ESTELLE: An Efficient and Cost-effective Cloud Log Engine

Yupu Zhang*
University of Electronic Science and Technology of China
Chengdu, China
zhangyupu@std.uestc.edu.cn

Guanglin Cong*
Cloud Database Innovation Lab of Cloud BU, Huawei Technologies Co.
Chengdu, China
congguanglin@huawei.com

Jihan Qu
University of Electronic Science and Technology of China
Chengdu, China
qujihan@std.uestc.edu.cn

Ran Xu
Cloud Database Innovation Lab of Cloud BU, Huawei Technologies Co.
Chengdu, China
xuran215@huawei.com

Yuan Fu
University of Electronic Science and Technology of China
Chengdu, China
fuyuan@std.uestc.edu.cn

Weiqi Li
Cloud Database Innovation Lab of Cloud BU, Huawei Technologies Co.
Chengdu, China
liweiqi4@huawei.com

Feiran Hu
Cloud Database Innovation Lab of Cloud BU, Huawei Technologies Co.
Chengdu, China
hufeiran@huawei.com

Jing Liu
Cloud Database Innovation Lab of Cloud BU, Huawei Technologies Co.
Chengdu, China
liujing160@huawei.com

Wenliang Zhang†
Cloud Database Innovation Lab of Cloud BU, Huawei Technologies Co.
Chengdu, China
zhangwenliang14@huawei.com

Kai Zheng†
University of Electronic Science and Technology of China
Chengdu, China
zhengkai@uestc.edu.cn

ABSTRACT

With the advancement of cloud computing, more and more enterprises are adopting cloud services to build a variety of applications. Monitoring and observability are integral to the complex and fragile cloud-native architecture. As an extremely important data source for both, logs play an indispensable role in applications such as code debugging, root cause analysis, troubleshooting, and trend analysis. However, the inherent characteristic of cloud logs, with TB-level daily data production per user and continuous growth over time and with business, poses core challenges for log engines. Traditional log management systems are inadequate for handling the requirements of massive log data high-frequency writing and storage, along with low-frequency retrieval and analysis in cloud environments. Exploring a low-cost, high-performance cloud-native log engine solution is an extremely extraordinary challenging task. To tackle these challenges, we propose a cost-effective cloud-native log engine, called ESTELLE, equipped with a low-cost pluggable log index framework. This engine features a compute-storage separation and read-write separation architecture, enabling linear scalability. We designed a near-lock-free writing process for handling high-frequency writing demands of massive logs. Object storage is used to significantly reduce storage costs. We also tailored ESTELLE Log Bloom filter and approximate inverted index for this cloud-native engine, applying them flexibly to enhance query efficiency and optimize various queries. Extensive experiments on real open-source log datasets have demonstrated that the ESTELLE Log Engine achieves ultra-high single-core CPU write speeds and pretty low storage costs. Furthermore, when equipped with the complete index framework, it also maintains fairly low query latency across various log scenarios.

CCS CONCEPTS

• **Information systems** → **DBMS engine architectures**; **Structured text search**.

KEYWORDS

cost-effectiveness;Bloom filter;cloud-native;log engine;index framework

ACM Reference Format:
Yupu Zhang, Guanglin Cong, Jihan Qu, Ran Xu, Yuan Fu, Weiqi Li, Feiran Hu, Jing Liu, Wenliang Zhang, and Kai Zheng. 2024. ESTELLE: An Efficient and Cost-effective Cloud Log Engine. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653387

*Equal contribution.
†Corresponding authors. The corresponding author, Kai Zheng, is with Shenzhen Institute for Advanced Study, University of Electronic Science and Technology of China, Shenzhen, China.

SIGMOD-Companion '24, June 9–15, 2024, Santiago, Chile.

ACM ISBN 979-8-4007-0421-5/24/06…$15.00
https://doi.org/10.1145/3626246.3653387

1 INTRODUCTION

Across the last several years, more and more enterprises have rapidly migrated cloud-native applications to cloud-native infrastructures [7], in the form of microservices [5], serverless and container [25, 32] technologies. With a vast number of applications running on hundreds to thousands of machines, this distributed architecture is highly complex yet extremely fragile [41], prone to interruptions due to failures [19], and can even lead to partial paralysis of the Internet [29]. Therefore, monitoring is crucial for checking the operational status of applications. It not only requires issuing alerts when failures occur but also demands early detection of bugs and issues hidden in the development environment that may be exposed in the production environment, aiming to prevent system interruptions [29]. However, compared to previous architectures, the unique characteristics of cloud-native architecture (e.g., Intrusiveness, Resilience, Reliability, etc. [1]) make traditional monitoring solutions and strategies inadequate for monitoring tasks [16, 35].

In recent years, observability, as an extension of monitoring, has become an indispensable feature of the environment of cloud-native architectures [27]. Logs, metrics, and traces, known as the three pillars of observability, are the raw data needed to obtain an internal view of the health and behavior of applications and microservices [26]. Logs, as a crucial data source for monitoring and observability, capture the details of each request and can be used for debugging [37], root cause analysis [41], exploratory troubleshooting [14], and other applications, making them indispensable for any production-grade system [29].

In any production-grade system, the volume of logs increases significantly over time and with business growth. Building a low-cost log engine for an observability platform is an extraordinary mission. We summarize the challenges we encountered in our production environment as follows:

Challenge 1: Heavy and Skewed Log Writes. The hundreds or thousands of various microservices and programs running on the cloud-native infrastructures generate a large amount of logs every day, with log generation times concentrated and frequently encountering bursts of write demands. For example, in our production environment, many users generate several hundred terabytes of logs daily, and the total volume of logs produced each day continues to increase with business growth. Within a day, log writes are mainly concentrated within a few hours. Therefore, the ability to store and rapid write such massive log data at a low cost is crucial.

Challenge 2: Low Frequency and Heavy Log Queries. Compared to write operations, the frequency of log queries is much lower, and the majority of logs will be never queried. However, executing precise queries within such a vast volume of data and within an acceptable latency (ranging from hundreds of milliseconds to a few seconds) is undeniably challenging. Moreover, many queries involve a wide time span, often ranging from a day to a week, and sometimes even longer, up to a month or more. Therefore, establishing reasonable data partitioning and designing efficient and practical indexes and caches are essential.

Challenge 3: Various Log Queries and Important Log Aggregations. In addition to the basic full-text queries, a log engine needs to support several other crucial types of queries to meet the requirements of monitoring and observability. Utilizing AND queries is essential for filtering relevant events or operations that meet multiple conditions, providing a more comprehensive context. Additionally, prefix fuzzy queries can be employed to quickly locate or filter logs related to services or components with specific prefixes, facilitating further analysis and issue resolution. Log aggregation is crucial for identifying trends and helping users recognize bottlenecks, performance issues, or even network threats based on data collected over a period. However, histogram queries for high-frequency words suffer from significant time and resource consumption, limiting their capability for rapid trend analysis. Therefore, designing a system that can optimize various queries and efficiently index data is paramount.

Challenge 4: Low-Cost Log Engine System. The trait of logs growing with time and business makes low cost a necessary requirement for a log engine. It is also an indispensable part of a low-cost observability platform. Here, low cost refers to the efficient writes, storage, and queries of massive log data with fewer resources within an acceptable time frame. Resources here primarily include CPU, memory, I/O, etc. Therefore, utilizing low-cost storage for massive logs and designing a dedicated cost-controllable index framework for this specific scenario is both necessary and practical.

However, there is no existing log engine that meets all of the above requirements. Among these log engines, some choose to have no index at all [9, 18, 23], some choose to build inverted indexes in real-time when writing logs [2, 3, 7, 9, 40]. Specifically, SLS [9] offers two modes: one with no index and another utilizing inverted indexes. Additionally, ClickHouse [39] offers an index-free architecture and uses the standard Bloom filter [6] as the index. Having no index at all allows the log engine to write logs quickly but sacrifices support for efficient queries. Constructing an inverted index of a size comparable to the data size during log writing can lead to slow writing speeds and high storage costs. The use of an index-free architecture with Bloom filters as the log indexes provides efficient log indexing for log queries with minimal impact on log writing speed. However, the standard Bloom filter is not suitable for a low-cost log engine. When using the standard Bloom filter for word filtering, fetching all the Bloom filters into memory at once would incur significant I/O overhead. On the other hand, if only the word related bits from all the Bloom filters are retrieved into memory, the storage medium needs to have efficient random access capability. Both of these approaches do not align with our definition of low cost. Furthermore, none of the aforementioned log engines optimize for various critical queries, especially histogram queries for high-frequency words.

