ESDB: Processing Extremely Skewed Workloads in Real-time

Jiachi Zhang[*][†][§] Shi Cheng[§] Zhihui Xue[§] Jianjun Deng[§] Cuiyun Fu[§]
Wenchao Zhou[§] Sheng Wang[§] Changcheng Chen[§] Feifei Li[§]
jz598@georgetown.edu,{chengshi.cs,zhihui.xzh,jianjun.djj,cuiyun.fcy,zwc231487,sh.wang}@alibaba-inc,tianyu@taobao.com,lifeifei@alibaba-inc.com
[§]Alibaba Group and [†]Georgetown University
[§]Hangzhou, China and [†]Washington DC, USA

ABSTRACT

With the rapid growth of cloud computing, efficient management of multi-tenant databases has become a vital challenge for cloud service providers. It is particularly important for Alibaba, which hosts a distributed multi-tenant database supporting one of the world's largest e-commerce platforms. It serves tens of millions of sellers as tenants, and supports transactions from hundreds of millions of buyers. The inherent imbalance of shopping preferences from the buyers essentially generates a drastically skewed workload on the database, which could create unpredictable hotspots and consequently large throughput decline and latency increase. In this paper, we present the architecture and implementation of ESDB (ElasticSearch Database), a cloud-native document-oriented database which has been running on Alibaba Cloud for 5 years as the main transaction database behind Alibaba's e-commerce platform. ESDB provides strong full-text search and retrieval capability, and proposes dynamic secondary hashing as the solution for processing extremely skewed workloads. We evaluate ESDB with both simulated workloads and real-world workloads, and demonstrate that ESDB significantly enhances write throughput and reduces the completion time of writes without sacrificing query throughput.

CCS CONCEPTS

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

KEYWORDS

multi-tenant; cloud-native; load balancing; document-oriented database

ACM Reference Format:
Jiachi Zhang[†][§] Shi Cheng[§] Zhihui Xue[§] Jianjun Deng[§] Cuiyun Fu[§], Wenchao Zhou[§] Sheng Wang[§] Changcheng Chen[§] Feifei Li[§]. 2022. ESDB: Processing Extremely Skewed Workloads in Real-time. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3514221.3526051

1 INTRODUCTION

With the prevalence of cloud computing, enterprises are migrating their applications as well as databases to the cloud. Nowadays, cloud vendors are supposed to provide database products for millions of customers, including applications and online services that host millions of users. This drives cloud service providers to adopt distributed multi-tenant databases where data from multiple tenants are allocated to share a same set of computation and storage resources. Such multi-tenant architecture allows for high resource utilization, and, consequently, high elasticity and low cost for database products. On the other hand, however, due to the inherent imbalance among the workloads from different tenants, multi-tenant databases can suffer from severe throughput decline when facing extremely skewed workloads.

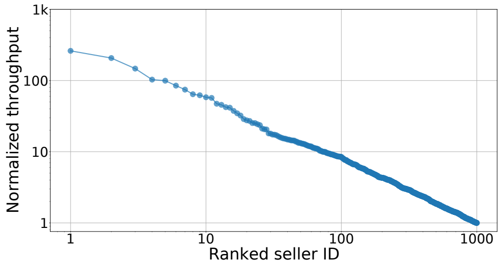


Figure 1: Normalized throughput of top 1000 sellers in the first 10 sec of Single's Day Global Shopping Festival, 2021.

Alibaba is running the world's largest e-commerce platforms, which include Taobao, TMall and others. As the backbone of Alibaba's e-commerce platforms, the transaction database is expected to provide stable read/write throughput and low response time (RT) during both ordinary times and under heavy workloads (such as during major promotion events). In Alibaba's transaction database, *transaction logs* consist of both structured data (e.g., transaction ID, seller ID, created time and transaction status) and full-text data (e.g., auction title, sellers' and buyers' nickname). Produced from e-commerce platforms, transaction logs record shopping transactions initiated by the buyers, and are then sequentially written into the sellers' database through a coordination platform. As a targeting goal, the sellers' database is dedicated to processing high workloads and providing highly efficient seller-oriented query and analysis service. Deployed on Alibaba Cloud [33], the sellers' database was initially maintained in a shared-nothing MySQL cluster [52, 53] where transaction logs were organized by their seller IDs. As the throughput of Alibaba's transaction logs has grown rapidly in recent years, we observed that the sellers' database started to face the following two challenges:

Diverse data schema and full-text search. As the transaction logs contain information of various commodities, users are allowed

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06.
https://doi.org/10.1145/3514221.3526051

[*]This author takes part in this work as a research intern at Alibaba DAMO Academy.

to add customized attributes (e.g., sizes, materials of clothes, weights of food) to the commodities, which leads to a diverse data schemas (i.e., different transaction logs have different attributes). Since it is unrealistic to add columns for all customized attributes to the table, we build an "attributes" column where all customized attributes are concatenated together as a string. In practice, over 1500 sub-attributes have been added to the transaction database. Before the adoption of ESDB, querying about "attributes" column was highly inefficient, since it is non-trivial to build a suitable index of such a non-standard string column.

In practice, sellers often find themselves in need to perform full-text search queries that contain non-standard strings or keywords. For example, bookstore sellers search transaction status of book transactions by keywords in the auction titles. Although MySQL provides fuzzy search (e.g., `LIKE`, `REGEXP`) as well as full-text search using full-text indexing, the support for these functions are limited and the performance is unstable especially with the rapid growth of the transaction logs over years. Consequently, it drives us to shift our backbone from MySQL to document-oriented databases.

Skewed and unpredictable workloads. Another challenge is that the workload distribution in the sellers' database is extremely skewed because of the tremendous variation of the numbers of transactions conducted by different sellers. The variation is further magnified at the kickoff of major sale and promotion events during which the overall throughput increases dramatically. For example, Figure 1 shows the normalized throughput of the top 1000 sellers in the first 10 seconds of the Single's Day Global Shopping Festival, 2021. In the figure, the normalized throughput roughly follows a power law curve where the aggregated throughput of the top 10 sellers comprises 14.14% of the total throughput. The unpredictability of the throughput distribution adds another layers of complexity: the throughput of different sellers fluctuates substantially over time, depending on multiple factors such as the availability of merchandise promotions, the readiness of stock preparation, etc. In practice, the popularity of sellers can change significantly in a short time period, and the peaks of top sellers are difficult to forecast.

Before the adoption of ESDB, transaction logs of each seller are uniquely assigned to a MySQL instance based on their seller IDs. In practice, when a top seller's promotion starts and brings in voluminous transactions, the write throughput can easily overwhelm the instance's write capability. On the other hand, most instances (for ordinary sellers) are deeply under-utilized. The consequence of such skewed workload is waste of resources on idle instances, failed real-time queries on hotspot instances, and an overall performance degradation. For example, the write delays (i.e., a metric that evaluates how long ESDB takes to complete a write of a transaction log into the seller's database) of large tenants could rise as high as over 100 minutes in early years, in which case, the sellers would lose the capability of adjusting their sale strategy based on the market's response. A straightforward multi-tenant solution is to allow multiple sellers to share a database instance, e.g., by routing sellers' transaction logs to instances through consistent hashing [42, 49]. However, the distribution of transaction logs is still skewed due to the inherent imbalance of transactions performed by different sellers. Therefore, we need a workload-adaptive load balancing mechanism, especially with the high fluctuation and unpredictability in the workloads.

Contributions. In this paper, we introduce *ESDB*, a cloud-native multi-tenant database which features processing extremely skewed workloads. ESDB is built upon Elasticsearch [17] and inherits its core characteristics, such as full-text search, distributed indexing and querying, and using double hashing [5] as its request routing policy. The novelties of ESDB lie in a new load balancing technology that enables workload-adaptive routing of multi-tenant workloads, and optimizations that overcome the shortcomings of Elasticsearch as a database, such as high RT for multi-column queries and high cost of index computation. More concretely, we make the following contributions of this paper:

- We present the architecture of ESDB, a cloud-native document-oriented database which has been running on Alibaba Cloud for 5 years as the main transaction database behind Alibaba's e-commerce platform. ESDB provides support for elastic writing for extremely skewed workloads, as well as highly efficient ad-hoc queries and real-time analysis.
- We introduce *dynamic secondary hashing* as the solution to the performance degradation caused by the high skewness of the workloads. This mechanism enables real-time workload balancing to allow balanced write throughput on different instances. At the same time, unlike double hashing which requires expensive distributed read to fetch results from virtually all instances, it avoids the heavy burden induced by distributed queries.
- We further introduce several optimizations, such as the adoption of a query optimizer, physical replication, frequency-based indexing. These optimizations enable ESDB to incorporate features of a document-oriented database (e.g., high scalability and strong support for full-text retrieval), while providing low RTs for ad-hoc queries.
- Finally, we evaluate ESDB both in laboratory environment with simulated workloads and in production environment with real-world workloads. The experimental results show that ESDB is able to balance write workloads in different skewness scales and achieves high query throughput and latencies. Our results show that the deployment of ESDB succeeds reducing write delays even at the spike of Single's Day Global Shopping Festival.

