X-Stor: A Cloud-native NoSQL Database Service with Multi-model Support

Hongyu Lei
Chunhua Li*
Ke Zhou
hylei@hust.edu.cn
li.chunhua@hust.edu.cn
zhke@hust.edu.cn
WNLO, HUST

Jianping Zhu
Kezhou Yan
Fen Xiao
felixzhu@tencent.com
kezhouyan@tencent.com
cherryxiao@tencent.com
Tencent Inc.

Ming Xie
Jiang Wang
Shiyu Di
reganxie@tencent.com
wangjiang@hust.edu.cn
sparrow@hust.edu.cn
Tencent Inc.; WNLO, HUST

ABSTRACT

In recent years at Tencent, we have observed that the use of multiple NoSQL databases for storing business data with diverse models has led to increased programming and deployment costs, as well as inefficient maintenance and underutilized resources. In this paper, we report X-Stor, a cloud-native NoSQL database system that supports multiple data models by extending different storage engines and efficiently managing them through a unified control plane. This design significantly reduces expenses and enables rapid expansion of new models, while seamlessly supporting their complete functionality through storage engine extensions. By consolidating multi-tenant services and data models on the same physical machines, X-Stor significantly enhances the utilization of cluster resources. Additionally, X-Stor introduces a standardized metric called Request Unit (RU) to measure tenant resource consumption for consumption-based pricing purposes. Leveraging this metric, we design RU-based resource management strategies and achieve efficient multi-tenant resource isolation and system load balancing. Currently, X-Stor manages a storage capacity of over 12PB for online operational data, including more than 100,000 tables with multiple data models. It handles 700 billion requests per day with a peak of 30 million requests per second. We evaluate the performance of X-Stor on popular benchmarks and production workloads. The results show that X-Stor performs well under diverse data models.

PVLDB Reference Format:
Hongyu Lei, Chunhua Li, Ke Zhou, Jianping Zhu, Kezhou Yan, Fen Xiao, Ming Xie, Jiang Wang, and Shiyu Di. X-Stor: A Cloud-native NoSQL Database Service with Multi-model Support. PVLDB, 17(12): 4025 - 4037, 2024.
doi:10.14778/3685800.3685824

1 INTRODUCTION

NoSQL databases have become indispensable tools for managing massive amounts of data in many businesses ecosystem due to their specialized optimization and support for various data models. For instance, in the field of finance, key-value databases are used to store and process large volumes of transaction records and achieve quick read and write operations. In the field of Internet of Things, time-series databases can effectively store vast amounts of time-series data generated by devices, and support rapid query and analysis based on time ranges. In e-commerce field, document databases can be used to store and manage a large amount of product information, user reviews and order data. The flexible document structure can adapt to different types of goods and support complex query, helping enterprises achieve personalized recommendations and precision marketing. Meanwhile, graph databases are used to modeling complex relationships between entities and provide powerful querying service in social networks.

Tencent Inc., as the largest social network service company in China, whose cloud databases provide data storage and management services for massive users, supporting diverse businesses covering social networks, games, e-commerce, finance, advertising, etc. These businesses make use of various NoSQL databases to manage data for different purposes. For instance, graph databases are used in social networks to store user relationships and wide-column stores [12] are employed to maintain user profiles, while document databases are utilized in advertising services for storing material data and time-series databases record user behavior data.

As businesses operations expand, some issues have arisen when storing operational data in different NoSQL database. First, incorporating support for novel data models into an existing system incurs high costs because it necessitates developing a new NoSQL system from scratch, which also entails redundant functionalities. Second, deploying multiple heterogeneous databases at scale leads to system resources isolation for different NoSQL databases, which not only complicates maintenance but also hinders efficient resource sharing among clusters.

To address these issues, we develop X-Stor, a cloud-native multi-model NoSQL database system that supports the storage and operation of multiple data models within a single cluster. It allows rapid extension to new data models, thereby reducing development costs. Additionally, it decouples database services from resource control functions and manages cluster resources and various models of databases through a unified control plane. This design not only improves the reusability of components, but also greatly simplifies system maintenance and further reduces deployment costs.

*Chunhua Li is the corresponding author.
HUST: Huazhong University of Science and Technology.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685824

In the design of X-Stor, we focus on the following two fundamental properties:

Comprehensive multi-model support. Most existing NoSQL database systems adopt a single storage engine architecture to facilitate multiple data models, wherein additional data models are derived from an underlying implementation of one specific data model. For example, MongoDB's WiredTiger [25] storage engine facilitates storage and operations for document model data, as well as query operations for graph and time-series data. ArangoDB [8] uitilizes RocksDB [26] as its storage engine to store JSON objects and provides support for key-value and graph data models. Conversely, Redis [37] stores key-value model data in memory while enabling writing and querying capabilities for document and time-series data through Redis Modules [38]. These solutions lack extensibility to novel data models, and the extended data models often only support a subset of data operations, resulting in inferior performance compared to dedicated single-model systems.

In contrast, X-Stor achieves multi-model support through the extension of different storage engines. When there is a need to support novel data models, it simply requires extending the corresponding storage engine and data access interfaces within the X-Stor system. Independent storage engines can fully support their respective data models, with performance comparable to that of their single-model counterparts.

Multi-tenancy and serverless. X-Stor is a multi-tenant architecture that stores the same mode of data from different tenants on the same physical machines to ensure high resource utilization, achieving cost savings for our customers. For reasonable resource management, X-Stor introduces a standardized metric called Request Units (or RUs, for short) to quantify the consumption of system resources per request by a tenant, including CPU, memory, disk IO and network IO, etc., enabling consumption-based pricing. Further, X-Stor implements RU-based admission control and performance isolation and prevents database from overload. It also achieves load balancing in multi-tenant clusters through RU-based replica placement and scheduling, effectively improving the utilization of cluster resources.

X-Stor is also a serverless database service that can be deployed at a large scale in the cloud, ensuring high availability and elastic scalability. Tenants are only required to initialize the database for their respective data models, enabling them to focus on data storage and management with different APIs. Moreover, tenants only need to pay for the resources they actually use based on the amount of Request Units.

In summary, our key contributions are as follows:

• We have developed X-Stor, a cloud-native NoSQL database that effectively supports multiple data models. It is fully capable of large-scale cloud deployment and facilitates the storage and manage of diverse data models within a single cluster.

• We describe how X-Stor supports various models through multiple pluggable storage engines, a unified data structure and a set of X-Stor commands. This approach is essential for expanding X-Stor to new data models and providing the models with comprehensive functionality support and respectable performance.

• We introduce a standardized measure called Request Unit to quality the resource consumption of database requests in multi-tenant clusters, and provide a detailed description of its definition and calculation, as well as how to perform resource isolation and load balancing based on RUs.

• We exhibit extensive experimental results on production workloads and popular benchmarks (including YCSB [7] and TSBS [6]). The results show that X-Stor performs well under diverse data models.

2 THE ARCHITECTURE OF X-STOR

X-Stor consists of multiple microservices, which are fully deployed in containers and orchestrated through Tencent Kubernetes Engine (TKE) [5]. Figure 1 depicts the internal architecture of an X-Stor cluster, which consists of multiple Data Planes and one Control Plane. Each Data Plane only serves for a specific data model, namely, the similar businesses operations from different tenants are served by the same data plane, thus X-Stor is also a multi-tenant architecture. The control plane is responsible for centrally executing management operations on database resources and cluster resources across all data planes, such as partitioning, table creation, node deployment, and pod eviction.

Data Plane. The Data Plane enables users to perform CRUD operations on their data with APIs from different databases. These APIs come not only from popular open-source NoSQL databases like Redis and InfluxDB [23], but also from some of our own in-house NoSQL databases. The Data Plane comprises three layers: Access Layer, Cache Layer and Storage Layer. Storage Layer serves as the core of X-Stor, encompassing a cluster of storage nodes. Each node hosts multiple database pods, each of which runs an autonomous database storage engine container that stores partitioned data for tenants. These pods work together to serve the tenants. X-Stor's multi-tenant architecture enables data from different tenants to be stored in a single storage engine, facilitating resource sharing among tenants within a container.

The Access Layer consists of a group of gateway nodes, each of which deploys multiple pods to offer gateway services. Typically, tenant requests first reach the Access Layer. Then, the Protocol Adapter converts different types of API calls made by tenants into unified commands for X-Stor (details in section 3), and sends them to the appropriate storage node based on routing information managed by the Router. The gateway service also offers functionalities such as filtering, aggregation and sorting for across-storage-node requests. Besides, it employs the Consistency Manager to govern the read logic based on various consistency models including strong, eventual and bounded staleness consistent reads. The Cache Layer is formed by a group of distributed cache nodes that store hot partition data for tenants and perform replica management and consistency based on multi-Raft protocol [20].

