CloudyBench: A Testbed for A Comprehensive Evaluation of Cloud-Native Databases

Chao Zhang[§], Guoliang Li[*], Leyao Liu[†], Tao Lv[‡], Ju Fan[§]
[*]*Tsinghua University,* [§]*Renmin University of China,*[†]*Imperial College London,*[‡]*China Software Testing Center*
cycchao@ruc.edu.cn, liguoliang@tsinghua.edu.cn, leyao.liu24@imperial.ac.uk, lvtao@cstc.org.cn, fanj@ruc.edu.cn

***Abstract*—As more and more on-premise databases are moving towards the cloud service, it is crucial to have a benchmark to holistically evaluate the performance of their core features including elasticity, multi-tenancy, and cost-efficiency. However, existing benchmarks lack specific workload patterns and metrics for evaluating cloud-native databases, and the real workload is often unavailable due to privacy requirements.**

In this paper, we propose a new testbed for cloud-native databases, named `CloudyBench`. Its core contribution is to provide tailored workloads and metrics to evaluate the service quality of cloud-native databases in various dimensions. First, we design cloud-native workload patterns with peaks and valleys for elasticity evaluation. Second, we devise new multi-tenancy patterns by posing varied resource contention to evaluate the resource scheduling among tenants. Third, we propose a unified metric that considers performance, cost, elasticity, multi-tenancy, replication lag time, and fail-over. Fourth, we provide an evaluation testbed for evaluating cloud-native databases. To verify the effectiveness of `CloudyBench`, extensive experiments have been conducted over five commercial representatives from multiple cloud providers. We also obtain a number of insights for the performance implications of cloud-native databases from the architectural perspective.

I. Introduction

Recently, we have witnessed a proliferation of cloud-native databases (CDB) that seek for higher elasticity and lower cost by developing new database techniques in the cloud [8], [45]. CDBs own the *disaggregation of compute and storage* architecture [38], [39], [16], [41], [18], which decouples the storage from the compute nodes, and then connects the compute nodes to the shared storage through a high-speed network. The compute layer consists of a primary read-write (RW) node and multiple secondary read-only (RO) nodes, where each node has a local cache. Further, the disaggregated memory architecture decouples the memory from the local instance to a remote buffer pool. Consequently, CDB claims to provide better elasticity and multi-tenancy service, higher cost-efficiency, and superior transaction processing performance. However, it is not clear under what circumstances there is a significant benefit of using a CDB compared with the AWS Relational Database Service (RDS). To answer such a question, we, in this work, make the first comprehensive investigation of CDB's key features by proposing a new end-to-end benchmark, named `CloudyBench`.

Guoliang Li is the corresponding author. This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62232009, 62436010, 62441230, 62436010). Thank database teams of GaussDB, PolarDB, TDSQL-C, Dameng, OceanBase, and GoldenDB for the help of benchmark design and evaluation.

Database benchmarking [12], [35], [36], [5], [9], [22], [44], [47], [46], [4], [17] is a common practice for evaluating the database performance. Since the real workload is unavailable and there is a dearth of benchmarks for cloud-native databases, many practitioners tried to utilize established database benchmarks such as TPC-C [35], YCSB [5], and SysBench [15] to evaluate the cloud-native databases. Unfortunately, existing benchmarks are unsuitable for benchmarking CDBs as follows:

Motivation 1: Elasticity Evaluation. The real-world workloads of many applications usually fluctuate unexpectedly (e.g., peaks and valleys), and the cloud vendors have rolled out the elastic data service that can dynamically schedule the resources to the varied workloads on demand. Existing benchmarks lack the elasticity evaluation and they can not readily vary the workload patterns at runtime. To address such a problem, we develop an elasticity evaluator that can generate customized patterns with peaks and valleys to evaluate the elasticity of the cloud data services.

Motivation 2: Multi-Tenancy Evaluation. When deploying on-premise databases, the hardware resources are bound to the fixed customers, leading to a waste of idle resources. With multi-tenancy [24], the cloud data service could dynamically schedule resources to different tenants based on their needs and priorities so as to achieve better resource utilization. Existing benchmarks do not support multi-tenancy evaluation, failing to simulate various degree of resource contention among multiple tenants. Consequently, we develop a multi-tenancy evaluator to quantify the CDB's performance with interweaving workloads from multiple tenants.

Motivation 3: Holistic Metric. Existing benchmarks fall short of metrics in two aspects. First, they pay most attention to performance such as latency and throughput but lack tailored metrics to quantify the resource cost in the cloud including CPU, memory, I/O, and network. Second, they lack a unified metric that can make a horizontal comparison among the CDBs in a holistic way. To this end, we design a unified metric, taking into account performance, elasticity, multi-tenancy, resource cost, data replication and fail-over. These factors are chosen because they reflect the most important aspects of service quality of CDBs.

Challenges. There are three main challenges for benchmarking CDBs. First, elasticity and multi-tenancy are two critical features of CDBs, but it is non-trivial to design

TABLE I
A COMPARISON BETWEEN CLOUDYBENCH AND EXISTING OLTP BENCHMARKS

Features	SysBench	YCSB	TPC-C	CDSBen[48]	Stitcher[40]	CloudyBench
Domain-Specific Cloud-Native Application	×	×	×	×	×	√
OLTP Evaluation with ACID	√	√	√	×	×	√
Elasticity Evaluation with Peaks and Valleys	×	×	×	√	√	√
Multi-Tenancy Evaluation with Contention Patterns	×	×	×	×	×	√
Fail-Over Evaluation with Built-in Module	×	×	×	×	×	√
Replication Lag Time Evaluation	×	×	×	×	×	√
Cloud-Native Metrics with Performance and Cost	×	×	×	×	×	√

representative workloads that can quantify the performance of elasticity and multi-tenancy of existing CDBs effectively (**C1**). Second, different cloud vendors have different hardware environments and pricing models, and it is challenging to compare the performance and cost-efficiency of different vendors simultaneously (**C2**). Third, different CDBs have their pros and cons, and it is challenging to design a unified metric for a horizontal comparison (**C3**). To address **C1**, we design foundation *deterministic patterns* and assemble them to cover representative patterns with peaks and valleys, allowing users to evaluate the elasticity and multi-tenancy in an effective and economic way. To address **C2**, we calculate the cost from the resource perspective by defining the standard price based on the resource unit cost of CPU, memory, I/O, and network, allowing us to evaluate the CDB's performance and cost-efficiency in a unified framework. To address **C3**, we devise a unified metric to quantify the CDBs' performance from seven dimensions considering performance, cost, elasticity, multi-tenancy, fail-over, and replication lag time.

In this paper, we propose an end-to-end benchmark for cloud-native databases, named CloudyBench. As shown in Table I, it is the only one covering all the seven salient features. First, we design a SaaS scenario of sales microservice that contains typical read/write transactions in a cloud application. The access distribution and pattern of transactions can be controlled to evaluate the cloud-based OLTP performance and replication lag time. Second, we design basic elastic workload patterns to evaluate the elasticity with specialized peaks and valleys. Third, we design basic multi-tenancy patterns to measure the capacity of resource scheduling among multiple tenants. Fourth, we develop a fail-over evaluator that can inject the node failure and report the fail-over performance automatically. To the best of our knowledge, this is the first benchmark that can evaluate the performance of throughput, elasticity, multi-tenancy, cost-efficiency, and fail-over with tailored workloads and metrics for CDBs.

The main contributions are summarised as follows:

1) We made an in-depth investigation on the key features of various state-of-the-art cloud-native databases.
2) We designed an end-to-end benchmark and a testbed to evaluate the key features of cloud-native databases, including elasticity, multi-tenancy, and cost-efficiency. The code is open-sourced at Github.
3) We proposed a unified metric to quantify the service quality of cloud-native databases by considering the performance, cost, elasticity, multi-tenancy, lag time, and fail-over.
4) We obtained a number of insights by leveraging CloudyBench to evaluate five cloud data services.

II. CLOUDYBENCH BENCHMARK

To evaluate the core features of CDB, we develop a new testbed based on a scenario of cloud-based microservice. First, we design typical operations to evaluate the disaggregated transaction processing with controllable access distribution. It also enables to evaluate the replication time independently. Second, we design basic deterministic elasticity patterns with peaks and valleys. Third, we also devise basic representative multi-tenancy patterns to evaluate the resource scheduling with the multi-tenancy deployment. Fourth, we develop a fail-over evaluator to test the recovery speed concerning node failure. Finally, we define seven metrics to quantify the CDBs' service quality, we also design a unified metric to enable a horizontal comparison. In order to simulate the real workload, the designed workloads have referenced the workload characteristics in real cloud OLTP applications (such as ERP microservice [19] and E-Commerce application) [43].

Figure 1 depicts an overview of CloudyBench, including the data generation, workload manager, elasticity evaluator, multi-tenancy evaluator, OLTP evaluator, fail-over evaluator, performance collector, and metrics. Given a configuration file, data is generated based on the scale factor and the workload manager spawns the workers based on the concurrency and access distribution. The workload consists of basic CRUD transactions (T1-T4), which are used to evaluate the throughput and replication lag time. They also serve as the base for generating the elasticity and multi-tenancy patterns with read and write operations. Lag time evaluation measures the replication latency. Elasticity generator will produce an elastic workload for each tenant. Multi-tenancy evaluator concurrently runs the workloads from multiple tenants. Fail-over evaluator tests the recovery speed by injecting node failures. Performance collector accumulates the performance metrics and corresponding cost.

