and 20(f), we run distributed query *dStockLevel* by varing *disDegree* to 1 (local) and 2, respectively. OceanBase has a slight performance increasing on a large cluster *w.r.t* its local query processing, i.e., *OB-1*. This is due to its good ability of distributed parallel computations. Unfortunately, as the cluster size grows, TiDB and CockroachDB perform worse when handling local queries because it is less likely that the query will be routed to a compute node and a storage node that are both on the same physical node. When the selected compute node is not the same as the data stored node, there is a costly data pulling from a remote node to the single compute node. For distributed query, TiDB and CockroachDB exhibit a performance decline followed by an increase because Algo. 1 needs to access a fewer warehouses if cluster size rises. It ultimately results in less data accessing, improving performance. But, enlarging the cluster decreases the likelihood of routing to the local compute node, leading to a decline in performance. The trade-off between these two factors leads to this phenomenon.

6.3.5 Contention. To demonstrate the impact of contentions, we running *dNewOrder* by varying contention intensity *CI* in Fig. 21. Notice that *CI*=1, i.e., *CI-1*, represents no contention. In OceanBase under *RR* IL, when two transactions write the same data, one of them will simply rollback. In order to keep all transactions correctly executed, it proposes to launch contention transaction management in client-sides. Specifically, when the client catches a rollback transaction due to contention, it adds the transaction to the contention queue and suspends the thread until the previous contention transactions have been completed. Except for CockroachDB, we demonstrate the effectiveness of our precise contention control methods as in Fig. 21(a). CockroachDB meets *CI* more than the specified one,

Luyi Qu[1†], Huidong Zhang[1†], Rong Zhang[1†], Peng Cai[1†],
Quanqing Xu[2‡], Zhifeng Yang[2‡], ChuanHui Yang[2‡]



(a) Contention Intensity (b) Throughput

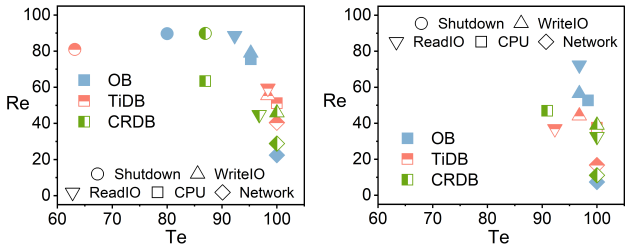
(c) Latency

Figure 21: Performance vs. Contention

because it would abort the transaction so as to avoid introducing serializable anomalies caused by contentions which requires client side to retry rollback transactions. *CI* does have a side-effect on performance of all database, i.e., the more intense the *CI*, the more degradation in performance. OceanBase has a severer performance degradation because handling contention on the application side is more expensive. Besides, we also check the impact of *CI* on a standalone database, i.e., MySQL with IL=*Repeatable Read*. MySQL has a lower performance degradation than distributed databases because of its shorter transaction processing length, i.e., the lowest latency.

6.4 Availability

We inject 5 types of exceptions to databases, i.e., *CPU exception*, *Shutdown*, *Write IO exception*, *Read IO exception* and *Network exception*. Specifically, we limit the database processes to 3 cores (totally 8 cores) when introducing *CPU exceptions* (half of the actual CPU usage when executing Dike), launch a node shutdown for *Shutdown exceptions*, impose additional IO read/write bandwidth for *Write/Read IO exceptions* by loading extra IO workloads, and apply an additional 50ms delay(50ms is just the right amount of pause to make a perceptible) to NIC (Network Interface Card) service to simulate *Network exceptions*. (more details in Appendix A.4). Single-Node and Three-Node in Fig. 22 mean to inject exceptions to one node and three nodes respectively. Notice that for a high availability service in a database, it requires three replicas managed by different consistent protocols. To avoid the risk of data loss, we do not introduce *Shutdown Exceptions* on three nodes. *Dike* has defined *Te* and *Re* to describe the recovery efficiency and the degree of performance preservation *w.r.t* exceptions. Essentially, databases in the up-right corner of Fig. 22 have a good availability. From the results, we can see that : 1) *Write/Read IO exception* has a stronger impact on TiDB and CockroachDB than Oceanbase. This is because the former two databases require more write iops than the latter