The remainder of this paper is organized as follows. In Section 2, we introduce as background document-oriented databases, and the general concept of load balancing. Section 3 presents the architectural overview of ESDB. We then introduce the design of dynamic second hashing in Section 4, and the optimizations in Section 5. Section 6 presents the evaluation of ESDB. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 BACKGROUND AND MOTIVATION

In this section, we introduce as background document-oriented databases, and a particular example, Elasticsearch, which is used as the basis of ESDB. We then discuss the general concept of load balancing, as well as the motivation of dynamic second hashing.

2.1 Document-oriented Database

Document-oriented database is a subclass of NoSQL database where the data is stored in form of documents. Unlike relational database, document-oriented database does not have predefined data format thus supports flexible schema. Compared to other NoSQL

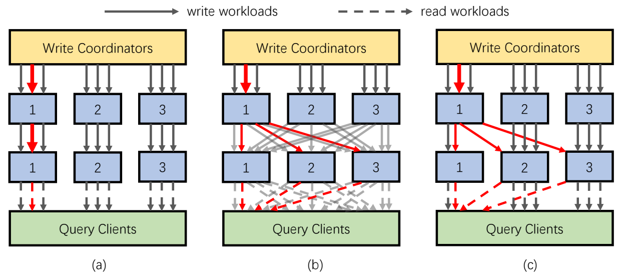


Figure 2: Workflows where the workloads originate from the write coordinators (yellow boxes) to shards (blue boxes) through double hashing and finally reach query clients (green boxes). Black arrows represent workloads of small tenants. Thick red arrows represent workloads of large tenants; they are separated into thinner arrows as the workloads are routed to multiple shards by double hashing. Figure (a), (b) and (c) depicts hashing, double hashing and dynamic secondary hashing respectively.

databases, the advantages of document-oriented databases are handling complex queries and managing composite data structures, such as nested documents. Generally, the documents are encoded in file formats, such as JSON, BSON, XML, YAML. These file formats expose their internal structures which the database engine takes advantage of in order to optimize indices and query.

Although the mainstream document-oriented databases (e.g., MongoDB [40], CouchDB [16] and Couchbase [39]) gain popularity from the industry, their performance of full-text search is unstable and cannot meet our expectation of real-time query in a large scale. Elasticsearch [17] and Solr [20] are search engines which are based on the search library Lucene [19]. As they provide operations on documents (e.g., data is stored in JSON), they are also considered as document-oriented databases. As of February 2022, Elasticsearch is arguably the most powerful and popular open-source full-text search tool [1]. In addition to the strong support for full-text search, Elasticsearch provides high horizontal scalability and availability as it is a naturally distributed system which supports low-cost cluster extension, multiple replicas, and distributed queries. This feature is also essential for load balancing because the transaction logs and workloads of large tenants are supposed to be distributed instead of uniquely allocated. Moreover, Elasticsearch implements double hashing [5] which is the basis of our load balancing method. Therefore, we choose Elasticsearch as the backbone of ESDB.

Unfortunately, we find Elasticsearch has two main drawbacks after deploying it as the sellers' database: (1) High RT for multi-column queries. When a query involves with multiple columns and complex conditions, RTs of Elasticsearch are usually higher than the original MySQL cluster. (2) High cost of index computing. On the one hand, as every shard has a replica which is distributed on a different machine, operations on Elasticsearch's index file (segment file) double the computation cost. On the other hand, in order to improve query efficiency, we are supposed to build indices for the sub-attributes of the "attributes" column of transaction logs. It will cause unacceptable computation and storage considering the number of sub-attributes is around 1500. In Section 3 and 5, we introduce our solutions to these two problems.

2.2 Load Balancing

Typical load balancing methods include sharding and double hashing. Sharding is a horizontal partitioning method which distributes large tables across multiple machines by a shard key (e.g., time, region, seller ID). Through distributing storage as well as workloads, sharding alleviates heavy burden on single machine. In addition, some well-adopted databases (e.g., Spanner [28], MongoDB [40], ONDB [54], Couchbase [39]) implement automatic resharding which automatically reshards and migrates data across machines in order to balance workloads. HBase [18] uses an auto-augmentation reshard method which splits one shard into two shards once the shard grows too large. Yak [44] uses a rule-based reshard method which uses manually defined split rules for workload schedule and data migration based on metrics detected by a global monitor. Although optimal sharding strategies are likely to guarantee load balancing, high cost and risk caused by data migration cannot satisfy the requirement of real-time processing and volatile workloads.

Double hashing [35, 48] is a classic algorithm proposed to resolve hash collision in a hash table, in which a pair-wise independent secondary hash function is introduced to produce an offset in case the first hash function causes collision. In some applications [30, 43, 56], two independent hash functions are applied to one key. Particularly, the double hashing result of a key k is $p = (h_1(k) + ih_2(k)) \bmod N$ where i increases as the sequence of collision grows and N is the size of hash table.

When databases (e.g., Elasticsearch [17], HBase [18], OceanBase [34] , OpenTSDB [60]) take advantage of double hashing for more balanced routing, it commonly includes two independent hash functions separately applied on two attributes. For example, Elasticsearch implements a double hashing module which allows users to choose two attributes k_1 and k_2 and a global static variable s. The corresponding routing destination is

$$p = (h_1(k_1) + h_2(k_2) \bmod s) \bmod N \tag{1}$$

where k_1 is the main attribute for load balancing (e.g., tenant ID), k_2 is a key uniquely generated for each transaction (e.g., transaction ID), $1 \le s \le N$ and $s \in \mathbb{Z}$.

In Equation 1, the maximum offset s plays an important role in the trade-off between load balancing and query efficiency. When $s = 1$, double hashing degrades to hashing which fails in workload balancing but enables queries on single shard (Figure 2 (a)). On the contrary, when $s = N$, double hashing routes records to all distributed shards regardless of whether they are hotspots or not. It enables complete balanced workloads of all tenants but requires query executions to search and aggregate query results from all distributed shards thus causes low query throughput (Figure 2 (b)). Some operations, such as sort and top-k, are much more time-consuming once the data is stored in a distributed manner. In this paper, we fulfill the goal of supporting both load balancing and high query throughput by replacing the static s with a function which dynamically changes with respect to the real-time workload of a tenant. We present the detailed design in Section 4.

3 SYSTEM OVERVIEW

ESDB is a distributed database system deployed on Alibaba Cloud. It adopts a shared-nothing architecture where each database node (i.e., physical or virtual machines) stores the allocated shards and

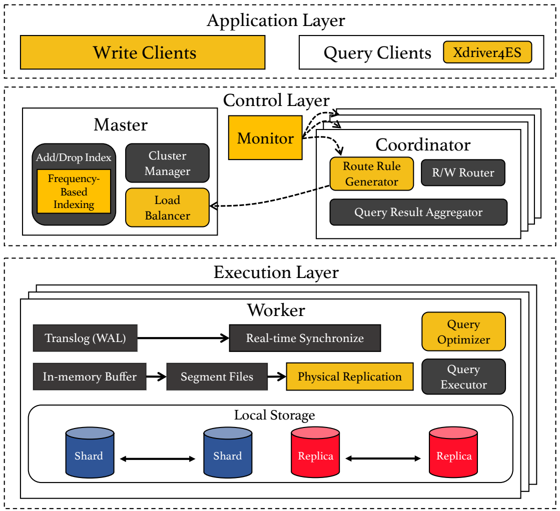


Figure 3: Architecture of ESDB.

processes workloads independently. In an ESDB cluster, shards and replicas (each shard has one replica) are randomly allocated to different nodes. Nodes play different roles: each node works as both a coordinator (on control layer) and a worker (on execution layer); each cluster further elects one node as the master node.

Figure 3 shows an architectural overview of ESDB. In addition to the three layers of workload processing, this figure also includes ESDB's key features that enable real-time workload balancing and read/write optimizations. The rest of this section, we introduce these components in more details.

3.1 Application layer

The Application layer consists of ESDB's write clients and query clients. We briefly introduce these two kinds of clients:

Write clients. Accompanied by ESDB's load balancer, ESDB's write clients use the following techniques to accelerate writing and alleviate hotspots. (1) One-hop routing. The original write clients of Elasticsearch are transport clients which are unaware of workloads' destinations [12]. These transport clients route workloads to coordinators in a round-robin fashion which leads to two-hop routing (write client → coordinator → worker). In ESDB, we allow the transport clients to be aware of the routing policies. In this way, we achieve one-hop routing (write client → worker) and thus accelerate writing. (2) Hotspot isolation. In write clients, all workloads are temporally buffered in a queue before they are routed to their corresponding workers batch by batch. Once a worker is overloaded (probably caused by both skewed write workloads and slow read queries), the queue will be blocked and the write delay will rise. In order to solve this problem, ESDB implements hotspot isolation which isolates workloads of hotspots to another queue, such that they will not negatively affect other workloads. (3) Workload batching. When a write client detects that a row (identified by its row ID) will be frequently modified in a short period of time, it will batch-execute the workloads by aggregating together these modifications and only materializing the eventual state of this row. By adopting the workload batching, the write clients avoid the repeated writes to the same row and thus improve write throughput.

