LogStore: A Cloud-Native and Multi-Tenant Log Database

Wei Cao[†‡] Xiaojie Feng[‡] Boyuan Liang[‡] Tianyu Zhang[‡] Yusong Gao[‡] Yunyang Zhang[‡] Feifei Li[‡]
{mingsong.cw,xiaojie.fxj,boyuan.lby,moyun.zty,jianchuan.gys,taofang,lifeifei}@alibaba-inc.com
[†]Zhejiang University and [‡]Alibaba Group
Hangzhou, China

ABSTRACT

With the prevalence of cloud computing, more and more enterprises are migrating applications to cloud infrastructures. Logs are the key to helping customers understand the status of their applications running on the cloud. They are vital for various scenarios, such as service stability assessment, root cause analysis and user activity profiling. Therefore, it is essential to manage the massive amount of logs collected on the cloud and tap their value. Although various log storages have been widely used in the past few decades, it is still a non-trivial problem to design a cost-effective log storage for cloud applications. It faces challenges of heavy write throughput of tens of millions of log records per second, retrieval on PB-level logs and massive hundreds of thousands of tenants. Traditional log processing systems cannot satisfy all these requirements.

To address these challenges, we propose the cloud-native log database *LogStore*. It combines shared-nothing and shared-data architecture, and utilizes highly scalable and low-cost cloud object storage, while overcoming the bandwidth limitations and high latency of using remote storage when writing a large number of logs. We also propose a multi-tenant management method that physically isolates tenant data to ensure compliance and flexible data expiration policies, and uses a novel traffic scheduling algorithm to mitigate the impact of traffic skew and hotspots among tenants. In addition, we design an efficient column index structure *LogBlock* to support queries with full-text search, and combined several query optimization techniques to reduce query latency on cloud object storage. *LogStore* has been deployed in Alibaba Cloud on a large scale (more than 500 machines), processing logs of more than 100 GB per second, and has been running stably for more than two years.

CCS CONCEPTS

• **Information systems** → **Data management systems**; • **Networks** → *Cloud computing*.

KEYWORDS

cloud-native; multi-tenant; log storage; column store; full text retrieval; shared-nothing architecture; shared-data architecture

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGMOD '21, June 20–25, 2021, Virtual Event, China.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8343-1/21/06.
https://doi.org/10.1145/3448016.3457565

ACM Reference Format:
Wei Cao[†‡] Xiaojie Feng[‡] Boyuan Liang[‡] Tianyu Zhang[‡] Yusong Gao[‡] Yunyang Zhang[‡] Feifei Li[‡]. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457565

1 INTRODUCTION

When enterprises use cloud services to build applications, they expect that their applications in the cloud can run stably, efficiently and securely, and are eager to collect comprehensive logs from various sources including cloud services, gateways, applications, virtual machines or containers. Therefore, cloud vendors must provide products and utilities to store and analyze various kinds of logs for scenarios, such as service metrics measurement, performance tuning and security audit. On the other hand, obtaining operational data (such as DAU/WAU/MAU) through application logs is also very helpful for customers to make business decisions. Log storage and analysis system are essential to support the above scenarios.

The applications deployed on the cloud come from customers in various industries. And hundreds of thousands of tenants are continuously generating logs. Compared with the on-premise environment, the demands to build log storage services in the cloud environment are diverse, which has to overcome the following challenges:

Extremely High Write Throughput and Storage Volume. Cloud-wide applications, services and servers generate logs at a tremendous rate. Take the audit log of Alibaba Cloud DBaaS (Database as a Service) as an example, the overall traffic during working hours is close to 50 million log entries (about 100 GB) per second, as shown in Figure 1. Due to such high write throughput, storage capacity and cost issues also arise when storing a large number of historical logs, especially for some customers who want to archive logs for several years due to compliance requirements (such as finance and banking). In our production environment, the total storage capacity is as high as 10 PB and continues to grow, and the capacity of a single tenant can reach 100 TB.

Large Number of Tenants and Highly Skew Workload. As a cloud-scale log store and analysis service, it needs to serve more than 100,000 tenants. The life cycle of tenant data is different. Some require the latest logs for diagnosis, while others require long-term logs for analysis or archiving. It is necessary to design a flexible tenant management mechanism to achieve differentiated data recycling and billing policies for different tenants. Furthermore, the workloads of tenants are highly skewed. Some tenants contribute most of the storage volume, as shown in Figure 2. Besides, the write throughput of different tenants reaches its peaks at different time, requiring dynamic traffic scheduling.

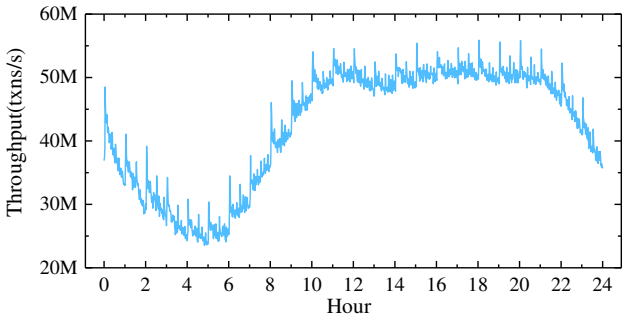


Figure 1: The total write throughput of Alibaba Cloud DBaaS audit logs in a day.

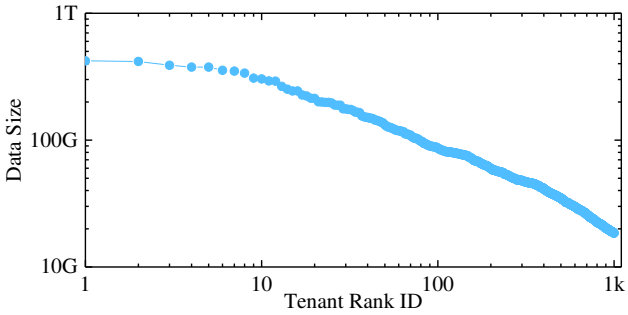


Figure 2: A statistics of tenants' daily data size in the *LogStore* production environment, which is highly skewed and close to the Zipfian Distribution.

Accurate and Interactive Log Retrieval on Massive Data. With petabyte-sized historical logs and highly skewed data distribution, it is non-trivial for large tenants to accurately retrieve the corresponding logs with a tolerable delay, for example, hundreds of milliseconds or seconds. This requires efficient indexing and techniques such as data skipping, caching and prefetching. In addition to the full-text retrieval feature, analytics and aggregations are also needed to fulfill lightweight BI query requirements, such as "which IP addresses frequently accessed this API in the past day?"

Although some of these challenges can be solved by existing systems, no single system can solve all of these challenges. NoSQL systems such as HBase[13] and Cassandra[11] support transactional high-throughput writes, but this comes at the cost of eliminating complex query support. ElasticSearch[12] is a system dedicated to full-text search and has a comprehensive analysis function for log data. But because it chooses to build inverted indexes in real time while ingesting logs, the write throughput cannot meet our requirements. Besides, all these systems adopt the shared-nothing architecture. When the amount of data grows rapidly, the cost of data migration caused by scaling up/down is inherently high, and limits the elasticity.

In recent years, cloud data warehouse systems have emerged, such as Snowflake[27] and Vertica Eon Mode[52]. They separate computing and storage and store data in the cloud object storage Amazon S3[9]. This architecture has excellent scalability and low storage cost. However, these systems reserve separated computing resources for different tenants which is not economical enough for a multi-tenant log service and they also do not support efficient full-text search. Real-time data warehouses like Druid[55] and Pinot[37] have rich data analysis capabilities. They are asynchronous ingestion analytical storage and data is ingested asynchronously from streaming sources like Kafka[18]. The data written to Kafka is not immediately visible by the query before being indexed and consumed. Cloud computing vendors provide real-time log collection and analysis products such as CloudWatch Logs[8], Cloud Logging [34] and Log Service [6], and users can access logs from the cloud console or through specific Open APIs. *LogStore* can be used as the underlying support for such products, providing low-cost storage, tenant isolation experience, and efficient full-text retrieval and analysis.