Control Plane. The Control Plane manages all resources within an X-Stor cluster and offers metadata service for all Data Planes. It consists of Admin Service, Resource Manager and Metadata Service. The Admin Service provides authorization and authentication for data management requests from tenants and OSS (Operations Support System) via User Admin and Service Admin components.

The Workflow Service is the core service of the Control Plane, responsible for executing all management operations within the cluster. X-Stor abstracts all asynchronous and time-consuming management tasks into workflows, including resource control workflows like CreateDatabase workflow and data management workflows

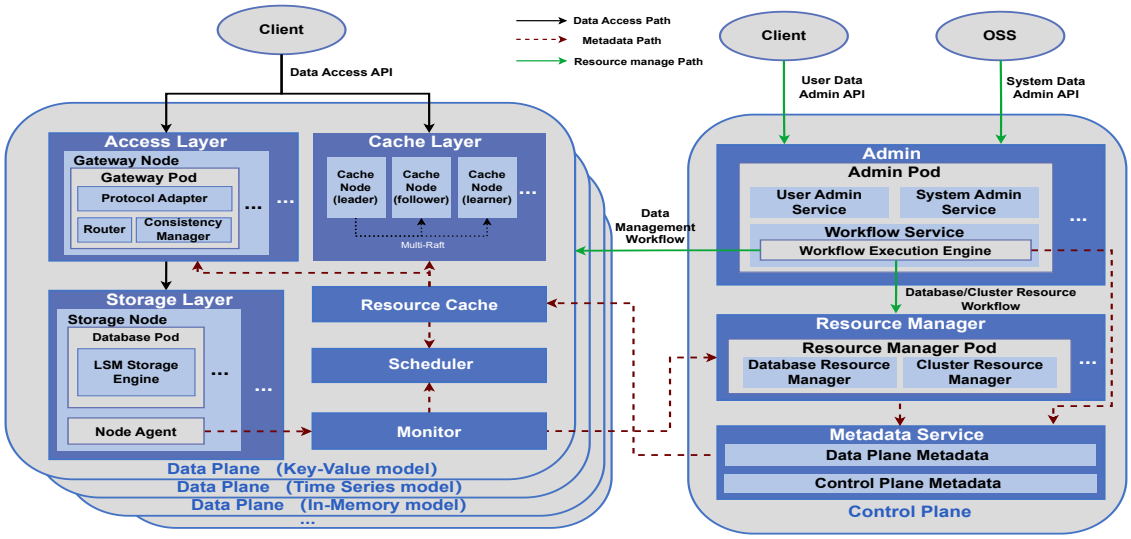


Figure 1: The architecture of an X-Stor cluster. It consists of multiple Data Planes and one Control Plane, with each Data Plane supporting database services for a specific data model. That is, the similar businesses from different tenants are served by the same Data Plane, which achieves multi-tenancy isolation at the Access Layer and the Storage Layer. The Control Plane offers metadata service for itself and all Data Planes, and centrally performs management operations on database resources and cluster resources (such as partitioning and table creation, node deployment and pod eviction). In addition, the Monitor collects and aggregates real-time database service status from storage nodes, such as SLA metrics and pod resource usage, and sends it to the Resource Manager on Control Plane for making decisions on replica placement and scheduling for Data Plane.

such as MigrateNode workflow. Tenants, OSS, and other services call the Workflow Service's internal RPC server interfaces, triggering the Workflow Service to construct and execute the workflows they need. The Resource Manager stores such metrics as resource usage about cluster and database provided by Monitor, and formulates resource management strategies based on these metrics, such as decisions on partition placement and procedures for addressing disk failures. The Metadata Service, comprising MongoDB clusters, manages metadata for all Data Planes and Control Plane and executes real-time updates using change streams.

Each storage node in the Data Plane deploys a Node Agent for collecting real-time database service status information (such as SLA metrics and pod resource usage) from its node and pushes it to the Monitor. The Monitor aggregates this information and sends it to the Resource Manager in the Control Plane. When X-Stor launches, all gateways and cache nodes in the Data Plane register with the Resource Cache, which is responsible for validating and distributing metadata for the Data Plane. The Scheduler periodically inspects Data Planes based on these metadata information and decides whether to trigger the resource management interface of Control Plane. The Control Plane then constructs and executes the corresponding workflows for Data Plane, such as replica placement and scheduling workflow as discussed in Section 5.3.

3 LARGE SCALE DEPLOYMENT

X-Stor decouples database service from resource management and adopts a shared-nothing architecture at Storage Layer, where each storage node independently manages its own system resources and disk space. This architecture benefits X-Stor deployment at a large scale in cloud, offering both scalability and isolation. Currently, the storage capacity of X-Stor cluster for online operational data exceeds 12PB, including over 100,000 tables from different tenants, with more than 700 billion daily requests and 30 million peak QPS. To reduce deployment costs, we consolidate different services and data models on a single physical machine to enhance resource utilization within the cluster.

Service consolidation. We deploy some gateway pods on storage nodes with lower workloads to utilize the idle resources available on those nodes, i.e. the gateway pods and database pods share resources on the same physical machine. Although database pods on storage node belong to the same Data Plane, gateway pods can come from any Data Plane within an cluster due to their statelessness. We exploit *HPA* [30] to dynamically scale the number of gateway pods in response to fluctuations in traffic, thereby optimizing performance during peak periods and costs during lulls.

In order to reduce the probability of data loss caused by disk failures, inspired by Copyset replication [15], we further divide Data Plane into two hierarchical levels: Subcluster and Subcluster Group. As illustrated in Figure 2, a Subcluster Group is formed from 27 storage nodes, serving as the fundamental unit for scaling. Within a single Subcluster Group, there are 12 Subclusters, each consisting of database pods from 27 distinct storage nodes. In terms of data distribution, the three replicas of a data partition are confined to a single Subcluster. Most Subcluster Groups consist of 27 storage nodes, while some smaller clusters only contain 9 storage nodes. These configurations depends on the trade-off between data reliability and data recovery speed.

Data model consolidation. In the early development of X-Stor, we employed storage nodes with high-capacity SSD (Solid State Disk) to support data models that require persistent and high performance storage, such as key-value and time-series models. Additionally, we equipped in-memory models with nodes featuring large memory capacity. However, we noticed that a significant portion of the SSD nodes' memory was not effectively utilized. Based on our shared-nothing and multi-engine architecture, we can easily consolidate a group of database pods with different data models onto the same storage node, enabling data models with varying resource requirements to take full advantage of the resources of a single storage node.

As we all know, the in-memory model requires less disk space and can coexist with other models that require large capacity on storage nodes. The Resource Manager periodically initiates a migration workflow based on the memory usage of nodes. This workflow aims to reallocate DBS pods dedicated to the in-memory model to nodes with high capacity SSDs. When a node clears pods on it, the hosted in-memory model automatically goes offline. The process first tries to clear the node with the lowest memory usage by moving all its pods and then disabling the node. If no node is available, the workflow will resort to a worst-fit [41] strategy, moving pods from nodes with higher memory usage to SSD nodes with lower memory usage, in ascending order of memory usage. This continues until all SSD nodes reach a predetermined memory threshold or the maximum number of migration operations is reached.

4 COMPLETE MULTI-MODEL SUPPORT

In this section, we describe how X-Stor supports various models through multiple pluggable storage engines, a unified data structure and a set of X-Stor commands.

Multiple Storage Engines. X-Stor utilizes multiple storage engines at storage layer to support the scalability of data models. Each data model is supported by a single storage engine. For example, to support a time-series model, X-Stor develops a Time-Structured Merge Tree (TSM) [24] engine. In contrast to the typical design of storage engines, X-Stor's storage engines only retain the most basic data access interfaces. For instance, for key-value models, only Get/Set/Del and MGet/MSet/Mdel interfaces are retained, as well as data snapshot import and export interfaces. Other functionalities such as write-ahead logging (WAL) and replicators are deployed as separate components in storage layer.

This pluggable storage engine design facilitates rapid development of a new data model. We just need to design a simplified storage engine that focuses on its basic CRUD operations. In addition, optimizations can be applied to individual storage engines, enabling performance improvements for the respective data models. For example, we optimize the inverted index for the TSM engine, enhancing the efficiency in time range queries.

Universal Row. The challenge of multi-storage engine system lies in handling a multitude of custom structures during data transfer and processing in the system. Moreover, in order to accommodate multiple protocols, it is necessary to define various structures and fields for storing command parameters. To address these issues, X-Stor introduces a unified data structure called Universal Row to represent the data models of different storage engines.