(a) Single-Node Exception (b) Three-Node Exception

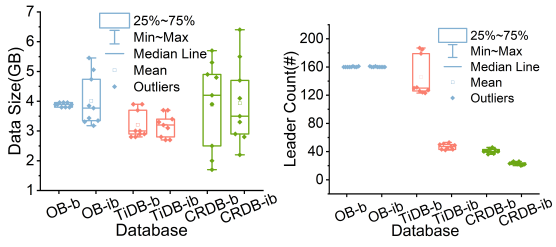
Figure 22: Service Availability of Three Databases

one by using percolator-like transaction processing mechanism. 2) TiDB has a worse availability than the other two databases *w.r.t CPU exception*. Because the compute layer and storage layer run as two processes, implemented by different languages (e.g., go in tidb, rust in tikv), it leads to obvious overhead in computation coordination, e.g., the serialization and deserialization of messages. 3) *Network exception* causes dramatic performance degradation for all of them. Since increasing network latency blocks transaction execution, the side-effect of network is then inversely related to the response time of the database, i.e., the low latency database meets more transaction blocked. So OceanBase with the lowest response time is affected more by *Network exception*. 4) the performance degradation caused by *Shutdown* is almost the same, but *CockroachDB* has the quickest recovery due to its less states to maintain, e.g., no learner for consensus protocol.

6.5 Schedulability

6.5.1 Data balance ability. We generate data of *Dike* with a uniform distribution (-b) and the skewed distribution by Zipfian 1.5 (-ib) into the three databases respectively, all of which take its default hash/range-partition to data on *w_id*. Then we plot the data distribution across nodes in Fig. 23(a). For a uniform distribution, Oceanbase can perfectly divide the data on *w_id* and then have the best balance distribution, i.e., *OB-b*, with little deviation between the *max* and *min* data sizes on nodes. TiDB does not strictly follow the partition rule. It may launch partition split or merge automatically during data loading, so data size varies among nodes even though the data is uniform on partition keys, i.e., *TiDB-b*. Besides, CockroachDB cares nothing about data balance and applies no partition rules and then it has the worst balance on data for both *CRDB-b* and *CRDB-ib*. For skewed data distribution, all databases have imbalance data distributions, but TiDB designs a better schedule mechanism to balance data, that is to split big partition or merge small partitions during data loading.Then *TiDB-ib* has the least deviation of data distribution among nodes even better than *TiDB-b*, for the skewed data distribution is easy to trigger the merge or split process.

Since all databases have master (leader) and slave nodes organized by Paxos or Raft. We also collect the distribution of partitions located on each leader node, which provide runtime services, as in Fig. 23(b). Even though they may have imbalance distribution among nodes, such as TiDB and CockroachDB in Fig. 23(a), they all have a balanced data partition distribution among leaders, i.e., a balanced online transaction processing.