In this paper, we propose a cloud-native and multi-tenant log database. First, after investigating various distributed databases, we propose an architecture that combines the advantages of both shared-nothing and shared-data designs to meet the requirements for high write throughput and low response time. The data is first written to a write-optimized data structure in the local storage, and then later converted to a read-optimized column storage structure and transferred to the object storage in the background. Meanwhile, the introduction of cloud object storage solves the cost problem but causes problems such as high latency, limited and fluctuating bandwidth. We have summarized the problems encountered and provided best practices. Secondly, due to hotspots or surges in traffic, throughput can sometimes be severely affected. Therefore, we have designed a global flow control algorithm that can effectively balance tenant traffic between nodes to approximate the maximum cluster-wide throughput, and prevent instability due to data skew between tenants. The final challenge is to provide desirable query performance on massive logs. Here we have designed a column-oriented structure *LogBlock*, which supports full-text inverted index as well. Moreover, we have implemented various auxiliary strategies to achieve fast retrieval from massive logs, including the data skipping strategy, multi-level data caches and parallel prefetch method.

The remainder of the paper is organized as follows. Section 2 introduces the background and motivation, and Section 3 presents the cloud-native architecture. Multi-tenant traffic flow control is described in Section 4, and query optimization is discussed in Section 5. Section 6 details the experiments and evaluation. Finally, Section 7 provides a brief overview of related work, and Section 8 concludes the paper.

2 BACKGROUND AND MOTIVATION

As far as we know, there are currently three mainstream architectures for cloud distributed databases.

Shared-Nothing is the most straightforward architecture for the transition from a traditional single-node database to a distributed database. A large number of databases have adopted this architecture, such as Spanner[25], DynamoDB[28] and Redshift [35]. Tables are divided into multiple partitions or shards. Each shard is assigned to a database node. Every node stores the allocated shards on the local disk and is responsible for managing and querying them independently. Therefore, it has good write scalability. The drawbacks of shared-nothing architecture are also obvious.

It tightly couples computing resources and storage resources, resulting in high costs for horizontal scaling and recovering replicas during failover.

Shared-Storage architecture decouples computing and storage resources and is adopted by AWS Aurora[54], Azure HyperScale[19] and Alibaba PolarDB[4, 22]. Several data-intensive tasks, such as data replication, crash recovery, and data materialization could be offloaded from the database kernel to the underlying storage service, thereby making the database node more efficient and elastic. However, there is a bottleneck in the write operation under the shared-storage architecture, and there will be contentions for updating shared resources when multiple nodes modify the same table. Shared-storage is mainly used in OLTP scenarios, so a trend is to use emerging hardware (such as RDMA network and NVM storage) to alleviate the additional delay overhead caused by cross-network I/O operations, but this is not suitable for low-cost storage scenarios such as log archiving.

Shared-Data architecture also separates compute from storage, but it relies on more cost-effective cloud object storage services like Amazon S3 [9] instead of developing in-house storages. The representative of this architecture is Snowflake[27]. In order to comply with the characteristics of object storage, tables are organized as a set of immutable files. Each compute node first caches some table data on the local disk and then converts the cached data into immutable files. These files are eventually uploaded to S3 to complete the writing process. The architecture is more suitable for bulk inserting scenarios of data warehouses. The client cannot receive responses in time until this data is flushed to S3, otherwise there would be a risk of data loss. Therefore, due to batched writes and cloud object storage characteristics, the client has to endure high write latency, which is unacceptable in real-time situations.

We choose to use an architecture that combines the advantages of both shared-data and shared-nothing. This architecture has good write scalability and elasticity, as well as low write latency and storage costs. Next, we list specific design choices.

Separation of Compute and Storage For the storage of massive data, it is very time-consuming to restore data from a replica or scale the cluster through data migration. Therefore, we decouple computing and storage. This also brings the benefit that compute nodes and storage nodes can be scaled independently.

Scalable Reads and Writes In order to scale the write throughput horizontally, input logs will be partitioned and dispersed to different computation nodes according to tenants. And for large tenants, log entries could be further partitioned according to the hash value to balance the load.

Real-time and Low-latency Writes Unlike most data warehouses that use bulk loading, *LogStore* supports low-latency writes and real-time data visibility. In order to achieve optimal latency and stability, we choose to persist data to the local disk, keep multiple copies of data and synchronize WAL (write-ahead logs) between three replicas using Raft[46]. These are all commonly used technologies for shared-nothing architecture. Furthermore, we have a write-optimized row-oriented storage format, avoiding the use of CPU-intensive optimizations, such as building extra indexes or data compression, to maximize the write throughput.

Low-cost and Read-optimized Storage Because local disks cannot save the ever-increasing large amounts of data at an affordable cost, as shared-data architecture, we choose to put data on the object storage. *LogStore* uploads the data on the local disk into the object storage in the background, and we use a read-optimized column-oriented indexed storage format *LogBlock* with a high compression rate to store logs on object storage. *LogBlock* is organized according to the tenant and timestamp dimensions to facilitate fast seek and retrieval. We also learn experiences from other columnar formats such as C-store[49] and DataBlock[41].

Dynamic Flow Scheduling on Heterogeneous Resources Because the data volume and load distribution among tenants are severely skewed, and each tenant has personalized behaviors, such as different business peak time, and variable span of data that needs to be loaded in queries. Therefore, to continuously keep the global optimal strategy for completing writing and query tasks for all tenants, it's necessary to schedule dynamically computing and I/O resource within the entire cluster. In addition, a continuously running production cluster will inevitably have heterogeneous hardware, such as virtual machines with different core numbers and clock speeds. Balancing the load dynamically according to the hardware configuration is also an issue in the production system.

3 CLOUD NATIVE ARCHITECTURE

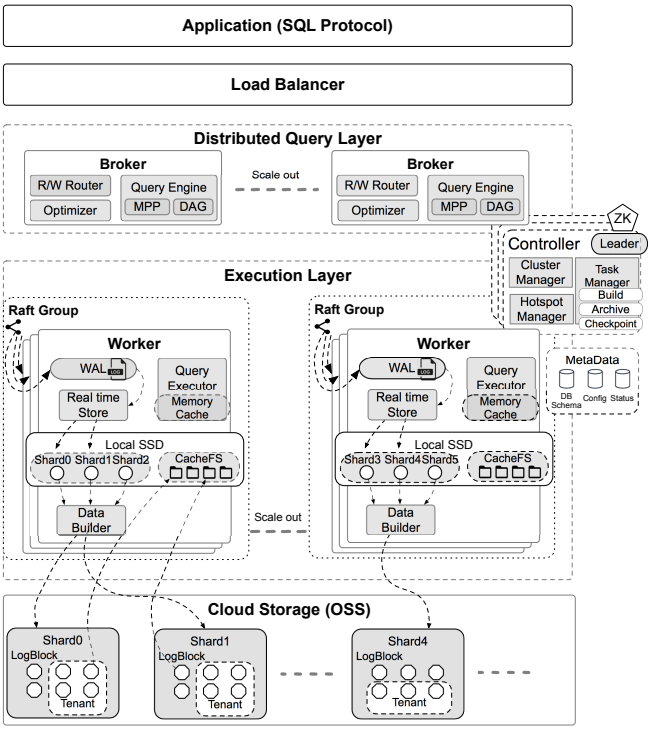


Figure 3: The *LogStore* Architecture.

Figure 3 shows the architecture of the *LogStore*. The entire system is deployed on Alibaba Cloud, using cloud services such as Server Load Balancer (SLB)[5], Elastic Compute Service (ECS)[2], Object Storage Service (OSS) [3], etc. Here is a brief introduction to each module.

Controller. The controller acts as the manager of the entire system. A three-node ZooKeeper cluster is deployed to ensure high availability of the controller service, and only one elected controller node is active. In *LogStore*, the controller is in charge of cluster monitoring, metadata management, and task scheduling. To monitor the status of various modules (including brokers and workers), it collects the runtime metrics from them. When an abnormal node is detected, the controller removes it from the router table and schedules tasks for node recovery. The controller also manages the database schema and guarantees schema consistency. When performing DDL operations, the controller will update the catalog and synchronize the changes to each broker. In addition, the controller is also responsible for scheduling background tasks periodically, such as checkpointing and cleaning up expired data.

Query Layer. The distributed query layer has a set of brokers to process requests. The SLB dispatches SQL requests from the application to the corresponding brokers. Next, the broker parses and optimizes the received request, and generates a query plan expressed as a directed acyclic graph (DAG). The broker distributes the sub-queries to the corresponding shards according to rules in the routing table, which is maintained by the controller using global traffic control algorithms, so that *LogStore* can eliminate the impact of skewed workloads or hotspots. Finally, the broker merges the responses from each shard and returns the final result to the client.