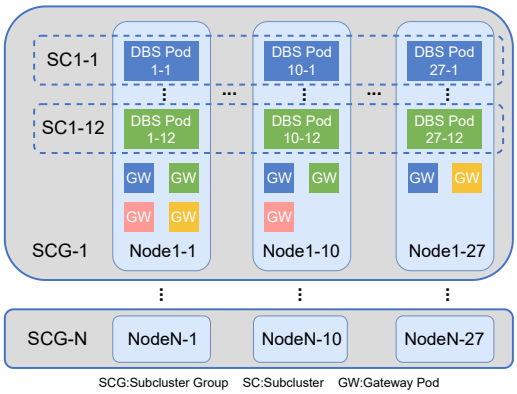


Figure 2: Subcluster Group and Subcluster on a Data Plane

Universal Row (URow, for short) consists of a Row Key, a Row Value and Row attributes. The Row Key includes a partition group and a range group, which is an array made up of partition keys and range keys. The partition key determines which partition the data belongs to, while the range key is used for sorting data within the same partition to facilitate sort searches and range queries. The Row key also contains the hash value of the partition group because we employ hashed sharding. Figure 3 illustrates how to express mainstream NoSQL data models using URow. As the figure shows, the Row Value encapsulates one of four different data types: primitive values such as int, float, or string; container values such as map, set, or array; column objects; or document objects (JSON). The Row properties are initialized by the system and include basic properties of URow, such as the last modified version and timestamp.

URow has two patterns in X-Stor: serialized and structured. The serialized URow is used for network transmission, where it is encoded in BSON (binary form of JSON) or FlexBuffers [3] format to achieve higher transmission efficiency while reducing memory footprint. The structured URow, on the other hand, is used for manipulating data in requests.

However, due to the flexibility of URow definition, some storage engines may not be able to handle an individual URow-structured data. To solve this issue, we define a more fine-grained data structure at the storage layer, called Record, which further subdivides the URow structure. The value of Record can only be a primitive or container value, while its key includes the hash value of the original URow, the Row key, and additional subkeys (if the URow has been subdivided). Record standardizes the interaction interface between components at storage layer, enabling WAL, replicator and storage engine to interact based on it.

X-Stor Commands. In order to convert operations from various NoSQL systems into a generic expression within X-Stor, we design a set of X-Stor commands, which provide a unified expression for manipulating data at storage layer. Each command is composed of a URow and an Action, where the Action contains one specific operation type, i.e., Fetch, Update, Put, Delete, Scan, or Query. When manipulating data with the X-Stor command, differences in data models between different storage engines are not taken into account. For example, if a table is configured as read-only, the X-Store command like Update, Put, and Delete operations on any data model will be rejected.

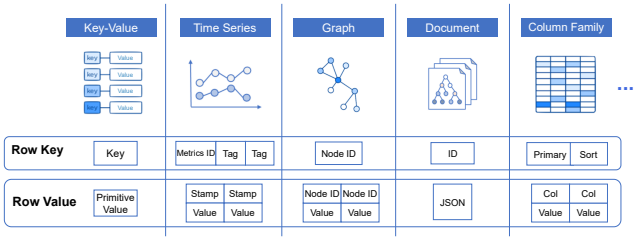


Figure 3: Supporting diverse data models via Universal Row

Based on above designs, X-Stor requires only three key interfaces to manipulate data. As shown in Table 1, ITransform.Transform() accepts a URow and transforms it into a new URow. Depending on the Action, the new URow is either used to create the response or saved back to the engine. The ITranslator.Map() is used to split a URow into multiple Records, while ITranslator.Reduce() is responsible for combining a batch of Records into a single URow.

Figure 4 illustrates an example where a tenant uses the Redis HMSET command to insert some data into X-Stor. The Protocol Adapter in access layer first translates the Redis command into an X-Stor command, in which the data is converted into a URow structure. Then, in storage layer, we utilize the Taskflow Framework [4] to implement different data read and write processing flows. Next, the Translator parses the X-Stor command and splits it into several Records. Afterwards, the Encoder encodes each Record into a Binary Large Object (BLOB), thereby reducing the storage footprint for records in the storage engine. Finally, the basic write interface of the storage engine is invoked to insert the binary data.

Table 1: Key interfaces for Universal Row

Method	Parameters	Returns
ITransform.Transform( )	Input URow	Output URow
ITranslator.Map( )	URow	Batch of Records
ITranslator.Reduce( )	Batch of Records	URow

5 MULTI-TENANT RESOURCE MANAGEMENT

The multi-tenancy feature of X-Stor allows tenants to share resources within a single database container, which means database requests from different tenants can access replicas managed by the storage engine. In this section, we first introduce the definition of Request Unit (RU) and its quantification methods, then we describe how X-Stor achieves multi-tenant resource isolation and resource scheduling based on RUs.

5.1 Request Unit

The resource consumption varies significantly among database requests, and each request has its dominant resources. For example, key-value model requests primarily consume CPU and I/O resources, whereas some time-series model queries require not only CPU but also a significant amount of memory. Therefore, it is crucial to accurately measure the resource consumption of tenant requests for serverless billing and resource management in cloud databases. X-Stor introduces a standardized metric to measure the resource consumption of tenant requests from different data models,

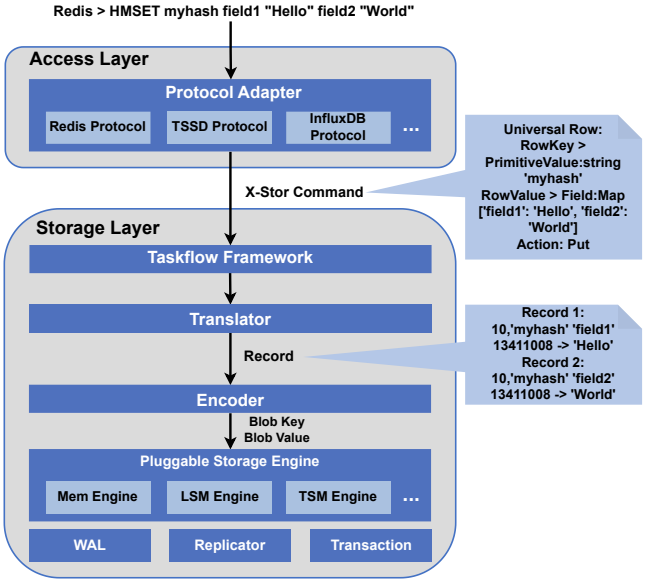


Figure 4: An illustration of X-Stor multi-model support

which is called Request Unit (RU). All tenants' database requests will consume a certain amount of RUs, representing the system resources consumed by the requests. We summarize the design principles of RU as follows:

Represent dominant resources. The RUs needs to reflect the dominant resources consumed by a request, which is relative to the configuration resources of the pod being accessed.

Hardware independence. Requests that consume the same amount of resources should incur the same RU charges, regardless of differences in hardware configurations across storage nodes.

Real-time computing. The RUs for a request needs to be calculated rapidly within a time constraint to meet real-time requirements.

Definition. To meet the different requirements of different scenarios, we define two distinct types of RU: the *Physical RU (PRU)* and the *Logical RU (LRU)*. The Physical RU is represented using a four-dimensional vector that abstracts the resource consumption of CPU, memory, disk IOPS, and network bandwidth incurred by requests. In our system, retrieving 1KB data is stipulated to consume 1 Physical RU, which represents the minimal resource consumption associated with a single request, i.e.,

$$PRU = \langle PRU_{CPU}, PRU_{memory}, PRU_{I/O}, PRU_{network} \rangle \quad (1)$$

The Physical RU is used for scenarios requiring fine-grained resource management across different dimensions within the system, such as replica scheduling (discussed in section 5.3). On the other hand, the Logical RU is a scalar value derived from the Physical RU. It is tailored for tenant-oriented scenarios, such as multi-tenant isolation (discussed in Section 5.2) and billing.