(a) Data Size (b) Leader Count

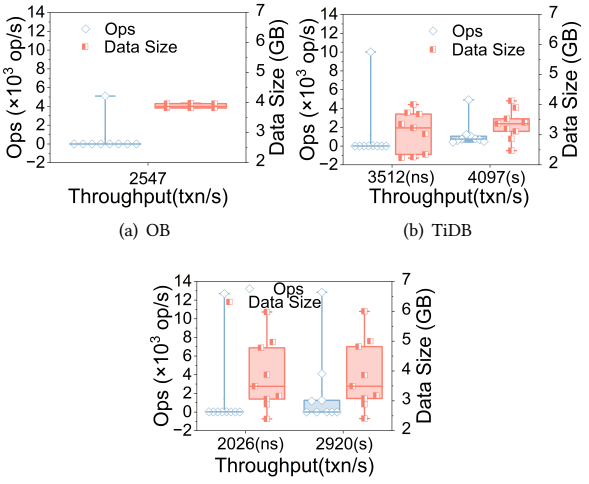
Figure 23: Data Balance Capability

6.5.2 Workload balance ability. We first investigate whether databases can be scheduled in terms of **Skew Partition Access**. To avoid bottlenecking database performance and making it easy to analyze schedulability with different workload distributions, we run *dUpdateStock* with totally 20 threads on 180 warehouses under a uniform data distribution. Then, in Fig. 24, we plot distributions of *ops* (operation per second) and data size among cluster nodes. Both TiDB and CockroachDB can switch on/off the scheduler for the hot spot access, but OceanBase designs no additional swith for the hot spot access. In OceanBase, only the node that contains the accessed data performs operations, all other nodes are idle. Besides, after executing *dUpdateStock*, the data distribution is still uniform among nodes. In TiDB and CockroachDB, if switching off the scheduler (ns), only one node provides data service; if enabling the scheduler (s), other nodes can provide services with throughput increasing by 16.7% and 44% respectively. So TiDB and CockroachDB have the ability to schedule hot partitions and improve performance. Specifically, TiDB balances data during the running; CockroachDB improves performance based on the idea of balancing workload by migrating data as little as possible. So we can see that CockroachDB has boosted its performance by 44% on almost the same imbalance data (small change on data distribution). Though, we have also launched experiments to investigate the schedulability *w.r.t* **Skew Co-access** using *dUpdateStock*. Unfortunately, we find that none of these databases trigger the scheduler to reduce distributed transactions. The main reason is that a general **Skew Co-access** detection demands an efficient online analysis to dynamic workload, which is greatly challenged by the quick response requirement.

6.6 Summary

In *Dike*, we have extended TPC-C by constructing *dNewOrder*, *dStockLevel* and *dUpdateStock* for distributed transactions/queries, and the distribution can be well controlled by our parameter instantiation algorithms. Based on the access probability *w.r.t* the assigned threads, we can simulate contentions in a non-blocking way. From our comprehensive experiments on representative distributed databases, we have obtained the following observation on their designs.

- In scalability, *Shared-Nothing* model, e.g., OceanBase, can have much better scalability than *Shared-Nothing with Separation*. For a well partition workload, *Shared-Nothing* model can achieve an almost linear scalability.



(a) OB (b) TiDB

(c) CRDB

Figure 24: Workload Balance Capability

- In availability, the more internal statues maintained by nodes, the slower the recovery. A high throughput distributed database is more sensitive to the network exception.
- In schedulability, data balance may not have a balanced workload distribution; dynamic migration of hot spot can siginificantly improve performance and to support co-access migration is now still a hard work.

7 RELATED WORK

Benchmarking is not a trival research topic for databases. As the expansion of application scenario, e.g., increase of data size, enlargement of database services or a further requirement of low latency, new databases with new design architectures come out, among which distributed transaction processing is a clear trend and attracts a lot efforts [9, 13, 16, 18, 21, 32, 34, 61, 65, 69, 74]. Compared to the stand-alone databases, distributed databases introduce new properties including **Scalability**, **Availability** and **Schedulability**. To further promote the development of databases, guarantee the healthy competition in database markets, and stimulate engineering and innovations, it becomes increasingly important over the last years to find a way to benchmark the distributed transactional databases.

7.1 Classic Transactional Benchmarks

Previous transactional benchmarks are mainly designed for stand-alone databases and most of them are constructed to simulate the characteristics of the specific applications. TATP [3], proposed by IBM, abstracts the operations in a telecommunication business application. The Transaction Performance Council (TPC) proposes TPC-C [4], which simulates a warehouse and order management application. After that, TPC proposes a more complicated benchmark, TPC-E [5] , which simulates the typical behaviors in a stock brokerage company. Besides, SmallBank [8] simulates a simple bank application. PeakBench [72] 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. YCSB [15] provides six types

of simple transactions referring to CRUD operation for key-value stores. Based on it, YCSB+T [22] is proposed to determine whether the consistency constraint is violated during transaction execution. These benchmarks are not designed specifically to evaluate distributed databases, which have not taken into account the representative characteristics of distributed transactional databases seriously.

7.2 New Requirements in Benchmark Design

7.2.1 Controllable Distribution Ratio and Degree. Distributed transactions and queries challenge the performance of distributed databases significantly. Although the majority of transactional benchmarks [4, 5, 8, 15, 72] can introduce distributed transactions, only TPC-C [4] and TPC-E [5] can generate a fixed ratio of distributed transactions with a fixed distributed degree. However, a thorough assessment of the performance of distributed transactional databases requires a controllable way to change the distribution ratio and degree.