CloudyBench is extensible for adding new patterns and OLTP workloads. To extend elasticity and multi-tenancy patterns, users can simply modify the length of *elastic_testTime* (e.g., 4) and add corresponding concurrency in the *props* file. (e.g., *fourth_con*), then modify the *CloudyBench* class to launch the new patterns. For fail-over pattern, it supports to add more replicas in the *props* file, then it can pose fail-over test on any node. Since the framework has decoupled the SQL statements, new workload can also be readily incorporated

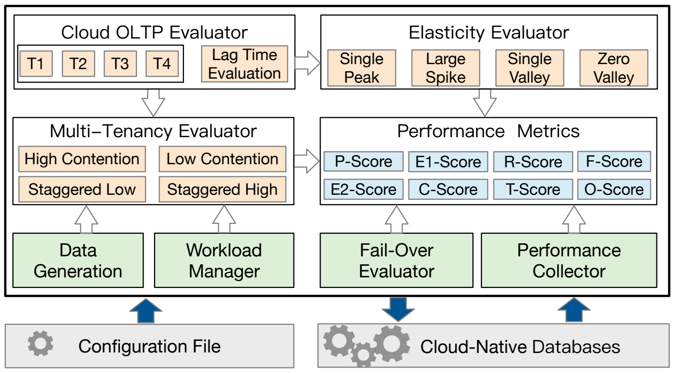

Fig. 1. **CloudyBench Overview**

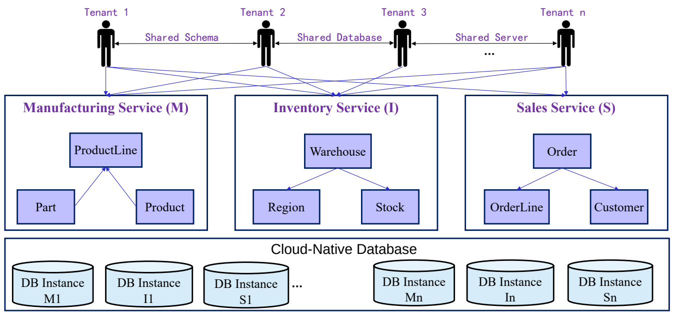

Fig. 2. **MicroService in CloudyBench**

by adding the statements in *stmt_db.toml* and modifying the classes of *SqlReader* and *Sqlstmts*.

A. *MicroService Schema and Data Generation*

Software as a service (SaaS) applications are prominent in the cloud. Common examples are Salesforce [33], Microsoft 365 [31], and Shopify [32]. SaaS applications normally adopt the microservice architecture. However, microbenchmarks such as SysBench [15] and YCSB [5] contain simple read/write operations on single table, and lack specific transaction logic for evaluating CDB. Consequently, we design a cloud-native application, simulating modern SaaS ERP applications like Salesforce [33] and ODOO [11]. As shown in Figure 2, it contains three microservices, manufacturing service, inventory service, and sales service. Tenants can share schema/database/server among the services. It has two advantages. First, the schema mimics the real cloud-native sales application. Second, it contains realistic transaction logic that involves multiple tables, posing a larger challenge than microbenchmark on OLTP. In this work, we focus on the sales service, we will add the microservicse of ”Manufacturing” and ”Inventory” in the future. Specifically, the sales schema contains three tables (CUSTOMER, ORDER and ORDERLINE), and the workload simulates the typical operations in the cloud (e.g., make online orders and payments, check the status). The scaling model makes the ORDERLINE table being an order of magnitude larger than the CUSTOMER table and ORDER table, with a same size of 300,000. Upon sales service, we assemble the basic elasticity and multi-tenancy patterns to evaluate the core features.

B. *Cloud OLTP Patterns*

Existing benchmarks [35], [15] generate the transactions based on the uniform or independent assumption, thus they are insufficient to simulate a cloud OLTP workload. First, realistic access distribution is skewed [48], [9], but macrobenchmarks like TPC-C generate the transactions with uniformly sampled parameters. Second, the workload should simulate the common operations, but microbenchmarks like SysBench can only generate the read/write operations on single table without any correlation between operations. Third, data replication is a crucial aspect in a disaggregated architecture, but existing benchmarks cannot evaluate the replication time. To address these issues, we design a new cloud-native workload.

1) Throughput Evaluation.: The workload covers the most common transactions in a sales microservice of ODOO [11]. As shown in Table II, T1 (New Orderline) is a write-only transaction that inserts a new orderline; T2 (Order Payment) is a read-write transaction that finds a target order, and then it updates the customer’s credit and order’s status. T3 (Order Status) is a read-only transaction that checks the status of a given order; T4 (Orderline Deletion) is to delete a given orderline. The workload manager will launch a worker for each transaction based on the concurrency number and transaction ratio. We support two types of distributions, `uniform` and `latest`. For the former one, the substitution parameters are chosen uniformly. For the latter one, we generate the skewed access distribution by controlling the access range of O_ID. For instance, concerning the `latest-10` distribution, T2 will update 10 specific items, and T3 will read these items randomly. As a result, the more skewed the distribution is, the more likely the fresh data is read.

2) Lag Time Evaluation.: To evaluate the log-replaying efficiency, we design three patterns to evaluate the replication lag time. Namely, (a) insert lag time; (b) update lag time; (c) delete lag time. The basic idea is to measure the lag time synchronizing the data changes from the RW node to the RO node with varied concurrency. Specifically, we run T1, T2, T4 with varied ratio to measure the latency that a replica has synchronized the data changes. For each pattern, once the primary RW node commits the transaction, the client will try to read the data change from the replica until the data is consistent between the RW node and RO nodes.

C. *Elasticity Patterns*

Realistic cloud applications often exhibit intermittency and therefore stressing the elasticity [37], [39]. However, existing benchmarks can not vary the workload patterns at runtime. To address such a problem, we design basic elastic patterns and assemble them to simulate the realistic arrival patterns in a sales microservice.

As shown in Figure 3, pattern (a) launches a single peak to test if the CDB can handle the spike (e.g., an ETL maintenance job); pattern (b) has two small spikes and a large spike which starts from a small concurrency, then gradually increases to a spike, and finally decreases to a small concurrency (e.g., ordering a hot-selling product); pattern (c) has a reverse pattern

TABLE II
CLOUDYBENCH'S OLTP WORKLOAD

Task	Transaction Name	SQL Statement Reference	Pattern
T1	New Orderline	INSERT INTO orderline VALUES (DEFAULT, ?,?,?,?,?)	Write-Only
T2	Order Payment	(1) SELECT O_ID, O_C_ID, O_TOTALAMOUNT, O_UPDATEDDATE FROM orders WHERE O_ID=? (2) UPDATE orders SET O_UPDATEDDATE=?, O_STATUS='PAID' WHERE O_ID=? (3) UPDATE customer SET C_CREDIT=C_CREDIT+?, C_UPDATEDDATE=? WHERE C_ID=?	Read-Write
T3	Order Status	SELECT O_ID, O_DATE, O_STATUS FROM orders WHERE O_ID = ?	Read-Only
T4	Orderline Deletion	DELETE FROM orderline WHERE OL_ID=?	Deletion

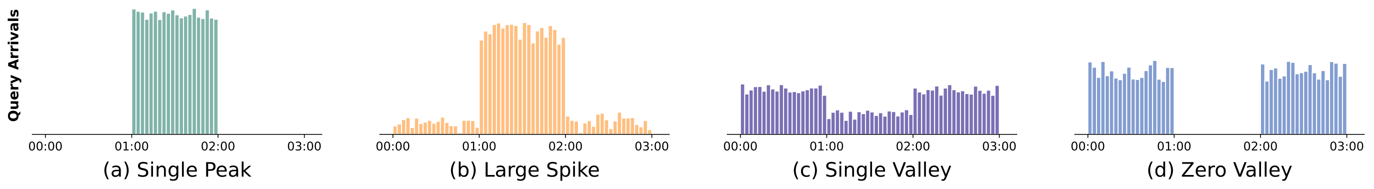

Fig. 3. **Elasticity Patterns in CloudyBench**

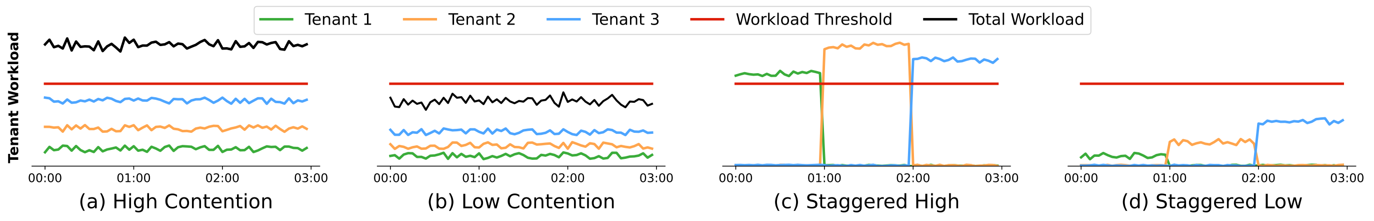

Fig. 4. **Multi-Tenancy Patterns in CloudyBench**

to (b) that starts from a large concurrency, then decreases to small concurrency, and finally increases to a large concurrency (e.g., declined sales due to price variation); pattern (d) aims to evaluate the pause-and-resume mechanism, which starts from a large spike, then decreases to a zero valley, finally increases to a large spike again (e.g., out of stock shortly). The concurrency will be changed in each time slot, and we specify a minute as a time slot. To determine the specific concurrency number in each time slot, we obtain the concurrency number τ where a tested database reaches the resource limit, then we generate the patterns proportionally. For instance, given a configured CDB and the concurrency τ=110, we generate the basic patterns in the following typical proportions: pattern (a): (0%, 100%*τ, 0%)=(0, 110, 0); pattern (b): (10%*τ, 80%*τ, 10%*τ)=(11,88,11); pattern (c): (40%*τ, 20%*τ, 40%*τ)=(44,22,44); pattern (d): (50%*τ, 0%, 50%*τ)=(55,0,55). Note that when evaluating multiple databases, we set τ to the maximum concurrency among all databases so as to evaluate the elasticity for each one. Additionally, the default proportion is set by the Pareto distribution.