Calculation. The RU is quantified in two parts: the RU consumption by requests and the RU capacity of pods. First, we measured resource consumption in four dimensions resulting from retrieving 1KB data in different models, described as $1KB\ Read_{dim}$, Where dim represents the four dimensions of system resources, i.e., CPU,

memory, I/O and network. Next, we establish the maximum resource capacity for each of the four dimensions of a pod based on its configuration, which is described as *PodResourceCapacity*. After that, we can calculate the pod's Physical RU capacity:

$$Pod\ PRU_{dim} = \frac{\text{PodResourceCapacity}_{dim}}{1KB\ Read_{dim}} \tag{2}$$

We choose the minimum Physical RU capacity of this pod as its Logical RU capacity, so the Logical RU represents the dominant resources of this pod:

$$Pod\ LRU = \min_{dim}(Pod\ PRU_{dim}) \tag{3}$$

The next step is to calculate RU consumption of requests. To calculate PRU consumption, we first analyze the modules in the system that consume the most resources when processing requests. These modules typically occur in protocol encoding and decoding processes, X-Stor command conversion processes, and read/write operations of the storage engine. Then, we consider the resources used by each module when processing a certain amount of data. For example, to calculate the PRU consumed by a module when retrieving 1KB data, we first measure the module's resource consumption in four dimensions, represented as *ModuleCost*. Then, we can calculate the RU consumption based on this metric:

$$Module\ PRU_{dim} = \frac{ModuleCost_{dim}}{1KB\ Read_{dim}} \tag{4}$$

After completing the iterative computation, we compile a *Physical RU table* that documents the PRU consumption of various modules when handling data with different sizes. For a tenant's request, we only need to identify the involved modules and their invocation frequency. Then, we can refer to the PRU table to determine the PRU consumption. For the Logical RU, it can be calculated as:

$$Request\ LRU = Pod\ LRU * \max_{dim}\left(\frac{request\ PRU_{dim}}{Pod\ PRU_{dim}}\right) \tag{5}$$

Through the above calculation, the Logical RU represents the consumption of dominant resources by a request in a hardware-independent manner.

5.2 RU-based Multi-Tenant Isolation

Based on RU, X-Stor is able to determine the resource consumption of requests in real time, thereby achieving more precise resource management and performing performance isolation in a multi-tenant environment. First, X-Stor performs admission control at access layer to prevent storage nodes overload while ensuring that tenants can obtain the equivalent system resources to their billing. Then, within the database pods in a storage node, X-Stor further protects nodes against overload while maximizing system resource utilization when nodes are idle and avoiding performance degradation due to resource competition among tenants when nodes are constrained.

Admission Control at Access Layer. As illustrated in figure 5, X-stor has multiple gateway services at the access layer. In order to minimize the blast radius in the event of a gateway failure, we logically divide all gateways into separate access pools, with each tenant being assigned to only one access pool. This strategy also reduces the memory overhead associated with storing data

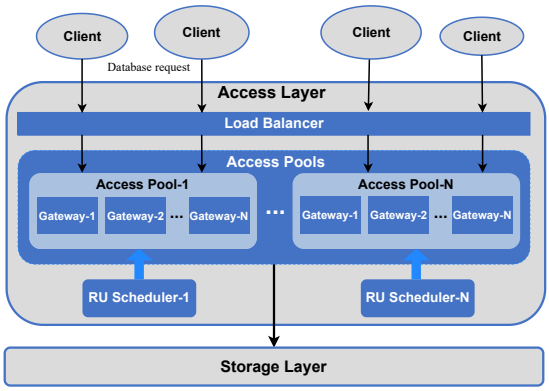


Figure 5: X-Stor access layer overview. Gateway services are logically isolated into separate access pools, serving only a group of tenants.

partition routing information in the gateway services and the cost of Resource Cache routing updates. We assign tenants a certain number of LRU quotas based on their billing, and initially these quotas are evenly distributed across all gateways within the access pool. If the LRU consumption exceeds the quota allocated on the gateway per second, it will reject further requests to prevent an overload of requests entering the storage layer.

The Load Balancer uses a simple load distribution strategy based on QPS to distribute tenant requests across the various gateways. However, balanced QPS distribution doesn't guarantee even RU consumption among gateways due to varying RU costs for different requests, leading to the tenant actually receiving fewer LRUs than their billing. The RU Scheduler in each access pool is responsible for monitoring the RU consumption of each gateway per second, and adjusting the LRU quota on the gateways based on changes to the tenant billing strategy or changes in the number of gateways within the access pool. The RU scheduler addresses load imbalance and traffic bursts between gateways in two mechanisms: *autoscaling* and *burst packages*. With autoscaling, the RU scheduler periodically evaluates the distribution of LRU usage across gateways and reassigns LRUs from less active gateways to those facing higher demand based on their proportion of idleness. For traffic bursts lasting less than one second that occur at any gateway, the RU scheduler provides "bursting packages" for each gateway. These packages consist of a reserved number of LRUs, which can be used by any tenant within the gateway when their LRU consumption exceeds their allocated quota. To avoid gateway overload, there is a limit to the number of burst packages that each tenant can use per second.

In certain data models within X-Stor, queries typically consume more LRU compared to write operations, and some large queries occasionally use up the majority of an entire gateway's LRU quota. To do this, gateways store unused tenant LRU quotas in the last 60 seconds, which is a feature X-Stor calls *accumulated RUs*. For a query, the LRUs cost is not immediately subtracted from the current LRU quota of the gateway (this mechanism can also provide higher privileges for write requests), but from the accumulated RUs starting from when the query is executed. If the accumulated RUs is inadequate, the gateway will not forward the query to storage

layer. Instead, it keeps the query in a waiting queue and will handle it as soon as there is enough accumulated RUs. In addition, to further handle queries that consume a large amount of LRUs, the gateway can schedule a certain amount of accumulated LRUs from other gateways in the access pool. We prioritize gateways with the longest waiting queries first, based on a simple priority strategy, in an attempt to schedule and complete those queries.

Storage Layer Isolation. At storage layer, a storage engine stores replicas from different tenants. Our main concern is how to allow tenants in a container to utilize as much of their pod's resources as possible, while preventing a "Noisy Neighbor" from monopolizing all the resources and affecting the performance of other tenants. X-Stor uses multiple token bucket algorithm to manage RUs at storage layer. Each pod owns two tiers of token buckets: tenant token buckets and pod token buckets. The size of the tenant token bucket is determined based on the number of partitions the tenant has on the storage node, while the size of the pod token bucket is determined by the pod LRU that we have discussed in Section 4.1. One token corresponding to one LRU. When a tenant's request arrives at the storage node, it must acquire enough tokens before it can be processed. The request first attempts to obtain tokens from both the tenant's token bucket and the pod token bucket. If there are insufficient tokens available, it will then try to obtain extra tokens from the pod token bucket.

The tenant token bucket ensures a tenant's basic resource usage by allocating a portion of resources that cannot be preempted, thus providing stable performance when the tenant's resource consumption is below billing. On the other hand, the pod token bucket allows tenants to use extra resources when pod resources are idle, improving resource utilization while keeping the aggregate resource consumption of all tenants within the limit threshold (considering over-subscription) to prevent node overload.

When both bucket tokens are exhausted, subsequent requests from the tenant will be rejected. However, for certain models, large queries still pose a problem at storage layer. For example, in time-series models, some queries may take longer to complete and the original token bucket algorithm could lead to prolonged starvation for such queries. Besides, we aim to ensure that write requests have a higher priority in time-series models. To do this, we define a queuing threshold for the pod's token bucket. If the token consumption rate of the pod's token bucket exceeds this threshold, we will enqueue subsequent read and write requests into separate read and write queues. The system will process the queued requests in order once the pod has sufficient resources. Write requests will be dequeued before read if they have a higher priority.

5.3 RU-based Load Balancing

In X-Stor, a partition has multiple replicas, which use the Raft algorithm for consensus and leader election. These replicas experience varying workloads due to their business characteristics and data sizes. We define the load of these replicas with the amount of RUs consumed when they are accessed per second. Through RU-based replica placement and scheduling strategies, we achieve more efficient workload balancing within the cluster.

5.3.1 Replica Placement. After a tenant creates a table, the Resource Manager calculates the number of partitions needed based on the tenant's estimation on their own business workload and required storage space. Subsequently, it generates multiple replicas of these partitions. We first evenly distribute replicas of the table across all pods. For the remaining replicas, we calculate a score for each pod:

$$Score = \sqrt{U_{disk}^2 + A_{RU}^2 + A_{disk}^2} \tag{6}$$

where U_{disk} reflects the current disk availability, considering the disk storage capacity already used by the pod. A_{RU} is the LRU allocation rate, which considers the ratio of the total RUs allocated across all replicas on the pod to the LRU capacity limit of the pod. A_{disk} is the allocation rate of the pod's disk space, which is the ratio of the total disk space allocated across all replicas on the pod to the pod's disk capacity. Note that the latter two values take into account the proportion of over-subscriptions in the cluster. These three values are normalized before calculating, and any pod with a value exceeding a predefined threshold is excluded. Replicas are greedily placed on the pod with the lowest score.