7.2.2 Controllable Contention. Contention mitigation is critical to improve performance for both standalone databases and distributed transactional databases. Since contention is usually unavoidable among transactions, it has been a hot topic to tackle with contentions for a high performance [28, 63, 68]. Tranditional benchmarks [4, 15] simulate contention by either resizing the data access range or altering the data access distribution, e.g., *Zipfian Distribution*. They provide a coarse way to change the probability of contentions instead of an accurate contention simulation. Peakbench is the first work which proposes a queue-based blocking way for fine-grained contention control. Unfortunately, the performance can be bottlenecked by the client node, especially for a high contention application scenario, for it has to wait for the expected number of available threads.

7.2.3 Imbalance Control among Nodes. Distributed databases are expected to have a schedulability to balance the workload among distributed nodes and maximize the resource usages, which may be realized by data migration, or leader transferring [54, 60]. So simulating imbalance in both data and workload among nodes is vital in evaluating the schedulability of distributed databases. Previous benchmarks [4, 5, 23] pay more attention to skewed data distribution and access distribution on local nodes. They fail to take into account the skew distribution among nodes, which is highly related to the data placement policy.

7.3 Metrics for Benchmarking Distributed Transactional Databases

Scalability, availability and schedulability are three expectative properties of distributed transactional databases. Though the majority of transactional benchmarks [4, 5, 8, 15] support to scale data and concurrency, they do not explore the performance of the database when scaling out. OLxPBench [35] proposes to explore the scalability of TiDB and OceanBase, but it focuses on the performance isolation of these databases rather than transaction processing capacity when scaling out. For distributed environment, stable services from databases are expected with respective to hardware or software exceptions, i.e., availability [7, 42, 47, 66]. However, there is no consistent definition to evaluate the services from distributed transactional databases under different environmental anomalies. In terms of schedulability, HATtrick [41] and OLxPBench [35] only launch resource scheduling, e.g., CPU or Thread, that is to reassign resources between transactional processing and analytical processing to look into the change of performance. It can not be used to provoke an active action from the schedule module, if any.

8 CONCLUSION

Dike is a test suite for evaluating distributed transactional databases, extended from TPC-C in both data distribution and workload generation. It is designed mainly to evaluate three representative properties, i.e., **scalability**, **availability** and **schedulability**, in DDBMSs in a fair way. To compare the performance of distributed transactional databases, we provide guidelines on evaluation methodology and metrics, that is *quantitative control* and *imbalance control* for data and workload generation. We show that, compared with TPC-C, *Dike* provides **controllable distributed transaction ratio and degree**, **controllable contention intensity**, **imbalance data and workload generation** and **general environment exception**. From our experiment, we can arrive at some interesting conclusion that OceanBase performs well in the well-partition application scenarios which introduce less distributed transactions. In TiDB and CockroachDB, the separation of computation and storage will cause more RPC costs; even if the accessed data is from different partitions in a single physical node, it is still considered to be a distributed transaction. In respect to schedulability, TiDB pays attention to both the data balance across nodes and workload balance to achieve better performance, while CockroachDB can achieve the best performance by migrating a small amount of data. Unfortunately, the data balance and workload balance in OceanBase strictly follow the schema definition without adaptive scheduling. We expect fair comparison among the distributed transactional databases can promote the technical design in database architecture, transaction processing, fault solution and schedule ability [1].