In this paper, we propose a cost-effective cloud-native log engine, called ESTELLE, equipped with a low-cost pluggable log index framework to address the challenges mentioned above. To address heavy and skewed log writes, we introduce object storage to enable low-cost storage of logs and their indexes. We apply a cloud-native architecture with storage-compute separation to support linear scaling of write capacity, and we carefully design an approximately lock-free log writing process. To handle low-frequency and heavy log queries, we adopt a dual time filtering strategy, implement multiple caches, and introduce an efficient indexing framework. To address various log queries and important log aggregations, we configure an index set with multiple pluggable components for

each data block. We design a ESTELLE Log Bloom filter to optimize full-text and prefix fuzzy queries and a fixed-length Approximate Inverted Index for optimizing AND queries on low-frequency words and histogram queries on high-frequency words. Specifically, to enable quick returns of log aggregation results, we set the histogram query in the progressive query mode. To meet the requirements of a low-cost engine, we utilize object storage for log storage and design the index set corresponding to each data block as a cost-effective, cloud-native-friendly version. Specifically, for ESTELLE Log Bloom filter and the Approximate Inverted Index, we carefully design I/O-friendly columnar-store formats and provide strategies and theoretical supports for balancing performance and cost.

Our contributions can be summarized as followed:

1) We implement a cost-effective, cloud-native log engine featuring read-write separation and storage-computation separation. This design facilitates rapid scaling in response to burst write and query scenarios. Furthermore, we propose an near-lock-free writing process based on this framework to accommodate the demands of massive log data ingestion.

2) We propose a low-cost pluggable log index framework primarily composed of ESTELLE Log Bloom filter and Approximate Inverted Index. Both are tailor-made, I/O-friendly index structures specifically designed for cloud-native architectural log engines. The former is employed for effective word filtering, while the latter optimizes histogram queries for high-frequency words and AND queries for low-frequency words. To our knowledge, the ESTELLE Log Bloom filter is the first Bloom filter variant specifically customized for this scenario.

3) We report on experiments using a real open-source log dataset, showing that the ESTELLE Log Engine not only attains exceptionally high single-core CPU write speeds, but also incurs relatively low storage expenses. Moreover, when integrated with the complete index framework, it consistently ensures fairly low query latency in diverse log scenarios.

The remainder of the paper is organized as follows. Section 2 presents low-cost cloud-native ESTELLE Log Engine. Section 3 introduces a low-cost pluggable log index framework. Section 4 describes the detailed processes of several types of optimized queries. Section 5 details the experiments and evalution. Section 6 provides a brief introduction of related work. Section 7 gives the conclusion.

2 ESTELLE LOG ENGINE

This section introduces the ESTELLE Log Engine, covering its cloud-native architecture, storage, writing, and querying processes.

2.1 Architecture Overview

Figure 1 shows the architecture of ESTELLE Log Engine. It is a cloud-native architecture featuring read-write and storage-compute separation. Below is a brief introduction to some main modules.

Meta Cluster is primarily responsible for mananging system metadata and the entire cluster. It stores metadata, including database schema, table schema, permissions information, etc. Additionally, it maintains the Retention Policy (i.e., RP) that specifies the period data is to be retained. Meta Cluster also monitors the status of nodes within the entire cluster. It plays a crucial role in fault

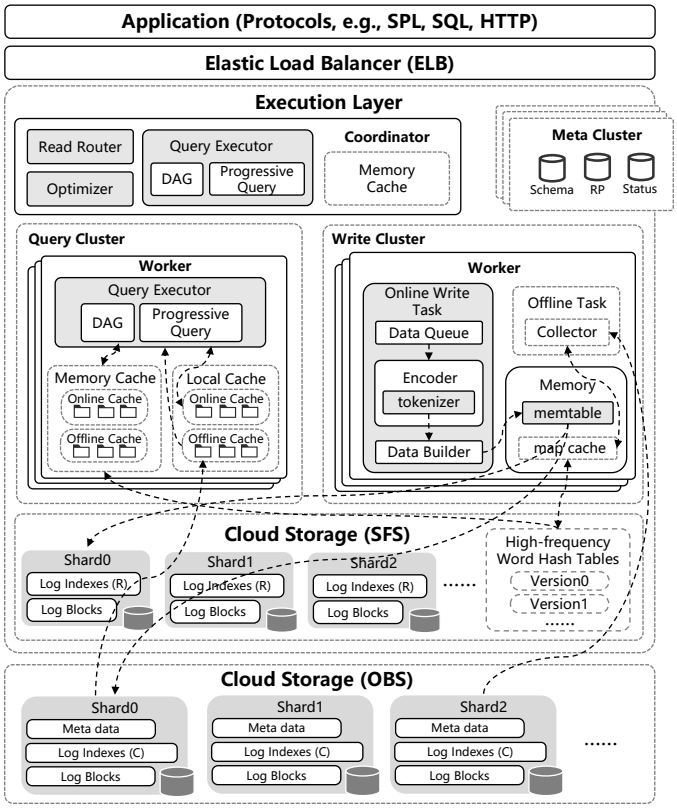

Figure 1: The Architecture of ESTELLE Log Engine

detection and failover. Furthermore, it is also in charge of the allocation and takeover of shards, determing, for instance, which write state nodes are responsible for specific shards. Meta Cluster itself employs raft algorithm [31] to ensure information consistency.

Execution Layer executes query and write operations, serving as the computational core of the ESTELLE Log Engine. It is essentially a cluster composed of numerous nodes, where each node can operate in either a query or write state. When ELB receives a write request from the upper-level application, it randomly assigns it to a node, which is then referred to as a write state node. The query state node is further subdivided into two types: coordinator and subquery node. The query state node that receives query requests dispatched by the ELB is referred to as the coordinator. The query state node that receives subquery requests dispatched by a coordinator is referred to as the subquery node of the coordinator. Each write state node and subquery node in the cluster contains a set of workers, which can be understood as threads. Each worker corresponds to one shard. The log writing process performed by write state nodes and the query process executed by query state nodes are introduced in Section 2.3 and Section 2.4, respectively.

Object Storage Service OBS [12] (i.e., Object Storage Service) is a product from Huawei Cloud, offering massive, secure, highly reliable, and cost-effective object storage services. OBS provides web service interfaces based on the HTTP/HTTPS protocol, allowing multiple cloud servers to access it over the Internet. Moreover, OBS uses the Erasure Code (EC) algorithm, instead of multiple copies, to ensure data redundancy. Object storage has a lower cost compared

to other storage services, but with relatively high access latency. In the production environment, log data exhibits a characteristic of high-frequency writes and low-frequency queries. Therefore, in the ESTELLE Log Engine, OBS is utilized for the permanent storage of both log data and its associated indexes. Typically, data is flushed to OBS only after it has accumulated to a certain extent in the memory of the write state node.

Scalable File Service SFS [13] (i.e., Scalable File Service) is also one of Huawei Cloud's products, offering on-demand scalable high-performance file storage using NFS (i.e., Network File System). It can be mounted simultaneously on multiple cloud servers, enabling shared storage. The storage system adopts a distributed storage architecture with a fully modular and redundant design, eliminating single points of failure. The underlying storage comprises two types of storage media: HDD and SSD. The access speed of SFS is significantly faster compared to OBS, but the cost is much higher. Therefore, in the ESTELLE Log Engine, when the volume of log data in the memory of a write state node has not yet reached the amount of a block, but a flush command is received from the user or there are no log entries written for more than 30 seconds, this log data is then temporarily written to the SFS. This avoids the performance loss associated with frequent flushing to OBS. SFS is also used for storing the high-frequency word hash tables.

2.2 Storage Layout

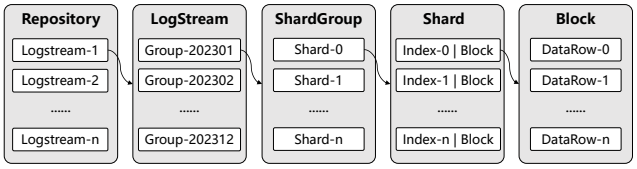

Figure 2: The Storage Layout of ESTELLE Log Engine

The storage layout of the ESTELLE Log Engine is illustrated in Figure 2. A data row mainly consists of two parts: a timestamp and the actual log content, and the latter can be understood as a string. Each data row stores one log entry. Data rows accumulate to form a block once they reach a certain quantity. A block serves as the fundamental unit for data compression and index design. Within a shard, multiple blocks and their corresponding indexes are stored. A shard is the smallest unit to manage the read and write capabilities of the index table, and all the shards are grouped into a Shard Group based on their write time to expedite queries and log expiration. A logstream is a combination of logs and indexes, and can be understood as a table. A repository is a collection of logstreams, with each user corresponding to a distinct repository.

2.3 Writing Process

Figure 1 shows the write state node's log writes as online write tasks for immediate write requests and offline tasks for periodic updates to high-frequency word hash tables.