5.3.2 Replica Scheduling. The Node Agent on each storage node continuously collects the PRU load of each replica and each pod. This data is then reported to the Monitor, which aggregates the information and forwards it to the Scheduler on Data Plane. Based on this information, the Scheduler analyzes the situation like overloads and imbalances for clusters every 24 hours. If such situations exist, it invokes the Workflow Service interface to initiate a replica scheduling Workflow. We schedule our cluster based on the most utilized resources, considering a cluster of N database pods $P = [pod_1, pod_2, ..., pod_N]$, $PRU^t_{pod_i}$ is the PRU of this resource on pod_i at time point t. Replica scheduling aims to minimize the following three metrics:

• **Violations.** The number of times that the PRU of pods in the cluster exceeds the PRU threshold, which is generally set at 60% of the pod's PRU limit:

$$violation = \sum_i \left| \{t | PRU^t_{pod_i} > threshold\} \right| \tag{7}$$

• **Resource Imbalance.** We use standard deviation to define it:

$$Imbalance = std_{p \in P}(PRU_p) \tag{8}$$

• **Cost.** The cost of scheduling also need to be considered:

$$cost = total\ number\ of\ migrated\ replicas \tag{9}$$

Candidate Replica. Algorithm 1 describes how to select candidate replicas for scheduling. In a given time period T, if a pod's load exceeds k times its average load, we define this time point as an anomaly, or if the load exceeds the PRU threshold at N consecutive time points (typically 3), we define all these time points as anomalies (Line 1-5). Then, for pod with a number of anomalies exceeding the warning threshold, we consider it as high-load pod and select the top k_1 replicas with the highest PRU consumption, increase the contribution of these replicas by one at every time point (Line 6-10). Last, the top k_2 replicas with the highest contribution become the candidate replicas for that pod (Line 11).

Role Switching. The leader replica handles both read and write operations from tenants, and as a result, it often experiences a much higher load compared to the follower replicas. Our first strategy is role switching, which requires very few system resources because

Algorithm 1: Selecting candidate replicas

```
function Selecting(P)
    for Pod p ∈ P do
        for timestamp t ∈ T do
            if load_p^t > k * avg_{q∈P}(PRU_q^{t+i}) or PRU_p^{t+i} >
              threshold, ∀i ∈ [0, N − 1] then
                p.anomaly.add(t)
    for Pod p ∈ P, |p.anomaly| > threshold do
        for timestamp t ∈ p.anomaly do
            C ← top k1 RU consumption replicas in p
            for replica r ∈ C do
                r.contribution++
        p.candidates← top k2 contribution replicas
```

only the role of the replica needs to be changed. After calculating candidate replicas for each high-load pod, we iteratively perform role switching. In each iteration, we first select the low-load pod with the fewest anomalies as the destination pod for the role switching. Next, we select the source pod for the role switching, for every other pod with anomalies, if there exists at least one leader replica which has a follower on the destination pod, we assign the number of anomalies as the score for that pod, and choose the pod with the highest score as the source pod. We select the leader replica acceptable to the destination pod from candidate replicas in the source pod, and then perform a role switching with the follower on the destination pod. If no pod can exchange with this destination pod in a single iteration, the destination pod will be excluded from subsequent iterations. The iteration process ends when ether a high-load pod cannot be found or there are no suitable destination pods.

Replica Migration. Role switching restricts the range of replica scheduling, as it only involves switching roles within a fixed set of pods. Therefore, after role switching, the Scheduler migrates replicas among the pods. For each pod whose anomalies still exceed the threshold after role switching, we iteratively check the remaining replicas in its candidate list. For each candidate replica, we select a destination pod based on the following priorities in descending order: (1) The destination pod with the least number of anomaly points after receiving the replica. (2) The destination pod is in the same Subcluster as the source pod, which is to reduce migration costs, as cross-Subcluster migration requires migration of all replicas in the partition. (3) The destination pod and the replica have the minimum PRU similarity, for which we use cosine similarity:

$$PRU\ similarity = \frac{PRU_p \cdot PRU_{replica}}{|PRU_p| \cdot |PRU_{replica}|} \tag{10}$$

where PRU_p is the vector constituted by the PRU load per minute of the destination pod in the previous period, and $PRU_{replica}$ is the PRU load vector corresponding to the replica. The lower similarity indicates that the two PRUs are complementary, thereby making the load on the destination pod more balanced over time. (4) The destination pod with the smallest total PRU in the previous period. (5) The destination pod with the lowest current disk usage. When comparing two pods, if the difference in the values of a priority is within a set threshold, we then compare the next priority level.

Note that during migration, we always restrict the number of all replicas of a table on each pod to remain average (typically, the number on each pod must not exceed the average by more than 5) to prevent the migration process from disrupting the balanced distribution of replicas in terms of their quantity.

6 EVALUATIONS

Currently, X-Stor provides online support for time series model and various key-value model, including an in-memory and LSM-tree pluggable engine, as well as a modification of FasterKV [13]. We are also actively developing support for other data models. In this section, We conduct extensive experiments to evaluation the performance within an X-Stor cluster, which comprises 12 subclusters, each containing 9 database pods. Through Kubernetes pod configuration files, we enforce resource limitations for each pod, allowing a maximum of 6 cores. The key-value model pods are allocated no more than 10GB of memory, while the in-memory model pods have a memory limit of 134GB. For time series model pods, the memory allocation is restricted to a maximum of 48GB. Additionally, the gateway pod is configured with 4 cores and 10GB RAM.

6.1 System Performance under different models

Key-value model. X-Stor design a pluggable LSM-tree storage engine to store and manage key-value data. We evaluate it using microbenchmark db_bench [2] and select LevelDB [1] as the baseline database, which is a fast, embedded, and lightweight LSM-tree storage. Db_bench provides multiple benchmark tests. We choose *fillseq*, *fillsrandom*, and *readrandom*, which are the most common operations in actual business scenarios. The dataset size for each test is approximately 62GB with various value sizes. Figure 6 shows the throughput and P99 latency under various benchmark tests. Whether for reading or writing, the throughput of X-Stor exceeds LevelDB for all value sizes, only slightly lagging behind LevelDB in random reads with a value size of 1KB. The P99 latency of read and write operations in X-Stor is generally lower than that of LevelDB, especially at a value size of 64KB, where LevelDB shows significant fluctuations in sequential write P99 latency.

In-memory model. We first demonstrate the scalability of X-Stor using in-memory model. X-Stor adopts a shared-nothing architecture in storage layer and can effectively handle load growth by horizontally scaling the number of pods. We use three simulated workloads with different access patterns. Figure 7 illustrates the throughput of the three workloads under different numbers of pods, showing that the throughput of X-Stor's memory model almost linearly increases with the number of pods. we also evaluate the read and write latency of the memory model under different throughputs on YCSB A and B. We use a uniform key distribution with a 128-byte value of key. Figure 8 presents the p50 and p99 latency for reads in the two workloads. The latency variation is minimal regardless of the throughput, especially when the throughput is below 1M. The p50 latency remains nearly unchanged. Figure 9 shows the p50 and p99 latency for writes, indicating that the write latency is more stable, with only a slight increase even at a throughput of 1M.

Next, we focus on the performance of the X-Stor memory model when dealing with frequently accessed keys, known as hot keys.

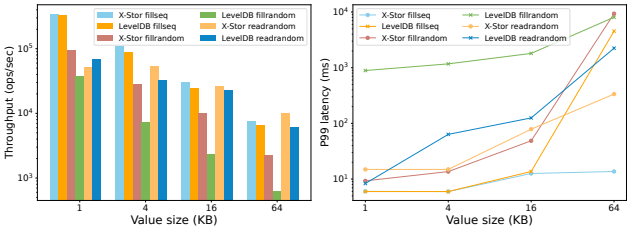


Figure 6: Performance of key-value model on db_bench

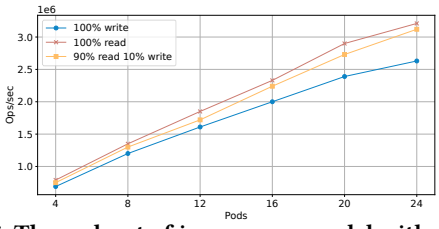


Figure 7: Throughput of in-memory model with pod scaling

We choose Redis as the baseline database to compare with X-Stor's in-memory model, as Redis is the most popular in-memory database. The experiment is based on the YCSB and we test workloads A, B, C, D, and F. In a single test, we create a data with a value of 100B and then launch the corresponding workload. By modifying the *thread* parameter, we simulate multiple concurrent users accessing the database simultaneously. Figure 10 shows the throughput of hot key requests under different workloads. As the number of clients increases, the throughput of both X-Stor and Redis shows an almost linear upward trend. However, X-Stor's performance growth outpaces Redis. When there are 64 clients, except for workload A, X-Stor's performance exceeds that of Redis. Figure 11 shows the latency of hot key requests under different workloads. With fewer clients, X-Stor's latency performance is slightly inferior to Redis. However, as the number of clients increases, X-Stor demonstrates lower latency compared to Redis. X-Stor specifically optimize the read performance of memory storage engine, including the design of lock-free structures and the introduction of multithreading.