REFERENCES

[1] Github URL. https://github.com/DBHammer/Dike.
[2] Quorum. https://github.com/jpmorganchase/quorum.
[3] TATP. http://tatpbenchmark.sourceforge.net.
[4] TPC-C. http://tpc.org/tpcc/default5.asp.
[5] TPC-E. http://tpc.org/tpce/default5.asp.
[6] Fault tolerance in distributed systems: A survey, author=Ledmi, Abdeldjalil and Bendjenna, Hakim and Hemam, Sofiane Mounine. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–5. IEEE, 2018.
[7] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 74–85. IEEE, 2013.
[8] M. Alomari, M. Cahill, A. Fekete, and U. Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585. IEEE, 2008.
[9] 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, and V. Wakade. Socrates: The new sql server in the cloud. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1743–1756, 2019.
[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.
[11] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.
[12] R. L. Burden and J. D. Faires. 2.1 The bisection algorithm. *Numerical analysis*, pages 46–52, 1985.
[13] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma. PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. volume 11, pages 1849–1862. VLDB Endowment, 2018.
[14] 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)*, 26(2):1–26, 2008.
[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.
[16] 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, and D. Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.
[17] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
[18] U. Cubukcu, O. Erdogan, S. Pathak, S. Sannakkayala, and M. Slot. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2490–2502, 2021.
[19] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.
[20] 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*, 41(6):205–220, 2007.
[21] A. Depoutovitch, C. Chen, J. Chen, P. Larson, S. Lin, J. Ng, W. Cui, Q. Liu, W. Huang, Y. Xiao, and 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*, pages 1463–1478, 2020.
[22] A. Dey, A. Fekete, R. Nambiar, and U. Röhm. YCSB+ T: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.
[23] B. Ding, S. Chaudhuri, J. Gehrke, and V. Narasayya. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment*, 14(13):3376–3388, 2021.
[24] Y. Gan, X. Ren, D. Ripberger, S. Blanas, and Y. Wang. IsoDiff: debugging anomalies caused by weak isolation. *Proceedings of the VLDB Endowment*, 13(12), 2020.
[25] L. George. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.", 2011.
[26] A. Geraci. *IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries.* IEEE Press, 1991.
[27] F. Gessert, F. Bücklers, and N. Ritter. Orestes: A scalable Database-as-a-Service architecture for low latency. In *2014 IEEE 30th international conference on data engineering workshops*, pages 215–222. IEEE, 2014.
[28] Z. Guo, K. Wu, C. Yan, and X. Yu. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data*, pages 658–670, 2021.
[29] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5):553–564, 2017.
[30] K. K. Hercule, M. M. Eugene, B. B. Paulin, and L. B. Joel. Study of the Master-Slave replication in a distributed database. *International Journal of Computer Science Issues (IJCSI)*, 8(5):319, 2011.
[31] H. Howard and R. Mortier. Paxos vs Raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–9, 2020.
[32] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.
[33] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 477–484. IEEE, 2002.
[34] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.
[35] G. Kang, L. Wang, W. Gao, F. Tang, and J. Zhan. OLxPBench: Real-time, Semantically Consistent, and Domain-specific are Essential in Benchmarking, Designing, and Implementing HTAP Systems. *arXiv preprint arXiv:2203.16095*, 2022.
[36] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.
[37] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.
[38] Y. Lu, X. Yu, L. Cao, and S. Madden. Epoch-based commit and replication in distributed OLTP databases. 2021.
[39] L. Ma, D. Van Aken, A. Hefny, G. Mezerhane, A. Pavlo, and G. J. Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645, 2018.
[40] E. J. McShane. Jensen's inequality. *Bulletin of the American Mathematical Society*, 43(8):521–527, 1937.
[41] E. Milkai, Y. Chronis, K. P. Gaffney, Z. Guo, J. M. Patel, and X. Yu. How Good is My HTAP System? In *Proceedings of the 2022 International Conference on Management of Data*, pages 1810–1824, 2022.
[42] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.
[43] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.
[44] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, 2014.
[45] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1137–1148, 2011.
[46] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.
[47] A. Pandey et al. Impact of memory intensive applications on performance of cloud virtual machine. In *2014 Recent Advances in Engineering and Computational Sciences (RAECS)*, pages 1–6. IEEE, 2014.
[48] A. Pavlo and M. Aslett. What's really new with NewSQL? *ACM Sigmod Record*, 45(2):45–55, 2016.
[49] A. Pavlo, C. Curino, and 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*, pages 61–72, 2012.
[50] A. Quamar, K. A. Kumar, and A. Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 430–441, 2013.
[51] D. P. Reed. *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, 1978.
[52] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1583–1598, 2016.
[53] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046, 2014.
[54] M. Serafini, R. Taft, A. J. Elmore, A. Pavlo, A. Aboulnaga, and M. Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[55] Y. Sheng, A. Tomasic, T. Zhang, and A. Pavlo. Scheduling OLTP transactions via learned abort prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–8, 2019.
[56] R. Shrestha. High Availability and Performance of Database in the Cloud-Traditional Master-slave Replication versus Modern Cluster-based Solutions. 2017.
[57] 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.
[58] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.
[59] R. Taft, N. El-Sayed, M. Serafini, Y. Lu, A. Aboulnaga, M. Stonebraker, R. Mayerhofer, and F. Andrade. P-store: An elastic database system with predictive provisioning. In *Proceedings of the 2018 International Conference on Management of Data*, pages 205–219, 2018.
[60] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.
[61] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.
[62] N. Tanković, N. Bogunović, T. G. Grbac, and M. Žagar. Analyzing incoming workload in Cloud business services. In *2015 23rd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 300–304. IEEE, 2015.
[63] B. Tian, J. Huang, B. Mozafari, and G. Schoenebeck. Contention-aware lock scheduling for transactional databases. *Proceedings of the VLDB Endowment*, 11(5):648–662, 2018.
[64] K. Q. Tran, J. F. Naughton, B. Sundarmurthy, and D. Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 39–50, 2014.
[65] 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 *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.
[66] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, et al. Manifold: A parallel simulation framework for multicore systems. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 106–115. IEEE, 2014.
[67] T. Warszawski and P. Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 5–20, 2017.
[68] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment*, 9(5):444–455, 2016.
[69] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi, H. Yu, B. Liu, Y. Pan, B. Yin, J. Chen, and Q. Xu. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proceedings of the VLDB Endowment*, 15(12):3385–3397, 2022.
[70] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. 2014.
[71] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 17–30, 2015.
[72] C. Zhang, Y. Li, R. Zhang, W. Qian, and A. Zhou. Benchmarking on intensive transaction processing. *Frontiers of Computer Science*, 14(5):1–18, 2020.
[73] T. Zhang, A. Tomasic, Y. Sheng, and A. Pavlo. Performance of OLTP via intelligent scheduling. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1288–1291. IEEE, 2018.
[74] 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, and V. Yadav. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2653–2666, 2021.

