Dike: A Benchmark Suite for Distributed Transactional Databases

Huidong Zhang
hdzhang@stu.ecnu.edu.cn
East China Normal University
Shang Hai, China

Luyi Qu
luyiqu@stu.ecnu.edu.cn
East China Normal University
Shang Hai, China

Qingshuai Wang
qswang@stu.ecnu.edu.cn
East China Normal University
Shang Hai, China

Rong Zhang*
rzhang@dase.ecnu.edu.cn
East China Normal University
Shang Hai, China

Peng Cai
pcai@dase.ecnu.edu.cn
East China Normal University
Shang Hai, China

Quanqing Xu
xuquanqing.xqq@oceanbase.com
OceanBase, AntGroup
Hang Zhou, China

Zhifeng Yang
zhuweng.yzf@oceanbase.com
OceanBase, AntGroup
Hang Zhou, China

Chuanhui Yang
rizhao.ych@oceanbase.com
OceanBase, AntGroup
Hang Zhou, China

ABSTRACT

Distributed relational database management systems (abbr. DDBMSs) for online transaction processing (abbr. OLTP) have been gradually adopted in production environments. With many relevant products vying for the markets, an unbiased benchmark is urgently needed to promote the development of transactional DDBMSs. Current benchmarks for OLTP applications have not taken the challenges encountered during the designs and implementations of a transactional DDBMS into consideration, which expects to provide high elasticity and availability as well as high throughputs. We propose a benchmark suite *Dike* to evaluate the efforts to tackle these challenges. *Dike* is designed mainly from three aspects: quantitative control to evaluate *scalability*, imbalanced distribution to evaluate *schedulability*, and comprehensive fault injections to evaluate *availability*. It also provides a dynamic load control to simulate real-world scenarios. In this demonstration, users can experience core features of *Dike* with user-friendly interfaces.

CCS CONCEPTS

• **Information systems** → **Data management systems**.

KEYWORDS

Distributed Transactional Database, Benchmark Suite

ACM Reference Format:
Huidong Zhang, Luyi Qu, Qingshuai Wang, Rong Zhang, Peng Cai, Quanqing Xu, Zhifeng Yang, and Chuanhui Yang. 2023. Dike: A Benchmark Suite for Distributed Transactional Databases. In *Companion of the 2023 International Conference on Management of Data (SIGMOD-Companion '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3555041.3589710

*Rong Zhang is the corresponding author

SIGMOD-Companion '23, June 18–23, 2023, Seattle, WA, USA

ACM ISBN 978-1-4503-9507-6/23/06…$15.00
https://doi.org/10.1145/3555041.3589710

1 INTRODUCTION

DDBMSs provide elastic and available services to handle continuing business expansion. They have devoted lots of efforts to meet the requirements of *scalability*, *schedulability* and *availability*. Specifically, the database is partitioned among multiple servers for the horizontal scaling of storage and computing resources, i.e., *scalability*. Database migrates partitions between servers to balance storage usages and avoid performance bottlenecks caused by a single server overload, i.e., *schedulability*. Database stores multiple replicas for each partition [2], to guarantee no data loss or service interruption in the event of any system failures, i.e., *availability*.

Challenges arise when the database scales to distributed scenarios. Firstly, transactions involving partitions from different servers, aka, distributed transactions, suffer a lot from network communication for coordination on transaction statuses [3]. Complex business logic may lead to a large number of distributed transactions. Secondly, competitive accesses to hotspot data cost more in global transaction ordering, global deadlock detection and cascading rollback, since DDBMSs have longer transaction processing paths than standalone databases. Thirdly, when data and workload distribution among servers are imbalanced, online database service suffers from performance degradation and service level agreement violation. Fourthly, various system failures occur frequently in the cluster, challenging DDBMSs to guarantee service quality.