Query clients. ESDB inherits the RESTful API and the query language ES-DSL from Elasticsearch. However, ES-DSL is less user-friendly compared to SQL, and it does not support all expressions and data types of SQL (e.g., type conversion expression `date_format`). We therefore need a tool to rewrite SQL queries into ES-DSL queries. In order to solve this problem, we develop a plugin Xdriver4ES as a bridge between SQL and ES-DSL.

Xdriver4ES adopts a smart translator which generates cost-effective ES-DSL queries from SQL queries. Unlike SQL, ES-DSL encodes query ASTs directly which are then parsed to generate execution plans. Therefore, instead of building ASTs from SQL queries, Xdriver4ES adopts the following two optimization techniques: (1) CNF/DNF conversion. Considering queries as boolean formulas, Xdriver4ES converts them into CNF/DNFs in order to reduce the depth of ASTs. (2) Predicate merge. Xdriver4ES merges predicates that involve the same column in order to reduce the width of ASTs (e.g., merge `tenant_id=1 OR tenant_id=2` to `tenant_id IN (1,2)`). Xdriver4ES further utilizes a mapping module which converts the query results into a format that a SQL engine understands. For example, we implement in this module built-in functions of SQL, such as data type conversion and `IFNULL`. In this way, Xdriver4ES allows the execution of SQL queries on ESDB.

3.2 Control layer

The control layer consists of a master node, coordinator nodes and a monitor which collects metrics for workload balancing. In an ESDB cluster, master node works as the manager of the whole cluster. It is responsible for cluster management tasks such as managing metadata, shard allocation and rebalance, and tracking and monitoring nodes. The coordinator nodes are mainly responsible to route read/write workloads to the corresponding worker nodes [8]. Specifically, during the query execution phrase, coordinators first collect row IDs of the selected rows from all involved shards, and then fetch the corresponding raw data (indicated by the retrieved row ID list). Therefore, coordinators contain a query result aggregator that is in charge of row ID collection and perform aggregation operations (e.g. global sort, sum, avg).

Load balancer. ESDB's load balancer is developed to generate and commit routing rules which instruct the reader/write routers. Since it is enabled by dynamic secondary hashing, the routing rules are also called secondary hashing rules. Once the monitor detects hotspots, the coordinators will generate new routing rules that essentially split and extend the shards that are hosting the current hotspots. Through exploiting new routing rules, ESDB distributes workloads of large tenants to more shards thus alleviates the over-burdened nodes. Once it receives new routing rules from coordinators, the master node is then responsible to commit these rules. In this way, the load balancer ensures the consistency of workload routing in the whole cluster. We present more design details of the load balancer in Section 4.

Frequency-based indexing. In order to mitigate the cost of maintaining indices of the sub-attributes of the "attribute" column (see Section 2.1), we adopt frequency-based indexing (i.e., build indices only for the most frequently queried sub-attributes). This idea is derived from the observation that sub-attributes' read/write frequencies are skewed. For example, some generic sub-attributes, such as "activity" that indicates what e-commerce activity the seller is participating in, are frequently used and likely to be queried. In practice, frequency-based indexing provides significant improvement in query latency with the cost of a slightly increased storage overhead. We demonstrate the effectiveness of frequent-based indexing in Section 6.3.3.

3.3 Execution layer

The execution layer consists of worker nodes which maintain shards and replicas in their local SSD, and execute read/write workloads. The worker nodes have a shared-nothing architecture where each worker maintains its own storage, independent to other workers.

Write execution and data replication. When executing write workloads, ESDB inherents the feature of near real-time search [7] from Elasticsearch. Raw data and indices are temporally written into an in-memory buffer before they are periodically refreshed to segment files and become available to search. In order to address persistence and durability, ESDB adopts Translog [11] which pertains to the disk. Every write workload will be added to the Translog once it is successfully submitted. In this way, the data that has not been flushed [3] to the disk can be safely recovered from Translogs, if ESDB encounters process crash or hardware failures. Moreover, segment merge [9] is another important mechanism that merges smaller segments to a large segment. It costs computation resources but effectively improves query efficiency.

Elasticsearch adopts a logical replication scheme [4]: a primary shard will forward a write workload to all its replicas once this workload has been locally executed. In other word, same write workload is executed respectively by the replicas, which causes n-fold computation overhead to the cluster (n is the number of replicas). In ESDB, write workloads are still forwarded and added to Translogs on replicas in real-time, but are never executed by replicas. Instead, it implements physical replication of segment files. We describe the details of physical replication in Section 5.2.

Query optimizer. For query workloads, ESDB builds optimized query plans using a rule-based optimizer. Instead of using index scan for all columns, ESDB provides more operations such as composite index scan and sequential scan. Through exploiting combinations of different operations, ESDB significantly reduces query latencies. Details of optimizations are presented in Section 5.

4 LOAD BALANCING

The load balancer of ESDB is designed to meet the following two requirements: (1) Query efficiency. Data of multiple tenants should be placed on as few shards as possible in order to avoid query executions across too many shards. (2) Load balancing. Distribution of workloads across multiple shards should be as uniform as possible in order to avoid overload on a single shard. In practice, we make a trade-off between these two contradictory requirements by limiting data of small tenants on single shard and distributing data of large tenants across multiple shards (see Figure 2 (c)).

4.1 Dynamic Secondary Hashing

The key intuition of dynamic secondary hashing is to adopt a workload-adaptive offset function $L(k_1)$ in the secondary hashing period of double hashing. Compared to Equation 1, the fixed maximum offset s is replaced with $L(k_1)$:

$$p = (h_1(k_1) + h_2(k_2) \bmod L(k_1)) \bmod N \qquad (2)$$

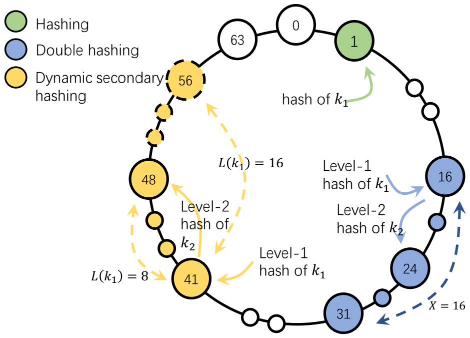


Figure 4: Comparison of workload routing across 64 shards between hashing, double hashing and dynamic secondary hashing. Colored nodes represent the shards to which workloads of attribute k_1 are routed.

where $1 \leq L(k_1) \leq N$, $L(k_1) \in \mathbb{Z}$ and $L(k_1)$ depends on the real-time workloads of tenant k_1.

Figure 4 depicts three different routing policies: hashing, double hashing and dynamic secondary hashing. Apparently, hashing can only route a workload based on the 1-level hash of the partition key (e.g., tenant ID) thus falls short when it needs to balance multi-tenant workloads. The ordinary double hashing is able to route workloads of a tenant to a *fixed* set of consecutive shards based on 2-level hashing of two keys (i.e., tenant ID and record ID) but fails to manage dynamic workloads. The dynamic secondary hashing is inspired by double hashing, which also route workloads to consecutive shards. However, it is capable of extending to the successive shards when the workloads increase (e.g., $L(k_1)$ increases from 8 to 16 in Figure 4).

Algorithm 1 ESDB load balancer

```
$K \leftarrow$ collect the set of tenants
$P \leftarrow$ collect the set of shards
$R \leftarrow \emptyset$                    ▷ initialize secondary hashing rule list
$t_{init} \leftarrow$ select effective time
$S(K) \leftarrow$ collect current storage of each $k$
for each $k$ in $K$ do
    $r \leftarrow \frac{S(k)}{\sum_{k \in K} S(k)}$
    $s \leftarrow ComputeOffsetSize(r)$
    $R \leftarrow UpdateRuleList(R, t_{init}, s, k)$
end for
while Service on do
    $t_{update} \leftarrow$ select effective time
    $T(K) \leftarrow$ collect periodic write throughput of each $k$
    for each $k$ in $K$ do
        $r \leftarrow \frac{T(k)}{\sum_{k \in K} T(k)}$
        if $CheckHotSpot(r)$ then
            $s \leftarrow ComputeOffsetSize(r)$
            $R \leftarrow UpdateRuleList(R, t_{update}, s, k)$
        end if
    end for
end while
```

In practice, the maximum offset in the secondary hashing $s = L(k_1)$ relies on two metrics: (1) Current storage. We assume that tenants with larger storage proportion are more likely to have large forthcoming workloads. Therefore, we select larger s for tenants

Algorithm 2 Secondary Hashing Rule List Update

```
function UpdateRuleList($R, t, s, k$)
    if $(t, s)$ in $R$ then
        $k\_list \leftarrow R.GetKList(t, s)$
        $k\_list.append(k)$
        $R.UpdateKList(t, s, k\_list)$
    else
        $R.insert((t, s, [k]))$
    end if
    return R
end function
```

with larger storage proportion during the initialization phase. Notably, in order to satisfy the requirement of query efficiency, we set $s = 1$ for most of the tenants who have a small storage proportion. (2) Real-time workload. Based on the real-time write throughput proportion, which is periodically reported from the workload monitor, the load balancer will enlarge the maximum offset s for the tenants who are considered as hotspots. This adjustment of s happens during the runtime of ESDB.