A CONFIGURATION

A.1 Default

```
db=oceanbase
resultDirectory=results/oceanbase/dike/
    my_result_%tY-%tm-%td_%tH%tM%tS

# Connection Properties
host=*.*.*.*
port=2883
set=dike
user=root@test
password=
## Mysql
useSSL=false
rewriteBatchedStatements=true
allowMultiQueries=true
useLocalSessionState=true
socketTimeout=300000
queryTimeout=300000
allowLoadLocalInfile=false
autoReconnect=true
useServerPrepStmts=false
useConfigs=
## Postgresql
sslmode=disable
reWriteBatchedInserts=true
transactionIsolation=1

# Workload Properties
warehouses=180
terminals=180
terminalRange=1,180
physicalNode=9
runMins=5
transactions=45,43,4,4,4,0,0,0,0

# Control Properties
## Distributed Transaction
newOrderDistributedRate=0
newOrderSpanNode=1
warehouseDistribution=uniform
terminalWarehouseFixed=false
## Distributed Query
stockLevelDistributedRate=0
stockLevelWIDNode=1
statisticsCalc=true
## Broadcast
broadcastTest=false
batchUpdate=false
accesssUpdateItemRate=20
## Read-Write seperation
readWriteSeperation=false
bandTransaction=false
## Dynamic District
dynamicDistrict=false
```

```
## Dynamic Load
dynamicLoad=false
## Dynamic Conflict
dynamicConflict=false
cilist=100
## Global Snapshot
snapshotTimes=4
## Global Deadlock
deadlockTimes=5
## Coaccess pattern
coaccessNumber=1
## Dynamic Transaction
dynamicTransaction=false
changeTransactions=50,40,6,2,2;40,25,5,15,15
changePoints=1,2

## Chaos Test
chaosNode=root@10.24.14.75
cpuLoad=false
stressMemory=false
diskRead=false
diskWrite=false
networkDelay=false
shutdown=false
chaosTime=2

# Statistics Properties
osCollector=false
osCollectorScript=/root/benchmarkStartSimple/run/misc/os_collector_linux.py
osCollectorSSHAddr=root@*.*.*.*,root@*.*.*.*,root@*.*.*.*,root@*.*.*.*
osCollectorDevices=net_eth0,blk_vdb
txnReportInterval=1

# Schema
schemaScript=tableCreates
partitions=9

# Load Data
loadWorkers=180
writeCSV=false
fileLocation=/data/dike

# Load Data Shell
dataSource=insert
```