Time-series model. For the time-series model of X-Stor, we choose InfluxDB [23] as the baseline database, which is a very popular open-source time series database. It is deployed as a pod on a storage node that matches the configurations of X-Stor, with its maximum cache size set to 10GB, consistent with X-Stor.

As shown in Table 2, we use the DevOps dataset generated by TSBS and three production workloads collected from Tencent's business. For DevOps, we adjust the *host_scale* parameter to control the number of hosts, thereby adjusting the quantity of time series. Each time series contains 12 hours of data with 1 minute interval.

Figure 12a illustrates the write throughput of InfluxDB and the X-Stor time-series model across different host scales of the DevOps dataset. In any timeseries scales, X-Stor demonstrates approximately twice the performance of InfluxDB. Figure 12b shows the write throughput of the X-Stor time-series model compared to InfluxDB under production workloads. As we can see, X-Stor outperforms InfluxDB across all three datasets. Notably, in the Meeting dataset, X-Stor achieves about 4× the throughput of InfluxDB. These are because the additional optimizations we implement for

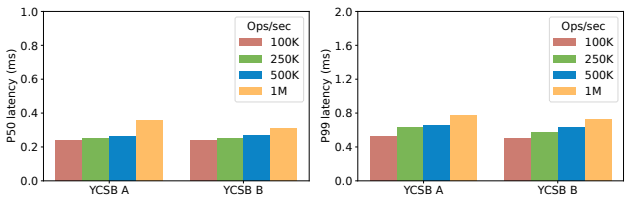


Figure 8: Read latency of In-memory model on YCSB

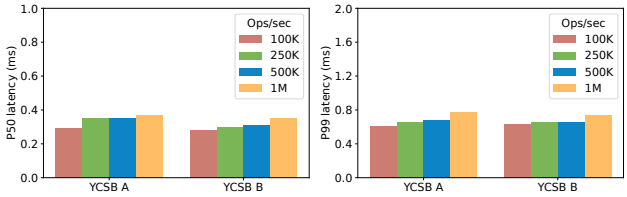


Figure 9: Write latency of In-memory model on YCSB

TSM engine, such as introducing series files at the shard level and parallel data ingestion.

We also evaluate the query performance of both databases. For DevOps dataset, we focus on three queries: single-groupby, which aggregates the maximum value every 5 minutes for a host; max-all, which aggregates all CPU metrics for a host; last point, which retrieves the latest readings from all hosts. Table 3 shows the average latency for these three queries at different host scales; as the number of hosts increases, the latency of the queries also shows a growing trend. However, X-Stor's latency is typically lower than that of InfluxDB, especially in the last point query.

Than we evaluate several commonly used queries in the production workloads dataset, including retrieving distinct tag values, fetching series within a time range, downsampling service failure counts per hour, and aggregating passive request numbers. Table 3 shows the average latency for these commonly used queries. While the show tag query exhibits similar performance between the two, the X-Stor time-series model significantly outperforms InfluxDB in all other queries. The improvement in query performance is largely due to indexing optimizations and the concurrent execution of multiple time series queries within the TSM engine.

6.2 Effectiveness of multi-tenant isolation

To evaluate the autoscaling and bursting packages of RU scheduler, we introduce the concept of RU satisfied rate, which represents the ratio of the actual number of RUs consumed by a tenant compared to the expected number of RUs required every second. In this evaluation, a lower RU satisfied rate indicates that tenants are experiencing request rejections at the gateway due to their skewed access patterns, even if their usage is below their allocated capacity. We use a workload trace from an real tenant. This tenant employs a time-series model and has three gateways in access pool, which we label as Gateway 1, Gateway 2, and Gateway 3. The tenant sends read and write requests to these gateways at random for 15 minutes, with some queries consuming significantly more RU than the allocated quota. Initially, we divide tenants' purchased RU quota equally among the three gateways. Then, we compute the

Table 2: Dataset summary for Time-series model

Dataset	Source	Size
Devops	TSBS	101,000 to 10 million time series, depends on *host_scale*.
Monitor	System metrics from microservices within Tencent Cloud.	Over 2.4 million time series, 44 million data points.
Meeting	Meeting schedule information from Tencent Meeting.	Over 3.5 million time series, 25 million data points (sparse).
Live Streaming	Network metrics from QQ live video streams.	Over 100 million time series (large cardinality).

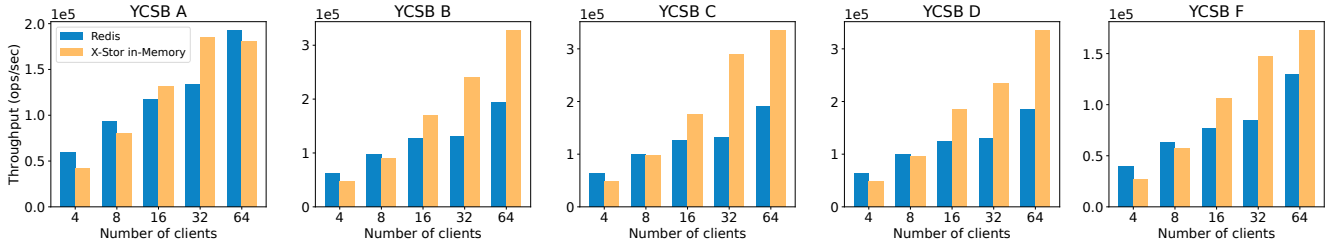


Figure 10: Hot key throughput of In-memory model on different YCSB workloads

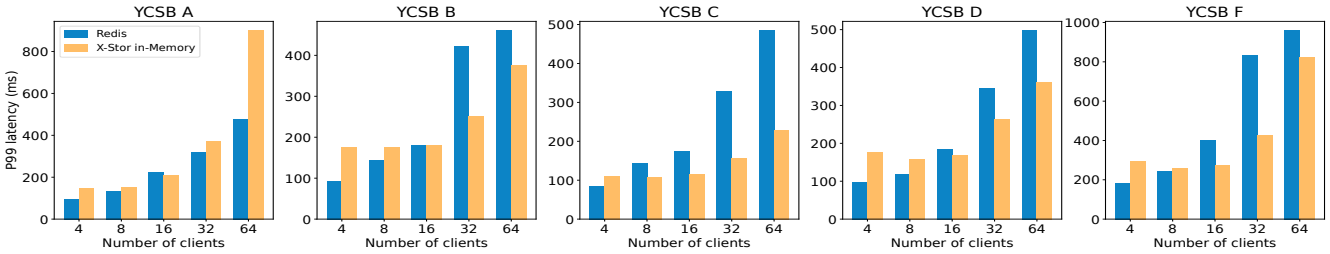


Figure 11: Hot key p99 latency of In-memory model on different YCSB workloads

(a) TSBS workloads **(b) Production workloads**

Figure 12: Write throughput of time-series model

Table 3: Query latencies on TSBS (ms)

Host scale	Single groupby		Max all		Last point	
	①	②	①	②	①	②
1000	17	18	76	68	3158	809
5000	113	118	546	390	19332	4637
10000	318	307	1507	914	60703	17194
100000	4459	2499	20338	8799	749691	216344

① InfluxDB ② X-Stor

RU satisfied rate for the tenant at each gateway every minute:

$$RU\ satisfied\ rate = \frac{true\ RU\ throughput}{expected\ RU\ throughput} \times 100\% \quad (11)$$

Table 5 displays the average RU satisfied rates for the tenant across the three gateways. Upon disabling the autoscaling feature,

Table 4: Query latencies on production workloads (ms)

Query pattern	InfluxDB	X-Stor
Show value	1700	1700
Time range	1100	30
Down sampling	1400	330
Aggregation	200	30

the satisfied rates for Gateways 2 and 3 are only about 70% and 60%, respectively, indicating a skewed distribution of the tenant's requests load toward these gateways. However, after enabling autoscaling at access layer, there is a improvement in the RU satisfied rates for Gateways 2 and 3, with Gateway 3 nearing full satisfaction at 98.5%. Finally, by introducing bursting packages to all three gateways, Gateway 2's satisfied rate shows a significant increase. This improvement can be attributed to the bursting package's ability to accommodate the tenant's short-term burst requests within the RU limitations of the gateway.