2.3.1 Online Write Task. Figure 3 serves as a supplement to the content related to online write tasks in Figure 1.

The writing process above the shard level is depicted in Figure 3. Upon receiving a write request from the upper-level application, ELB randomly assigns it to one node in the cluster. At this point,

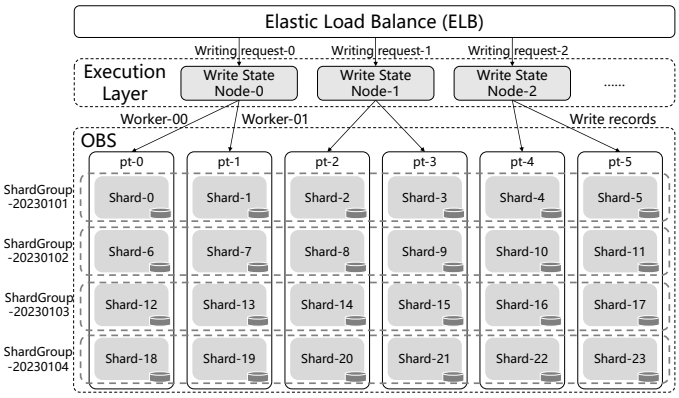

Figure 3: The Online Write Task

the node is referred to as a write state node. Subsequently, each write state node creates multiple workers and distributes log data randomly among these workers. The number of workers created is equal to the number of pts (i.e., partitions) it holds. The pts represent partitions of a repository, and each worker only writes log data in its corresponding pt. Therefore, the number of pts in a repository corresponds to its concurrent write capacity. Each worker corresponds to one shard per day and writes data to it. When a new day begins, the content of the shard from the previous day will no longer change, and the worker will create a new shard in its corresponding pt.

The writing process at the shard level is indicated in the middle-right section of Figure 1. When a worker writes log to a shard, it maintains a Data Queue. The queue releases a batch of log data to the Encoder in a single operation. The tokenizer within the Encoder tokenizes the logs, and these tokens are utilized for index construction. The processed log data and its corresponding indexes are then temporarily cached in the memtable. When a user issues a flush command or if no new log data row written for over 30 seconds, log data along with its indexes, which has not yet reached the volume of a block, will be written to the SFS. Under normal circumstances, when the log data rows in the memtable reach the volume of a block, they are compressed and written to OBS. At this point, the number of indexes constructed in the memtable is checked. If they reach the quantity of a group, they will be transformed into columnar-store format and flushed to OBS. After flushing the log data rows or indexes to OBS, their corresponding contents in both SFS and memtable are cleared. Indexes in memtable and SFS use row-store format, while OBS indexes use column-store. More on index construction and the two formats in Section 3.

The carefully designed write process outlined above provides several benefits. From the perspective of the entire online write task process, apart from the Data Queue, the entire process remains lock-free. This significantly enhances the writing speed of the ESTELLE Log Engine. The number of pts in the repository can adjust to user concurrency needs, allowing linear write capacity scaling.

2.3.2 Offline Task. The Offline Task is a pluggable component designed to generate or update high-frequency word hash tables, intended for use by other pluggable indexes.

The collector, as shown in Figure 1, updates word frequency information daily. At the end of each day, the collector randomly

samples that day's log data from the Shard Group, calculates word frequencies to form a hash table. The hash tables are cached in the map cache, and updated in a sliding window fashion.

Collector also periodically updates the high-frequency word hash tables. When a checkpoint is reached, the collector filters words with frequencies greater than the high-low frequency threshold r from the recently cached hash tables. It calculates each word's average daily frequency, retaining those still above r as high-frequency words. A new high-frequency word hash table, $W_s = \{w_1, w_2, \cdots\}$, is then generated. Each item $w_i =< str, p_s >$ pairs a word str with its frequency p_s. If a high-frequency word hash table has not been generated before, W_s will be assigned a version number and written to SFS. To save storage space, the values in this hash table can be set to null before writing, as we only need it to determine whether a word is a high-frequency word. If a previously generated high-frequency word hash table exists, load the latest version (denoted as $W_l = \{w_1, w_2, \cdots\}$, and $w_i =< str, >$.) from SFS into the map cache. Then calculate the similarity between W_s and W_l using the following Equation 1.

$$\frac{\sum_{w.str \in W_l.str \cap W_s.str} w.p_s}{\sum_{w.str \in W_s.str} w.p_s} \tag{1}$$

If similarity is below 0.9, W_s gets a new version number and is saved to SFS; otherwise, no update is needed.

2.4 Query Process

The primary query process above the shard-level in the ESTELLE Log Engine is depicted in the middle-left section of Figure 1. This process can mainly be divided into the coordinator's request allocation process and the subquery node's query process.

2.4.1 Coordinator Allocation Process. The main role of the coordinator in the cluster is to judiciously allocate sub-requests and merge responses from its subquery nodes. Upon receiving a query from ELB, the coordinator parses, optimizes it, and creates a query plan as a Directed Acyclic Graph (DAG). Subsequently, the coordinator sends sub-requests to multiple other query state nodes (i.e., its subquery nodes) following the guidance of Read Router. Upon receiving the query results from its corresponding subquery nodes, the coordinator merges these results and ultimately returns the final query result to the upper-level application. A query requiring multiple dependent sub-executions is termed a stateful query (e.g., Progressive Query), with previous results temporarily stored in the coordinator's memory cache.

It is the responsibility of the read router to select the subquery nodes corresponding to the sub-requests. Subquery nodes are inherently stateless. The coordinator provides a weak state to these subquery nodes through reading routes. To fully utilize caches of subquery nodes, reduce interactions with OBS and enhance query efficiency, the read router primarily adheres to the following several rules when selecting subquery nodes.

- Within a single query, subquery tasks corresponding to shards within one pt are dispatched to one subquery node.
- Across multiple queries, subquery tasks associated with one shard are dispatched to the subquery node that previously executed subquery tasks on this shard.
- Across multiple stateful queries, the coordinator that received a query previously executed will directly forward the query request to the coordinator that handled the execution of that query task last time.

2.4.2 Subquery Node Query Process. The primary task of subquery nodes is to execute queries and provide the subquery results back to their coordinators. After receiving a sub-request, the subquery node also firstly parses and optimizes the sub-request, and then generates a DAG-form query plan. When executing the query plan, the subquery node creates multiple workers, with each worker corresponding to a shard-level query task. The query process at the shard-level is introduced in Section 4.1.

Each worker corresponds to several types of caches. From the storage medium perspective, the entire cache can be divided into two types: memory cache and local cache. The query involved small size of indexes are cached into memory, including meta data, high-frequency word hash table, etc. Other indexes are cached to the local cache, including Bloom filters, approximate inverted indexes, etc. From the perspective of cache mode, the entire cache can be divided into two types: online cache and offline cache. Online cache primarily caches the indexes involved in currently executed queries to increase the likelihood of hitting the cache when querying the same content again within a short time range. The offline cache primarily utilizes a time sliding window approach to offline cache the complete indexes related to queries executed in the recent time range. This is done to increase the likelihood of hitting the cache when querying content related to recent activities.

3 LOG INDEX FRAMEWORK

To simultaneously meet the demands of swift and accurate access to extensive log data, supporting diverse common queries, and ensuring low and controllable costs, we design a log index framework for ESTELLE Log Engine. Next, we introduce its overview, ESTELLE Log Bloom filter, and Approximate Inverted Index.

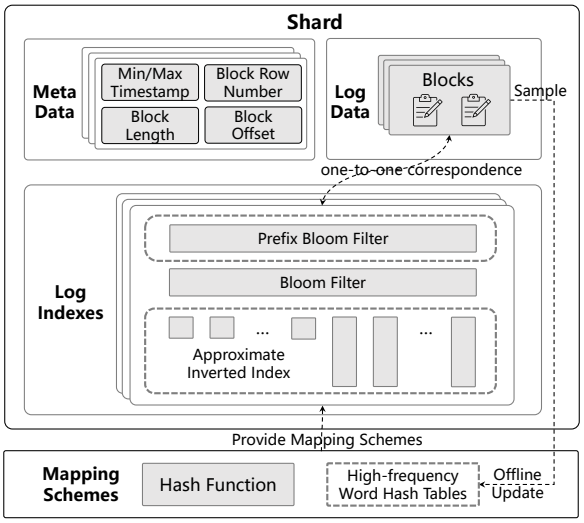

Figure 4: Storage Layout at Shard Level

3.1 Index Framework Overview

As illustrated in Figure 4, the index framework is primarily designed at the shard level. The following is an overview of its components.

Meta Data. The Meta Data in a shard stores the information for each block, including the block's minimum and maximum timestamp, data row number, size, and offset within the shard. The size of each block's metadata is fixed, eliminating the need for additional metadata index. The primary roles of these metadata of blocks in accelerating query speed are as follows:

- **Min/Max Timestamps:** They enable a time filter on block-level to accelerate query for the targeted time range.
- **Block Row Numbers:** They speed up indexing by enabling quick results for histogram queries without any condition.
- **Block Lengthes and Block Offsets:** These facilitate the swift identification of a block's location and determination of its length, enabling the quick, selective retrieval of only the necessary blocks into memory.

Log Data. The log data rows are primarily stored in blocks, with the compressed state stored in OBS and the uncompressed state in SFS. Each data row stores one log entry.