A.2 Key Design Features in Dike

A.2.1 Effectiveness of the quantitatively distributed transaction simulation.

```
newOrderWeight=100
paymentWeight=0
orderStatusWeight=0
deliveryWeight=0
stockLevelWeight=0
newOrderDistributedRate=1/10/20/40/80/100
newOrderSpanNode=3/5
```

A.2.2 Effectiveness of the quantitatively contention control.

```
transactionIsolation=0(CRDB)/1(TiDB)/2(OceanBase)
dynamicConflict=true
conflictChangeInterval=1
cilist=30,110,150,80,50
```

A.2.3 Effectiveness of the dynamicity.

```
dynamicLoad=true
dynamicTransaction=true
```

A.3 Scalability

A.3.1 The impact of distributed transcation.

```
newOrderWeight=100
paymentWeight=0
orderStatusWeight=0
deliveryWeight=0
stockLevelWeight=0
newOrderDistributedRate=1/10/20/40/80/100
newOrderSpanNode=3/5
```

A.3.2 The impact of distributed query.

```
terminals=18
newOrderWeight=0
paymentWeight=0
orderStatusWeight=0
deliveryWeight=0
stockLevelWeight=100
stockLevelDistributedRate=100
stockLevelWIDNode=1/2/3/4/5
```

A.3.3 The impact of optimization means.

```
terminalWarehouseFixed=true
schemaScript=tableCreates/tableCreatesNoItem/tableCreatesNoTablegroup
```

A.3.4 Scalability over various cluster sizes. Dike/d-NewOrder

```
warehouses=60/180/300
terminals=60/180/300
terminalRange=1,60/1,180/1,300
terminalWarehouseFixed=true
newOrderWeight=100
paymentWeight=0
orderStatusWeight=0
deliveryWeight=0
stockLevelWeight=0
newOrderDistributedRate=0/20/60/100
newOrderSpanNode=3
```

dStockLevel

```
warehouses=60/180/300
terminals=6/18/30
terminalRange=1,60/1,180/1,300
newOrderWeight=0
paymentWeight=0
orderStatusWeight=0
deliveryWeight=0
stockLevelWeight=100
stockLevelDistributedRate=100
stockLevelWIDNode=1/2
```

A.3.5 Contention.

```
        transactionIsolation=0(CRDB)/1(TiDB/OceanBase)/2(TiDB/OceanBase)
        dynamicConflict=true
        conflictChangeInterval=1
        cilist=10/50/100
```

A.4 Availability

A.4.1 CPU exception.

```
chaosNode=root@*.*.*.*/root@*.*.*.*,root@*.*.*.*,root@*.*.*.*
cpuLoad=true
# shell
./blade create cpu load --cpu-list 0-5 --timeout 180
```

A.4.2 Read IO exception.

```
chaosNode=root@*.*.*.*/root@*.*.*.*,root@*.*.*.*,root@*.*.*.*
diskRead=true
# shell
./blade create disk burn --timeout 180 --read --path /data
```

A.4.3 Write IO exception.

```
chaosNode=root@*.*.*.*/root@*.*.*.*,root@*.*.*.*,root@*.*.*.*
diskWrite=true
# shell
./blade create disk burn --timeout 180 --write --path /data
```

A.4.4 Network exception.

```
chaosNode=root@*.*.*.*/root@*.*.*.*,root@*.*.*.*,root@*.*.*.*
networkDelay=true
# shell
./blade create network delay --timeout 180 --time 50 --interface eth0
```

A.4.5 Shutdown.

```
chaosNode=root@*.*.*.*/root@*.*.*.*,root@*.*.*.*,root@*.*.*.*
shutdown=true
# shell
shutdown -h now
```

A.5 Schedulability

A.5.1 Data balance ability.

```
dynamicDistrict=true
```

A.5.2 Work balance ability.

```
newOrderWeight=0
paymentWeight=0
orderStatusWeight=0
deliveryWeight=0
stockLevelWeight=0
updateStockWeight=100
coaccessNum=1/2
```