Existing classic transactional benchmarks for relational databases, such as TPC-C and TPC-E, are meticulously designed in database schemas and workloads [4], but not sufficient to cover the advanced designs of DDBMSs and evaluate the pros and cons of each DDBMS, which has been thoroughly discussed in work [1]. They can not (1) quantify the generation of distributed transactions or contentions, (2) generate skew data distribution or workloads to trigger data migration for DDBMSs, and (3) provide comprehensive fault injections. We propose a benchmark suite *Dike*, which follows the practical application scenario of TPC-C but enriches its capabilities to benchmark transactional DDBMSs. *Dike* is endowed with the following four representative characteristics, which are

- **Quantitative Control:** The first two challenges imply that distributed transactions and contentions block transaction execution and restrict the *scalability* of DDBMSs. Quantitative control of workload generation is necessary to facilitate impartial comparisons among DDBMSs.
- **Imbalanced Distribution:** The third challenge requires a benchmark to generate non-uniform data distributions and skew access patterns to partitions. These features can evaluate the *schedulability* of DDBMSs to support data migration and balance the system resource usage of servers.
- **Comprehensive Fault Injection:** The fourth challenge mentions that frequent and various exceptions plague DDBMSs. Chaos testing with comprehensive fault injections could verify the robustness and *availability* of DDBMSs.
- **Dynamic Load Variation:** In real application scenarios, load distributions vary over time in both volumes and types. DDBMSs expect to scale out (resp. down) to handle heavy (resp. light) loads to guarantee performance with low cost.

In this demonstration, we demonstrate *Dike* from (1) special designs to benchmark the *scalability*, *availability* and *schedulability* of transactional DDBMSs, (2) interactive user interfaces to operate *Dike* and benchmark results to verify the effectiveness of *Dike*.

2 DESIGN OVERVIEW OF DIKE

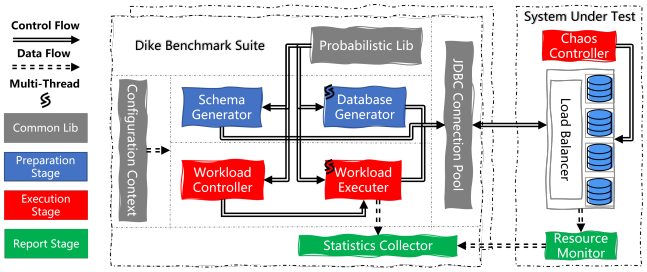

Figure 1: Dike Architecture

The system architecture of *Dike* is depicted in Figure 1. The control flow is in double solid lines, which mainly includes JDBC commands and quantitative, imbalanced, dynamic control of the benchmark process. The data flow is in double dashed lines, mainly including statistic results.

Dike is a standalone benchmark suite deployed on multiple clients for scalable workload generation. Three common components grayed out provide support for the benchmark process. *Configuration Context* parses control parameters for the user-preferred scenario. *Probabilistic Lib* implements probabilistic distribution models and quantitative control algorithms for database and workload generation. *JDBC Connection Pool* maintains connections to DDBMSs and works in the MySQL or PostgreSQL compatible way.

From the perspective of the benchmark process, the workflow can be divided into three stages, i.e., the preparation stage (marked in blue), the execution stage (marked in red) and the report stage (marked in green). In the preparation stage, *Schema Generator* creates TPC-C style database schemas based on partitioning and data placement strategies. *Database Generator* populates the database and controls data volume. In the execution stage, *Workload Executer* controls transaction proportions, instantiates transaction templates and interacts with DDBMSs via JDBC connection. *Workload Controller* is responsible for the variation of load patterns and sends control messages to *Workload Executer*, e.g., dynamic load volume. In the report stage, *Statistics Collector* receives system resource metrics from *Resource Monitor* and transaction traces from *Workload Executer*, and finally generates benchmark reports. Database and workload are generated in a multi-thread mode for efficiency.

The system under test (abbr. SUT) is a DDBMS, which may be deployed with *Load Balancer*, such as OBProxy for OceanBase [5] and HAProxy for CockroachDB. *Dike* deploys *Resource Monitor* on each server to record the utilization of system resources for performance diagnosis.

We illustrate our designs in the following four aspects, which are how to generate (1) quantitative distributed transactions, (2) quantitative contentions, (3) non-uniform data distribution and skew partition access pattern, and (4) various types of exceptions.

2.1 Quantitative Distributed Transaction

Distributed transactions suffer from high latency for acquiring data from remote servers, and adopting atomic commit protocols to achieve a consensus on all updates among servers.

The classic OLTP benchmark TPC-C generates distributed transactions by accessing data from different warehouses with a low unquantifiable probability [4]. Quantitative control of distributed transactions, which have a fixed number of cross-servers, can control to quantify the impact on database throughputs and make fair comparisons among DDBMSs.