Table 5: RU satisfied rate

	Without AC	AC	AC with bursting
Gateway 1	99.7%	100.0%	100.0%
Gateway 2	73.8%	87.3%	96.8%
Gateway 3	62.0%	98.5%	99.0%

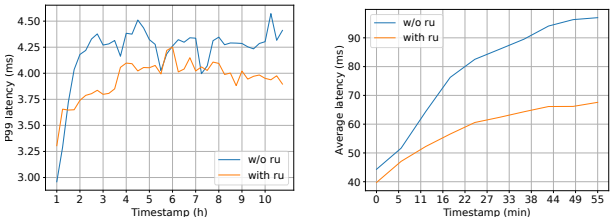
AC stands for gateway autoscaling.

To validate the RU-based storage layer isolation, we verify whether resource contention leads to performance degradation by observing

the latency of requests. We first create two databases form different tenants on the same pod, each with half the RU capacity of the database pod. Then, We send two workloads to this pod: the first tenant initiates writes and gradually increases the amount of concurrency until reaching the database pod's RU capacity limit, the second tenant initiates a smooth workload that is always below the database pod RU capacity limit but sometimes slightly above the tenant's purchased RU.

We first evaluate RU-based database isolation under the key-value model, where the workload is mostly CPU-intensive and lasts about 10 hours. Figure 13a shows the p99 latency of the second tenant, as the workload of the first tenant gradually increases, the implementation of RU isolation results in a significantly lower p99 latency for the second tenant. This is because the isolation mechanism prevents the first tenants from overusing a large amount of the database pod's resources for an extended period through RU-based token bucket. Besides, we calculate the write throughput for second tenants. The overall throughput decreases by less than 0.1% after enabling database isolation. This minimal reduction occurs because when a tenant's RU consumption exceeds their purchased allowance and the database pod depletes its RU capacity, subsequent requests will be rejected until additional tokens are replenished in the bucket. However, such occurrences are infrequent.

We also evaluate database isolation under the time-series model, as memory is the main bottleneck in this model. This workload lasts for one hour, containing a large amount of writing and a certain number of memory-intensive queries. Figure 13b shows the average latency of the second tenant. The latency is significantly reduced when isolation is implemented. Unlike the key-value model, the overall throughput for the time-series model increases by approximately 20% after database isolation is enabled. The primary reason for this improvement is the threshold set for the time-series model, which queues the large queries and permits a batch of write requests to be executed beforehand.



(a) Key-value model **(b) Time-series model**

Figure 13: Latency of the second tenant with/without isolation under different data models

6.3 Effectiveness of RU-based Load Balancing

We evaluate our replica scheduling strategy using production resource traces from a X-Stor cluster. Every minute, we gather the RU consumption and performance metrics for each pod to analyze the differences between pre- and post-scheduling. Figure 14 illustrates the RU consumption within the cluster over a period of approximately 10 hours. Following the completion of scheduling, the majority of pods fall below the RU threshold. This is mainly because the role-switch of high load replicas and migration of these replicas to pods with lower RU consumption based on priority. Additionally, it can be observed that most anomalies of the second type are eliminated (under the k*average line).

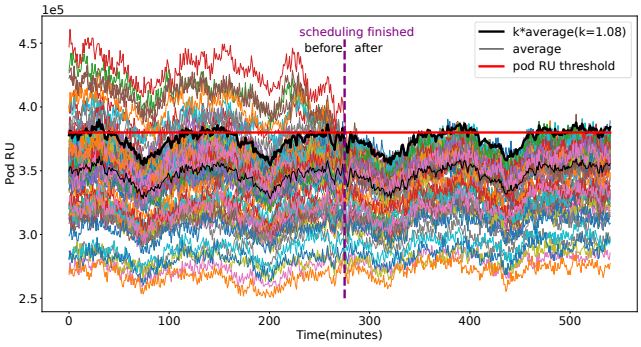


Figure 14: RU consumption before and after scheduling

Table 6: RU metric before and after scheduling

	Before	After	
Violations	2113	265	(↓ 87.46%)
Anomalies	2547	258	(↓ 89.87%)
Standard deviation	28576	21860	(↓ 23.50%)

Table 6 shows the details of scheduling. After scheduling, violations in the cluster are reduced by 87.46% and anomalies are reduced by 89.87%. Additionally, the average standard deviation is reduced by 23.50%, indicating that the imbalance in cluster resource usage is mitigated. Scheduling also decreases the request latency. Table 7 shows the p99 request latency of the cluster before and after scheduling relative to minimum p99 request latency of pods. The average relative p99 latency across all pods decrease by approximately 24%. This is due to the replica scheduling, which avoids resource contention for high-load pods. Meanwhile, other pods experience no contention and as a result, their latency increases only slightly after replicas move in. Consequently, the overall latency is reduced.

Table 7: P99 relative latency before and after scheduling

	Before	After	
mean	0.1211	0.0916	(↓ 24.37%)
median	0.0480	0.0248	(↓ 48.36%)
p99	0.4973	0.4146	(↓ 16.63%)

Finally, we calculate the cost of scheduling. There are 69 role-switching partitions and 130 migrated replicas, compared to the total of 10,170 partitions and 30,510 replicas across the entire cluster. Consequently, role-switching partitions account for approximately 0.67%, and migrated replicas make up about 0.42% of the totals. This demonstrates that the scheduling delivers good results at a fraction of the cost.

7 LESSONS LEARNED

As X-Stor remains in the initial phases of product adoption, our real-world deployment has already yielded some important insights.

Workflow. In X-Stor, all management tasks are designed as workflows, which are composed of multiple asynchronous executable steps. These workflows are both combinable and nestable. For example, the DropNode workflow includes three sub-workflows: DisableNode, DropDisk, and DeleteNode, where the DropDisk workflow itself includes other workflows such as MigrateReplica. This design significantly enhances the reusability of resource management tasks within X-Stor. The status information of each workflow, including the intermediate results after each step is executed, is recorded in the metadata service. In case of a failure, the system will roll back the entire affected workflow to maintain consistency.

Host network for pod. When deploying database pods through a virtualized network, we discover that Kubernetes CoreDNS [16] is a system bottleneck, resulting in a performance degradation of approximately 20% for large-scale clusters (exceeding 1000 nodes and 3000 pods) and hindering the implementation of network optimization protocols such as RDMA. As a result, we set all pods to host network mode and developed a component called PortAlloc to assign node ports. Upon pod initiation, it requests a port from PortAlloc on the node, and replica metadata records the pod's IP and port directly. This simplifies routing forwarding, enhances system throughput, and avoids strong dependency on Kubernetes. Additionally, we are able to use RDMA for network performance optimization in future.

Single point of failure detection. X-Stor employs only two-layer data access path (gateway to database or cache to database) but faces various single points of failure like disk failures, memory MCE faults, and gray failures [21]. To address challenges like detecting new types of single point failures and ensuring quick detection, a hybrid fault detection mechanism combining proactive and passive probing is implemented. This includes deploying Node Agent for active probing and simulating client access to X-Stor, as well as real-time monitoring of gateway access anomalies for passive probing and verification of database pod statuses across multiple nodes. This integrated approach enhances single point fault recall rates and significantly reduces the time required for fault diagnosis.

8 RELATED WORKS

NoSQL systems with multi-model. Many NoSQL databases offer service for multiple data models, such as MongoDB, Couchbase, Aerospike [22], FoundationDB [28], Oracle NoSQL [34], and Cassandra [12]. OrientDB [35] and ArangoDB are designed to be a native multi-model DBMSs from the beginning and support cross-model queries using a unified language. DynamoDB [40] and CosmosDB [10] offer support for various data models but fully hosted by cloud service providers for high availability and scalability purpose. X-Stor is a cloud-native NoSQL database system that opts to support various data models through multiple storage engines within a single cluster, aiming at full support for different data models and rapid model expansion.

Resource management. Effective resource management is crucial for improving the utilization of cluster resources and the service quality for tenants. Currently, many cloud service providers standardize the resource consumption of tenant database operations through an abstract form of requests. For instance, DynamoDB [18] introduces Read Capacity Units (RCUs) and Write Capacity Units (WCUs) to represent the resource consumption of tenant read and write requests, and designed partition-level traffic control based on this. CosmosDB introduces the concept of Request Units (RUs), and designs resource consumption control for tenant databases and containers based on this concept. However, the specific calculation details are not disclosed.

Multi-tenancy is an important feature of cloud databases, allowing multiple tenants to share resources within a cluster to reduce costs. Existing implementations of multi-tenancy include using virtual machines [39], or sharing resources at the operating system level through mechanisms such as CGroups, Job objects, or containers. SQLVM [17, 32, 33] researches sharing CPU and buffer pool memory resources in multi-tenant RDBMS. [9] achieves a high degree of oversubscription efficiency in a multi-tenant DBaaS through CPU and memory reallocation techniques, albeit with isolation in separated containers. isolation in separated container. X-Stor focuses on DBMS-level isolation, meaning tenants share resources within a single database system. It abstracts the consumption of resources for tenant requests and implements multi-tenant isolation strategies in the cloud based on this abstraction.