Execution Layer. The executive layer contains a group of workers. The controller assigns a number of shards to each worker. And workers are responsible for handling read and write operations of the corresponding shards. We adopt a row-column hybrid storage architecture and a two-phase writing process. The first phase is local writing. It is responsible for generating the WAL, synchronizing other replicas, and writing to local disks. In order to reduce the storage overhead of replicas, it can store only WAL on other replicas. This is a trade-off between storage cost and availability. In the production environment, we use three replicas, of which two replicas have a complete row-store, and the remaining one only contains WAL.

The second phase is remote archiving. The data builder module asynchronously converts the row-store into the column-oriented *LogBlock*s and then uploads them to OSS. Moreover, each work has a multi-layer cache to avoid frequent remote reads from OSS.

Cloud Storage Layer. OSS is a reliable and cost-effective object storage product on Alibaba Cloud. It could guarantee 99.9999999999% (12 9's) durability and 99.995% availability. OSS allows users to access objects (files) in buckets over the HTTP(s) RESTful APIs or SDKs. It is well known that object storages face some challenges in storing data, such as high latency due to additional network overhead and the inability or inefficiency to append data to the end of the file. We have solved the problems with the two-phase writing process. Each *LogBlock* is an immutable file and will no longer be modified after packing, and new *LogBlock*s will be generated for the newly arrived logs. The *LogBlock*s are uploaded to OSS in the background without affecting the write latency in the foreground.

When operating in production environments, we encountered more subtle problems, which were not fully addressed in the past works [27, 52]. For example, we found that traversing a large number of files is time-consuming when performing tasks like backup, migration, and data expiration. A *LogBlock* of a tenant is composed of a lot of small files, such as metadata, indexes, and data blocks, and all these files are packaged into a large tar file instead of using small files. The header of the tar file contains a manifest, allowing subsequent read operations to seek and read any part of the tar file. This avoids the performance penalty when dealing with many small files.

3.1 Multi-Tenant Storage

There are multiple ways to manage the storage of multi-tenant data. A trivial method is to set up a dedicated storage cluster for each tenant, as most managed Hadoop services and data warehouses are, such as Amazon EMR[7] and Redshift[35]. This method can completely isolate tenant data, but it is not economical to serve a large number of tiny tenants. The opposite direction is to store all tenant data in a large table and all tenant data is mixed and interleaved. This is inefficient for querying, managing and billing a single tenant. In particular, retrieving the data of one tenant will inevitably load the data of other tenants, resulting in additional overhead, and the high latency of object storage exacerbates this problem.

In *LogStore*, we choose a hybrid and compromise method between the two ends to manage tenant data, provide services to more customers with limited budgets, while maintaining data isolation. In the row-store, all log data is stored in a single huge table, and organized only by the timestamp, rather than separated by tenants, to improve space efficiency and reduce random I/O accesses, as shown in Figure 3. The data builder then converts data in row-store into many *LogBlock*s in the remote archiving stage. During this period, the row-store table will be divided into separated columnar tables according to tenants. If a tenant is too large due to data skew, it will be divided into multiple *LogBlock*s. Each columnar table corresponds to an OSS directory, which belongs to a tenant and contains a series of *LogBlock*s stored in chronological order. At the same time, the metadata manager in the controller will update the information of each tenant, including the path, size and timestamp range of the new *LogBlock*s. After the data expires, the task manager will issue a task to delete the expired *LogBlock*s. In this way, tenant data is isolated and stored on OSS, and can be retrieved and retired separately without affecting other data.

3.2 *LogBlock*

LogBlock is the basic and constituent unit of the log data stored on OSS. We design the read-optimized *LogBlock* according to the following design principles.

Self-contained. A *LogBlock* is a self-explanatory and location-independent data package, which contains the complete table schema, and detailed information of all columns. With this design, *LogBlock* can still be resolved after being renamed or moved.

Compressed. *LogStore* supports a series of compression algorithms, such as Snappy[33], LZ4[42], and ZSTD[30]. Because the compression ratio is preferred in *LogStore* to reduce the amount of data transmitted over the network, ZSTD is used as the default compression algorithm, which consumes more CPU than other algorithms.

Columnar-oriented. Most queries do not need to retrieve all columns, so the column-oriented format can prevent reading irrelevant columns and improve query performance.

Full-column indexed and Skippable. We create indexes for all columns, which is different from other columnar storages. This is to reduce the scan operation, which is relatively expensive on OSS. The extra space cost of the index is acceptable after using OSS. We support two types of indexes: the inverted index and BKD tree index[47], corresponding to string type and numerical type respectively. We also generate a Small Materialized Aggregates (SMA) [44] for each column, including maximum and minimum values for skipping data blocks.

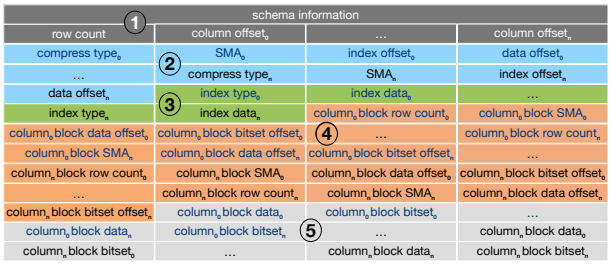


Figure 4: Layout of a *LogBlock* for n attributes

Figure 4 shows the structure of *LogBlock* in detail. *LogBlock* consists of five parts: header, column meta, index, column block header and column block. These parts are numbered 1 to 5, and marked in different colors. The header ① records table schema, the number of records in the table, and the offset of metadata of each column ②, which describes the compress algorithm used, the SMA and offset to indexes ③ and column block headers ④. ③ consists of index type (e.g. inverted index or BKD tree index) and index data. ④ contains the number of rows, the SMA and offset of the data and bitset, while the column block ⑤ stores the bitset and compressed data.

4 LOAD BALANCING

We discussed how to achieve high write throughput through the cloud-native architecture. Ideally, the maximum write throughput of *LogStore* can be estimated by summarizing the total processing capacity of the ECS nodes. However, in actual systems, the system could become unbalanced and under-loaded. Some ECS nodes become overloaded, while others remain idle. It is difficult to achieve the theoretical writing throughput in practice. Load imbalance comes from three factors:

High Skewed Workload. After hash partitioning according to tenants, the workload will be heavily biased towards "hot" shards, that is, the partitions where large tenants are located. A few tenants contribute most of the log volumes, and most long-tail tenants are in a state of low activity. For example, as shown in Figure 2, the workload of tenants in Alibaba Cloud DBaaS production environment is close to Zipfian Distribution.

Variations of Traffic. The activities of tenants will fluctuate with business. The surge in traffic, such as online promotions, will generate new hotspots dynamically.

Heterogeneity of ECS nodes. For long-running systems, the heterogeneity of computing nodes is inevitable, because the ECS node configuration provisioned on the cloud continues to evolve. The load that each ECS node can carry is different.

In the course of running *LogStore* for many years, we have realized that in the face of the above challenges, a dynamic workload-ware load balancing based on real-time feedback is the most important thing for maintaining the stability of the service. Prior work on distributed systems has considered this aspect. Some of them target factors such as the balance of storage capacity and reducing the overhead of distributed transactions. Databases that are range-partitioned, such as HBase [13], can dynamically split a partition into two when one partition becomes a hotspot. This can solve the problem of skew in the size of a single partition, but it cannot alleviate hot spots in all cases. For example, when the partition key is an incremental value, say the timestamp. After the split, the hotspot still falls on the last partition, which is still unbalanced. Schism[26] abstracts each database node as a vertex on the graph, minimizing the number of distributed transactions across shards by cutting the graph into multiple balanced partitions, but it cannot handle the hot spots of a single partition. Other shared-nothing systems use rule-based/heuristic algorithms to schedule migrations to eliminate hotspots. Yak[38] defines and monitors load metrics to detect imbalance, and a set of rules (such as load split rule and size split rule) to invoke balancing actions, namely split and move. E-store[51] presents a two-tier partitioning framework that implements a greedy algorithm to dynamically assign hot partitions to database nodes, so that the load is evenly distributed.

As a cloud-native log database, we expect a lightweight load balancing framework without data migration. On the one hand, tenant logs need to be distributed to enough shards that can handle the workload. On the other hand, in order to prevent unavailable machines from affecting the query and write latency of a large number of tenants, each tenant's workload should be assigned to as few shards as possible. This requires a weight-based balancing algorithm, which is studied in this section.