Most tables in TPC-C, except for *Item*, are closely associated with *Warehouse*. Data scales with the cardinality of *Warehouse* [4] and partitions by the unique identifier of each warehouse, i.e., *wid*, which are the same for *Dike*. Specifically, transaction *NewOrder* in TPC-C simulates the behavior of product orderings, which mainly involves products supplied by the local warehouse. *Dike* extends the logic of *NewOrder* by controling the products from remote warehouses. It generates distributed transactions by updating table *Stock* of each warehouse in different servers.

Based on the access probability, we formalize the transaction distribution as follows. Suppose the number of servers in the cluster is N, the target number of cross-servers is n and the number of distinct warehouses is c. The probability that the data of a warehouse resides on i^{th} server is p_i with $\Sigma_{i=1}^{N} p_i = 1$. Whether *NewOrder* visits i^{th} server or not is denoted as $P(x_i = 0)$ and $P(x_i = 1)$ respectively, which are calculated in Equation 1.

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

To make the expectation of distinct servers visited by *NewOrder* close to the target n, we formalize the quantitative distribution control problem by Equation 2. Given N and n, c can be calculated through *Probabilistic Lib*. *Workload Executer* selects c distinct warehouses for *NewOrder* to instantiate transaction templates to create distributed transactions.

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

2.2 Quantitative Contention

Current DDBMSs mainly depend on lock-based methods to mitigate performance degradation caused by competing accesses to records [3]. For example, CockroachDB achieves its concurrency control by validating read-write dependencies, i.e., optimistic strategy, while it still provides *SELECT FOR UPDATE* dialect for applications to define explicit controls on row-level locking. Contention on locks is scheduled by Wound-Wait, Wait-Die, or other strategies. At a time point, only one transaction can hold the lock and proceed, while others wait in order. The observation is that transactions contending on a specified record execute sequentially, and non-contention transactions do not interfere with each other, both of which have stable throughputs.

To ease the illustration, we assume that contention exists on record x. Let Contention Intensity (abbr. *CI*) as the number of transactions competing to update x at a time point. *Dike* generates quantitative contentions in two phases. In the first phase, the number of *Workload Executers* that continuously execute contention transactions is set to *CI*, having throughput TPS_c; the other *Workload Executers* execute non-contention transactions, having throughput TPS_{nc}. In the second phase, *Workload Executers* launch contention transactions following the probability $\frac{TPS_c}{TPS_c+TPS_{nc}}$. Statistically, concurrent accesses to record x are equivalent to that in the first phase.

2.3 Skew Distributions of Data and Access Pattern

Data migration balances resources among servers to improve service quality. Most DDBMSs that provide data migration, determine whether to make a schedule by monitoring distributions of data or access patterns. To evaluate this design, *Dike* provides two scenarios that can trigger data migration.

Skew Data Distribution In TPC-C, data in all warehouses follows a uniform distribution. Take *District* for example. Let W denote the number of warehouses and each warehouse has the data of 10 *Districts* by default. *Dike* designs to generate non-uniform scale factors for each warehouse following a *Zipfian Distribution*. For example, the number of *District* for a warehouse is $d(k;s,W)$ with a scale factor $f(k;s,W)$, where k is the *wid* of a warehouse and s is the skew factor, defined in Equation 3.

$$d(k;s,W) = 10W \cdot f(k;s,W); \quad f(k;s,W) = \frac{\frac{1}{k^s}}{\sum_{n=1}^{W} \frac{1}{n^s}} \tag{3}$$

Skew Partition Access Pattern *Dike* extracts transaction *UpdateStock* from *NewOrder* to provide skew partition access workload. Specifically, *UpdateStock* updates the quantities of different items of a single warehouse in *Stock*. In this way, hotspots within a single partition are generated, since *Stock* is partitioned according to *wid*. *Dike* avoids contentions by assigning different *Workload Executers* to disjoint data ranges. The hotspot partition will be divided into smaller units and scattered to different servers if the DDBMS is sensitive to skew access patterns and provides migration.

2.4 Comprehensive Fault Injection

As DDBMSs expand to large scales, it is common that parts of the system encounter exceptions, such as network jitter, CPU preemption, disk blocking, data corruption and server shutdown. *Dike* provides comprehensive fault injections to explore the impact of exceptions on DDBMSs, which are controlled by *Chaos Controller* in Figure 1. Specifically, *Chaos Controller* maintains a timing task to trigger exceptions at a time point. Performance degradation after that point and service unavailable time can be observed.