Log Indexes. Each block corresponds to a pluggable and cost-controllable index set, mainly comprising two types: Bloom filter and Approximate Inverted Index. Both types of these indexes are stored in row-store format in memory and SFS, and in columnar-store format in OBS, which is in detail in Section 3.2.1 and Section 3.3. The foundational version of the index set includes only a ESTELLE Log Bloom filter (abbreviated as EL-BF), which is introduced in Section 3.2.1. To optimize histogram queries for high-frequency words and AND queries for low-frequency words, two EL-BFs are combined to form a Frequency Division Bloom filter (abbreviated as ELFD-BF), storing high and low-frequency words separately. Each ELFD-BF is paired with an approximate inverted index. The related content is introduced in Section 3.3. For constructing indexes for prefix fuzzy queries, an additional EL-BF is added as a Prefix Bloom filter, which can be found in Section 3.2.2.

Mapping Schemes. The operation of Bloom filters and approximate inverted indexes relies on the Mapping Schemes module that include a Hash Function and multiple versions of High-frequency Word Hash Tables. This Mapping Schemes module is shared across all shards that store the log data for the same business. The function of the hash function is to map a word to a 64-bit unsigned integer. High-frequency word hash tables classify words as high or low frequency to guide their allocation to the respective sections of the ELFD-BF. A hash table stores high-frequency words from recent logs of a business, updated by the Offline Task (Section 2.3.2). Due to the extremely low proportion of high-frequency words in most of log data, the space occupied by these high-frequency word hash tables is also minimal, allowing them to be resident in memory.

3.2 ESTELLE Log Bloom Filter

To our knowledge, ESTELLE Log Bloom filter (abbreviated as EL-BF) is the first Bloom filter variant designed for the log engine of cloud-native architecture. In this sub-section, we firstly introduce the design of EL-BF, then introduce how to apply it flexibly to enhance various queries.

3.2.1 ESTELLE Log Bloom Filter Design. This section primarily includes the workflow of the EL-BF, its columnar-store format, reasons why it is more suitable for cloud-native frameworks compared to standard Bloom filter, and its theoretical false positive rate.

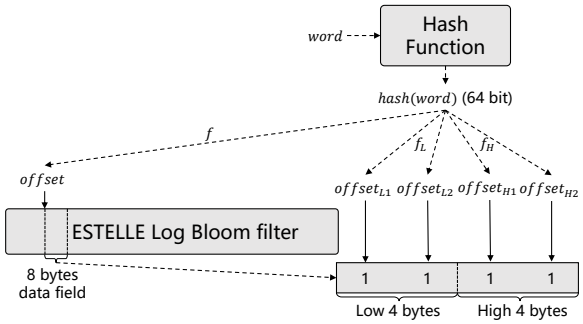

Figure 5: The Workflow of ESTELLE Log Bloom filter

Workflow. The workflow of the EL-BF is illustrated in Figure 5. It involves an array of 2^x+8 bytes, a hash function $hash(\cdot)$, and three mapping functions (i.e., f, f_L, f_H). In the production environment, we use MurmurHash3 algorithm [4]. It is used for mapping a word (denoted as $word$) to a 64-bit hash value (denoted as $hash(word)$). Each mapping function only consists of shift operations. The function f is used to extract the first x bits of $hash(word)$ as an $offset$ on EL-BF. Starting from $offset$, an 8-byte data field is obtained by retrieving the next eight bytes in the EL-BF. All subsequent operations related to $word$ are then conducted on this data field, which is denoted as df. The function f_H is used to extract bits $x+1$ to $x+9$ of $hash(word)$, and based on these bits, two offsets, $offset_{H1}$ and $offset_{H2}$, are obtained by indexing a pre-set array. These two offsets correspond to the high four bytes of df and their values range from 0 to 31. Similarly, the function f_L generates two offsets, $offset_{L1}$ and $offset_{L2}$, corresponding to the low four bytes of df, based on the information from bits $x+10$ to $x+18$ of $hash(word)$. When adding $word$ to the EL-BF, the four bits corresponding to $offset_{L1}$, $offset_{L2}$, $offset_{H1}$, and $offset_{H2}$ in df are set to one. When checking if the EL-BF contains $word$, if the four bits are all set to 1, $word$ is considered to be present in the EL-BF, vice versa.

Columnar-Store Format. In order to minimize EL-BF I/O and make the most of sequential read capability of OBS during queries, we design a columnar-store format for EL-BF. Figure 5 depicts a row-store format EL-BF, typically stored in memory and SFS. During the log writing process, when the row-store Bloom filters in memory accumulate to a quantity of g, forming a group, this group of row-store Bloom filters is converted into columnar-store format and flushed to OBS. The left side of Figure 6 is a group of row-store EL-BFs, and the right side is a group of columnar-store EL-BFs. Dashed arrows indicate their storage order in their respective storage media. Starting from the beginning of the EL-BF, each consecutive eight bytes are split into one piece. Therefore, each EL-BF has a total of $h = (2^x + 8)/8$ pieces. If we denote the j-th piece of the i-th Bloom filter in the group as $piece_{ij}$ $(1 <= i <= g \wedge 1 <= j <= h)$, then the storage order of this group of row-store EL-BFs is $piece_{11} \rightarrow piece_{12} \rightarrow \cdots \rightarrow piece_{1h} \rightarrow piece_{21} \rightarrow \cdots \rightarrow piece_{gh}$, and for this group of columnar-store EL-BFs, its storage order is $piece_{11} \rightarrow piece_{21} \rightarrow \cdots \rightarrow piece_{g1} \rightarrow piece_{12} \rightarrow \cdots \rightarrow piece_{gh}$.

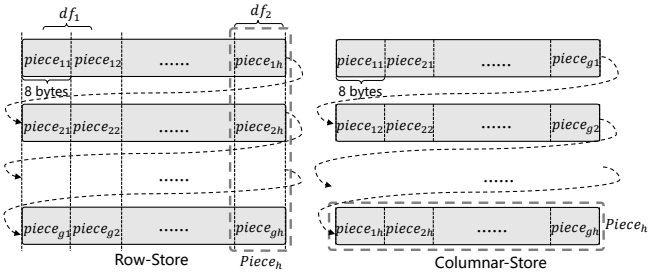

Figure 6: A Group of ESTELLE Log Bloom filters

ESTELLE Log Bloom Filter VS. Standard Bloom Filter. Compared to the standard Bloom filter, the advantages of the EL-BF are quite evident. When using the standard Bloom filter, there are two ways to check whether a word (denoted as *word*) exists in the Bloom filter. The first method involves completely sequentially scanning OBS, fetching all the entire Bloom filters into memory. However, this approach incurs significant I/O overhead. The second method uses random access to only retrieve the *word* related bits from all the Bloom filters. It's worth noting that in object storage like OBS, sequential access is several tens to several hundreds of times faster than random access. Therefore, the latency of fetching the relevant bits using this method is also comparatively high. When utilizing the EL-BF, a single sequential I/O operation enables the retrieval of all the *word* related parts of a group of Bloom filters from OBS, thereby enhancing efficiency. Since the $offset$ may land on the boundary or in the middle of a piece, there are two possible scenarios. When the data field in the EL-BF corresponding to *word* precisely overlaps with a certain piece, only a sequential retrieval of one group of pieces is required. However, when the data field spans two pieces, it is necessary to sequentially retrieve two groups of pieces. For a concrete example, let's assume that in a group of EL-BFs, $Piece_j$ is denoted as the collection comprising the j-th pieces of all Bloom filters, i.e., $Piece_j = \{piece_{1,j}, piece_{2,j}, \cdots, piece_{g,j}\}$. As shown in Figure 6, if *word* corresponds to the data field df_2 in each EL-BF, it is sufficient to fetch $Piece_h$ into memory which sequentially reads $8 \cdot g \cdot h$ bytes. If *word* corresponds to df_1, both $Piece_1$ and $Piece_2$ need to be fetched into memory, requiring a sequential read of $16 \cdot g \cdot h$ bytes.

False Positive Rate. Next, we present the theoretical false positive rate of the EL-BF. With this information, we can determine the appropriate size of the Bloom filter based on the requirements of the log business, striking a balance between performance and cost.