4.1 Global traffic control

In *LogStore*, if the load of a tenant is too high to be carried by one node, the tenant's load will be evenly (e.g. in a round-robin manner) distributed to multiple nodes according to routing rules, as described in Section 4.1.2. In practice, when we re-balance workload between shards, in addition to quickly alleviating the load imbalance caused by skew, it is also crucial to avoid the occurrence of new hot spots. Otherwise, new shards will be overloaded and hot spots will oscillate repeatedly. On the other hand, when the traffic of a certain tenant increases rapidly, if the system cannot respond in a timely manner, for example, balancing the tenant's load to multiple nodes will cause service degradation such as increased request delay.

In order to approach the theoretical maximum throughput, we model the load balancing problem of multi-tenant logs as the flow network[32]. The flow network is used in scenarios, such as resource allocation [36], QoS of routing protocols [20, 43, 50], but as far as we know, there is no prior work using max-flow to solve the load balancing problem in the distributed database.

The following chapters will explain the model, constraints and optimization goals, and give detailed algorithms.

4.1.1 Modeling. To better define the problem, we abstract the global traffic control into a flow network as shown in Figure 5. It is a single source/single sink flow network $G(V, E)$, where vertices represent multiple tenants, shards and worker nodes, the edge between a tenant and a shard means the shard bears traffic from the tenant, and the edge between a shard and a node means the shard is located on the node. Besides, there is an virtual source point V and sink point T. The balance problem of multi-tenant traffic is transformed into a maximum binary matching problem.

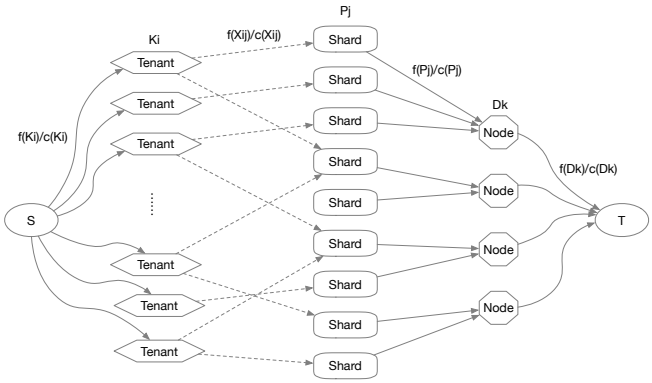


Figure 5: An example of the single source/single sink flow network in *LogStore*.

The first layer of the network is the tenants K, $\{K_i : K_0, K_1, K_2, K_3 ... K_m\}$ (m is the number of tenants). $f(K_i)$ is the real flow (traffic) of tenant K_i proceeded in the database. Then the second layer is a set of shards in the table P, $\{P_j : P_0, P_1, P_2, P_3 ... P_w\}$ (w is the number of the shards). $f(P_j)$ is real flow handled by the shard P_j, $c(P_j)$ is the capacity (maximum processable flow) of the shard P_j. X_{ij} is the proportion (weight) of the flow distributed to the shard P_j by the tenant K_i. $f(X_{ij})$ is the flow distributed to the shard P_j by tenant K_i . Formally expressed as:

$$f(X_{ij}) = X_{ij} \cdot f(K_i)$$

$$\sum_{\tau(i)}^{j} X_{ij} = 1$$

$$f(K_i) = \sum_{j=0}^{w} f(X_{ij}) = \sum_{j=0}^{w} X_{ij} \cdot f(K_i)$$

The last layer is all workers in the cluster D, $\{D_k : D_0, D_1, D_2, D_3 ... D_n\}$(n is the number of the nodes). $f(D_k)$ is real flow handled by the node D_k, $c(D_k)$ is the capacity of the node D_k. Our goal is to adjust the edges and weights (routing rules) of the graph G, while keeping the edges as few as possible, so that the maximum flow (flow) of the graph meets the required flow (actual flow).

First some constraints must be satisfied:

The Capacity Constraints

$$\forall P_j \in P, f(P_j) \leq c(P_j)$$

$$\forall D_k \in D, f(D_k) \leq \alpha \cdot c(D_k)$$

α is the high watermark for each node (e.g. 85%).

Secondly, we strive to find the best balance plan to satisfy the following conditions:

Minimize the size of routes

$$\sum_{i=0}^{m} \sum_{j=0}^{w} \{1 : X_{ij} \neq 0\}$$

Maximum the traffic from S to T

$$\sum_{i=0}^{m} f(K_i)$$

4.1.2 Architecture. Figure 6 shows how the framework handles and distributes traffic from multi-tenants. The SLB service distributes the tenant's traffic to the brokers. The controller will push a tenant routing table to the brokers, which specifies the rules (including destinations and weights) for forwarding requests from different tenants in the following form:

$$Rules\{T_0 : \{P_0 : X_{00}, P_1 : X_{01}, P_3 : X_{03}\}, T_1 : \{P_3 : X_{13}\} ...\}$$

This table is updated in real-time by the controller's hotspot scheduler, which is a core component of *LogStore* and consists of three modules: monitor, balancer and router. The monitor detects hotspots by collecting runtime traffic or load metrics of tenants, shards, and workers. Then, it calls the balancer to generate a new routing plan through the flow control algorithm. The router then executes the updated plan immediately. Next, we will describe each module in more detail.

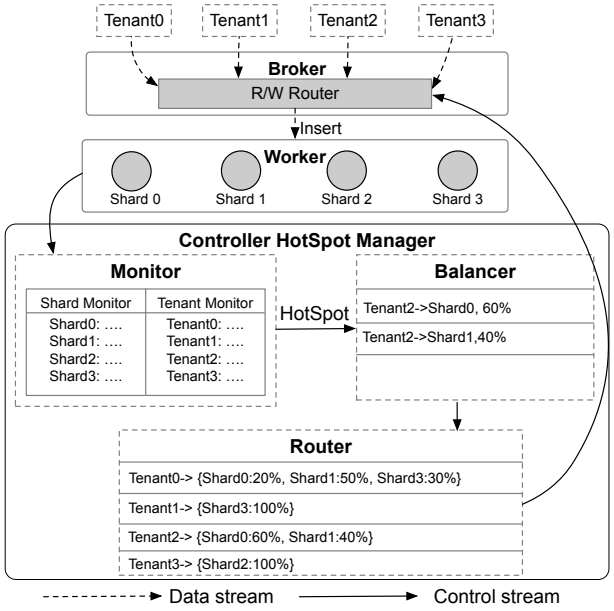


Figure 6: The framework of global traffic control in *LogStore*.

4.1.3 Monitor. Since traffic fluctuates dynamically, it is necessary to detect hot spots in real-time. In order to effectively identify hot spots, we need to find indicators to measure skewness. Therefore, timeliness and effectiveness are the keys to the monitoring module.

CPU utilization is the most commonly used indicator to identify hot spots. In our practice, skewed shards have higher CPU utilization, but the reverse is not necessarily true. More indicators are needed to improve accuracy, such as access frequency and blocked requests in the buffer queue. These indicators can reflect that the performance of the database decreases as the workload tilts.

The monitor also needs to fill in the input data (nodes and edges in $G(V, E)$) required to run the flow network algorithm. It collects tenant traffic $f(K_i)$, shard load $f(P_j)$ and worker node load $f(D_k)$. The capacity of shard $c(P_j)$ and worker $c(D_k)$ is inferred through system limits and configurations. It will detect load imbalance every 300 seconds, which can be adjusted according to the sensitivity of the business.

4.1.4 Balancer. The main function of the balancer module is to generate a new scheduling plan to balance the load or scale the cluster when hot spots are detected, and then send the updated routing table to the workers.

The Framework (Algorithm 1) The outermost layer is a loop that does not terminate until the service stops (line 8 to 30). For every 300 seconds, the traffic control logic is executed to determine whether there are any shards that need to be offloaded under heavy pressure. If $\sum_{k=0}^{n} f(D_k) \leq \alpha \cdot \sum_{k=0}^{n} c(D_k)$, then execute $TrafficSchedule()$ algorithms to adaptively rebalance the tenant traffic among all workers (line 17 to 21). Other wise, it means that the upper limit of the system has been reached, and only working nodes can be added to meet the traffic demand (line 25).