3 DEMONSTRATION

We develop front-end and back-end interactive interfaces to demonstrate *Dike*. Firstly, we introduce the benchmark process of *Dike*, including how to submit a task, track task progress and check benchmark report. Secondly, we focus on six typical scenarios to describe the benchmark report and verify the effectiveness of our designs.

3.1 Configuration Interface

Dike is designed to evaluate DDBMSs' capability in supporting OLTP applications from multiple aspects. We provide users with a high-modularity interface as shown in Figure 2 (A). The interface contains three modules, (1) Basic Setting, including information about database cluster, data/workload scale and transaction control, (2) Connection Setting, including parameters to fill in JDBC connection, (3) Test Controller, including pluggable control switches to configure workloads for certain purposes, e.g., distributed transaction and fault injections. Detailed parameters for each workload are available in our Technical Report [1].

3.2 Task Tracing Interface

After configuration, we *Submit* to create a task in the back-end message queue. The running status of each task is displayed in the task tracing interface, as shown in Figure 2 (B). Each task has three possible statuses, *SUCCESS*, *PENDING* and *ERROR*. In the case of *SUCCESS*, the benchmark process finishes without errors, and benchmark report can be depicted in the result dashboard. In the case of *PENDING*, the benchmark task is in execution or waiting in the queue. In the case of *ERROR*, the benchmark process encounters errors and error logs can be depicted in the log dashboard.

3.3 Result Dashboard

We provide vivid graphic reports to present the statistic results, and conduct experiments from six typical scenarios to make a demonstration. We deploy OceanBase [5] v3.1.3 on 9 servers, and deploy *Dike* with OBProxy on the client. The data of 180 *Warehouses* is populated into the database. Screenshots of the result dashboard are displayed respectively in Figure 2 (C - H).

3.3.1 ***Scenario 1: Quantitative Distributed Transaction.*** We set the number of servers (N) to 9 and the target number of cross-servers (n) in *NewOrder* to 3 in Equation 2 for distributed transactions. During the parameter instantiation of transactions, we refer to the partition table inside OceanBase and calculate the number of involved servers. Figure 2 (C) shows that the average cross-servers of *NewOrder* are stably close to 3, which verifies the effectiveness of our design in the quantitative control of distributed transactions.

3.3.2 ***Scenario 2: Quantitative Contention.*** We dynamically change Contention Intensity (*CI*) to 30, 110, 150, 80, 50 every 60 seconds. Figure 2 (D) shows the real-time statistics of the actual contention intensity. In the first stage, we collect the number of