Lemma 3.1. *Assuming that the hash function $hash(\cdot)$ and function f select each position of data field df with equal probability and the process of adding any two words is mutually independent, given a EL-BF with a length of 2^x+8 bytes, after adding n distinct hash values to it, its false positive rate is:*

$$\epsilon = (1 - (2 \cdot \frac{4}{2^x} \cdot \frac{\binom{31}{2}}{\binom{32}{2}} + \frac{2^x - 8}{2^x} \cdot 1)^n)^4 \quad (2)$$

Proof. Due to the equiprobable mapping of *word* to any data field, and with a total of 2^x available data fields, the probability of *word* being mapped to a specific data field df is established as $\frac{1}{2^x}$. Therefore, the probabilities of a particular bit being mapped to the high four bytes of df, to the low four bytes of df, and not being mapped to df are $\frac{4}{2^x}$, $\frac{4}{2^x}$, and $\frac{2^x-8}{2^x}$, respectively. According to classical probability theory, after writing *word* into the EL-BF, the probability that a specific bit in the high four bytes or low four bytes of df remains 0 is given by $\frac{\binom{31}{1}}{\binom{32}{2}}$. The probability that a bit outside of df remains 0 is 1. Therefore, according to the Law of Total Probability, the probability that any specific bit in the EL-BF remains 0 after writing *word* into it is represented by p_0, where $p_0 = \frac{4}{2^x} \cdot \frac{\binom{31}{2}}{\binom{32}{2}} + \frac{4}{2^x} \cdot \frac{\binom{31}{2}}{\binom{32}{2}} + \frac{2^x-8}{2^x} \cdot 1$. Furthermore, since adding any two words to the EL-BF is mutually independent, the probability that a specific bit remains 0 after adding n distinct words to the EL-BF is p_0^n, and the probability that the bit becomes 1 is $1 - p_0^n$. The event of a false positive for a specific word in the EL-BF occurs as follows: after inserting n distinct hash values, a word not present in the EL-BF is queried, and it is found that the four bits in its associated data field are all set to 1. The probability of this event is $\epsilon = (1 - p_0^n)^4$. The lemma is proven.

3.2.2 Applications. To support optimizing histogram queries, AND queries, prefix fuzzy queries, we make small modifications or apply flexible adaptations to the EL-BF according to the specific needs. ELFD-BF is introduced in Section 3.3. Here, we only introduce how to utilize the Prefix Bloom filter to expedite prefix fuzzy queries.

Prefix Bloom Filter. A straightforward approach is to augment the basic index set, which only contains a single EL-BF, with an additional EL-BF dedicated to storing word prefixes. This approach not only preserves the efficiency of log writes but also enables a trade-off between cost and efficiency by controlling the size of the Prefix Bloom filter and setting an upper limit on the length of prefixes written to it. Taking the MurmurHash3 algorithm as an example of a hash function, when calculating the hash value for a given word, the algorithm traverses each byte. It converts the current byte into a 64-bit unsigned integer and performs a bitwise XOR operation with the previous result to obtain the updated result. For example, calculating the hash value for the word *hero* undergoes the following process: $hash(h) = hash(h) \oplus 0 \rightarrow hash(he) = hash(e) \oplus hash(h) \rightarrow hash(her) = hash(r) \oplus hash(he) \rightarrow hash(hero) = hash(o) \oplus hash(her)$. Therefore, inserting the intermediate results to the Prefix Bloom filter has minimal impact on the efficiency of write calculations. On the other hand, the number of hash values written into the Bloom filter and the size of the Bloom filter itself affect its false positive rate. We can control the number of hash values written into the Prefix Bloom filter by only writing the first l prefixes of each word into it. The trade-off between the size of the Prefix Bloom filter and the false positive rate can be balanced according to Equation 2.

3.3 Approximate Inverted Index.

Approximate inverted index is used to optimize histogram queries for high-frequency words and AND queries for low-frequency words. Histogram queries for high-frequency words often come with significant I/O overhead and query latency. However, in most cases, such as routine inspections, users are generally interested in an approximate aggregate result rather than an extremely precise value. Therefore, in ESTELLE Log Engine, histogram queries are designed to be progressive mode. Upon receiving a histogram query request, the log engine quickly returns an approximate histogram

result, and users can choose to stop the query at this point, reducing the scan volume and saving costs. For cases requiring extremely precise histogram query results, such as generating statistical reports, the log engine, after returning an approximate histogram result to the user, iteratively refines the result through multiple rounds to provide an accurate value. However, to achieve progressive approximate histogram queries, the rapid return of a pretty accurate approximate result is crucial. For AND queries of low-frequency words, there is often a multitude of false positives. In a basic index set containing only one EL-BF, if two low-frequency words simultaneously appear in a block but not in the same log data row, that block will still be loaded into memory for scanning, introducing significant scan latency. However, recording the row numbers that low-frequency words appear in a block in an inverted index during log writing process allows for quick intersection of the corresponding inverted lists during AND queries. This enables the rapid determination of whether there are log data rows in the block that simultaneously contain both words. Moreover, it swiftly returns the row numbers of data rows that satisfy the conditions of the AND query within the block, thereby circumventing a portion of the line-by-line scanning process. Unfortunately, however, directly creating an inverted index for each block would result in an index size comparable to the block's size. This not only increases the cost of storing the index but also introduces significant I/O overhead during queries.

To address these two challenges, as shown in Figure 7, we design a fixed-size Approximate Inverted Index. The high-frequency part is used to quickly return an approximate histogram result, while the low-frequency part is utilized to optimize AND queries for low-frequency words. Next, this section will sequentially introduce the ELFD-BF upon which the approximate inverted index depends, the layout of the approximate inverted index, its workflow, its columnar-store format, and the theoretical values of the probabilities of determining whether there are data rows in a block that meet the query criteria using only the approximate inverted index in low-frequency word AND queries.

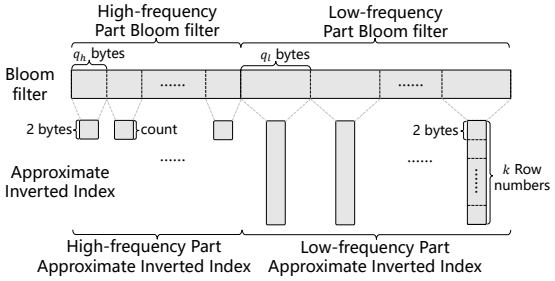

Figure 7: Approximate Inverted Index

Frequency Division Bloom filter. As shown in the upper part of Figure 7, the Frequency Division Bloom filter (abbreviated as ELFD-BF) is essentially formed by concatenating two EL-BFs, where the sizes of the high-frequency and low-frequency parts are $2^{x_h} + 8$ and $2^{x_l} + 8$ bytes, respectively. The number of hash values for high-frequency words in a block is significantly smaller than the number for low-frequency words, so generally, x_h is less than x_l. The specific values of x_h and x_l can be determined based on Equation 2. For instance, setting x_h to 13 (approximately 8KB for the high-frequency part) results in a false positive rate lower than 7.8386×10^{-4} after writing fewer than 3000 distinct hash values of high-frequency words. Setting x_l to 18 (approximately 256KB for the low-frequency part) leads to a false positive rate less than 4.0083×10^{-6} after writing fewer than 24000 distinct hash values of low-frequency words.

Layout of Approximate Inverted Index. As shown in Figure 7, the high-frequency and low-frequency parts of the ELFD-BF correspond to high-frequency and low-frequency parts of the approximate inverted indexes, respectively. The high-frequency part Bloom filter starts from the beginning, with every q_h bytes corresponding to an approximate inverted list. Each of these lists stores only a 2-byte count value, indicating how many hash values are mapped to that approximate inverted list. The low-frequency part of the Bloom filter starts from the beginning, with every q_l bytes corresponding to an approximate inverted list. Each of these lists stores fixed length k row numbers. Each row number consists of 2 bytes, recording which row the word corresponding to the hash value mapped to the inverted list is within the block.

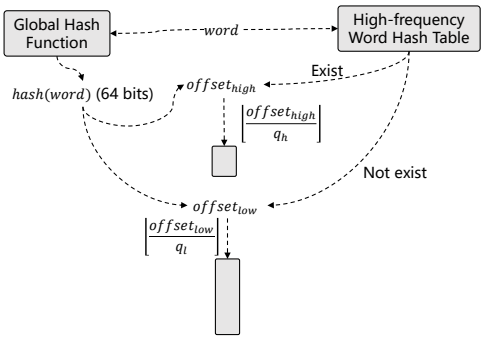

Figure 8: The Workflow of Approximate Inverted Index

Workflow. The workflow of approximate inverted index is shown in Figure 8. If a word (denoted as *word*) exists in the high-frequency word hash table, it is considered as a high-frequency word, and it will be associated with the high-frequency part of both the ELFD-BF and the approximate inverted index, vice versa. Assuming *word* is a high-frequency word, by using hash function $h(\cdot)$ and mapping function f, we can get an offset $offset_{high}$ in its corresponding Bloom filter. The $offset_{high}$ denotes the start position of the data field df_{high} associated with *word*. Then, $hash(word)$ can be written into df_{high} in the Bloom filter (refer to Section 3.2.1). At this point, we define that *word* corresponds to the $\lfloor \frac{offset_{high}}{q_h} \rfloor$-th approximate inverted list. We use the same method to write a low-frequency word to the low-frequency part of the ELFD-BF and obtain its corresponding approximate inverted list, i.e., the $\lfloor \frac{offset_{low}}{q_l} \rfloor$-th list. For the list in the high-frequency part, each time a hash value is mapped to it, its stored count value increases by one. However, for the list in the low-frequency part, in order to fix its length, only the row numbers corresponding to the first k hash values mapped to it are recorded. During an AND query, if the two lists corresponding to the two words are both full and no qualifying data row is found after intersecting the two lists, it