4.2 Read-your-writes Consistency

Although the dynamic secondary hashing enables a flexible routing policy, the change of secondary hashing offset brings risks to read-your-writes consistency, because it breaks the static mapping between records and shards. For example, for a tenant k_1, if its maximum offset s changes to s' during runtime, it becomes very difficult to find all the shards that have hosted k_1's historical records: the secondary hashing result changes from $h_2(k_2)$ mod s to $h_2(k_2)$ mod s'. This inconsistency can cause severe problems such as duplicate indices across different shards, deletion failure, and incorrect query results. In order to solve this problem, we maintain a *secondary hashing rule list R* during the runtime of ESDB load balancer. Each second hashing rule is maintained as a tuple (t, s, k_list), where t represents the time when the rule takes effect, s is the maximum offset of secondary hashing and k_list records the tenant IDs that adopt s in its secondary hashing. The secondary hashing rule list consists of secondary hashing rules built during the initialization and runtime phases of ESDB's load balancing process.

Load balancing with hashing rules. Algorithm 1 demonstrates how ESDB performs load balancing through updates to the hashing rule lists. At the initialization phase (Line 5-10), the load balancer builds secondary hashing rules from the current storage for each tenant. During the runtime phase (Line 11-21), the load balancer updates secondary hashing rule list R according to the fluctuation of the real-time workloads. Specifically, we manually design new secondary hashing rules for "hot" tenants according to the storage proportion or real-time workload proportion ($ComputeOffsetSize$). In practice, we choose s among exponents of 2 (e.g., 1, 2, 4) in order to limit the number of secondary hashing rules and accelerate the search in the rule list. The load balancer uses $UpdateRuleList$, which is presented in Algorithm 2, to update R (Line 9 and 18).

Once a **write** operation (e.g., INSERT, UPDATE, DELETE), which can be identified by tenant ID k_1, record ID k_2 and record created time t_c, arrives at a coordinator node, the coordinator is responsible to select a matching secondary hashing rule from R. The rule (t, s, k_list) must satisfy the following three conditions: (1) t must be earlier than the creation time of record t_c. (2) k_1 is in k_list. (3) s is the largest among all the rules that satisfy the first two conditions. In this way, both the workloads to create new records (i.e., INSERT) and the workloads to modify existing records (i.e., UPDATE, DELETE) are routed to the correct shards.

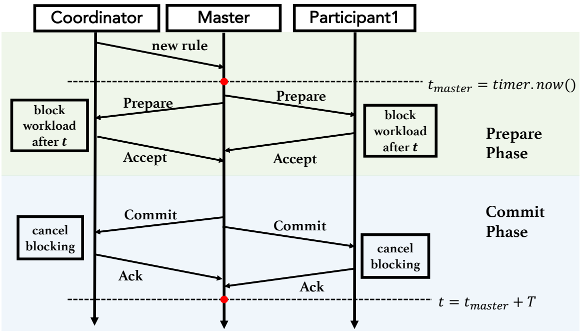


Figure 5: ESDB's secondary hashing rule consensus protocol

Once a **read** operation (i.e., SELECT), which is associated with tenant ID k_1, arrives, the ESDB query client will use k_1's matching secondary hashing rule (t, s, k_list) to decide the correct consecutive shards where the query should be executed on, that is, from shard $h_1(k_1)$ mod N to shard $(h_1(k_1) + s - 1)$ mod N.

4.3 Consensus

In ESDB, each node works as a coordinator which is responsible to route workloads to matching shards. Therefore, it is essential for the whole system to reach consensus on the secondary hashing rule list R once a new rule is generated on any node, in order to ensure strict consistency [37]. That is, all coordinators always use the same secondary hashing rule for the same write workload.

Over the past decades, classic consensus protocols were proposed for distributed transactions. 2PC and 3PC [22, 23, 59] are commitment protocols which aim for atomicity of transactions (binary consensus on commit or abort). They cannot ensure strict consistency of R in the occurrence of network partition or node failure. Paxos [45, 46], Raft [51] and ZAB [41] solve problems, such as fault tolerance, leader election, crash recovery [26, 32, 36], to reach consensus across replicas. However, they are not necessarily needed in the scenario of deciding the secondary hashing rules. This is because R is an append-only list where each rule is associated with an effective time. Therefore, we do not need to decide the ordering of rules (as they are sorted by their effective time), instead, it reduces to deciding, for each rule, whether it is committed or aborted. We can therefore use the more efficient commitment protocols (e.g., 2PC) to reach consensus on the secondary hashing rules. In ESDB, we propose a 2PC variant protocol which is inspired from Spanner's commit wait mechanism [28] to solve the problem of workload blocking. Figure 5 shows an overview of this protocol.

Prepare Phase. Whenever a coordinator node builds a secondary hashing rule, it sends the new rule to the master node of ESDB cluster. The master node decides the effective time t that the rule takes effect using its local timer with a manually designed time interval T, $t = timer.now() + T$. Next, the master node broadcasts the effective time as a proposal to all participant nodes. When a participant receives the proposal, it checks whether all its records has been created earlier than the effective time (i.e., ensure $t_c < t$ for

```
SELECT logs FROM transaction_logs
            WHERE tenant_id = 10086
            AND created_time >= ‘2021-09-16 00:00:00’
            AND created_time <= ‘2021-09-17 00:00:00’
            AND status = 1 OR group = 666
```

Figure 6: Query example

all executed records). If this condition is satisfied, participant blocks all workloads whose creation times are later than the effective time and replies the master node an acceptance message. Otherwise, participant reports error to the master node. If the master node receives any error message or detects any timeout (a participant does not respond within $\frac{T}{2}$), this secondary hashing rule is aborted. Otherwise, the commit phase begins.

Commit Phase. The master node broadcasts commit message as well as the secondary hashing rule to all participating coordinators. Since all nodes have reached consensus in the last phase, they will accept the commit message and add the rule to their local secondary hashing rule lists. Once the commit phase is complete, the nodes will remove the block of workload execution (i.e., continue to process workloads with creation time greater than the effective time).

Choose of time interval. The time interval T provides a buffering period for the system to reach consensus on the secondary hashing rules. T should be much larger than the roundtrip latency of broadcasting (e.g., 100ms) and the maximum of local clock deviations across the cluster (no more than 1s in ESDB) to ensure strict consistency. At the same time, T should be shorter than our expected time of load balancing (e.g., 60s) for effectiveness. As long as T is larger than the time for the cluster to reach consensus, ESDB's consensus protocol achieves non-blocking of workload processing.

Fault tolerance. Although ESDB's consensus protocol ensures strict consistency of R, it still suffers from network partition and node failures during the commit phase. ESDB adopts an automatic solution of fault tolerance. However, it relies on the detection of network partition and node failure, that is, it needs to differentiate temporarily unresponsive nodes from failed nodes. A typical solution uses a pre-defined timeout; in ESDB, we manually verify a raised alarm (a node becoming unresponsive), to definitively decide whether a node failure or a network partition has occurred.

5 OPTIMIZATION

ESDB focuses mainly on multi-column SFW (SELECT-FROM-WHERE) queries on a single table, where multiple predicates are connected by AND and OR operators. Before the deployment of optimizations, ESDB's query performance is decent when queries only involve few columns. However, we observed more than 10x overhead from ESDB's multi-column queries compared to that of a MySQL cluster. (Queries from sellers usually involves with more than 10 columns). After a performance analysis, we identify that the suboptimal performance mainly results from Lucene's rigid query plans. To address this problem, we introduce a query optimizer (see Section 5.1).

5.1 Query Optimizer

As an example, consider a query involving four columns (shown in Figure 6). The execution plan generated by Lucene is shown in Figure 7. First, Lucene generates posting lists, which record the row IDs of the selected rows, for each column by searching the corresponding indices. Then it aggregates the posting lists through intersections and unions. This query plan introduces large overhead since the posting lists are generated sequentially. It will become more time-consuming when the selectivity of a column is high and the posting list grows prohibitively large. In order to solve this problem, we let ESDB incorporate features from relational databases: composite indices, sequential scan and a rule-based optimizer (RBO) which produces cost-effective query plans.

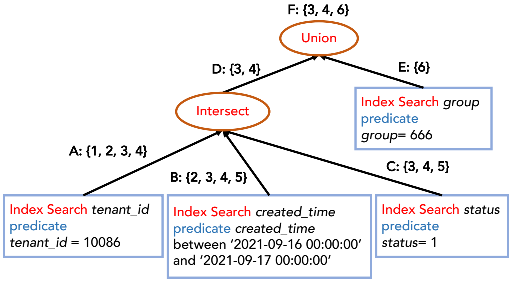


Figure 7: Example query plan of Elasticsearch. A, B, C, D, E and F represent posting lists generated by corresponding operations.