D. Multi-Tenancy Patterns

Multi-tenancy is a core feature of CDBs for improving the resource utilization by sharing and scheduling the resources among tenants. However, existing method can not pose various resource contention to evaluate how well CDB can schedule the resources among tenants, we thus design new multi-tenancy patterns.

We design basic multi-tenancy patterns, covering the most common contention cases. Namely, (a) high contention, (b) low contention, (c) staggered high and (d) staggered low. As shown in Figure 4, multiple tenants' workloads arrive with varied demand. The red line depicts the workload threshold and the black line illustrates the actual total workload. Pattern (a) and (b) pose resource contention among tenants. Such patterns evaluate if the CDB can schedule the resources from the low-demand tenants to the high-demand tenants so that the overall resource utilization is improved. In pattern (c) and (d), tenants' workloads arrive and stop at different time slots. Such patterns evaluate if the CDB can schedule the resources to the tenants on demand in the contention-free case so that the allocated resource is reduced. The tenants' total workload is higher than the workload threshold in pattern (a) and pattern (c), but it is lower than the threshold in pattern (b) and (d). To generate the specific patterns, we create the tenants' concurrency with the defined ratio, then we adjust the concurrency and execution mode based on the multi-tenancy patterns. For instance, we define the concurrency of three tenants with three time slots as follows: tenant 1: (10%*τ, 10%*τ, 10%*τ), tenant 2: (30%*τ, 30%*τ, 30%*τ), tenant 3: (60%*τ, 60%*τ, 60%*τ). By adding/subtracting the concurrency of the tenants with a concurrency of δ and running three tenants' workloads in parallel, we manage to generate pattern (a) and (b) respectively. For pattern (d), we define the concurrency of tenants as follows: tenant 1: (10%*τ, 0, 0), tenant 2: (0, 20%*τ, 0), tenant 3: (0, 0, 30%*τ), then we run the tenants' workload in sequence. By adding 100%*τ to the tenants, we are able to generate pattern (c). Concerning multiple databases, we set τ to maximum concurrency for pattern (a) and (c) and set τ to minimum concurrency for pattern (b) and (d). This allows us to evaluate the impact of resource contention and sharing for multiple databases. Note that by configuring the parameter file, `CloudyBench` supports arbitrary numbers of tenants and time slots, and the generation method remains the same. Additionally, `CloudyBench` has an extensible framework that can easily incorporate new multi-tenancy patterns.

E. Fail-Over Patterns

Existing benchmarks do not support automatic fail-over evaluation and DBAs have to manually craft the failure and analyze the performance [3], [6]. To enable the fail-over evaluation, we develop a module in the testbed that can inject the node failure and report the fail-over performance automatically.

We design basic fail-over patterns to evaluate the recovery performance of various CDBs, namely, (a) RO failure; (b) RW failure. The basic idea is to inject the node failure during workload processing, then evaluate how fast the CDB can recover the service and throughput, respectively. By investigating the existing APIs of CDBs, we develop a *restart model* [20] to simulate the node failure and evaluate the fail-over performance. This is because the *kill* or *stop* API will lead to the unavailable service, and we have to start the service manually. To evaluate the recovery time, we invoke the *restart* API and record the TPS before the failure, then calculate the duration in two phases. In phase one, we measure how long the TPS is greater than zero after the node failure. In phase two, we continually check if the current TPS is recovered to the original TPS in a given interval.

F. Resource Unit Cost

As CDBs have disparate hardware configurations and pricing models, it is challenging to quantitatively measure the cost-efficiency. For instance, the ratio between CPU and Memory and the pricing of CDBs are totally different; Aurora will charge the IOPS but PolarDB does not; their storage services employ different number of replicas. We define the *resource unit cost* (RUC) to address such a challenge. The basic idea is to define the standard unit price to measure the resource cost, then we can normalize the cost across different providers from the perspective of basic resource unit.

According to the existing settings, we set the resource unit separately for resource package calculation. Namely, 1 vCore for CPU, 1 GB for RAM, 1 GB for Storage, 100 for IOPS, 1 Gbps for TCP/IP or RDMA network. Such a method allows us to quantify any combinations of resource package. For instance, concerning the case that Aurora defines the ACU with 1 vCPU, 2GB while PolarDB defines the instance with 4 vCPU, 32 GB, we can calculate the resource cost based on the unit price. Since the cloud vendors may define the cost in hour or in month, we unify the unit price in hour. To calculate the hourly cost more accurately, we propose a new method to have the standardized costs closer to the real costs. The calculation performs in three steps. First, we finalize the relative ratio between the resource units in the package by referencing the hardware price. Second, we normalize the unit price based on their real cost. Third, we obtain each resource unit by averaging the price of the systems. For instance, by referencing the CPU price of referenced Intel Xeon Platinum 8562Y+ Processor and compatible RAM price of Micron DDR5, we define the ratio of CPU and Memory is 0.95 and 0.05. Since Aurora defines the ACU cost is \$0.2 per hour, we have the CPU price is \$0.1809/vCore/hour and the RAM price is \$0.0095/GB/hour. Finally, we calculate the unit price for each cloud-native database and average the unit costs. That is, we obtain the CPU cost and Memory cost as \$0.1847/vCore/hour and \$0.0095/GB/hour by averaging the unit costs of Aurora, PolarDB, HyperScale, and Neon. Since the RDMA network is an emerging resource only supported in PolarDB, we leverage the same way to calculate the network cost for a fair comparison.

TABLE III
RESOURCE UNIT COST PER HOUR

Resource Unit	Cost	Reference
`CPU (vCore)`	\$0.1847/h	Aurora/PolarDB/HyperScale/Neon
`Memory (GB)`	\$0.0095/h	Aurora/PolarDB/HyperScale/Neon
`Storage (GB)`	\$0.000853/h	Aurora/PolarDB/HyperScale/Neon
`IOPS (100)`	\$0.00015/h	AWS RDS IOPS Pricing
`TCP/IP Network (Gbps)`	\$0.07696/h	Huawei S1730S-S24T4X-QA2 10G
`RDMA Network (Gbps)`	\$0.23088/h	MELLANOX MSB7890-ES2F 100G

Consequently, we are able to make a horizontal comparison with the normalized resource cost. Note that for the cases that CDBs have the different hardware, we can calibrate the price with the actual cost. We have also discussed the difference between resource cost and actual cost in Section III-G.

G. Performance Metrics

To measure CDB's performance holistically, we propose a framework of "PERFECT". The basic idea is to reflect the most important aspects with the consideration of performance and cost. In specific, 'P' refers to productivity that measures the ratio of throughput and RUC; the first 'E' (i.e., E1) refers to scaling up/down elasticity and the second 'E' (i.e., E2) refers to scaling out/in elasticity; 'R' refers to throughput recovery efficiency; 'F' refers to fail-over speed; 'C' refers to replication lag time for consistency; 'T' refers to tenants' performance. Finally, we combine seven metrics into a unified metric that reflects the overall performance, called O-Score. In the following, we introduce each cloud metric in detail:

P-Score. To consider the performance and cost together, we define the Productivity (P-Score) as the average transaction performance per RUC as follows:

$$\texttt{P-Score} = \overline{TPS}/(Cost_{cpu}+Cost_{mem}+Cost_{s}+Cost_{io}+Cost_{net}) \tag{1}$$

where $\overline{TPS}$ is the average TPS and we consider the average resource cost of CPU, memory, storage, IOPS, and network per minute.

E1-Score. To quantify the scaling up/down elasticity, we define E1-Score as follows:

$$\texttt{E1-Score} = \overline{TPS}/(\widetilde{Cost_{cpu}} + \widetilde{Cost_{mem}} + \widetilde{Cost_{io}}) \quad (2)$$

where $\overline{TPS}$ is the average TPS and we consider the resource cost of CPU, memory, and IOPS that are mostly relevant to the elasticity.

F-Score. Concerning node failure, we calculate the time range starting from failure injection to the point where databases resume the throughput. The definition is as follows:

$$\texttt{F-Score} = \frac{1}{k}\sum_{i=1}^{k}(t_s^i - t_f^i) \quad (3)$$

where t_f^i and t_s^i is the timing of injecting node failure and the timing of service recovery in the i-th recovering phase.

R-Score. We evaluate the CDB's recovery speed for recovering the TPS after the service recovery with R-Score. Since CDBs have different TPS for a given concurrency, we set the same target TPS for recovery. The definition is as follows:

$$\texttt{R-Score} = \frac{1}{k}\sum_{i=1}^{k}(t_r^i - t_s^i) \quad (4)$$

where t_s^i and t_r^i is the epoch timestamp of service recovery and the epoch timestamp of recovering the TPS before the failure in the i-th recovering phase.

E2-Score. To evaluate the scalability, we add RO nodes and quantify the CDB's improved performance per node. The definition of E2-Score is as follows:

$$\texttt{E2-Score} = \frac{1}{\lambda}\sum_{i=1}^{\lambda}(TPS_i - TPS_{i-1})/\delta \quad (5)$$

where λ is the number of RO node and δ is the scaling factor; TPS_i is the throughput with i nodes.