Greedy Algorithm(Algorithm 2) This naive approach intuitively identifies the set of tenants with the largest traffic on the overloaded node, splits and distributes their traffic to the least loaded shards in the cluster. Γ_{P_j} is a collection of tenants contributing traffic on shard P_j, and the tenant with the largest traffic in Γ_{P_j} is chosen and put into K_{hot} (line 2 to 4). $CalculateAddRoutesNum(K_i)$ obtains the number of edges (tenants to shards) to be added by dividing the total tenant traffic by the upper limit of processing one tenant traffic on each shard (line 6). For example, a tenant has total traffic of 500K log entries per second, and one shard is limited to process up to 100K logs belonging to the same tenant. Thus at least 5 shards are needed, and new shards are added correspondingly. $GreedyFindLeastLoad(P)$ finds the least loaded shards in the cluster (line 10). After adding shards to a tenant, the tenant's traffic will be evenly distributed to all shards by averaging their weights (line 16 to 19).

Max-Flow Algorithm(Algorithm 3) $MaxFlowAlgorithm(G)$ calculates the maximum flow of the deterministic graph $G(V, E)$ using Dinic's algorithm[29] (line 8). If $\sum_{i=0}^{m} f(K_i) > F_{max}$ (line 9), it means that under the current topology, there is no way to make the maximum achievable flow of the graph meet the actual flow. More edges should be added to $G(V, E)$ to enlarge F_{max} (line 13 to 14). Otherwise, it just updates the weights X_{ij} from the maximum flow algorithm without adding edges (line 22 to 25). When it needs to change the topology, similar to the Greedy Algorithm, it selects the shard that is least loaded in the cluster (line 13). It does not terminate the iteration of edge addition until the condition is met (line 9).

4.1.5 Router. The router module uses the new balance plan generated by the balancer module to update the routing table. Then, the router will update the routing table on each broker in a transaction. After all brokers receive the new routing table, they will switch to distribute tenant traffic according to the new policy. After receiving the new routing plan, the router module will merge the existing global routing table with the new plan, because the tenant's read request needs to be forwarded to the nodes in both old and new plans within a period of time. If after rebalancing, a node no longer carries the traffic of a certain tenant, it will not migrate data between nodes like other shared-nothing database systems (e.g. MongoDB[45], HBase[13]). Instead, the tenant data will be packaged and flushed to OSS. This helps to reduce node load in the case of system hotspots.

Algorithm 1 Global Traffic Control Framework

```
1:  c(P) ← the capacity of all shards
2:  c(D) ← the capacity of all worker nodes
3:  R ← ∅ Route table, initially empty
4:  for K_i ← K do
5:      P_j ← ConsistentHash(K_i)
6:      X_ij ← 100%
7:  end for
8:  while Service On do
9:      f(K), f(P), f(D) ← collect the traffic metrics of the tenants,
        shards, workers
10:     P_hot ← ∅ the set of hot shards detected
11:     for P_j ← P do
12:         if CheckHotSpot(P_j) then
13:             P_hot ← P_hot ∪ P_j
14:         end if
15:     end for
16:     if P_hot ≠ ∅ then
17:         if Σ_{k=0}^{n} f(D_k) ≤ α · Σ_{k=0}^{n} c(D_k) then
18:             ; rebalance traffic between workers
19:             ; and update the route table R
20:             TrafficSchedule()
21:             NotifyWorkers(R)
22:             Sleep(300)
23:         else
24:             ; add more worker nodes
25:             ScaleCluster()
26:             c(P) ← add new shards
27:             c(D) ← add new workers
28:         end if
29:     end if
30: end while
```

4.2 Backpressure

Rebalancing and scaling are proactive approaches to deal with workload hotspots and surges. However, in some extreme cases, the memory can run out quickly or the CPU will saturate. Before these methods can take effect, the system will crash. Therefore, we have implemented a backpressure flow control (BFC) mechanism (Figure 7) to protect system availability under extreme load.

BFC was first proposed and applied in streaming computing systems, such as Heron[39] and Flink[23]. When a sudden spike occurs, it can effectively control the workload of the system by

Algorithm 2 Greedy Algorithm Version of TrafficSchedule()

$K_{hot} \leftarrow \emptyset$ the set of hot tenants found out
for $P_j \leftarrow P_{hot}$ **do**
  $K_{hot} \leftarrow K_{hot} \cup PickHotSpotTenant(\Gamma_{P_j})$
end for
for $K_i \leftarrow K_{hot}$ **do**
  $(N_{add}, N_{total}) = CalculateAddRoutesNum(K_i)$

  ; first update edges
  while $N_{add} > 0$ **do**
    $P_l = GreedyFindLeastLoad(P)$
    $R.put(K_i, \{P_l, 0\})$
    $N_{add} \leftarrow N_{add} - 1$
  end while

  ; then update weights
  $weight = \frac{1}{N_{total}}$
  for $P_k \leftarrow R.get(K_i)$ **do**
    $R.put(K_i, \{P_k, weight\})$
  end for
end for

Algorithm 3 Max Flow Algorithm Version of TrafficSchedule()

F_{max} is the max flow in the graph $G(V, E)$
f_{max} is the edge max flow
$K_{hot} \leftarrow \emptyset$ the set of hot tenants found out
for $P_j \leftarrow P_{hot}$ **do**
  $K_{hot} \leftarrow K_{hot} \cup PickHotSpotTenant(\Gamma_{P_j})$
end for

$F_{max} = MaxFlowAlgorithm(G)$
while $\sum_{i=0}^{m} f(K_i) > F_{max}$ **do**
  ; add a new edge for each unsatisfied hot tenant in G
  for $K_i \leftarrow K_{hot}$ **do**
    if $f(K_i) > \sum_{j=0}^{w} f(X_{ij})$ **then**
      $P_l = GreedyFindLeastLoad(P)$
      $G.addEdge(K_i, P_l)$
    end if
  end for
  ; recalculate the maximum flow and weights
  $F_{max} = MaxFlowAlgorithm(G)$
end while

; set up weight
for K_i, P_j in R **do**
  $X_{ij} = \frac{f_{max}(X_{ij})}{f_{max}(K_i)}$
  $R.put(K_i, \{P_j : X_{ij}\})$
end for

reducing the source's consumption rate and the data transmission speed. In *LogStore*, there are many buffer queues in the system to receive and send messages asynchronously between components and interact with the network, disks and external components such as OSS. BFC works mainly based on monitoring these buffer queues. For each queue, we monitor both the number and size of

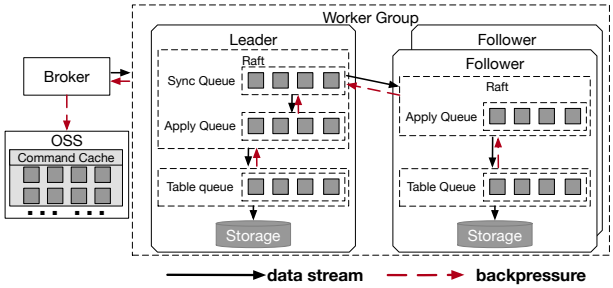


Figure 7: A demonstration of Backpressure Flow Control(BFC) mechanism.

pending requests, because for example, processing a small number of massive inputs can also cause the system to overload. When the monitoring metrics of a queue exceed the limit of the queue, the BFC will be triggered to reject writings which are delivered to the queue. BFC will gradually limit the productivity of upstream messages, and eventually limit the write throughput of requests issued by the client.

Furthermore, we integrate BFC into the Raft protocol [46] and proposed a Raft implementation with BFC. The Raft protocol has two blocking points to wait for I/O completion. One is the process of synchronizing WAL. After the leader receives the client's request, the synchronization can only be completed after most of the followers have persisted the WAL. The other is the process of applying WAL, and then the worker actually writes the data to local storage. So we added two buffer queues correspondingly, namely *sync_queue* and *apply_queue*. In this way, when a tenant's write rate is too high, causing synchronization between multiple nodes to be slowed down, the back pressure will take effect, reducing the tenant's write rate, and avoiding the explosion of nodes' internal queues, which will cause the nodes to gradually become unresponsive.

5 QUERY OPTIMIZATION

In the cloud-native architecture used by *LogStore*, log data is mainly stored on OSS. Compared with local disks, the query latency will increase due to network overhead. This problem can be alleviated from two aspects: accurate data retrieval to fetch as little unrelated data as possible, and effective caching strategies to avoid repeated loading of the same data block. We will discuss in detail two key technologies:

- A data skipping strategy to filter irrelevant data blocks.
- A multi-level data cache mechanism to cache data block downloaded from cloud storage and a parallel prefetch method to speed up data block loading when cache misses.