Composite index. In relational databases, composite index is built on multiple columns which are concatenated in a specific order. Taking advantage of composite indices usually avoids time-consuming operation, such as table scan, thus accelerates queries that involve multiple columns [24]. Meanwhile, composite indices have limited applicability, as the columns must comply with the leftmost sequence (i.e. the leftmost principle). For example, if the composite index is built on two columns *column1_column2*, we can either query about (*column1*) or (*column1, column2*); on the other hand, queries about (*column2*) cannot leverage this composite index. In order to increase availability of composite indices, DBAs are expected to manually build composite indices among a massive amount of column combinations [27].

Elasticsearch uses Bkd-tree [55] to index numeric data and multi-dimensional (e.g., geo-information) data. Bkd-tree is an index structure which combines kd-tree [57] and B+ tree. Unlike B+ Tree, Bkd-tree enables division of search space along different dimensions. Therefore, it is not necessary to follow the leftmost principle and this makes the composite index more flexible. Furthermore, it optimizes disk I/O and significantly reduces overheads of insertion and deletion by dynamically maintaining a forest of kd-trees [55]. However, Bkd-tree suffers from the curse of dimensionality where the search performance degrades as the number of dimensions grows high [15]. In ESDB, we build concatenated columns and one-dimension Bkd-trees on these columns as the composite indices. Although such design has less flexibility, ESDB's composite indices search performs fast and is able to cover most query workloads in practical application scenarios.

Another challenge of composite index is the growing key size when we concatenate columns, which makes operations like key comparisons expensive. In order to solve this problem, we take the advantage of common prefixes: since the concatenated keys are sorted in the composite index, the leaf nodes in the Bkd-tree usually contain keys that share a common prefix. By leveraging the common prefixes, we manage to increase the storage efficiency and reduce the cost of key comparisons.

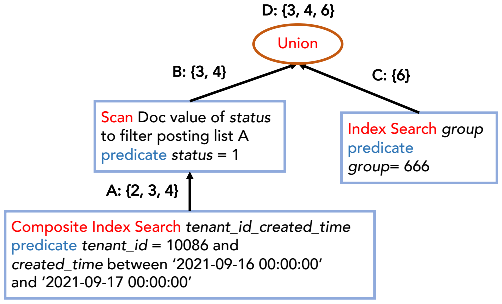

D: {3, 4, 6}
Union
B: {3, 4}
C: {6}
Scan Doc value of status to filter posting list A predicate status = 1
Index Search group predicate group= 666
A: {2, 3, 4}
Composite Index Search tenant_id_created_time predicate tenant_id = 10086 and created_time between '2021-09-16 00:00:00' and '2021-09-17 00:00:00'

Figure 8: Example query plan of ESDB. A, B, C and D represent posting lists generated by corresponding operations.

Sequential scan. Another important operation supported in traditional databases is sequential scan. Although it is considered less efficient, this operation requires less I/Os and performs better when the selectivity or cardinality of a column is low (e.g., gender column). However, sequential scan is not implemented in Elasticsearch because it is against core idea of search engines, which are typically designed for full-text search and ranking without fetching the raw documents. In ESDB, we implement sequential scan as an auxiliary of composite index. Based on the posting list generated from a composite index search, ESDB sequentially scans through the corresponding Doc values [2] to generate a filtered posting list. In this way, sequential scan becomes an effective search operation and is capable of accelerating query in certain cases (e.g., for queries that include columns without indices). In practice, we maintain a scan list which includes the names of columns that can benefit from sequential scans.

Rule-based optimizer. Finally, we introduce ESDB's rule-based optimizer (RBO) for multi-column SFW queries. RBO chooses execution plans for different columns based on the availabilities and rankings of the following access paths:

- **Composite index** is available when the predicates connected by AND use columns that are in some composite indices. In this case, we use longest-match to select the composite index which includes as many columns as possible.
- **Sequential scan** is available when the predicates connected by AND use columns that are not included in any composite index, but are included in the scan list.
- **Single column index** is available when the predicates connected by AND use columns that are not included in any composite index nor in the scan list, or the predicates are connected by OR.

Figure 8 depicts the optimized query plan of the example query (shown in Figure 6). Posting list A is generated from composite index *tenant_id_created_time*. After scanning the Doc value of *status*, the final result is the union of filtered posting list B and posting list C which is generated from a single index search.

5.2 Physical Replication

Instead of asking replicas to execute same write workloads (i.e. logical replication), a physical replication framework, such as Lucene Replicator [6] and Solr 7's TLOG [10], replicates segment files directly. When the primary shard refreshes a new segment file, it initializes a replication process, during which the replicas compute segment diff (i.e., the difference between segment files located on two shards) and request the missing segments from remote shards.

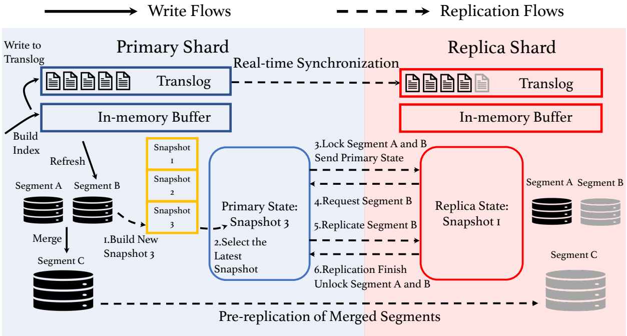


Figure 9: Framework of ESDB's physical replication.

ESDB adopts physical replication with the goal of reducing the CPU overhead incurred in the replication process. Specifically, we design it to overcome two drawbacks seen in previous works: 1) a long monolithic replication process that can be easily interrupted by a new round of replication process; 2) a long visibility delay (i.e., the interval between the timestamps when a segment becomes visible on the primary shard and on a replica) caused by replicating a large merged segment. ESDB's physical replication framework (shown in Figure 9) consists of three replication mechanisms:

Real-time synchronization of Translog. Since Translog is the durability guarantee of ESDB, we require that it should be synchronized in real-time between the primary shard and the replicas. The primary shard forwards a write workload to the replicas once it is executed successfully. The replica then adds the received write workload to its local Translog. In this way, ESDB ensures that all replicas are able to recover the data locally in case of replica failures or primary/replica switch.

Quick incremental replication of refreshed segments. ESDB further adopts a quick incremental replication mechanism to avoid long replication processes which can be easily interrupted. Figure 9 depicts the six steps of the quick incremental replication:

1. The primary shard builds a snapshot of the current local segments and adds it to a snapshot list (shown as yellow boxes) every time a refresh operation finishes.
2. The latest snapshot (i.e., Snapshot 3 in this example) is selected as the current primary state.
3. The primary shard locks the segments in the current snapshot (i.e., Segment A and B) and sends them to the replica.
4. The replica computes the segment diff according to its local snapshot and the snapshot received from the primary shard. Based on the segment diff, the replica either requests segments (i.e., Segment B) or deletes the segments that are already deleted by the primary shard.
5. The primary shard sends the segments requested by the replica.
6. After the replication finishes, the replica informs the primary shard to unlock the segments in current snapshot.

Above mechanism stabilizes the physical replication process when the refresh interval is short and guarantees the replication of the latest segments.

Pre-replication of merged segments. Since the merged segments are usually large, replicating merged segments with quick incremental replication can cause delays in replicating refreshed segments. For example, if snapshot 3 in Figure 9 contains Segment C, the replicas will request Segment B and C together. This causes

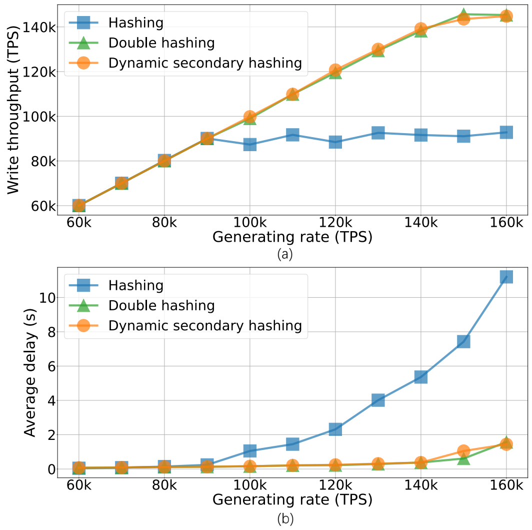


Figure 10: Comparisons of three routing policies when $\theta = 1$. Figure (a) and (b) respectively present write throughput and average delay with different generating rate.

larger visibility delay of Segment B. In order to solve this problem, we further introduce pre-replication of merged segments. When a merged segment is generated, the primary shard immediately starts to replicate it to replicas. This pre-replication is independently running along with the quick incremental replications. In this way, merged segments never appear in segment diff and thus have limited influence on the replication of refreshed segments.

6 EVALUATION

In this section, we present the evaluation results of ESDB to demonstrate its capability of processing skewed write workloads while retaining high throughput and low latency of distributed queries.

6.1 Experimental Setup

All experiments are performed on a cluster consisting of 11 ECS virtual machines (ecs.c7.2xlarge) on Alibaba Cloud. Each virtual machine contains 8 vCPUs, 16GB memory and 1TB SSD disk. We use three machines to simulate ESDB's write and query clients and the rest eight machines as the worker nodes of a ESDB cluster.