C-Score. To evaluate the replication lag time with DML operations, we define C-Score as follows:

$$\texttt{C-Score} = (\overline{T_{insert}} + \overline{T_{update}} + \overline{T_{delete}})/\lambda \quad (6)$$

where $|\lambda|$ is the number of replicas; $\overline{T_i}, \overline{T_u}$, and $\overline{T_d}$ is the average lag time for insertion, update, and deletion, respectively. Note that the smaller the C-Score is, the faster the data replication is.

T-Score. We define T-Score as follows:

$$\texttt{T-Score} = \sqrt[m]{\prod_{i=1}^{m} TPS_i} / \sum_{i=1}^{m} Cost_i \quad (7)$$

where the numerator calculates the geometric mean of the overall TPS and TPS_i is the average TPS of i-th tenant; $Cost_i$ is the consumed resource unit cost of i-th tenant;

O-Score. Having a unified metric is beneficial for comparing the performance of cloud databases holistically. Solely relying on one aspect cannot reflect the overall performance. Given that the serven components (cost-aware performance (P-Score), multi-tenancy (T-Score), scale-up elasticity (E1-Score), scale-out elasticity (E2-Score), fail-over time (F-Score), recovery time (R-Score) and replication latency (C-Score) are widely recognized as the most important factors for quantifying the service quality of cloud-native databases, we design a unified metric to quantify the overall performance. By multiplying all the seven scores and adding the logarithm, we propose O-Score defined as follows:

$$\texttt{O-Score} = SF * \lg\left(\frac{P * T * E1 * E2}{R * F * C}\right) \quad (8)$$

where SF is the scale factor; the numerator computes the multiplication of P-Score, T-Score and two elasticity scores; the denominator calculates the multiplication of R-Score, F-Score, and C-Score. The logarithm is for an accurate horizontal comparison. Note that O-Score has an equal weight to each aspect, and cloud vendors can adjust the weight to emphasize the individual part, e.g., add more weight to elasticity.

III. Experiments

A. Experimental Settings

Systems Under Test (SUTs). We evaluate `CloudyBench` over four state-of-the-art cloud-native databases. We use anonymization names of all cloud-native databases because they are commercial databases and have the "*Dewitt Clause*" [42] that forbids the publication of database benchmarks when the database vendor has not sanctioned. AWS RDS [1] is chosen as a representative of RDS. In the following, we briefly introduce their core features.

(1) `CDB1` separates the compute and storage, where the compute layer processes the transactions with the local cache, and the storage layer maintains the data's durability and availability. To reduce the I/O overhead, it offloads the redo processing to the storage tier. Concerning elasticity, it supports to scale the unit of CPU and memory. For scaling up, it will increase the resource immediately when the usage hits a built-in threshold. For scaling down, it will gradually decrease the resource to avoid performance fluctuation. On multi-tenancy, it supports to deploy the instances/clusters of different tenants into different regions, thus the resources of tenants are fully isolated in different nodes. Compared with the traditional ARIES recovery mechanism [23], it pushes down the redo process to the storage layer, and the compute layer does not need to write back the dirty pages.

(2) `CDB2` separates the storage into two parts: the log service for log management and page service for page management. The log service employs fast storage device and the page service leverages general storage device. On elasticity, it can automatically scale the CPU and memory resources independently based on the workload demand and load prediction [28]. It enables multiple tenants to share the compute and log service by developing an elastic pool for multi-tenancy where the tenants' instances within the pool can share vCores, memory, SSD cache, as well as the log service.

(3) `CDB3` develops a disaggregated compute-log-storage architecture based on PostgreSQL codebase. Its compute nodes are scheduled by Kubernetes [21] ; WAL is handled by the `SafeKeeper` procedures; the page servers replay the logs to serve the materialized pages; the hot data is cached in the compute nodes and the cold data is persisted to the cloud

object storage. On elasticity, CDB3 defines that a capacity unit (CU) is 1 vCore and 2 GB, where the minimum setting could be 0.25*CU. For both scaling up and scaling down, it will immediately adapt the CU usage to the workload pattern. It implements a *git-style* multi-tenancy model, where each project has a primary branch and each child branch is a copy-on-write clone of the parent branch. In this case, each branch is regarded as a tenant with a pre-allocated and isolated resource configuration. It supports the pause-and-resume mechanism, meaning that it can scale to zero and resume the service once a workload comes in.

(4) CDB4 develops a memory disaggregation architecture which relies a distributed storage service, and employs a shared remote buffer pool with a high-speed RDMA network. To ensure cache coherency, it utilizes cache invalidation to synchronize the updates between the local cache and the remote cache. It adopts an ARIES-style recovery algorithm with a remote buffer pool [49]. When a node failure occurs, the cluster manager initiates an auto switch-over process by promoting a RO node to a RW node. Then the new RW node distributes the redo logs with the checkpoint version from the storage service to the page server for log replaying.

Experiment Environment. For each SUT in different cloud vendors, we deploy the database service in the same region. The setting is summarized in Table IV, which presents the databases, engine, CPU, Memory, Storage, Network, Serverless, and Buffer Size.

Benchmark Configuration. To avoid the impact of network latency, we deploy the client in the virtual machine in the same VPC (Virtual Private Cloud) of the tested CDB. We produce testing datasets with three scale factors, SF1, SF10, and SF100, with raw data of sizes 194MB, 1.99GB, and 20.8GB respectively. In order to evaluate different workload patterns, e.g., read-only, read-write, and write-only, we vary the transaction ratios. Namely, $(t1 : t2 : t3) \in \{(0 : 0 : 100), (15 : 5 : 80), (100 : 0 : 0)\}$.

B. Transaction Processing Evaluation

Figure 5 illustrates the overall throughput of all SUTs with varied scale factors, transaction patterns and concurrency numbers. The three groups of bars are the TPS of read-only (RO), read-write (RW), write-only (WO) patterns, respectively. We deploy one RW node and one RO node for each SUT.

We have four major observations. First, it clearly indicates that CDB4 has the highest performance, which has an average throughput of 24502 for all workload patterns and scale factors. For the RO and WO patterns, CDB4 is the best because its 10G local buffer greatly promotes the performance (Note that we will also evaluate the impact of buffer size in Section III-I). As for the RW pattern, CDB4 outperforms others because it has a 24GB remote buffer with a high-speed RDMA network. Overall, the throughput of CDB4 is 3x higher than CDB2. Second, CDB3 has higher throughput than CDB1 and CDB2 because of its Local File Cache [25] and parallel log replaying [26]. Particularly, it has comparable read-write performance to AWS RDS for large dataset (i.e., SF100) and high concurrency (100-200). Third, the throughput of CDB2 is bounded when the concurrency increases: its TPS is no more than 11863, 8140, 9291 on RO, RW, and WO patterns, respectively. We believe the buffer has become its performance bottleneck due to the small size. Fourth, AWS RDS has the highest throughput on RW patterns regarding small dataset (i.e., SF1) and low concurrency (< 150) because the majority of data is cached in the buffer, and reading/writing the local storage is faster than the disaggregated storage that requires to access the network. While the performance decreases as the data grows (i.e., SF10 and SF100) and concurrency increases (> 150). The reason is that the dirty page flushing and checkpointing incur larger overhead. Such a finding verifies that CDB can benefit from disaggregated storage architecture via asynchronous log replaying.

Table V depicts the P-Score on different patterns of all SUTs, considering the average TPS and resource cost of CPU, memory, storage, IOPS, and network simultaneously.

By considering both throughput and resource cost, we observe that AWS RDS has the highest P-Score across all workloads as it has a relatively high throughput and incurs a relatively low cost. For instance, it has a high TPS of 12382 on RW patterns and its cost is the lowest, i.e., \$0.0437. CDB4 ranks the second because its remote buffer pool delivers the highest TPS of 36995. However, despite having 1.5x higher throughput than AWS RDS, CDB4 has the lower average P-Score due to its highest cost, especially for the RDMA network that is 3x more expensive than TCP/IP network. CDB3's P-Score on RW pattern is higher than CDB4, indicating that it also strikes a good balance between performance and resource cost. For instance, its cost is only 57.7% of CDB4's but its TPS is close. The P-Score of CDB2 is the lowest due to its low TPS. CDB1 has a higher cost than CDB2 due to two folds. First, its instance has a higher ratio of CPU and Memory, e.g. (1:8). Second, it has a higher storage cost as it adopts the six-way replication [38] while others employ the three-way replication [26]. We also observe that IOPS has a large impact on the cost. For instance, CDB2 has 327x higher IOPS cost than AWS RDS.

C. Elasticity Evaluation

We combine elastic workload patterns to evaluate elasticity of all the SUTs. we use SF1 and vary the transaction ratio to produce three workload modes. The concurrency number of each time slot in four patterns is: single peak: $(0, 110, 0)$; large spike: $(11, 88, 11)$; single valley: $(44, 22, 44)$; zero valley: $(55, 0, 55)$. We choose to calculate the cost in a ten-minute range starting from the beginning of each workload pattern.