5.1 Data Skipping

Based on the *LogBlock* structure, a multi-level data skip strategy is proposed to filter irrelevant data blocks. It can filter data by *LogBlock* through *LogBlock* map, and filter data by column and column block through SMA. Unlike ORC[16], Parquet[17] and CarbonData[10], in addition to column statistics, indexes are also used during data skipping. An example is given in Figure 8.

This is a sample SQL for log retrieval:

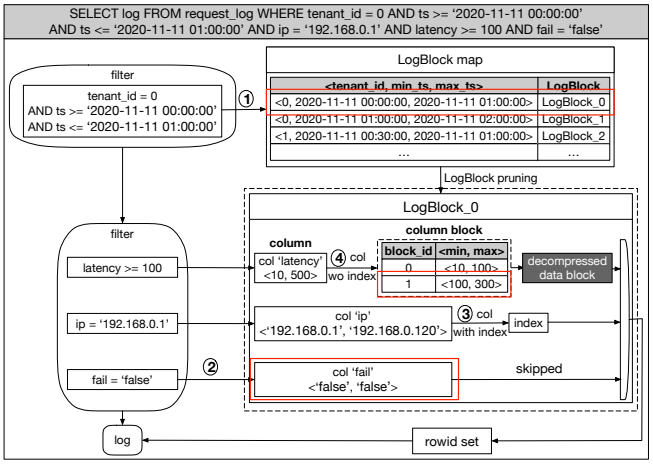


Figure 8: An example of data skipping strategy.

```
SELECT log FROM request_log WHERE tenant_id = 12276
AND ts >= '2020-11-11 00:00:00'
AND ts <= '2020-11-11 01:00:00'
AND ip = '192.168.0.1' AND latency >= 100 ANd fail = 'false'
```

① shows how to filter and skip *LogBlock*s through *LogBlock* map based on the filter conditions of column *tenant_id* and *ts*. Inside a *LogBlock*, columns and column blocks can be skipped through the filter conditions of other columns. The entire column can be filtered using the <*min*, *max*> statistics of each column (*fail* in this example), as shown in step ②. For column blocks that cannot be filtered by the predicate on the columns (*ip* in this example), the *rowid* of all rows that meet the predicate will be collected by fetching and looking up the index. as shown in ③. For columns that have not been indexed (such as *latency*), the entire column block (e.g. block 0) can be filtered using the column block's <*min*, *max*> statistics, as shown in step ④. In the remaining column blocks, the matching *rowid* will be collected by decompressing and sequentially scanning each column block. After merging the *rowid* set that meets the filter conditions, the log data can be finally loaded according to it.

5.2 Cache and Parallel Prefetch Strategy

In the query process, some data needs to be repeatedly accessed, such as metadata files, index files, and hot data files. If these files are waiting to be loaded from OSS on the critical path during query execution, the query performance will be greatly reduced due to high network latency.

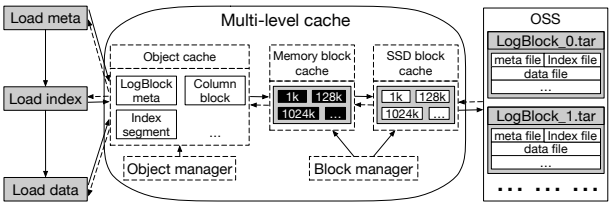


Figure 9: A demonstration of multi-level data cache mechanism.

First, we discussed the details of data loading during the query process. Figure 9 represents the loading process of a *LogBlock*, including metadata, index and data files. In order to avoid managing many small objects and files on OSS, we pack various files in each *LogBlock* into one large tar file and write them as a whole. However, in order to avoid reading irrelevant data, we allow each small file to be sought and loaded separately. Besides, we have introduced a file cache on the system. We put each file block loaded from OSS into the memory block cache (8GB). When its size exceeds the threshold, the memory cache will spill to the SSD block cache (200GB). The block manager is responsible for the expiration and swapping of the cache. We also added the object memory cache. The cache can effectively prevent frequent object allocations, which greatly increases the frequency of JVM GC. Object cache and file cache form a multi-level cache.

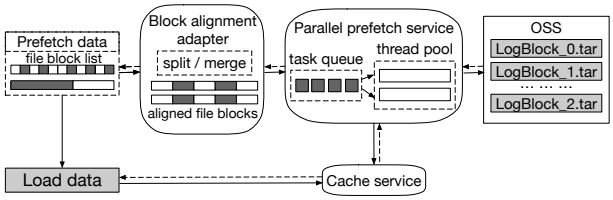


Figure 10: Parallel prefetch method workflow.

In addition, we designed a parallel prefetch strategy to avoid serial loading of data during the query process. Figure 10 shows the parallel loading process. Before parallel loading, the file to be prefetched should be divided into data blocks according to the metadata, and repeated data block read IO requests will be merged to avoid repeated loading.

6 EVALUATION

All experiments were performed on a cluster consisting of 9 ECS virtual machines (ecs.g6.8xlarge) in Alibaba Cloud. It has 32 CPU cores, 128GB RAM and 3TB SSD disk. The cluster includes 6 agents and 3 controllers, and each agent and controller process uses 6 CPU cores and 16 GB memory. There are 24 worker nodes, each node uses 8 CPU cores and 32 GB memory.

6.1 DataSets

LogStore can be used to store various types of logs, such as system logs, application logs, and IoT device logs. And we construct a sample table which stores the application logs used for the experiment. Among them, *tenant_id* and *ts* are used as the partition key to divide data into *LogBlock*s, , and indexes are created for all columns. We use Yahoo! Cloud Serving Benchmark (YCSB) framework[24] to measure the performance. The tenant logs inserted is under the Zipfian distribution controlled by the parameter θ. Here, the test data we simulated contains 1000 tenants, and the weight of tenant k is proportional to $(\frac{1}{k})^{\theta}$. When θ is higher, the workload of the tenant will be more skewed. If $\theta = 0$, then it corresponds to a uniform distribution. When the parameter is set to $\theta = 0.99$, the generated workload is similar to the highly skewed data distribution in the production environment, as shown in the figure 11.

6.2 Traffic Control

Figure 12 shows the impact of different traffic load balancing algorithms on the write throughput, the write latency (for writing a

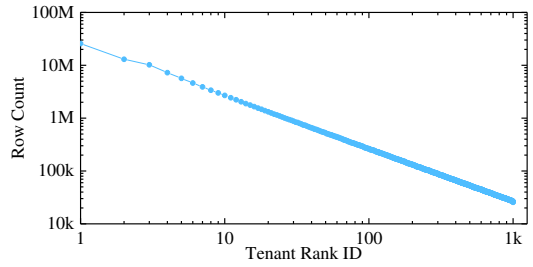
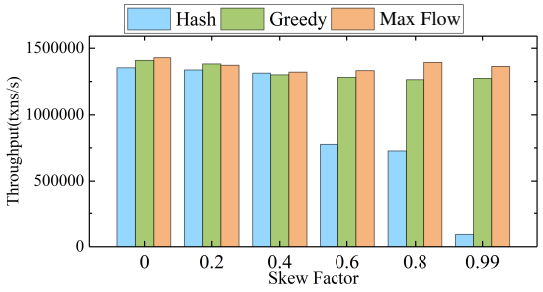
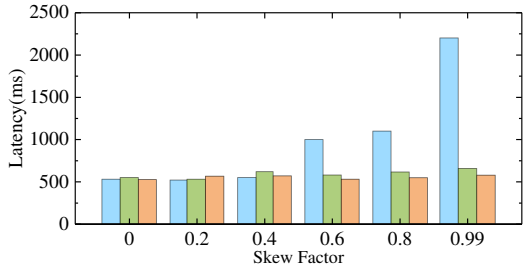


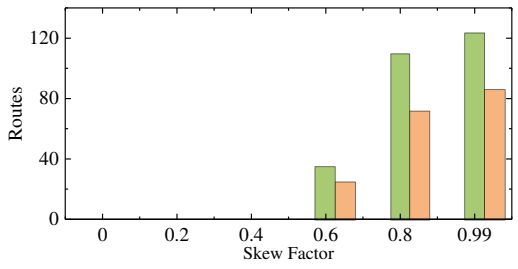
Figure 11: The tenant data distribution when $\theta = 0.99$, tenants are ranked by the number of rows.