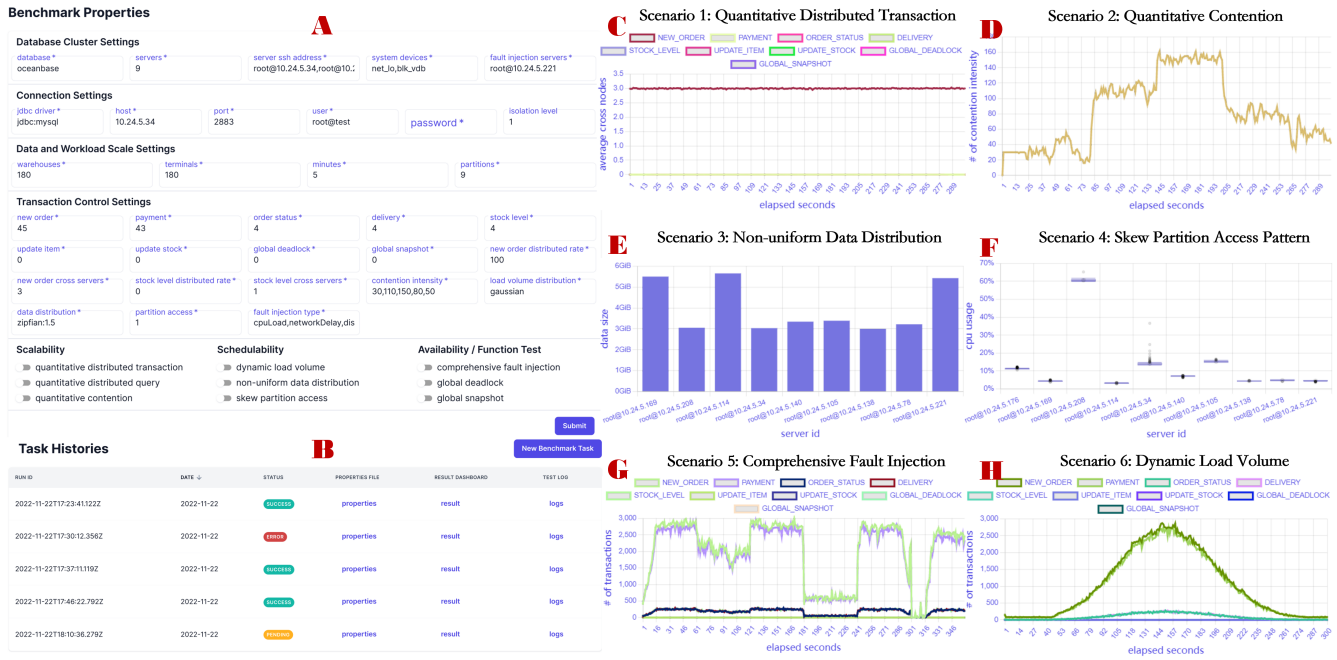

Figure 2: *Dike* User Interfaces

transactions finished by contention (TPS_c) and non-contention (TPS_{nc}) *Workload Executers* for 20 seconds. In the second stage, we randomly generate contention transactions based on the calculated probability. Actual contention intensity per second slightly fluctuates around the target settings and reacts quickly to *CI* changes.

3.3.3 ***Scenario 3: Non-uniform Data Distribution***. We set the skew factor (s) of *Zipfian Distribution* to 1.5 in Equation 3, where around 77% *Districts* are distributed in top 10 *Warehouses*. OceanBase uses hash partition on *wid*, and the data size of each server is depicted in Figure 2 (E). Three servers have much higher storage usage than others, one of which has the warehouse with the most districts and the other two have its replicas.

3.3.4 ***Scenario 4: Skew Partition Access Pattern***. We generate *UpdateStock* to update item quantities in one warehouse, which incurs terrific skewness. We collect the CPU usage of each server per second and make a boxplot in Figure 2 (F). Pressure is put on a single server in the cluster with the highest CPU usage. OceanBase v3.1.3 does not provide the data migration feature, so skew partition access workload blocks the throughput of the whole system.

3.3.5 ***Scenario 5: Comprehensive Fault Injection***. We choose three types of exceptions as examples to evaluate the *availability* of OceanBase, i.e., CPU, network and downtime exceptions. We trigger these exceptions on a server at the time point of 60, 180, 300 seconds respectively. The number of transactions per second is displayed in Figure 2 (G). These exceptions all impact system performance, where the network exception increases the message transmission latency and causes dramatic performance degradation, and the downtime exception leads to service unavailable time for Paxos re-election and transaction recovery.

3.3.6 ***Scenario 6: Dynamic Load Volume***. We provide *Workload Executer* with thinking time that follows various probabilistic distributions, to control the workload volume and simulate workload patterns in real application scenarios. *Workload Executer* will stop working during thinking time. Taking *Gaussian Distribution* as an example, the load volume at each time point has obvious differences in Figure 2 (H), and the overall trend conforms well to *Gaussian Distribution*.

4 CONCLUSION

Dike is the first benchmark suite to benchmark transactional DDBMSs, which provides unique designs in critical dimensions of transaction processing. The source code, representative experiments on popular DDBMSs, and detailed analysis are all available on github [1].

ACKNOWLEDGMENTS

This work is supported by NSFC (62072179, U22B2020), Ant Group Research Fund, and Key Laboratory of Performance and Reliability Evaluation of Basic Hardware and Software, Ministry of Industry and Information Technology.

REFERENCES

[1] DBHammer. 2023. How Good is the Distributed Databases in Supporting Transaction Processing. https://github.com/DBHammer/Dike
[2] Guo Jinwei et al. 2017. Low-overhead paxos replication. *Data Science and Engineering* 2 (2017), 169–177.
[3] Harding Rachael et al. 2017. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment* 10, 5 (2017), 553–564.
[4] Qu Luyi et al. 2022. Are current benchmarks adequate to evaluate distributed transactional databases? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* (2022), 100031.
[5] Yang Zhenkun et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3385–3397.