Figure 6 illustrates the results of elasticity evaluation, including average throughput, total cost (including execution cost and scaling cost) and E1-Score. Notably, we found that enabling serverless will largely impact the performance. For instance, CDB3 and CDB1 has degraded the performance with 32% and 82% compared with the fixed configuration. Moreover, it is visible that the write ratio will also impact the throughput, i.e., from Read-Only, to Read-Write and Write-

TABLE IV
THE EXPERIMENTAL SETTING OF CLOUD-NATIVE DATABASES

Databases	Engine	CPU & Memory & Storage	Network	Serverless	Buffer Size
AWS RDS	PostgreSQL 15	4 vCores, 16GB RAM, 150GB NVMe SSD	10 Gbps TCP/IP	×	128MB
CDB1	PostgreSQL 15	1 vCore, 2GB RAM – 4 vCores, 8GB RAM	10 Gbps TCP/IP	✓	128MB
CDB2	SQL Server 12	0.5 vCores, 2GB RAM – 4 vCores, 12GB RAM	10 Gbps TCP/IP	✓	44MB
CDB3	PostgreSQL 15	1 vCore, 2GB RAM – 4 vCores, 16GB RAM	10 Gbps TCP/IP	✓	128MB
CDB4	MySQL 8	4 vCores, 16GB local RAM and 24GB remote RAM	10 Gbps RDMA	×	10GB

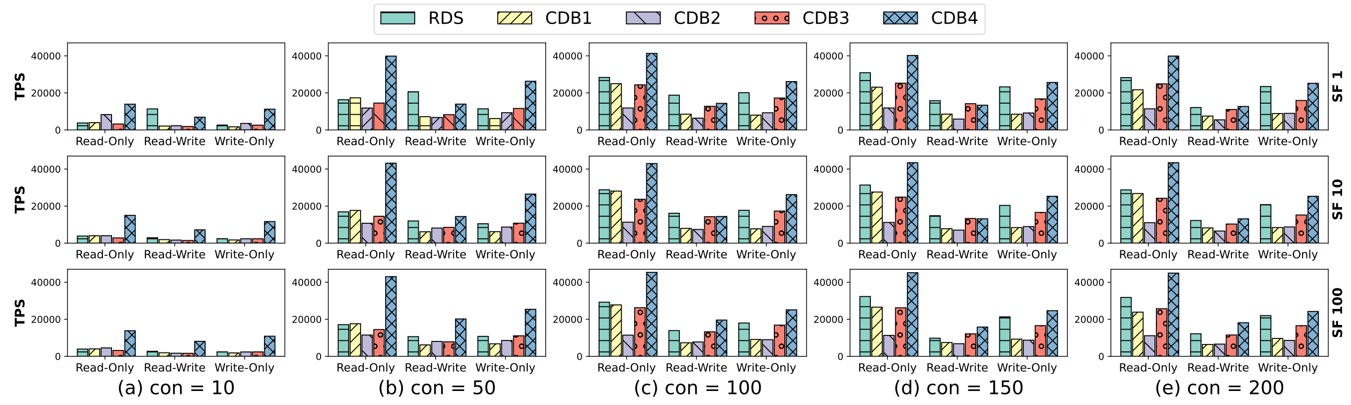

Fig. 5. Transaction Processing Performance of Different Cloud Databases; *con* denotes the concurrency number and y-axis denotes the TPS.

TABLE V
P-SCORE OF DIFFERENT CLOUD DATABASES WITH DETAILED RESOURCE COST

System	CPU/vCore		Memory/GB		Storage/GB		IOPS		Network/Gbps		Resource	P-Score			
	Value	Cost	Value	Cost	Value	Cost	Value	Cost	Value	Cost	Cost	RO	RW	WO	AVG
AWS RDS	4	0.0123	16	0.0025	42	0.0006	1000	0.000025	10	0.0128	$0.0437	**505538**	**283350**	**346174**	**378354**
CDB1	4	0.0123	32	0.0051	126	0.0018	1000	0.000025	10	0.0128	$0.0512	383837	123620	174070	227176
CDB2	4	0.0123	20	0.0032	63	0.0009	327680	0.008192	10	0.0128	$0.0538	189939	109292	142282	147238
CDB3	4	0.0123	16	0.0025	63	0.0009	1000	0.000025	10	0.0128	$0.0443	403273	213922	285425	300873
CDB4	4	0.0123	40	0.0063	63	0.0009	84000	0.0021	10	0.0385	**$0.0797**	464181	173773	284335	307429

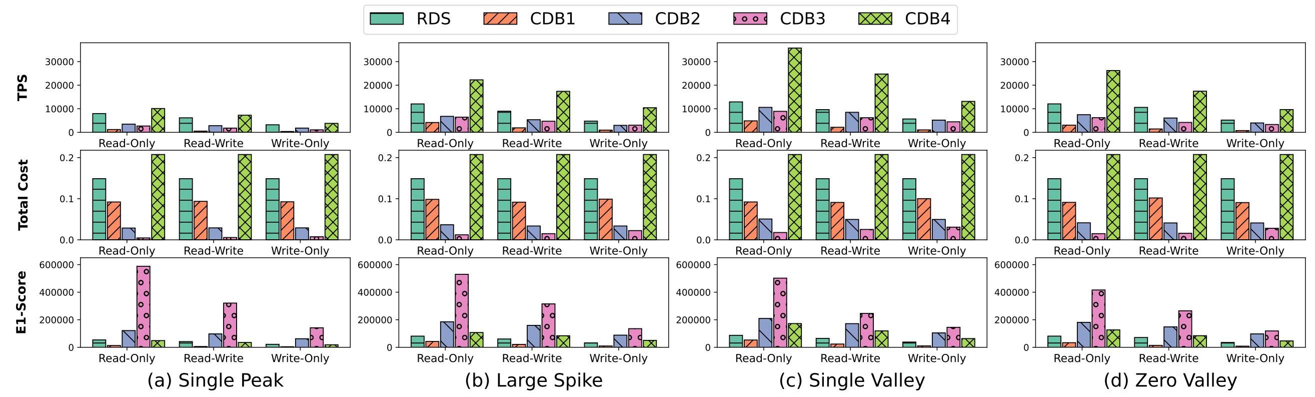

Fig. 6. Elasticity Evaluation of Different Cloud Databases with TPS, Total Cost, and E1-Score

TABLE VI
TIME INTERVAL AND SCALING COST DURING AUTOSCALING OF CLOUD-NATIVE DATABASES

System	Single Peak		Large Spike				Single Valley				Zero Valley			
	0→110	110→0	0→11	11→88	88→11	11→0	0→44	44→22	22→44	44→0	0→55	55→0	0→55	55→0
CDB1	14s	479s	17s		501s		11s		536s		11s		535s	
CDB2	30s	25s	30s	30s	30s	30s	25s	20s	15s	25s	30s	30s	30s	30s
CDB3	60s	60s	60s	60s	80s	60s	60s		180s		60s	60s	60s	80s
CDB1	$0.0018	$0.0789	$0.0035		$0.0756		$0.0019		$0.0827		$0.0019		$0.0827	
CDB2	$0.0071	$0.0017	$0.0027	$0.0082	$0.005	$0.0018	$0.0058	$0.0051	$0.0042	$0.0037	$0.0081	$0.0026	$0.0077	$0.0042
CDB3	$0.0037	$0.0022	$0.0022	$0.0065	$0.0059	$0.0019	$0.004		$0.0205		$0.0053	$0.0028	$0.0071	$0.0043

Only. Especially for CDB4 and AWS RDS, the higher write ratio incurs larger overhead of dirty page flushing, leading to lower throughput. Overall, the performance rank is CDB4 >AWS RDS>CDB2 >CDB3 >CDB1. Since CDB4 and AWS RDS have the fixed configuration, their TPS are 3x and 1.5x higher than CDB2. Nevertheless, the sub-figure of total cost

TABLE VII
MULTI-TENANCY EVALUATION RESULTS OF DIFFERENT CLOUD DATABASES

System	**TPS**				**Total Resources**	**Cost**	**T-Score**				
	(a)	**(b)**	**(c)**	**(d)**	**CPU, Memory, Storage, IOPS, Network**		**(a)**	**(b)**	**(c)**	**(d)**	**AVG**
CDB2	4000	6467	4948	3458	12 vCores, 36GB RAM, 189 GB, 54000 IOPS, 10Gbps TCP/IP	$0.06	70008	107799	82483	**57647**	79484
CDB3	5633	5389	5494	1237	12 vCores, 48GB RAM, 63 GB, 3000 IOPS, 10Gbps TCP/IP	$0.058	92524	92917	**94724**	21344	75377
AWS RDS	13489	6772	5321	1826	12 vCores, 48GB RAM, 126 GB, 3000 IOPS, 30Gbps TCP/IP	$0.085	**158702**	79673	62611	21488	**80619**
CDB1	9791	5607	3217	1622	12 vCores, 96GB RAM, 378 GB, 3000 IOPS, 30Gbps TCP/IP	$0.096	101991	58412	33515	16903	52705
CDB4	**20480**	**19319**	**7084**	**6130**	12 vCores, 120GB RAM, 189 GB, 84000 IOPS, 30Gbps RDMA	**$0.176**	116365	**109770**	40253	34831	75305

demonstrates that their cost is also much higher, which is 12x and 9x higher than CDB3's. Besides, CDB2's total cost is higher than that of CDB3 due to its minimum 0.5 vCore usage and higher scaling resources. We attribute CDB3's low cost to its on-demand scaling and pause/resume strategy. For instance, its superiority becomes evident concerning pattern (a) that contains two idle time slots. As a result, the E1-Score rank is CDB3 >CDB2 >CDB4 >AWS RDS>CDB1.

Table VI presents the detailed scaling time, scaling cost, and consumed resources within each time slot. We compare three CDBs with autoscaling feature, including CDB3, CDB2, and CDB1. We measure the scaling time in each time slot by calculating the duration from workload's changing to the scaling completion. Then we calculate the cost and average consumed resource per second. Our first observation is that CDB1 has a good elasticity on scaling up but its scaling down is much slower due to the gradual scaling strategy. For instance, it takes 14s to scale up in Single Peak, but it spends 479s scaling down to zero. Obviously, gradual scaling down incurs high cost because CDBs will also charge during scaling. The second observation is that CDB2 has better elasticity than CDB1 because it achieves on-demand scaling up/down. Particularly, it is capable of scaling the resources in each period. However, we do not observe any proactive autoscaling [29]. The third observation is that CDB3 has the best elasticity as it combines on-demand scaling up/down and pause/resume approach to minimize the cost. Nevertheless, it could not be sensible to instant workload change. For instance, it fails to scale down for the Single Valley (44, 22, 44) and Zero Valley (55, 0, 55). Finally, we observe that CDB3 consumes less resource than others. On average, CDB3 saves 56% vCores and 11% memory than CDB1.