(a) Throughput as skew factor grow



(b) Latency as skew factor grow

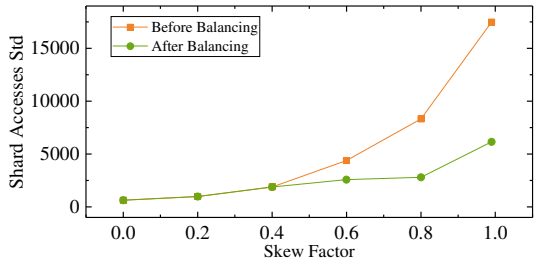


(c) Routes as skew factor grow

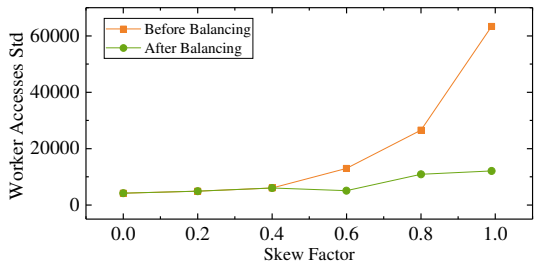
Figure 12: System Performance Under Different Balance Algorithm

batch of 1000 log entries), and the number of route rules under different degrees of data skew. As the skewness of log data increases, without flow control, throughput will drop sharply with higher latency. Especially under $\theta = 0.99$, the throughput drops to less than 1 million, and the delay is as high as 2000 milliseconds. While using the traffic control algorithms (both greedy algorithm or max flow algorithm) can keep system throughput and latency at a good performance, close to uniform distribution performance.

However, the greedy algorithm chooses the least loaded shards to carry the overflow workload, which does not consider the global system runtime state and cannot achieve the optimal solution. In addition, the greedy algorithm always adds more shards to the hot tenants to share the load, which tends to distribute the workload to more shards. In contrast, the maximum traffic algorithm generates a balanced plan that takes into account the runtime status of tenants, shards, and workers. And it tries to eliminate system hot spots by first adjusting the weights without increasing routing rules. Figure 12(c) shows that the max flow algorithm can use fewer route rules and achieve better performance under various θ configurations.



(a) Shard accesses standard deviation balanced by Max Flow algorithm.

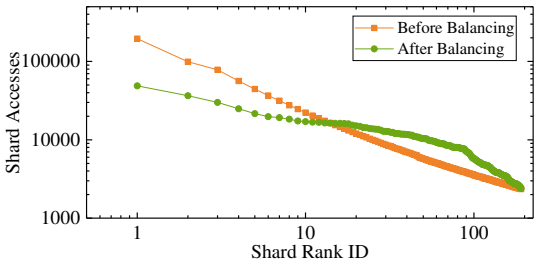


(b) Worker accesses standard deviation balanced by Max Flow algorithm.

Figure 13: Accesses standard deviation under Max Flow algorithm

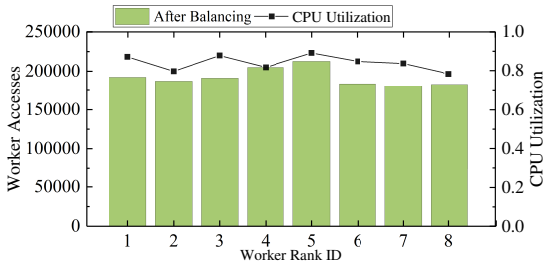
Figure 13 shows the effect of the max flow algorithm on the load imbalance between different shards and nodes. Obviously, after using the max flow algorithm, the standard deviation of the shard and worker accesses is reduced. If the skew factor is low (for example, $\theta \leq 0.4$), the standard deviation of the shard access frequency increases slightly, and the standard deviation of the worker access frequency rarely changes, which means that even without traffic control, *LogStore* can cope with the slight skew between tenants. However, when the skew factor continues to increase, the standard deviation of the shard and worker accesses begins to increase sharply without traffic control. Because the max flow algorithm considers the current load and capacity of all shards and workers to calculate the weight of the flow distribution, it can generate a balanced plan and reduce the shard accesses standard deviation by 2.8 times, and the staff accesses standard deviation by 5 times.

Figure 14 shows the running status of each shard and worker when $\theta = 0.99$. Before rebalancing, the access frequency of shards



(a) Shard accesses per second balanced by Max Flow algorithm when $\theta = 0.99$.

(b) Worker accesses per second before Max Flow algorithm balancing when $\theta = 0.99$.



(c) Worker accesses per second balanced by Max Flow algorithm when $\theta = 0.99$.

Figure 14: Detail accesses per second under Max Flow algorithm

is approximately Zipfian distribution. After the traffic control is turned on, the shard access is significantly reduced. The workload of workers is almost balanced, and the CPU utilization of all workers is close to α (85%). If the max flow algorithm finds that the system is about to be overloaded, the controller will scale the cluster or trigger the backpressure mechanism to protect the system.

6.3 Query Performance

Here we estimate the effect of the optimizations in Section 5. First, we evaluate each optimization separately to demonstrate the performance improvement of each strategy independently. Then, we test with all optimization enabled on a mixed workload that simulates online log retrieval queries to verify that the comprehensive query performance of our system can meet the requirements.

When designing the experimental cases, we select the most commonly used query template in practice, retrieve the logs of a single tenant within a specific time range, and add various filter conditions for each field. By adjusting different time ranges and query conditions, we can generate queries with different scenarios.

In order to simulate the online environment, we used the benchmarking tool to generate test data with a history of 48 hours for 1000 tenants (with the skew parameter $\theta = 0.99$). The skewed test data help observe the impact of different data sizes of tenants on query performance when evaluating various optimizations. Our query set contains 6000 queries, and six queries with different filtering predicates are generated for each tenant.

6.3.1 Data Skipping. Since the data of tenants are under the Zipfian distribution, we only display the latency statistics of the top 100 tenants in Figure 15 and Figure 16, and the difference between the rest 900 tenants is small.

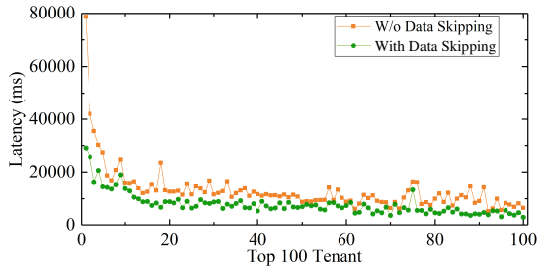


Figure 15: Impact of data skipping strategy on query latency.

As shown in Figure 15, after enabling the data skipping strategy, the average query latency has improved by 1.7 times. The largest tenant has the most significant improvement, reaching 2.6 times. Because the overhead of loading the index also needs to be considered, when the amount of data is relatively small, the performance improvement is not significant. All in all, with more data, data skipping strategies improve performance more effectively.

6.3.2 Parallel Prefetch Strategy & Multi-level Cache. Figure 16 shows the experimental results of the parallel prefetch strategy. For comparison, we also studied the query performance when the data is stored locally. Next, we focus on the query performance in three cases: storing data in local storage, using the parallel prefetch strategy (using 32 threads) and storing data on OSS, storing data on OSS without the parallel prefetch strategy.

Figure 16: Impact of parallel prefetch method on query latency.

Without the parallel prefetching strategy, the query performance of local storage will be 18.5 times faster than the data on OSS. However, when using the parallel prefetch strategy, the gap with local

storage is reduced to only 6 times. It shows that the parallel prefetch strategy can effectively narrow the performance gap between OSS access and local storage access. In addition, the multi-layer cache module caches data loaded from OSS. When the data is revisited, the data will be read directly from the local cache, which means that when the same query is executed the second time, it will be 6 times faster than the first time.

6.3.3 Overall Performance. Finally, we examine the query performance by comparing the performance before and after enabling all optimizations under a real query workload.

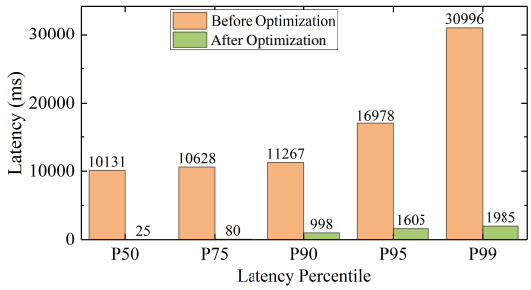


Figure 17: Effect of all query optimization methods.