In order to simulate real-time processing of Alibaba's e-commerce transaction logs, we build a benchmark which generates random workloads based on the template of our transaction logs and collects metrics of ESDB cluster in real-time. During the evaluation, the simulated workloads are routed to 512 shards located on the eight worker nodes. The simulated workloads contain columns of transaction ID (an auto-increment unique key), tenant ID and creation time which are essential for ESDB's workload balancer. In order to simulate different level of skewness situations, we let the workload generators sample tenant IDs from Zipf distribution tunable by a skewness factor θ. The sampling size of tenant k is set to be proportional to $(\frac{1}{k})^{\theta}$. We select 5 different θs: 0, 0.5, 1, 1.5 and 2. When $\theta = 0$, Zipf distribution is effectively reduced to a uniform distribution. When $\theta = 1$, simulated workloads are the closest to real workloads. Simulations with $\theta = 1.5$ and $\theta = 2$ rarely happen in our production environment, but serve to evaluate the performance of ESDB in the case of extreme skewness.

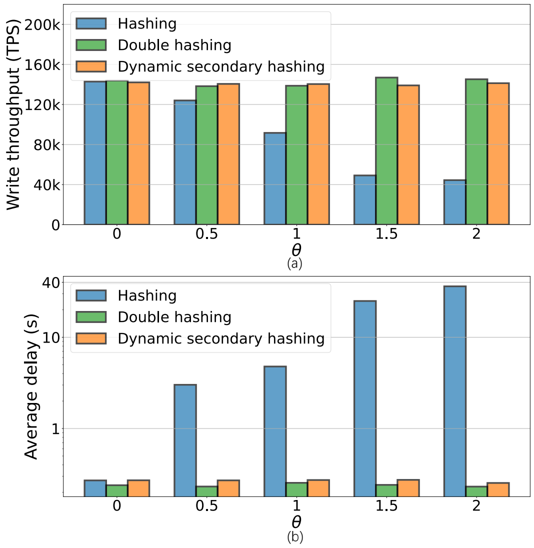


Figure 11: Write throughput and average delay of three routing policies with different skewness factor θs.

6.2 Balanced Write

6.2.1 Write Throughput and Delay. In the first set of our experiments, we measure the cluster throughput and the write delay to evaluate the performance of three different routing policies:

- Hashing, the baseline policy without any workload balancing;
- Double hashing, another baseline policy that distributes data of each tenant to 8 shards;
- Dynamic secondary hashing, the routing policy used by ESDB's load balancer.

Figure 10 presents the cluster throughput and write delays when $\theta = 1$ with different data generating rate. In Figure 10 (a), we observe that the throughput of hashing reaches its limit at around 90K TPS while the other two does not stop until they reach 140K. This is mainly because hashing fails to balance the skewed workloads, and thus waste the resource that could have been used to handle workflows targeted at hotspots. On the contrary, dynamic secondary hashing manages to balance the skewed workloads, and therefore has close performance to double hashing, which is the optimal option since the data is uniformly distributed on the nodes.

Figure 10 (b) shows how the average delay changes as the data generating rate grows. Delays of three routing policies all rise when the generating rate surpasses their throughput upper bounds. However, we observe that the delay of hashing increases rapidly after it reaches its throughput upper bound while the other two have smoother trends. This figure further demonstrates that dynamic second hashing significantly outperforms hashing and has close-to-optimal write performance.

We use Figure 11 to show that dynamic secondary hashing is capable of balancing workloads with different skewness factors. In this set of experiments, we collect the average write throughput during a period of more than 15 minutes for more stable results. Figure 11 (a) shows the write throughput with the three routing policies when the data generating rate is 160K TPS. When the skewness factor $\theta = 0$, the workload is naturally balanced and all three policies exhibits similar write throughput and practically reach the

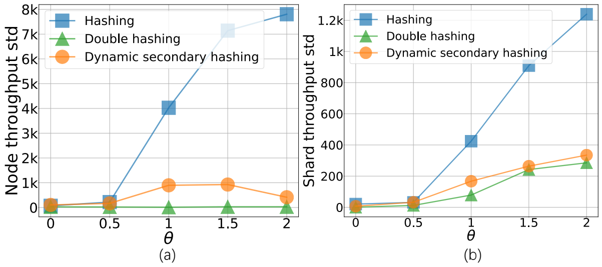


Figure 12: Standard deviation of write throughput of 8 nodes (a) and 512 shards (b) with different skewness factor θs.

performance upper bound of this cluster. When θ increases, the throughput of hashing drops while the other two remain stable. Compared to hashing, dynamic secondary hashing significantly enhances cluster throughput regardless of the level of skewness.

Figure 11 (b) reports the average write delay as we use different θs. In this figure, we observe that the average delay of hashing grows rapidly as θ increases. In the worst case, the average delay is more than 100 times higher than the delays without skewness. On the contrary, the average write delay for double hashing and dynamic secondary hashing remain stable and satisfactory (around 0.2 seconds) even when the skewness factor is extremely high. Notably, the average delays of double hashing are always lower than dynamic secondary hashing even when $\theta = 0$. It is the case because it is nearly impossible for dynamic secondary hashing to reach complete uniform distribution. Nevertheless, the write delays of dynamic secondary hashing retains low and closely track those of double hashing, which is valuable in production environments.

6.2.2 Distribution of Throughput on Nodes and Shards. In addition to the cluster throughput and delays, we also collect throughput of individual worker nodes and shards to show ESDB's load balancing capability more directly. Figure 12 (a) shows the standard deviations of throughput on nodes with different θs. When $\theta = 0$ and $\theta = 0.5$, standard deviations of these three routing policies only have slight differences. When θ grows larger, the imbalance between different nodes become more obvious as the standard deviation of hashing becomes larger. On the contrary, dynamic secondary hashing dramatically reduces standard deviations of the throughput on nodes. Although they are still higher than those of double hashing, the throughput reduction (shown in Figure 11 (a)) caused by higher standard deviation is acceptable.

Figure 12 (b) shows the standard deviations of the throughput on shards with different θs. We observe that dynamic secondary hashing is still able to reduce skewness across shards. Although the skewness across shards only has indirect impact on write throughput, it is an important factor for ESDB's query performance. Queries running on large shards incurs higher overhead compared to queries running on small shards. Therefore, we want the distribution among shards to be as uniform as possible in order to reduce query latency variance of multiple tenants.

Next, we study the distribution of individuals worker nodes and shards when $\theta = 1$. Figure 13 shows the throughput and CPU usage of the eight worker nodes with hashing (a), double hashing (b) and dynamic secondary hashing (c). In the figure, we observe that neighboring nodes have similar throughput and CPU usage; this is because each shard has a replica. For example, the shard of the largest hotspot resides in node 1, and it has a replica that resides in on a different node (node 2 in the figure). With hashing, node 1 and node 2 are the only two nodes that work at full capacity and the rest worker nodes' resources are wasted. With dynamic secondary hashing, the throughput and CPU usage of the rest nodes are significantly enhanced because they now participate in processing the excessive workloads, which were previously allocated only to node 1 and 2. We observe in Figure 13 (c) that, with dynamic second hashing, the throughput of individual nodes is close to evenly distributed, and the average CPU usage is around 85%.

Figure 13 (d) shows normalized shard sizes when $\theta = 1$. With hashing, shard size approximately has a Zipf distribution. The largest shard is more than 100 times larger than the smallest shard. On the contrary, with dynamic secondary hashing, the shard sizes are more balanced and the largest shard is only 16 times larger than the smallest shard. Double hashing has the most uniform distribution; the largest shard is 13 times of the smallest.

6.2.3 Write adaptivity. Figure 14 shows how dynamic secondary hashing adaptively alleviates hotspots in real-time. In this experiment, we introduced two groups of hotspots by changing the mapping between the tenant IDs and Zipf sampling results during a period of six minutes. We observe that when the first group of hotspots comes, the write throughputs of hashing and dynamic secondary hashing drop sharply. However, after the commitment of new secondary hashing rules, the write throughput of dynamic secondary hashing increases back to 120K while the write throughput of hashing never rises again. Similarly, write throughput of dynamic secondary hashing drops and recovers on the arrival of the second group of hotspots. Write throughput of double hashing is not affected by random hotspots because double hashing distributes workloads to all 8 worker nodes.