cannot be determined that there are no qualifying data rows in that block. Therefore, a higher k value increases the likelihood of definitively identifying the presence of qualifying data rows in a block using only the approximate inverted index. The setting of k can be referred to in Section 3.3. Parameters q_h and q_l directly control the number of inverted lists and can be determined based on a consideration of the trade-off between collision rate and cost. Each approximate inverted index has $h_h = \lfloor \frac{2_h^x+8}{q_h} \rfloor$ and $h_l = \lfloor \frac{2_l^x+8}{q_l} \rfloor$ lists for its high-frequency and low-frequency parts, respectively. Assuming that the hash function $hash(\cdot)$ and mapping function f uniformly select each position in the data field of the Bloom filter, the average number of hash values mapped into an approximate inverted list in the high-frequency and low-frequency parts is $\frac{n_h}{h_h}$ and $\frac{n_l}{h_l}$ respectively. Here n_h and n_l denote the number of distinct hash values of high-frequency and low-frequency words in a block, respectively. For high-frequency words, to avoid excessively high recorded count values due to excessive collisions, their conflict rate is typically kept lower. For example, with $q_h = 2$, if a block has fewer than 3000 distinct high-frequency words, the average hash values per approximate inverted list are under 0.7324. For low-frequency words, mapping only one hash value on average to each list is inefficient. Hence, setting q_l higher is advisable. This method effectively balances performance with cost-effectiveness.

Columnar-Store Format. The design goal and approach of the columnar-store approximate inverted index are similar to that of the EL-BF. Each approximate inverted list is considered a piece, with further operations aligning with those in Section 3.2.1.

Theoretically Useful Probability. This value represents the likelihood that using only the approximate inverted index can identify whether there are data rows in a block that meet the condition of a low-frequency word AND query. It serves as the theoretical foundation for setting the parameter k.

Lemma 3.2. *Assuming a block contains n data rows, the probabilities of a data row containing words w_a and w_b are p_a and p_b respectively, and these two events are mutually independent. Additionally, it is assumed that the low-frequency part of the approximate inverted index's inverted list for this block records k row numbers. Then the probability of misjudgment when using only the Bloom filter but being able to filter out the block using only the approximate inverted index is:*

$$p_u = \frac{\sum_{m_1=1}^{k}\sum_{m_2=1}^{k}\binom{n}{m_1}p_a^{m_1}(1-p_a)^{n-m_1}\cdot\binom{n-m_1}{m_2}p_b^{m_2}(1-p_b)^{n-m_2}}{\sum_{m_1=1}^{n}\sum_{m_2=1}^{n}\binom{n}{m_1}p_a^{m_1}(1-p_a)^{n-m_1}\cdot\binom{n-m_1}{m_2}p_b^{m_2}(1-p_b)^{n-m_2}} \quad (3)$$

Proof. Let's define:

- Event A as: w_a and w_b do not simultaneously appear in any data row within the block.
- Event B as: w_a appears m_1 times and w_b appears m_2 times in the block.

Then, according to the binomial distribution, $P(B)$ is given by:

$$P(B) = \binom{n}{m_1}p_a^{m_1}(1-p_a)^{n-m_1}\cdot\binom{n}{m_2}p_b^{m_2}(1-p_b)^{n-m_2} \quad (4)$$

From the classical probability model, the probability of Event A occurring under the condition of Event B is given by:

$$P(A|B) = \frac{\binom{n}{m_1}\cdot\binom{n-m_1}{m_2}}{\binom{n}{m_1}\cdot\binom{n}{m_2}} \quad (5)$$

Then, according to the conditional probability formula:

$$\begin{aligned} P(AB) &= P(B)\cdot P(A|B) \\ &= \binom{n}{m_1}p_a^{m_1}(1-p_a)^{n-m_1}\cdot\binom{n-m_1}{m_2}p_b^{m_2}(1-p_b)^{n-m_2} \end{aligned} \quad (6)$$

The event AB means that w_a appears m_1 times, w_b appears m_2 times in the block, and they do not simultaneously appear in any data row within the block. When m_1 and m_2 are both greater than 0 and less than k, we can have a misjudgment when using only the Bloom filter, but the block can be filtered out using only the low-frequency part of the approximate inverted index. Let $P(AB) = P(m_1, m_2)$, then $p_u = \frac{\sum_{m_1=1}^{k}\sum_{m_2=1}^{k}P(m_1,m_2)}{\sum_{m_1=1}^{n}\sum_{m_2=1}^{n}P(m_1,m_2)}$. The lemma is proven.

Based on Equation 3, setting k to 18 ensures that in an AND query, if the probability of both queried words appearing in a data row does not exceed 0.0003, then the likelihood of using only the approximate inverted index to determine whether a block contains data rows with both words appearing simultaneously is not less than 0.9996.

4 QUERY OPTIMIZATION

This section primarily introduces the processes of several types of queries. Firstly, the shared query process at shard-level is introduced, followed by individual descriptions of specific aspects for each type of query.

4.1 Shard-Level Query Process

During the allocation of sub-requests, the Coordinator performs the first-level time filtering on shards based on the information from the Shard Group, assigning only the shards that meet the time conditions to its subquery nodes. Within the subquery node, the worker initially performs the second-level time filtering on blocks based on the Min/Max Timestamp in the Meta Data. Following this, it proceeds with word filtering on the remaining blocks, involving two steps: index filtering and scan filtering. Index filtering is the process of fetching relevant indexes from OBS or SFS into memory and filtering out blocks that do not meet the query conditions. Scan filtering involves fetching the blocks after index filtering into memory, decompressing them, and then scanning data rows that meet the query conditions line by line. The scan filtering provides shard-level query results based on the query conditions. Index filtering is a critical step for reducing query latency and represents the most significant distinction in the query process across various types of queries.

4.2 Full-Text and Prefix Queries

The index filtering and scan filtering logic for full-text queries and prefix fuzzy queries is essentially the same. Taking full-text queries as an example, each worker firstly uses ESTELLE Log Bloom filter or Frequency Division Bloom filter to eliminate blocks that definitely do not contain the queried word. Then, the remaining blocks are fetched into memory, decompressed, and scanned line by line. For prefix fuzzy queries, index filtering is done by Prefix Bloom filter.

4.3 AND Queries

When worker performs index filtering for an AND query, it firstly uses the Frequency Division Bloom filter to filter out blocks that

definitely do not simultaneously contain both of the specified words. Then, it performs a second-level filtering based on the approximate inverted index:

- If both words are mapped to the high-frequency part or are mapped to the high-frequency and low-frequency parts respectively, the second-level index filtering is terminated, and scan filtering is initiated directly.
- If both words are mapped to the low-frequency part, intersect the inverted lists corresponding to the two words:
  - If the intersection is empty and at least one of the two inverted lists is not full, discard the block.
  - If the intersection is empty but both inverted lists are full, record the maximum value of the row numbers in the two inverted lists. When the worker performs scan filtering on this block, start scanning from the next row after that maximum row.
  - If the intersection is not empty and at least one of the inverted lists is not full, the block only needs to consider the rows corresponding to the row numbers in the intersection as results during scan filtering, without any scanning row by row.
  - If the intersection is not empty and both inverted lists are full, record the row numbers in the intersection for direct retrieval of corresponding data rows during scan filtering, and record the maximum value of the row numbers in the inverted lists. During scan filtering, start scanning from that maximum row.

4.4 Progressive Approximate Histogram Queries

The histogram query in ESTELLE Log Engine is configured in a progressive query mode, consisting of two phases: approximate query and sample correction.

In the approximate query phase, each worker firstly filters out blocks that definitely do not contain the queried word based on the ELFD-BF. Then, it checks the approximate inverted index:

- If the word is mapped to the high-frequency part, the count value recorded in that inverted list is directly returned. Coordinator adds together these count values from all workers and returns this result to the upper-level application.
- If the word is mapped to the low-frequency part, the worker will firstly assess the cost of scanning the remaining blocks after filtering:
  - If the number of remaining blocks is relatively small, it will scan all of them and return an accurate count value. Coordinator adds together all these accurate count values and returns the aggregated result. Then, the query execution concludes.
  - If the number of remaining blocks is still significant, it will directly enter the sample correction phase.

The sample correction phase requires multiple iterations. In each iteration, each worker samples a proportion of data rows from each block, scans and sums the counts, and returns the accurate count value for that shard. The coordinator scales all obtained count values proportionally to get the count result for the current iteration. Since the coordinator stores the intermediate results and workers do not resample data rows that have already been sampled, the final result after the iterations is an accurate count.

5 EXPERIMENT EVALUATION

5.1 Experimental Setup

Device. All the experiments are performed on an ECS [10] virtual machine in Huawei Cloud. It has 8 CPU cores and 32GB RAM.