D. *Multi-Tenancy Evaluation*

In this section, we evaluate CDB's multi-tenancy. We set up CDB2 with an elastic pool including 12 vCores and 36 GB memory shared by 3 tenants. Hence, each tenant has 4 vCores and 12 GB memory on average. As for CDB3, we create three branches that share the storage and each branch has 4 vCores and 16 GB of RAM, resulting in a total compute resources of 12 vCores and 48 GB memory. For CDB1, CDB4 and AWS RDS, we create a separate instance for each tenant. Since their instances are isolated, the cost of the network and IOPS is tripled. The detailed resource and cost are given in Table VII. Following the multi-tenancy patterns introduced in Section II-D, we generate four patterns for 3 tenants as follows: pattern (a): $\{(264, 264, 264), (99, 99, 99), (33, 33, 33)\}$; pattern (b): $\{(40, 40, 40), (30, 30, 30), (10, 10, 10)\}$; pattern (c): $\{(363, 0, 0), (0, 429, 0), (0, 0, 396)\}$; pattern (d): $\{(10, 0, 0), (0, 20, 0), (0, 0, 30)\}$.

Table VII summarizes the evaluation results of all SUTs with the multi-tenancy patterns. We have four observations. First, multi-tenancy with isolated instances can achieve high performance, but such a model has a rather high cost and can not share and schedule the resources effectively. For instance, CDB4 has the highest throughput with the isolated instances, remote buffer pool, and high-speed RDMA network. However, this is achieved at the highest cost of $0.176. Currently its multi-tenancy capacity has not been fully released, thus the resources can not be shared and scheduled. Following CDB4, AWS RDS also excels at transaction processing with an average TPS of 6852, but it is also unable to share the resources among tenants. CDB3 can deploy multiple tenants in a unified cluster, and its cost is low (i.e., $0.058). Nevertheless, the tenants' compute and I/O resources are stringently isolated, thus its average resource utilization is low concerning staggered patterns, leading to a lowest TPS of 1237 at pattern (d). Second, multi-tenancy with shared resources has the best cost efficiency. For instance, CDB2 has the highest T-Score of 57647 at pattern (d) with a shared elastic pool where multiple tenants can share compute and I/O resources. Third, we found there is no silver bullet regarding different multi-tenancy patterns. We make a comparison of CDB2 and CDB1 as they have similar read-write TPS in P-Score evaluation. On pattern (a), CDB1's TPS is 2.45x higher than CDB2 because of its fixed and isolated configuration, which prevents the tenants from being affected by others' heavy workload and high resource demand. On the contrary, the tenants of CDB2 suffer from high resource contention, resulting in the lowest TPS of 4000. On pattern (b), when total workload is lower than the threshold, CDB2 outperforms CDB1 since its elastic pool could allocate the resources to each tenant as needed, achieving better overall performance because high-demand tenant acquires more resources. Regarding pattern (c) and (d), CDB2's TPS is 2.13x higher than Aurora. This is because all the available resources in elastic pool could be scheduled to the only tenant that currently has resource demand, which can greatly increase the throughput. In contrast, Aurora is unable to schedule the resources, leading to a low resource utilization.

E. *Fail-Over Evaluation*

Table VIII presents the fail-over evaluation results, including the F-Score and R-Score that measure the recovery efficiency regarding node failure of RW and RO. We perform a constant

TABLE VIII
F-SCORE AND R-SCORE OF DIFFERENT CLOUD DATABASES

System	F-Score RW	F-Score RO	F-Score AVG	R-Score RW	R-Score RO	R-Score AVG	Total Time(s)
AWS RDS	24	6	15	18	30	24	78
CDB2	6	6	6	36	18	27	66
CDB3	12	6	9	30	6	18	54
CDB1	6	6	6	18	0	9	30
CDB4	**3**	**2**	**2.5**	**3**	**4**	**3.5**	**12**

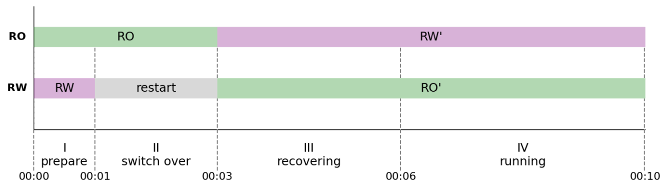

Fig. 7. **Timeline of CDB4's Failover Process; x-axis denotes the different phase and y-axis shows the status of RO and RW.**

read-write workload with a concurrency of 150 and inject the node failure with the restart model.

The results clearly show that AWS RDS has the highest recovering time across all failure patterns. Particularly, it takes an average time of 15s to resume service and 24s to recover the TPS, which are 2.5x and 2.6x longer than the recovering time of CDB1. The results indicate that the log-based replaying recovery mechanisms adopted by CDBs produce less overhead compared to ARIES recovery mechanism. Specifically, CDBs such as CDB1 and CDB3 can utilize asynchronous log replaying and the materialized pages in the page server to quickly recover the transaction data. A side observation is that although recovering from page server could mitigate the overhead of log replaying, the separation of log store and page store adds more network latency to the recovery process. For instance, CDB3's and CDB2's total recovering time is 1.8x and 2.2x higher than CDB1 due to the longer recovery route. CDB4 has the best fail-over ability, requiring 2.5s to resume and another 3.5s to recover the TPS. We attribute its superiority to its remote buffer pool that can quickly recover the data.

Figure 7 depicts a recovery process of CDB4. In the prepare phase, when a failure is detected via heartbeat signals, its cluster manager takes 1s to notify all nodes to refuse subsequent requests, then it collects the latest sequence number (LSN) of page and checkpoint. In the switch over phase, it takes 2s to promote a RO node to the new RW' node. Meanwhile, the original RW node performs a cleanup with the remote buffer pool, then it transforms to a RO' node via restarting. In the recovering phase, the RW' node takes 3s to construct the active transactions and rollback the uncommitted transactions by scanning the undo logs. After 6s, the recovered cluster can proceed to handle the subsequent requests.

F. Lag Time Evaluation between RW and RO

To evaluate the lagtime between primary and replica node, we vary the ratio of Insert, Update and Delete (IUD) in four patterns as follows: (I, U, D) $\in$ {(60%, 30%, 10%), (100%, 0%, 0%), (0%, 100%, 0%), (0%, 0%, 100%)}. In AWS RDS, checkpoint_timeout is set to 30s, and max_wal_size is 128MB.

Through the lag time evaluation, We have four main insights. First, CDB4 achieves the lowest latency of 1.5ms with the memory disaggregation. This is mainly because it utilizes the high-speed RDMA network to ship logs and fetch the global timestamps [43]. Additionally, it employs several optimizations such as local ordered timestamps and on-demand log replaying to further reduce the latency. Second, CDBs with storage disaggregation have disparate lag time. For instance, CDB3 has a relatively low lag time of 14ms because it also replays the relevant logs in parallel [26] which largely speeds up the replication process. By replaying the logs in sequence, CDB1 and CDB2's lag time is higher with an order (177ms) and two orders of magnitude (1082ms), respectively. CDB2's separation of log and storage leads to the highest lag time due to the longer replication path. Third, AWS RDS has a relatively small lag time because of its coupled compute and storage. Fourth, we found the combination of IUD has an impact on the lag time because different CDBs have disparate handling logic. For instance, all the SUTs have less lag time with higher delete ratio, the main reason is that most CDBs perform the deletion via the logical deletion. We also found that CDB1 is more sensitive to a higher insert ratio while CDB3 is affected by the higher update ratio.

G. Overall Performance

In this part, we quantify the CDB's overall performance with the proposed unified metric that contains seven scores, namely, "PERFECT" framework. We also compare an alternative method that computes the score based on the actual cost charged by the cloud vendors, i.e., P-Score*, E1-Score*, T-Score*, and O-Score*.

As presented in Table IX, different cloud databases have their pros and cons. Firstly, AWS RDS has the highest P-Score that is 3.6x higher than CDB2's. Furthermore, it has the highest T-Score and E2-Score after adding a RO node, and its TPS increases from 17003 to 36198 with the local SSD storage. However, its recovery speed is the lowest due to the dirty page flushing. Secondly, CDB3 has the highest E1-Score that is an order of magnitude higher than CDB1, and its other score is relatively balanced. Thirdly, CDB4 excels at the recovery speed with R-Score and F-Score of 3.5s and 2.5s, and it has the minimum C-Score of 1.5ms with the RDMA-enabled memory disaggregation. CDB2 performs the best among CDBs on the multi-tenancy patterns via its shared elastic pool, resulting in the highest T-score of 79484. By combining all the scores into a unified metric, we can see that CDB4 is the winner that has the highest O-Score of 17.7.