Tenant placement and scheduling are critical to the stability and performance of cloud databases. Previous research has explored how to place tenants on nodes within a cluster. Most approaches use heuristic methods[41]. [29] uses an estimation method to reduce the probability of resource violations during tenant placement. Placement can also be considered as an online bin packing problem [11, 14, 19, 27, 36], aiming primarily to minimize the number of machines. Our method employs a variant of the worst-fit approach, aiming at balancing system resources across nodes through replica migration. [31] eliminates hotspots and levels load by swapping primary and secondary replicas. Our framework extends this approach by introducing role-swapping under a unified resource abstraction, taking multiple resources into consideration.

9 CONCLUSION

In this paper, we introduce a cloud-native NoSQL database service called X-Stor that supports multiple data models in a single cluster. X-Stor provides a practicable solution to the growing challenges faced by multi-model NoSQL database systems. It does this through an architecture that decouples database services and resource management and the scalability of the storage engine. We introduce the standardized metric Request Unit to measure the resource consumption requested by tenants, and design RU-based performance isolation strategies which ensure stable system operation and effective resources utilization. In addition, the load balance of the system is realized and the access delay is reduced via the RU-based strategies of replica placement and scheduling.

ACKNOWLEDGMENTS

This work was supported by the Key Project of National Natural Science Foundation of China (No. 62232007), partially supported by the Innovation Group Project of National Natural Science Foundation of China (No.61821003). We thank the anonymous reviewers for their insightful opinions and valuable comments. Our thank also goes out to all individuals who contributed to the design and development of the X-Stor system.

REFERENCES

[1] 2021. LevelDB. https://github.com/google/leveldb Last accessed: 2024-07-15.

[2] 2023. DbBenchmark. https://github.com/google/leveldb/tree/main/benchmarks Last accessed: 2024-07-15.

[3] 2023. FlexBuffer. http://github.com/google/flatbuffers Last accessed: 2024-07-15.

[4] 2023. Taskflow. https://github.com/taskflow/taskflow Last accessed: 2024-07-15.

[5] 2024. Tencent Kubernetes Engine. https://www.tencentcloud.com/products/tke. Last accessed: 2024-07-15.

[6] 2024. Time Series Benchmark Suite. https://github.com/timescale/tsbs Last accessed: 2024-07-15.

[7] 2024. Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB Last accessed: 2024-07-15.

[8] ArangoDB. 2023. The Most Complete And Scalable Graph Database For Real-world Use Cases. https://arangodb.com/. Last accessed: 2024-07-15.

[9] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, Jiaqi Liu, Lukas M. Maas, Akshay Mata, Ishai Menache, Justin Moeller, Vivek Narasayya, Matthaios Olma, Morgan Oslake, Elnaz Rezai, Yi Shan, Manoj Syamala, Shize Xu, and Vasileios Zois. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proc. VLDB Endow.* 16, 13 (2023), 4202–4215. https://www.vldb.org/pvldb/vol16/p4202-narasayya.pdf

[10] Microsoft Azure. 2024. Azure Cosmos DB. https://azure.microsoft.com/en-us/products/cosmos-db/. Last accessed: 2024-07-15.

[11] Sebastian Berndt, Klaus Jansen, and Kim-Manuel Klein. 2020. Fully dynamic bin packing revisited. *Math. Program.* 179, 1 (2020), 109–155. https://doi.org/10.1007/S10107-018-1325-X

[12] Apache Cassandra. 2024. Apache Cassandra. https://cassandra.apache.org/_/index.html Last accessed: 2024-07-15.

[13] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 275–290. https://doi.org/10.1145/3183713.3196898

[14] Henrik I. Christensen, Arindam Khan, Sebastian Pokutta, and Prasad Tetali. 2017. Approximation and online algorithms for multidimensional bin packing: A survey. *Comput. Sci. Rev.* 24 (2017), 63–79. https://doi.org/10.1016/J.COSREV.2016.12.001

[15] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. 2013. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*. USENIX Association, 37–48. https://www.usenix.org/conference/atc13/technical-sessions/presentation/cidon

[16] CoreDNS. 2023. Kubernetes CoreDNS. https://coredns.io/plugins/kubernetes/. Last accessed: 2024-07-15.

[17] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. 2013. CPU Sharing Techniques for Performance Isolation in Multitenant Relational Database-as-a-Service. *Proc. VLDB Endow.* 7, 1 (2013), 37–48. https://doi.org/10.14778/2732219.2732223

[18] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 1037–1048. https://www.usenix.org/conference/atc22/presentation/elhemali

[19] Leah Epstein and Meital Levy. 2010. Dynamic multi-dimensional bin packing. *J. Discrete Algorithms* 8, 4 (2010), 356–372. https://doi.org/10.1016/J.JDA.2010.07.002

[20] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084. http://www.vldb.org/pvldb/vol13/p3072-huang.pdf

[21] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 150–155.

[22] Aerospike Inc. 2024. Aerospike. https://aerospike.com/products/database/. Last accessed: 2024-07-15.

[23] InfluxData Inc. 2024. InfluxDB. https://docs.influxdata.com/influxdb/v2/. Last accessed: 2024-07-15.

[24] InfluxData Inc. 2024. InfluxDB TSM. https://docs.influxdata.com/influxdb/v1/concepts/storage_engine/. Last accessed: 2024-07-15.

[25] MongoDB Inc. 2023. MongoDB. https://www.mongodb.com/. Last accessed: 2024-07-15.

[26] Meta Platforms Inc. 2022. RocksDB. https://rocksdb.org/. Last accessed: 2024-07-15.

[27] Shahin Kamali. 2015. Efficient Bin Packing Algorithms for Resource Provisioning in the Cloud. In *Algorithmic Aspects of Cloud Computing - First International Workshop, ALGOCLOUD 2015, Patras, Greece, September 14-15, 2015. Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 9511. Springer, 84–98. https://doi.org/10.1007/978-3-319-29919-8_7

[28] Alfons Kemper. 2022. Technical Perspective: FoundationDB: A Distributed Unbundled Transactional Key Value Store. *SIGMOD Rec.* 51, 1 (2022), 23. https://doi.org/10.1145/3542700.3542706

[29] Arnd Christian König, Yi Shan, Tobias Ziegler, Aarati Kakaraparthy, Willis Lang, Justin Moeller, Ajay Kalhan, and Vivek Narasayya. 2022. Tenant Placement in Over-subscribed Database-as-a-Service Clusters. *Proc. VLDB Endow.* 15, 11 (2022), 2559–2571. https://doi.org/10.14778/3551793.3551814

[30] Kubernetes. 2024. Kubernetes Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Last accessed: 2024-07-15.

[31] Hyun Jin Moon, Hakan Hacigümüs, Yun Chi, and Wang-Pin Hsiung. 2013. SWAT: a lightweight load balancing method for multitenant databases. In *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*. ACM, 65–76. https://doi.org/10.1145/2452376.2452385

[32] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2013/Papers/CIDR13_Paper25.pdf

[33] Vivek R. Narasayya, Ishai Menache, Mohit Singh, Feng Li, Manoj Syamala, and Surajit Chaudhuri. 2015. Sharing Buffer Pool Memory in Multi-Tenant Relational Database-as-a-Service. *Proc. VLDB Endow.* 8, 7 (2015), 726–737. https://doi.org/10.14778/2752939.2752942

[34] Oracle. 2024. Oracle NoSQL. https://www.oracle.com/database/nosql/. Last accessed: 2024-07-15.

[35] OrientDB. 2020. OrientDB Multi-Model. https://orientdb.org/docs/3.0.x/datamodeling/Tutorial-Document-and-graph-model.html Last accessed: 2024-07-15.

[36] Rina Panigrahy, Kunal Talwar, Lincoln K. Uyeda, and Udi Wieder. 2011. Heuristics for Vector Bin Packing. https://api.semanticscholar.org/CorpusID:17946270

[37] Redis. 2024. Redis. https://redis.io/. Last accessed: 2024-07-15.

[38] Redis. 2024. Redis Module. https://redis.com/community/redis-modules-hub/. Last accessed: 2024-07-15.

[39] Mendel Rosenblum and Tal Garfinkel. 2005. Virtual Machine Monitors: Current Technology and Future Trends. *Computer* 38, 5 (2005), 39–47. https://doi.org/10.1109/MC.2005.176

[40] Amazon Web Services. 2024. DynamoDB. https://aws.amazon.com/dynamodb/. Last accessed: 2024-07-15.

[41] Wikipedia. 2024. Bin packing problem. https://en.wikipedia.org/wiki/Bin_packing_problem Last accessed: 2024-07-15.