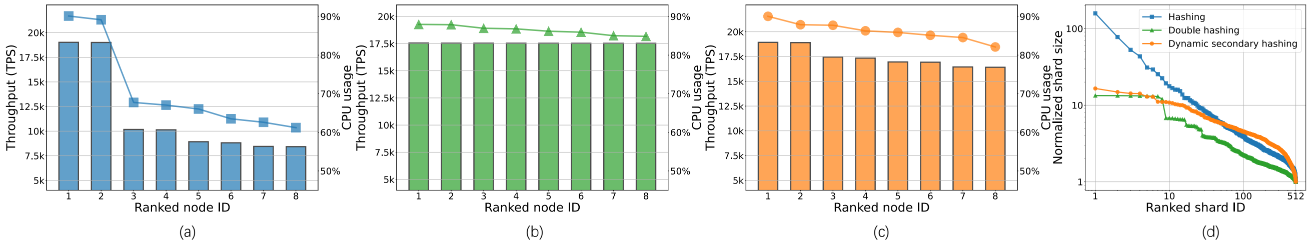
6.2.4 Physical replication. Figure 15 compares write throughput and average CPU usage with logical replication and with physical replication. In Figure 15 (a), write throughput with logical replication stops rising when the generating rate surpasses 140K while write throughput with physical replication rises to more than 180K. In Figure 15 (b), the average CPU usages with physical replication are always lower than logical replication. The experimental results prove that physical replication is able to reduce CPU consumption and increase write throughput of the cluster.

6.3 Query Performance

In this section, we evaluate the query throughput of skewed multi-tenant data when using different routing policies, as well as the effectiveness of the query optimizer and the frequency-based indices. We still use the simulated workload in Section 6.2. When building the top-k query, we select the most commonly used query template in practice: retrieving transaction logs of a tenant in a time period. For example, consider the following query template.

```
SELECT * FROM transaction_logs
WHERE tenant_id=1 AND created_time BETWEEN
‘2021-09-16 00:00:00’ AND ‘2021-09-17 00:00:00’
```

Based on this template, we build a query benchmark that generates random queries with multiple filters appended after the predicates of tenant ID and time range. (The number of involved columns is randomly chosen from 3 to 10.) Details of experiment setup are explained in the following sections.



(a) (b) (c) (d)

Figure 13: Write throughput and CPU usage with hashing (a), double hashing (b) and dynamic secondary hashing (c). Bars represent throughput, lines represent CPU usage. Figure (d) shows normalized shard sizes with three routing policies.

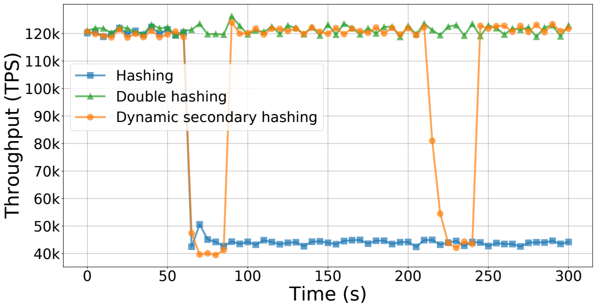
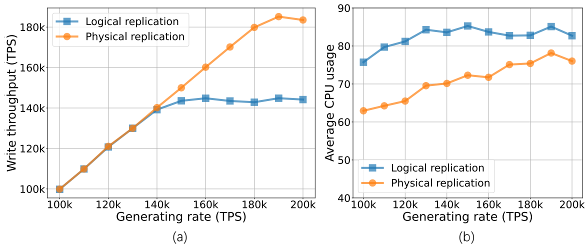


Figure 14: Real-time write throughput with three routing policies in 6 minutes.



(a) (b)

Figure 15: Write throughput (a) and average CPU usage of the cluster (b) with logical replication and physical replications.

6.3.1 Query Throughput. In this experiment, we evaluate the query throughput when we issue queries on an ESDB cluster consisting of eight worker nodes, 512 shards and 40M simulated transaction logs. These transaction logs belong to 100K tenants with a skewness factor $\theta = 1$. For each of the top 2000 tenants, we let three machines concurrently generate SQL queries and send requests to the ESDB cluster to evaluate the upper bound of query throughput. In order to collect more stable results, we add `LIMIT 100` statement after every SQL query statement, which avoids fetching too many rows.

Figure 16 shows the query throughput for the top 2000 tenants with the three routing policies. When using double hashing, each tenant's data is distributed to 8 shards, which means a query has to be expanded to 8 subqueries, one for each shard. Therefore, the query throughput for double hashing is much lower than the other two routing policies. On the contrary, dynamic secondary hashing achieves query throughput as high as hashing for both large tenants and small tenants. This is because, in our experiments, dynamic secondary hashing distribute a tenant's data to a smaller set of shards. Therefore, the number of subqueries is notably smaller than double hashing, and it increases the query throughput by as much as 63% (for the smaller tenants).

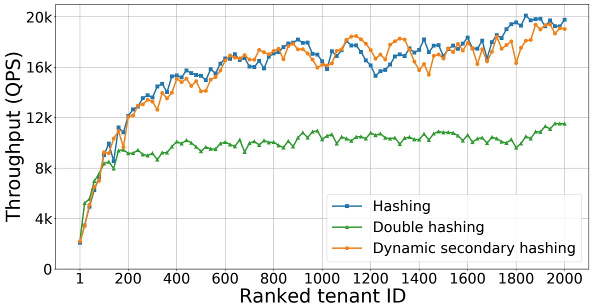
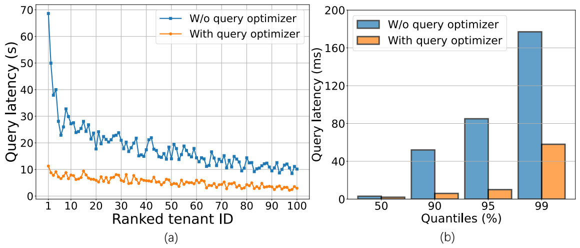


Figure 16: Query throughput of the top 2000 tenants with three routing policies.



(a) (b)

Figure 17: Average (a) and quantiles (b) query latencies of the top 100 tenants with and without ESDB's query optimizer.

Compared to hashing, dynamic secondary hashing also has its own advantage for queries issued to large tenants: since the shard sizes for large tenants are much smaller compared to those of hashing (Figure 13 (d)), subqueries can be executed in parallel and therefore complete faster. For this reason, we do not observe significant drop of query throughput for large tenants. When processing queries issued to small tenants, both dynamic secondary hashing and hashing only execute one subquery on the target shard and have similar performance.

6.3.2 ESDB's Query Optimizer. In order to prove the effectiveness of ESDB's query optimizer, we build a query sets of 1000 queries for each of the top 100 tenants. We then collect the total time consumed to finish the execution of the query set with a single-threaded query client. The target database is the same to the one used in the previous experiments for evaluating the query throughput. As shown in Figure 17 (a), query latencies decrease after enabling query optimizer for all the top 100 tenants. Figure 17 (b) further confirms that ESDB's query optimizer is able to reduce query latencies, and that the query latency is under 200 ms even for 99-percentile latency. Overall, with ESDB's query optimizer, the average query latency is improved by 2.41 times, where the latency of the queries issued to the largest tenant is most significantly improved by 5.08 times.

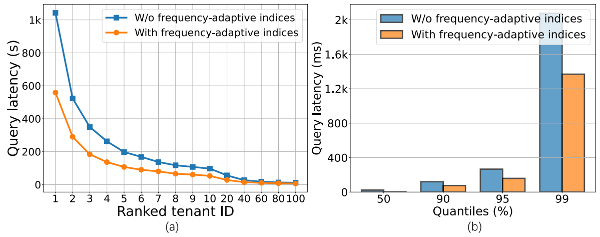


Figure 18: Average (a) and quantile (b) query latencies of the top 100 tenants with and without frequency-based indices.

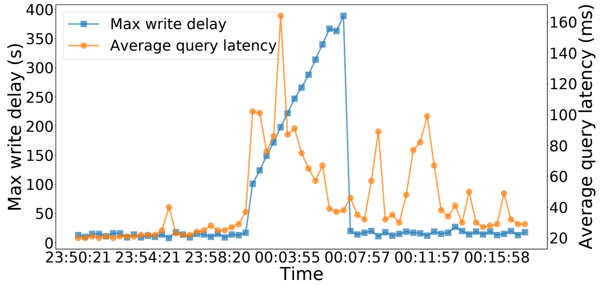


Figure 19: Max write delay and average query latency at the beginning of Single's Day Global Shopping Festival, 2021.

6.3.3 Frequency-based indexing. Imitating the real-world data from our production environment, the simulated "attributes" column in our benchmark consists of 1500 sub-attributes whose frequencies are skewed (top 30 sub-attributes appear in about 50% of both write and query workloads). When generating "attributes" for each simulated row, we sample 20 sub-attributes from Zipf distribution ($\theta = 1$). In this experiment, We build indices only for the top 30 sub-attributes, which incurs only 6.7% storage overhead. When generating query workloads, we append a filter of a sub-attribute, which is also sampled from Zipf distribution, to the query template used in Section 6.3. Figure 18 (a) and (b) show the average and quantile query latencies for the top 100 tenants. We observe that, with frequency-based indices, query latencies improve significantly; the average query latency of top 100 tenants is reduced by as much as 94.1%.