Interestingly, the actual cost leads to different ranks of the designed metrics due to the impact of pricing model. For instance, AWS RDS has the lowest P-Score* because its pricing model charges for at least 10 minutes. The T-Score of CDB2 has changed the second worst as the elastic pool is charged at least one hour. Since CDB3 is a startup, its pricing model tends to be cheaper than others, e.g., it has 3x lower

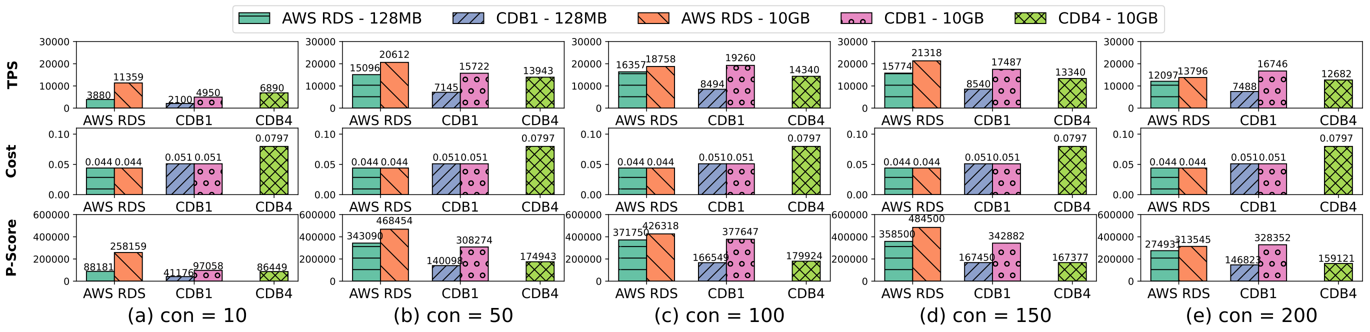

Fig. 8. **Performance Evaluation on AWS RDS, CDB1, and CDB4 by Varying the Buffer Size from 128MB to 10GB.**

TABLE IX
OVERALL PERFORMANCE OF CLOUD-NATIVE DATABASES. (X-SCORE)* DENOTES THE SCORE IS CALCULATED WITH THE ACTUAL COST

System	P-Score	P-Score*	E1-Score	E1-Score*	R-Score	F-Score	E2-Score	C-Score	T-Score	T-Score*	O-Score	O-Score*
AWS RDS	**359735**	359	59430	1052	24	15	**20**	14	**80619**	104	15.82	8.18
CDB1	131906	14369	16024	16311	9	6	3	178	52705	5326	13.48	11.53
CDB2	99212	2737	139933	70241	27	6	7	1082	79484	1923	13.64	10.17
CDB3	217002	**480660**	**286643**	**401643**	18	9	4	14	75377	**45540**	15.92	**16.19**
CDB4	153566	19124	80565	52241	**3.5**	**2.5**	10	**1.5**	75305	13806	**17.7**	15.87

price on CPU ($0.16 per vCore compared with $0.42 per vCore by CDB2). Hence, its P-Score*, E1-Score*, T-Score* are much higher, resulting in a highest O-Score*. We also observe that all CDBs outperform AWS RDS with the actual cost and defined metrics. Nevertheless, this is mainly affected by the pricing strategies. Hence, it is more fair to compare the service quality under a unified resource unit cost.

Overall, our metrics have three advantages. First, our metrics give a quantitative way to measure the cost from the resource perspective, but the actual cost is largely affected by the pricing model. Particularly, the vendors may have different pricing on the instance configuration, and it is hard to make a fair comparison. Second, our metrics are more accurate with a standard normalization while the actual cost ends up with different ranks. Third, our metrics can be computed individually and combined into a unified metric.

H. Varying the Buffer Size

As discussed in Section III-B, CDB4 has high performance because of its 10 GB local buffer size that is unmodified by users. Hence, it is crucial to investigate if the buffer size has an impact on performance and cost. To this end, we contrast it against AWS RDS and CDB1 by increasing their buffer size from 128 MB to 10 GB. CDB3 and CDB2 are excluded due to the unmodified buffer setting for the users. We run the RW pattern on SF1.

Figure 8 depicts the evaluation results, including TPS, Cost, and P-Score. The results indicate that buffer size has a significant impact on the performance with the same cost, leading to the different ranks of CDBs. For instance, the average TPS of CDB1 increases from 6753 to 14833, which outperforms CDB4 that has an average TPS of 12239. Consequently, CDB1 improves 21% TPS and reduces 34% cost of CDB4, resulting in 1.8x higher P-Score. We also observe that CDB1 outperforms RDS on the concurrency of 100. Nevertheless, AWS RDS still has 16% higher average TPS and 12% lower cost than that of CDB1.

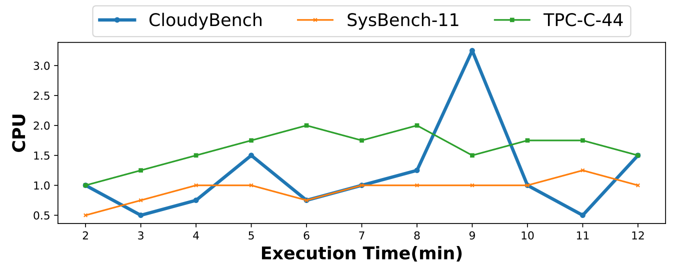

Fig. 9. A Comparison of CPU Fluctuation between CloudyBench and Two Existing Benchmarks (i.e., SysBench and TPC-C).

I. Comparison with Existing Benchmarks

To verify CloudyBench's effectiveness, we compare it with two widely-used benchmarks in the cloud, SysBench and TPC-C. We concentrate on the elasticity evaluation by conducting a 12-minute experiment on CDB3 and collecting the allocated CPU resources accordingly. We run CloudyBench's four elasticity patterns on CDB3 sequentially. We produce a 226MB dataset with 3 tables for SysBench, and each table has size of 300000. We employ OLTP-Bench [7] to run TPC-C with a scale factor of 1. To make a fair comparison, we launch 11 threads on SysBench and 44 threads on TPC-C, respectively. These two numbers lead to the peak and valley points in CloudyBench. As shown in Figure 9, CloudyBench's elasticity patterns lead to notable resource scaling while SysBench's and TPC-C's constant workloads produce the relatively flat resource usage. We observe that the scaling range of CDB3 is quite limited on SysBench's or TPC-C's workload. For instance, CDB3's CPU size scales between 0.5 vCore and 1.25 vCores on SysBench and it scales between 1 vCores and 2 vCores on TPC-C workload. The maximum change between time slots is just 1 vCore. In contrast, CDB3 scales up to 3.25 vCores and scales down to 0.5 vCore during processing CloudyBench's elasticity patterns with peaks and valleys. Particularly, CDB3 scales down from 3.25 vCores to 1 vCore from 9th to 10th minute, experiencing the largest drop of 2.25 vCores, posing more benchmarking challenges.

J. Takeaways and Discussions

Through detailed experiments, we have the following takeaways for both researchers and cloud providers:

(1) Concerning performance, AWS RDS [1] has better cost-efficiency because it achieves higher throughput via its local SSD storage and its resource cost is lower. Under heavy workloads, CDBs can have the comparable performance via asynchronous log replaying. Concerning various architectures of CDBs, disaggregated memory architecture performs the best on transaction processing because of the larger local buffer and remote shared buffer. The local buffer size has also a large impact on the performance. If the buffer size could be tuned for `CDB2` and `CDB3`, they could achieve higher performance.

(2) Concerning elasticity, CDBs significantly outperform AWS RDS by scaling the resources on demand for the elastic patterns. On the one hand, the on-demand scaling up/down can respond to the varied workload in the second level, resulting in a higher resource utilization. On the other hand, the pause-and-resume technique can largely reduce the resource consumption. If scaling down of `CDB1` is improved with on-demand scaling, it would be the clear winner. Moreover, implementing auto-scaling in `CDB4` has also a large potential to achieve the best elasticity because of its memory disaggregation architecture.

(3) Regarding multi-tenancy, AWS RDS achieves the highest T-Score because its high performance with isolated instances. Nevertheless, by scheduling the resources to multiple tenants on demand, `CDB2` and `CDB3` have a better performance and higher resource utilization on the staggered patterns. If they can address the performance drop concerning the contention patterns, we believe CDBs could have more advantages than AWS RDS.

(4) Concerning fail-over, memory disaggregation architecture has the highest recovery and replication speed. As for the recovery, it utilizes the remote buffer pool to enable the fast fail-over. Concerning replication, it leverages the RDMA-based network to quickly ship the logs for page materialization. We recommend all CDBS apply this schema to improve the recovery speed.

(5) Regarding cost-efficiency, we advocate that all CDBs should make more efforts to define the unit cost for each resource clearly and fairly, such that the users can make a clear comparison based on their own applications.

IV. Related Work

Traditional database benchmarks including TPC-C [35] and SysBench [15], have been adopted to benchmark cloud-native OLTP databases [38], [6], [26], but they were merely used to evaluate the read/write performance. YCSB [5] and its variant [27] were proposed for benchmarking cloud systems, but mainly focus on NoSQL data stores. Binnig et al. [2] once pointed out transactional TPC benchmarks like TPC-W are not sufficient for the cloud as they fall short of tailored metrics for scalability, cost, elasticity, and fault tolerance. OLTP-Bench (a.k.a BenchBase) [7] is a relevant work which has integrated many database benchmarks, including OLTP benchmarks (e.g., TPC-C, SmallBank, TATP), OLAP benchmarks (e.g., TPC-H and TPC-DS) and even an HTAP benchmark (i.e., hyadapt). Despite its diversity, it has no specific component for evaluating elasticity, multi-tenancy, and fail-over of the cloud-native databases. Moreover, it contains no tailored metrics for quantifying the performance and cost of existing cloud-native databases. Pang et al. [26] recently utilized TPC-C and SysBench to investigate the performance of various disaggregated architectures by developing an open-sourced CDB, called openAurora. However, it did not evaluate the elasticity, multi-tenancy, and fail-over. Moreover, it did not evaluate the memory disaggregation architecture.