In Figure 17, before enabling optimizations, queries with a delay of more than 10 seconds accounted for more than 50%, and even 1% of queries took more than 30 seconds, which is unacceptable for users. In contrast, after enabling optimizations, 99% of the queries return data within 2 seconds. In addition, 90% of the queries are returned within 1 second, while more than 75% of the queries are returned within 100 milliseconds. The results show that the above optimization strategies significantly improve performance.

7 RELATED WORK

Traditional row-oriented RDBMS can also be used for log storage, but has limited data capacity and scalability. NoSQL databases (such as MongoDB[45] and HBase[13, 31]) were proposed to solve the problems of scalability and schema flexibility. And they need additional plug-ins (such as sphinx) to implement full-text retrieval features. As a dedicated log storage, ElasticSearch[12] has powerful full-text retrieval capabilities. But due to its weak write performance and tenant isolation, it is more suitable for single-tenant log storage. Big data systems, such as Hive[14], Impala[21], Presto[53] only provide offline log storage and query functions.

C-Store[49], Vertica[40] proposed a hybrid architecture with a write-optimized store and read-optimized store. We also implemented a lightweight real-time write-optimized store to undertake high write throughput, then converted it into indexed and compressed data structures. Amazon's Redshift[35] and Aurora[54] provide a high-performance data consistency design of the shared storage architecture. The shared data architecture proposed by Snowflake[27] uses cloud object storage as the primary data storage. We refered to Snowflake to design our cloud storage layer to meet the needs of a scalable and low-cost log storage.

Several multi-tenant storage techniques have been proposed to consolidate multiple tenants on shared servers. Schism[26] is a shared-nothing distributed database, and it introduces a graph-partitioning algorithm to minimize the number of distributed transactions. E-store[51] balances tuple accesses across a set of elastic partitions based on heuristics. However, the reprovisioning of these systems needs data migration. Pinot[37] implements a random greedy strategy by taking a random subset of servers and adding servers. Inspired by the balance of hot tuples, we proposed a balance strategy on multi-tenant traffic.

The columnar storage format, ORC[16], Parquet[17] and CarbonData [10] are widely used in the OLAP and big data systems. They organize tuples in a compressed columnar format with row groups, see PAX[1]. DB2 BLU[48] further introduces a frequency-based dictionary compression method, allowing most SQL operations to be performed on compressed values. However, the above-mentioned works do not have an effective data structure or index for full-text search within a block. Data Blocks[41] implements a new lightweight index called PSMAs to improve early selection evaluation. However, the data compression of data blocks has a lower compression ratio than other works, which cannot meet our demand for low storage costs. Therefore, we use Zstd[30] compression method with high compression ratio, and add an inverted index based on Lucene[15] in *LogBlock*.

8 CONCLUSIONS

Log storage is a relatively mature technical field. Based on the Lucene index structure, extensive research has been conducted on full-text retrieval solutions. However, when using the existing systems to store logs on cloud, we encountered new problems, which have not yet been fully discussed. These issues include ultra-high throughput, multi-tenant management, tenant traffic control, scalable and low-cost storage.

In this paper, we designed a cloud-native storage architecture to achieve high throughput and low storage costs. When faced with high data skew and traffic fluctuations, we proposed a global traffic balance algorithm to keep the write traffic stable. At the same time, we implemented a column format, *LogBlock*, and data skip strategy, and parallel prefetch mechanism to reduce the delay caused by the network overhead of cloud storage. Finally, experimental data reveals that when using the max flow algorithm, hot spots can be eliminated with less routing rules added. Meanwhile, it shows that although the query latency is inevitably increased due to cloud storage, we can still control the latency within an acceptable range through various optimization strategies.

In future work, we will try to use vectorized query execution and "just in time" (JIT) query compilation to improve execution performance. In addition, we will focus on improving query performance by optimizing the data structure of the real-time store.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their valuable comments and helpful suggestions. We would also like to take this opportunity to thank Yucong Wang, Shuai Zhao, Yue Pan, Baofeng Zhang, Hao Chen, Wenbo Zheng and Chao Jiang for their contributions to LogStore.

REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.
[2] Alibaba Cloud. Ecs. https://www.alibabacloud.com/zh/product/ecs.
[3] Alibaba Cloud. OSS. https://www.alibabacloud.com/product/oss.
[4] Alibaba Cloud. PolarDB. https://www.alibabacloud.com/products/apsaradb-for-polardb.
[5] Alibaba Cloud. SLB. https://www.alibabacloud.com/product/server-load-balancer.
[6] Alibaba Cloud. SLS. https://www.alibabacloud.com/product/log-service.
[7] Amazon. EMR. https://www.amazonaws.cn/en/elasticmapreduce/.
[8] Amazon Web Services. CloudWatch. https://aws.amazon.com/cloudwatch/.
[9] Amazon Web Services. S3. https://aws.amazon.com/s3/.
[10] Apache. CarbonData. https://github.com/apache/carbondata.
[11] Apache. Cassandra. http://cassandra.apache.org/.
[12] Apache. Elasticsearch. https://www.elastic.co/elastic-stack.
[13] Apache. HBase. https://hbase.apache.org/.
[14] Apache. Hive. https://hive.apache.org/.
[15] Apache. Lucene. https://lucene.apache.org/.
[16] Apache. Orc. https://www.orc.org/.
[17] Apache. Parquet. https://parquet.apache.org/.
[18] Apache. Kafka. https://kafka.apache.org/, 2011.
[19] Azure. Hyperscale. https://docs.microsoft.com/en-us/azure/azure-sql/database/service-tier-hyperscale.
[20] S. Bhosale and N. Sarwade. Maximum flow based load balanced routing protocol for wdm networks. *European Journal of Scientific Research*, 56(3):364–375, 2011.
[21] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. Kuff, D. K. A. Leblang, N. Robinson, D. R. S. Rus, J. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th biennial conference on innovative data systems research*, 2015.
[22] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, et al. {POLARDB} meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 29–41, 2020.
[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
[24] 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.
[25] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.
[26] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.
[27] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, pages 215–226, 2016.
[28] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.
[29] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.
[30] Facebook. zstd. https://github.com/facebook/zstd.
[31] X. Gao, V. Nachankar, and J. Qiu. Experimenting lucene index on hbase in an hpc environment. In *Proceedings of the first annual workshop on High performance computing meets databases*, pages 25–28, 2011.
[32] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.
[33] Google. Snappy. https://github.com/google/snappy.
[34] Google Cloud. Cloud Logging. https://cloud.google.com/logging.
[35] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1917–1923, 2015.
[36] M. Hadji and D. Zeghlache. Minimum cost maximum flow algorithm for dynamic resource allocation in clouds. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 876–882. IEEE, 2012.
[37] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, et al. Pinot: Realtime olap for 530 million users. In *Proceedings of the 2018 International Conference on Management of Data*, pages 583–594, 2018.
[38] M. Klems, A. Silberstein, J. Chen, M. Mortazavi, S. A. Albert, P. Narayan, A. Tumbde, and B. Cooper. The yahoo! cloud datastore load balancer. In *Proceedings of the fourth international workshop on Cloud data management*, pages 33–40, 2012.
[39] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.
[40] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173*, 2012.
[41] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326, 2016.
[42] lz4. lz4. https://github.com/lz4.
[43] D. Macone, G. Oddi, A. Palo, and V. Suraci. A dynamic load balancing algorithm for quality of service and mobility management in next generation home networks. *Telecommunication systems*, 53(3):265–283, 2013.
[44] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. 1998.
[45] MongoDB. Mongodb. https://www.mongodb.com/.
[46] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.
[47] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*, pages 46–65. Springer, 2003.
[48] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, et al. Db2 with blu acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment*, 6(11):1080–1091, 2013.
[49] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 491–518. 2018.
[50] T. H. Szymanski. Max-flow min-cost routing in a future-internet with improved qos guarantees. *IEEE transactions on communications*, 61(4):1485–1497, 2013.
[51] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.
[52] B. Vandiver, S. Prasad, P. Rana, E. Zik, A. Saeidi, P. Parimal, S. Pantela, and J. Dave. Eon mode: Bringing the vertica columnar database to the cloud. In *Proceedings of the 2018 International Conference on Management of Data*, pages 797–809, 2018.
[53] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 197–210, 2013.
[54] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.
[55] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168, 2014.