Dataset. The logs of cluster1-worker1 from the Hadoop-14TB logs [22] are selected as the base dataset, containing a total of 84.7 GB of data. By replicating it 10 times, we generate 1030 files with a combined size of 847 GB, utilized as the log data for our experiment.

Baselines. We compare the following baseline log engines:

1) ES [3]. Elasticsearch is an open-source, distributed full-text search and analytics engine, widely used in the fields of log and real-time data analysis. We used Elasticsearch 7.6 in our experiments.

2) Doris [2]. Doris is an open-source, distributed, and scalable big data analytics engine, commonly utilized for real-time analytics and interactive query. We use doris 2.0.0 in our experiments.

3) CK [39]. ClickHouse is an open-source, column-oriented database management system designed for OLAP, which is popular in scenarios that require fast and efficient processing of large datasets. We use ClickHouse 22.12 in our experiments.

4) EST-b. Denotes ESTELLE-basic. This is the ESTELLE Log Engine equipped with only an EL-BF corresponding to a block.

5) EST. EST denotes ESTELLE. This is the ESTELLE Log Engine equipped with complete index set for a block, where the prefix Bloom filter sets no limit on the length of the prefixes.

In our experiments for ESTELLE-basic and ESTELLE, we use 16GB SFS [13] as temporary storage, 1GB SSD disk as local cache and OBS as the permanent storage of logs and indexes. For other baselines, we use enough Ultra-high I/O EVS Disk [11] for local cache and the permanent storage of logs and indexes, noting their higher cost compared to OBS. We use the tokenbf_v1 for CK indexing and inverted index for ES and Doris. All engines compress data with LZ4 [28], sort by timestamps, and disable the result cache, while ES retains other caches like index and shard-level data caches. Load tests are conducted using Wrk [17] and Wrk2 [34].

5.2 Experimental Results

5.2.1 Writing Performance. ESTELLE-basic and ESTELLE adopt an append write mode without compaction during the writing process, and the other baselines write and compact concurrently. To compare the writing performance of the mentioned log engines, the dataset from cluster1-worker1 is written to each log engine ten times, and the average single-core CPU write speed is calculated as the final experimental result.

The left side of Figure 9 illustrates the single-core CPU write speeds of the aforementioned log engines. As shown in Figure 9, the two log engines that build inverted indexes in real-time have slower single-core CPU write speeds. Between them, ES requires a more detailed inverted index to support complex text analysis, resulting in the slowest write speed. In our experiments, Doris initially achieves a single-core write speed of nearly 20MB/s, but as data continues to be written, its write speed decreases significantly. The other three index-free architecture log engines have faster single-core CPU write speeds. Due to the nearly lock-free writing process, adopting

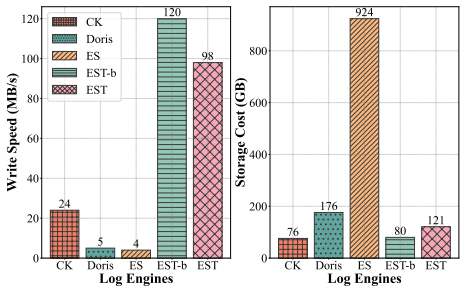

Figure 9: The Single-Core Write Speeds and Storage Costs

an OBS-friendly append write mode, and no compaction operation, the single-core CPU write speeds of ESTELLE-basic and ESTELLE are approximately 5 times and 4 times that of CK, respectively.

5.2.2 Storage Cost. To compare the storage cost of each log engine, we use the total size of the written and compressed logs and indexes as the evaluation criterion. The right side of Figure 9 illustrates the storage costs of the aforementioned log engines. As shown in Figure 9, Doris and ES, both employing inverted indexes, incur the highest storage costs, with ES being 5.25 times that of Doris. This is attributed to the fact that Doris optimizes its inverted index, and it adopts the columnar storage, reducing data redundancy and being more conducive to the effectiveness of compression algorithms. The storage costs of the other three log engines with index-free architecture are relatively lower. In our experiments, the size of the Bloom filter for ESTELLE-basic and ESTELLE is set larger than CK, and there is no compaction operation during the writing processes of ESTELLE-basic and ESTELLE. As a result, CK has the least storage space consumption, with ESTELLE-basic being comparable. ESTELLE incurs slightly higher storage costs than ESTELLE-basic due to the addition of two ESTELLE Log Bloom filters and an Approximate Inverted Index for each block. However, it's important to note that both ESTELLE-basic and ESTELLE use OBS as the storage medium, making their overall storage costs significantly lower compared to other log engines.

5.2.3 Query Performance. In order to assess the support of the aforementioned log engines for various scenarios and queries, we first construct three types of words: high-frequency words, low-frequency words, and non-existent words. Initially, we tokenize the entire dataset of logs and record the frequency of each word. Subsequently, we select 20 words from the frequency range [0.1, 1] as high-frequency words, 20 words from the frequency range (0, 0.00001] as low-frequency words, and arbitrarily choose 20 words that do not exist in the dataset as non-existent words. The experimentation for each word or word combination is repeated 50 times, and the average of these repetitions is computed to derive the final result. For each query, whether it's for an individual word or a combination of words, we use a load testing approach where we set the concurrency to 1. We conduct a 15-minute load test and then measure the average query latency and the average CPU usage ratio. For the term queries in our experiments, including full-text queries, prefix fuzzy queries, and AND queries, we use the "order by time" in each query statement and set the limit to 10.

Full-Text Query. The performance of full-text queries is shown in Table 1. As shown in this table, the query performance of the two log engines with inverted index is quite stable across three scenarios. ES has the highest overall full-text query performance. The three log engines using the index-free architecture perform better in high-frequency word scenarios than in low-frequency word and non-existent word scenarios. This is because when executing queries containing 'order by time', it is necessary to scan Bloom filters and blocks to find the relevant logs until the limit number of logs is reached. However, once the limit number of logs is accumulated, it is then possible to filter out many blocks that do not meet the time criteria using only the timestamp information in the metadata. This avoids numerous fetches and scans of Bloom filters and blocks. For high-frequency word queries, fewer blocks and Bloom filters need to be scanned to accumulate the limit number of logs, allowing for an earlier transition to the time pruning phase. In the high-frequency words scenario, CK and ESTELLE-basic/ESTELLE exhibit comparable query performance. However, in scenarios with low-frequency words and non-existent words, CK's query performance is significantly inferior to the other two log engines. Since CK takes an excessively long time when querying non-existent words, it mainly suggests that CK's Bloom filter has a higher rate of false positives. Additionally, since the Bloom filter of CK involves heavier I/O during queries, its performance in these two scenarios is significantly worse compared to ESTELLE-basic/ESTELLE. ESTELLE-basic/ESTELLE maintains a full-text query latency of less than one second in all scenarios.

Table 1: Performance of Full-Text Query

Engines		ES	Doris	CK	EST-b/EST
Low	Latency(ms)	56	114	15630	765
	CPU(%)	1.25	2.25	92.25	85
High	Latency(ms)	23	225	122	156
	CPU(%)	3	3.75	5	2
Non	Latency(ms)	15	174	214000	135
	CPU(%)	1.9	2.4	70.8	50

Prefix Fuzzy Query. The performance of prefix fuzzy queries is shown in Table 2. We do not find commands related to prefix fuzzy queries in Doris's documentation, therefore we only compared the other four log engines. CK and ESTELLE-basic are not optimized for prefix queries, and they can only perform brute-force scans. As shown in Table 2, ES achieves the best performance in all scenarios. In our experiments, we find that the latency of the first query in each scenario for ES is greater than or equal to 40 seconds. Its minimal query latency is mainly due to its numerous unclosed caches. However, this approach is not practical in scenarios with massive amounts of data. Among the three log engines using the index-free architecture, ESTELLE offers the best support for prefix fuzzy queries. In scenarios with low-frequency prefixes, the CPU usage of all three engines is comparable, but the latency of CK and ESTELLE-basic is much higher than that of ESTELLE. This means that the presence of Prefix Bloom filter can significantly reduce the amount of line-by-line scan filtering. CK and ESTELLE-basic lack support for scenarios involving non-existent prefixes, whereas ESTELLE can handle them. In high-frequency prefix scenarios, nearly all block meet the query condition. As a result, ESTELLE involves an additional step of scanning the prefix Bloom filter compared to

ESTELLE-basic, ESTELLE's query latency is slightly higher than ESTELLE-basic's. However, this difference of a few hundred milliseconds is negligible in terms of user experience.