6.4 Online Performance

In our last experiment, we evaluate ESDB's performance in a production environment. More concretely, ESDB is used to support Alibaba's e-commerce platform during the 2021's Single's Day Global Shopping Festival. Figure 19 shows the max write delay and average query latency during a period of approximately 30 minutes around the beginning of the festival. We observe that the max write delay starts to rise notably at 00:00 am due to the dramatic increase of workloads. After the detection of hotspots and the adoption of secondary hashing rules, it takes less than 7 minutes for ESDB to process the workloads generated during the first few seconds after 00:00 am and then fully eliminate write delays after the adaptation. This is a significant improvement over previous year's max write delay, which can be as high as over 100 minutes. In addition, ESDB retains decent average query latency during the first 30 minutes of shopping festival. The average query latency does not surpass 164ms even when both the write and query throughputs are extremely high.

7 RELATED WORK

Different load balancing techniques have been proposed for different applications. Google Slicer [14] proposes a weighted-move sharding algorithm, which automatically executes merge of cold slices and split of hot slices based on "weight" (a metric to evaluate skewness) and "key churn" (cost of split/merge). Facebook's Shard Manager [47] moves hot shards out of overloaded servers. CockroachDB [62], Spanner [28], HBase [18], Yak [44] use resharding methods which automatically split and move shards of "hot" tenants. Live migration is another type of migration-based load balancing technique which moves entire database applications of hotspots across nodes. Notable applications, such as Zephyr [31], ProRea [58] and Slacker [21], adopt different cost optimizations in order to minimize the service interruption and downtime. Albatross [29] is a live migration technique used in shared-storage database architectures. Instead of migrating data, Albatross migrates database cache and the state of transactions. Although effective, migration-based load balancers introduce extra bandwidth and computation overheads, consuming resources that are already very limited.

E-Store [61] identifies tuple-level hotspots and uses smart heuristics to generate optimal data migration plans for load balancing. SPORE [38] uses self-adaptive replication of popular key-value tuples in distributed memory caching systems. Compared to data migration, SPORE incurs fewer overhead and disperses workloads of "hot" tuples to multiple nodes. SWAT [50] implements a load balancing method which swaps replica roles as the primary and secondary replicas to process imbalanced workloads. Although lightweight, these three methods are not appropriate for our application because our major skewness is caused by imbalance of tenants other than tuples nor replicas. Centrifuge [13] uses temporary leases between continuous key ranges and servers to provide consistency for in-memory server pools. It balances workloads by changing the mapping from virtual nodes to physical worker nodes, which cannot be used to address single hotspot. LogStore [25] achieves real-time workload balancing by maintaining and updating a routing table during runtime. Using a max-flow algorithm, LogStore generates routing plans which maximize overall write throughput. However, LogStore's router has no read-your-writes consistency guarantee, and this makes it risky to process `UPDATE` and `DELETE` workloads.

8 CONCLUSION

This paper presents ESDB, a cloud-native document-oriented database which supports elastic write for extremely skewed workloads and efficient ad-hoc queries. ESDB adopts dynamic secondary hashing, a lightweight load balancing technique which eliminates hotspots of multi-tenant workloads in real-time. Compared to hashing and double hashing, dynamic secondary hashing fulfills efficient query and load balancing thus overcomes shortcomings of both techniques. In addition, we introduce optimizations that significantly reduce the computation overheads and query latencies. We evaluate ESDB both in a laboratory environment with simulated workloads and in a production environment with real-world workloads. Our results show that ESDB is able to enhance write throughput and reduce write delays when processing extremely skewed workloads, as well as maintain high throughput and low latency for ad-hoc queries on distributed multi-tenant data.

REFERENCES

[1] DB-Engines Ranking - popularity ranking of search engines. https://db-engines.com/en/ranking/search+engine

[2] Doc Values. https://www.elastic.co/guide/en/elasticsearch/reference/current/doc-values.html

[3] Elasticsearch Flush. https://www.elastic.co/guide/en/elasticsearch/reference/7.3/indices-flush.html

[4] Elasticsearch Replication. https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-replication.html

[5] Elasticsearch Routing. https://www.elastic.co/guide/en/elasticsearch/reference/8.0/mapping-routing-field.html

[6] Lucene Replicator. https://blog.mikemccandless.com/2017/09/lucenes-near-real-time-segment-index.html

[7] Near Real-time Search. https://www.elastic.co/guide/en/elasticsearch/reference/current/near-real-time.html

[8] Reading and Writing Documents. https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-replication.html

[9] Segment Merge. https://www.elastic.co/guide/en/elasticsearch/reference/8.0/index-modules-merge.html

[10] Solr 7 – New Replica Types. https://sematext.com/blog/solr-7-new-replica-types/

[11] Translog. https://www.elastic.co/guide/en/elasticsearch/reference/6.8/index-modules-translog.html

[12] Transport Client. https://www.elastic.co/guide/en/elasticsearch/client/java-api/current/transport-client.html

[13] Atul Adya, John Dunagan, and Alec Wolman. 2010. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services.. In *NSDI*. 1–16.

[14] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer:{Auto-Sharding} for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.

[15] Alexandr Andoni and Piotr Indyk. 2017. Nearest neighbors in high-dimensional spaces. In *Handbook of Discrete and Computational Geometry*. Chapman and Hall/CRC, 1135–1155.

[16] Apache. *CouchDB*. http://couchdb.apache.org/

[17] Apache. *Elasticsearch*. https://www.elastic.co/cn/elastic-stack/

[18] Apache. *HBase*. https://hbase.apache.org/

[19] Apache. *Lucene*. https://lucene.apache.org/

[20] Apache. *Solr*. https://solr.apache.org/

[21] Sean Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant Shenoy. 2012. " Cut me some slack" latency-aware live migration for databases. In *Proceedings of the 15th international conference on extending database technology*. 432–443.

[22] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Addison-wesley Reading.

[23] Philip A Bernstein and Eric Newcomer. 2009. *Principles of transaction processing*. Morgan Kaufmann.

[24] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. 2017. Accelerating multi-column selection predicates in main-memory-the Elf approach. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 647–658.

[25] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. 2021. LogStore: A Cloud-Native and Multi-Tenant Log Database. In *Proceedings of the 2021 International Conference on Management of Data*. 2464–2476.

[26] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 398–407.

[27] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*. Citeseer, 146–155.

[28] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 3 (2013), 1–22.

[29] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endow.* 8 (may 2011), 494–505. https://doi.org/10.14778/2002974.2002977

[30] Peter C Dillinger and Panagiotis Manolios. 2004. Bloom filters in probabilistic verification. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 367–381.

[31] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/1989323.1989356

[32] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)* 1 (2006), 133–160.

[33] Alibaba Group. *Alibaba Cloud*. https://www.alibabacloud.com

[34] Alibaba Group. *OceanBase*. https://www.alibabacloud.com/product/oceanbase

[35] Leo J Guibas and Endre Szemeredi. 1978. The analysis of double hashing. *J. Comput. System Sci.* 2 (1978), 226–274.

[36] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol.. In *EDBT*. 157–168.

[37] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3 (1990), 463–492.

[38] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC '13)*. ACM, New York, NY, USA, Article 13, 17 pages. https://doi.org/10.1145/2523616.2525970

[39] Couchbase Inc. *Couchbase*. https://www.couchbase.com/

[40] MongoDB Inc. *MongoDB*. https://www.mongodb.com/

[41] Flavio Junqueira and Benjamin Reed. 2013. *ZooKeeper: distributed process coordination.* " O'Reilly Media, Inc.".

[42] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 654–663.

[43] Adam Kirsch and Michael Mitzenmacher. 2008. Less hashing, same performance: Building a better Bloom filter. *Random Structures & Algorithms* 2 (2008), 187–218.

[44] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P.P.S. Narayan, Adwait Tumbde, and Brian Cooper. 2012. The Yahoo! Cloud Datastore Load Balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management* (Maui, Hawaii, USA) *(CloudDB '12)*. ACM, New York, NY, USA, 33–40. https://doi.org/10.1145/2390021.2390028

[45] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[46] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 4 (2001), 18–25.

[47] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. 2021. Shard Manager: A Generic Shard Management Framework for Geo-distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 553–569.

[48] George Lueker and Mariko Molodowitch. 1988. More analysis of double hashing. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 354–359.

[49] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review* 3 (2015), 52–66.

[50] Hyun Jin Moon, Hakan Hacıgümüş, Yun Chi, and Wang-Pin Hsiung. 2013. SWAT: A Lightweight Load Balancing Method for Multitenant Databases. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. ACM, New York, NY, USA, 65–76. https://doi.org/10.1145/2452376.2452385

[51] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 305–319.

[52] Oracle. *MySQL*. https://www.mysql.com/

[53] Oracle. *MySQL Cluster*. https://www.mysql.com/products/cluster/

[54] Oracle. *Oracle NoSQL Database*. https://www.oracle.com/database/technologies/related/nosql.html

[55] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*. Springer, 46–65.

[56] Robbi Rahim, Iskandar Zulkarnain, and Hendra Jaya. 2017. Double hashing technique in closed hashing search process. In *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, 012027.

[57] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 10–18.

[58] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. 2013. ProRea: Live Database Migration for Multi-Tenant RDBMS with Snapshot Isolation. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2452376.2452384

[59] Dale Skeen. 1981. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 133–142.

[60] StumbleUpon. *OpenTSDB*. http://opentsdb.net/

[61] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 3 (nov 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[62] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.