Existing cloud-oriented database benchmarks mainly retrofit established analytical benchmarks [34]. However, they cannot be used to evaluate cloud-native OLTP databases. For instance, CAB [37] creates a cloud analytic benchmark by incorporating workload patterns and multi-tenancy into TPC-H. Particularly, it generates multiple databases with varying scale factors to simulate multi-tenancy, the workload arrival patterns are randomly generated based on the defined patterns. Unfortunately, CAB contains no transactional workload. Other than domain-specific benchmarks, there exists microbenchmarks aiming to synthesize the specific workload characteristics in production. For instance, given an I/O trace in production, CDSBen [48] and Stitcher [40] developed a learning-based I/O workload benchmark for evaluating the performance of cloud-based storage. However, the major limitation of them is that the real I/O trace or query logs are often inaccessible due to privacy requirement and different applications may have disparate traces and metrics.

There exist numerous works [10], [30], [13], [14] studying the cloud service performance and elasticity in cloud computing. For instance, Garg et al [10] defines a framework to quantify cloud computing services, considering how to define the metrics such as agility, cost and usability as well as their weights. Bertino et al [30] studied the performance variance in the cloud using AWS EC2. Islam et al [14] and Hwang et al [13] evaluated the elasticity of cloud platforms mainly using TPC-W. Unfortunately, these works were published over ten years and solely studied the cloud performance in the virtual machine level. They also did not consider new cloud service features such as multi-tenancy and fail-over. In contrast, our work evaluated the performance of the state-of-the-art cloud-native databases. Moreover, we defined new metrics to quantify the overall performance of CDBs.

V. Conclusion

In this work, we propose a new benchmark `CloudyBench`, for evaluating the key features of cloud-native databases, including elasticity, multi-tenancy, and cost-efficiency. We design tailored workloads for benchmarking cloud-native databases. Furthermore, we propose new metrics to quantify their performance considering throughput, cost, elasticity, multi-tenancy, replication speed, fail-over efficiency. Experimental results over five representatives offer a number of key findings and verify the effectiveness of `CloudyBench`.

References

[1] Amazon Web Service. Relational Database Service. https://aws.amazon.com/rds/, 2024.

[2] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing. How is the weather tomorrow? towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, pages 1–6, 2009.

[3] W. Cao, Y. Zhang, X. Yang, et al. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD*, pages 2477–2489, 2021.

[4] Y. Chen, A. Pan, H. Lei, A. Ye, S. Han, Y. Tang, W. Lu, Y. Chai, F. Zhang, and X. Du. Tdsql: Tencent distributed database system. *Proceedings of the VLDB Endowment*, 17(12):3869–3882, 2024.

[5] 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.

[6] A. Depoutovitch, C. Chen, J. Chen, et al. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*, pages 1463–1478, 2020.

[7] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[8] H. Dong, C. Zhang, G. Li, and H. Zhang. Cloud-native databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[9] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[10] S. K. Garg, S. Versteeg, and R. Buyya. A framework for ranking of cloud computing services. *Future Generation Computer Systems*, 29(4):1012–1023, 2013.

[11] C. Y. Gómez-Llanez, N. R. Diaz-Leal, and C. R. Angarita Sanguino. A comparative analysis of the ERP tools, ODOO and Openbravo, for business management. *Aibi Revista de Investigación*, 8(3 (2020)):145–153, 2020.

[12] J. Gray. Database and transaction processing performance handbook., 1993.

[13] K. Hwang, X. Bai, Y. Shi, M. Li, W.-G. Chen, and Y. Wu. Cloud performance modeling with benchmark evaluation of elastic scaling strategies. *IEEE Transactions on parallel and distributed systems*, 27(1):130–143, 2015.

[14] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 85–96, 2012.

[15] A. Kopytov. SysBench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[16] G. Li, H. Dong, and C. Zhang. Cloud databases: New techniques, challenges, and opportunities. *Proc. VLDB Endow.*, 15(12):3758–3761, 2022.

[17] G. Li, H. Dong, and C. Zhang. Cloud databases: New techniques, challenges, and opportunities. *Proceedings of the VLDB Endowment*, 15(12):3758–3761, 2022.

[18] G. Li, W. Tian, J. Zhang, R. Grosman, Z. Liu, and S. Li. Gaussdb: A cloud-native multi-primary database with compute-memory-storage disaggregation. *Proc. VLDB Endow.*, 17, 2024.

[19] G. Li, W. Tian, J. Zhang, R. Grosman, Z. Liu, and L. Sihao. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment*, 17(5):1–12, 2024.

[20] T. Li, B. Chandramouli, S. Burckhardt, and S. Madden. Darq matter binds everything: Performant and composable cloud programming via resilient steps. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[21] M. Luksa. *Kubernetes in action*. Simon and Schuster, 2017.

[22] E. Milkai, Y. Chronis, K. P. Gaffney, Z. Guo, J. M. Patel, and X. Yu. How good is my HTAP system? In *SIGMOD*, pages 1810–1824. ACM, 2022.

[23] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[24] V. Narasayya, S. Chaudhuri, et al. Cloud data services: Workloads, architectures and multi-tenancy. *Foundations and Trends® in Databases*, 10(1):1–107, 2021.

[25] Neon. Local File Cache (LFC). https://neon.tech/docs/extensions/neon/, 2024.

[26] X. Pang and J. Wang. Understanding the performance implications of the design principles in storage-disaggregated databases. In *SIGMOD*, pages 1–26, 2024.

[27] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi. Ycsb++ benchmarking and performance debugging advanced features in scalable table stores. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[28] O. Poppe, T. Amuneke, D. Banda, A. De, A. Green, M. Knoertzer, E. Nosakhare, K. Rajendran, D. Shankargouda, M. Wang, et al. Seagull: An infrastructure for load prediction and optimized resource allocation. *arXiv preprint arXiv:2009.12922*, 2020.

[29] O. Poppe, Q. Guo, W. Lang, P. Arora, M. Oslake, S. Xu, and A. Kalhan. Moneyball: proactive auto-scaling in Microsoft Azure SQL database serverless. *Proceedings of the VLDB Endowment*, 15(6):1279–1287, 2022.

[30] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[31] A. Skendzic and B. Kovacic. Microsoft office 365-cloud in business environment. In *2012 Proceedings of the 35th International Convention MIPRO*, pages 1434–1439. IEEE, 2012.

[32] R. Su and X. Li. Modular monolith: Is this the trend in software architecture? *arXiv preprint arXiv:2401.11867*, 2024.

[33] S. Sunkari. A brief review on crm, salesforce and reasons stating salesforce as one of the top crm's. *Salesforce and Reasons Stating Salesforce as One of the Top CRM's (June 18, 2022)*, 2022.

[34] J. Tan, T. Ghanem, M. Perron, X. Yu, M. Stonebraker, D. DeWitt, M. Serafini, A. Aboulnaga, and T. Kraska. Choosing a cloud DBMS: architectures and tradeoffs. *Proceedings of the VLDB Endowment*, 12(12):2170–2182, 2019.

[35] Transaction Processing Performance Council. TPC-C, 2021.

[36] Transaction Processing Performance Council. TPC-H, 2021.

[37] A. Van Renen and V. Leis. Cloud Analytics Benchmark. *Proceedings of the VLDB Endowment*, 16(6):1413–1425, 2023.

[38] A. Verbitski, A. Gupta, D. Saha, et al. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*, pages 1041–1052, 2017.

[39] M. Vuppalapati, J. Miron, R. Agarwal, et al. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*, pages 449–462, 2020.

[40] C. Wan, Y. Zhu, J. Cahoon, W. Wang, K. Lin, S. Liu, R. Truong, N. Singh, A. M. Ciortea, K. Karanasos, et al. Stitcher: Learned workload synthesis from historical performance footprints. In *EDBT*, pages 417–423, 2023.

[41] J. Wang and Q. Zhang. Disaggregated database systems. In *Companion of the 2023 International Conference on Management of Data*, pages 37–44, 2023.

[42] Wikipedia. David dewitt, 2023.

[43] X. Yang, Y. Zhang, H. Chen, C. Sun, F. Li, and W. Zhou. Polardb-scc: A cloud-native database ensuring low latency for strongly consistent reads. *Proceedings of the VLDB Endowment*, 16(12):3754–3767, 2023.

[44] C. Zhang, G. Li, and T. Lv. HyBench: A New Benchmark for HTAP Databases. *Proceedings of the VLDB Endowment*, 17(5):939–951, 2024.

[45] C. Zhang, G. Li, J. Zhang, X. Zhang, and J. Feng. HTAP Databases: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20, 2024.

[46] C. Zhang and J. Lu. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases*, 39(1):1–33, 2021.

[47] C. Zhang, J. Lu, P. Xu, and Y. Chen. Unibench: a benchmark for multi-model database management systems. In *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27–31, 2018, Revised Selected Papers 10*, pages 7–23. Springer, 2019.

[48] J. Zhang, W. Jiang, B. Tang, H. Ma, L. Cao, Z. Jiang, Y. Nie, F. Wang, L. Zhang, and Y. Liang. CDSBen: Benchmarking the Performance of Storage Services in Cloud-Native Database System at ByteDance. *Proceedings of the VLDB Endowment*, 16(12):3584–3596, 2023.

[49] Y. Zhang, C. Ruan, C. Li, X. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo, et al. Towards cost-effective and elastic cloud database deployment via memory disaggregation. *Proceedings of the VLDB Endowment*, 14(10):1900–1912, 2021.