Table 2: Performance of Prefix Fuzzy Query

Engines		ES	CK	EST-b	EST
Low	Latency(ms)	15	16780	6483	745
	CPU(%)	0.46	89	85	85
High	Latency(ms)	50	438	134	623
	CPU(%)	1	3.8	3.1	4.3
Non	Latency(ms)	16	Timeout	Timeout	1100
	CPU(%)	0.15			98.75

And Query. The performance of AND queries is shown in Table 3. As shown in this table, in scenarios involving AND queries between low-frequency words, the performance of the two log engines using inverted indexes is higher than that of the three log engines using the index-free architecture. In scenarios involving AND queries between high-frequency words, the performance of both is comparable. Among the three log engines using the index-free architecture, CK and ESTELLE-basic have comparable performance, while ESTELLE-basic offers a slightly better user experience. The use of approximate inverted indexes makes ESTELLE's query latency in low-frequency word AND query scenarios far superior to both, even comparable to ES, which uses inverted indexes, resulting in a good user experience.

Table 3: Performance of AND Query

Engines		ES	Doris	CK	EST-b	EST
Low&Low	Latency(s)	4.18	0.095	193	149.76	5.74
	CPU(%)	1.96	1.37	91.37	83.75	93.75
High&High	Latency(s)	5.8	0.18	0.486	0.11	0.17
	CPU(%)	3.37	3.81	5	5	4

Histogram Query. The performance of histogram queries for high-frequency words is shown in Table 4. As shown in this table, due to the heavy and high false positive rate of CK's Bloom filter, it incurs additional I/O overhead during queries, making it the least supportive for histogram queries among these five log engines. Apart from ES, all other engines use a vectorized execution engine. Therefore, Doris, ESTELLE-basic, and ESTELLE have lower latency in querying exact histogram values compared to ES. Doris achieves the best performance in exact histogram queries due to its lighter inverted list during queries. However, as the volume of data increases, the latency for exact histogram queries rapidly rises for all log engines, while the latency for approximate queries remains relatively stable. In our experiments, ESTELLE returns relatively accurate results for approximate queries in less than 1 second. The average errors between these 20 approximate query results and the accurate count values for high-frequency words are all less than 0.02. This is because, after deduplication, the proportion of high-frequency words in each block is extremely low, averaging less than 0.07 in our experimental dataset. In the experiments, each block's corresponding Frequency Division Bloom filter for the high-frequency part has ample space to store these high-frequency words, resulting in a extremely low conflict rate. Therefore, the values returned by fast approximate queries are relatively accurate.

Table 4: Performance of Histogram Query

Engine		Latency(s)	CPU(%)
ES		39	27
Doris		7.9	14.5
CK		192	93.12
EST-b	apx	1.59	1
	acrt	24	97.5
EST	apx	0.3	98
	acrt	24	97.5

6 RELATED WORK

We categorize existing log engines into three types: no-index, index-based, and index-free architectures. No-index engines, like Scalyr [23], prioritize writing speed but lack query optimization. Loki [18] indexes only log tags, while SLS [9] offers a no-index mode. Most engines, including Doris [2], Splunk [24], ElasticSearch [3], SLS [9], LogStore [7] and TencentCLS [40], use inverted indexes for quick querying but face high storage costs and I/O overhead due to data volume and the variable length nature of inverted lists. Cloud-native engines like LogStore and TencentCLS support various inverted indexes [21, 33] but are not open source. ClickHouse [39] represents an index-free approach, using Bloom filters for efficient indexing with minimal impact on writing speed.

Over decades, various Bloom filter variants have emerged for different needs. The standard Bloom filter [6] is a compact structure for set membership tests, with innovations like the Counting Bloom filter [15] allowing for element deletion, the Compressed Bloom filter [30] reducing storage needs, the Partitioned Bloom filter [8] lowering false positives and aiding parallelism, and the Dynamic Bloom filter [20] adjusting to dataset size changes. Recent developments include machine learning-enhanced filters [36] for accuracy and elastic filters [38] for flexibility and deletability. Despite these advancements, no variant specifically addresses cloud-native log engines, a gap our ESTELLE Log Bloom filter aims to fill.

7 CONCLUSION

In this paper, we propose a cost-effective cloud-native log engine, called ESTELLE, equipped with a low-cost pluggable log index framework. The use of Object Storage (OBS) enables this log engine to achieve low storage costs. Its design of an near-lock-free writing process allows for high single-core CPU write speeds. Tailor-made ESTELLE Log Bloom filters and approximate inverted indexes ensure low query latency across various scenarios and queries. Overall, the ESTELLE Log Engine is exceptionally suitable for cloud-based log scenarios involving massive data with high-frequency writing and storage, and low-frequency querying and analysis, offering excellent cost-effectiveness.

ACKNOWLEDGMENTS

This work is partially supported by Shenzhen Municipal Science and Technology R&D Funding Basic Research Program (JCYJ20210324133607021), and Municipal Government of Quzhou under Grant (No. 2022D037, 2023D044), and Key Laboratory of Data Intelligence and Cognitive Computing, Longhua District, Shenzhen.

REFERENCES

[1] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. 2013. Cloud monitoring: A survey. *Computer Networks* 57, 9 (2013), 2093–2115.

[2] Apache. 2023. *Doris.* https://doris.apache.org/

[3] Apache. 2023. *ElasticSearch.* https://www.elastic.co/cn/elastic-stack

[4] Austin Appleby. 2016. *Murmurhash3.* https://github.com/aappleby/smhasher/wiki/MurmurHash3

[5] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software* 33, 3 (2016), 42–52.

[6] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[7] 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.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[9] Alibaba Cloud. 2023. *SLS.* https://www.alibabacloud.com/product/log-service

[10] HuaWei Cloud. 2023. *ECS.* https://www.huaweicloud.com/intl/zh-cn/product/ecs.html

[11] HuaWei Cloud. 2023. *EVS.* https://www.huaweicloud.com/intl/en-us/product/evs.html

[12] HuaWei Cloud. 2023. *OBS.* https://www.huaweicloud.com/intl/zh-cn/product/obs.html

[13] HuaWei Cloud. 2023. *SFS.* https://www.huaweicloud.com/intl/zh-cn/product/sfs.html

[14] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 1285–1298.

[15] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking* 8, 3 (2000), 281–293.

[16] Kaniz Fatema, Vincent C Emeakaroha, Philip D Healy, John P Morrison, and Theo Lynn. 2014. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *J. Parallel and Distrib. Comput.* 74, 10 (2014), 2918–2933.

[17] Will Glozer. 2021. *wrk.* https://github.com/wg/wrk

[18] Grafana. 2023. *Loki.* https://grafana.com/docs/loki/latest/

[19] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 1–16.

[20] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. 2009. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (2009), 120–133.

[21] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity: 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007. Proceedings 14*. Springer, 124–138.

[22] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

[23] Scalyr inc. 2023. *Scalyr home page.* https://www.dataset.com/

[24] Splunk inc. 2023. *Splunk Enterprise.* https://www.splunk.com/

[25] Shashank Mohan Jain. 2020. Linux Containers and Virtualization. *A Kernel Perspective* (2020).

[26] Joanna Kosińska, Bartosz Baliś, Marek Konieczny, Maciej Malawski, and Sławomir Zieliński. 2023. Towards the Observability of Cloud-native applications: The Overview of the State-of-the-Art. *IEEE Access* (2023).

[27] Joanna Kosińska and Krzysztof Zieliński. 2022. Experimental evaluation of rule-based autonomic computing management framework for cloud-native applications. *IEEE Transactions on Services Computing* 16, 2 (2022), 1172–1183.

[28] lz4. 2023. *lz4.* https://github.com/lz4

[29] Nicolas Marie-Magdelaine. 2021. *Observability and resources managements in cloud-native environnements.* Ph. D. Dissertation. Bordeaux.

[30] Michael Mitzenmacher. 2001. Compressed bloom filters. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 144–150.

[31] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[32] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2017. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* 7, 3 (2017), 677–692.

[33] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *Advances in Spatial and Temporal Databases: 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 2003. Proceedings 8*. Springer, 46–65.

[34] Jon Richards. 2019. *wrk2.* https://github.com/giltene/wrk2

[35] Hassan Jamil Syed, Abdullah Gani, Raja Wasim Ahmad, Muhammad Khurram Khan, and Abdelmuttlib Ibrahim Abdalla Ahmed. 2017. Cloud monitoring: A review, taxonomy, and open research issues. *Journal of Network and Computer Applications* 98 (2017), 11–26.

[36] Kapil Vaidya, Eric Knorr, Tim Kraska, and Michael Mitzenmacher. 2020. Partitioned learned bloom filter. *arXiv preprint arXiv:2006.03176* (2020).

[37] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. 2023. LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 452–468.

[38] Yuhan Wu, Jintao He, Shen Yan, Jianyu Wu, Tong Yang, Olivier Ruas, Gong Zhang, and Bin Cui. 2021. Elastic bloom filter: deletable and expandable filter using elastic fingerprints. *IEEE Trans. Comput.* 71, 4 (2021), 984–991.

[39] Yandex. 2023. *ClickHouse.* https://clickhouse.com/

[40] Muzhi Yu, Zhaoxiang Lin, Jinan Sun, Runyun Zhou, Guoqiang Jiang, Hua Huang, and Shikun Zhang. 2022. TencentCLS: the cloud log service with high query performances. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3472–3482.

[41] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.