TencentCLS: The Cloud Log Service with High Query Performances

Muzhi Yu
Peking University
Beijing, China
muzhi.yu@pku.edu.cn

Zhaoxiang Lin
Tencent Cloud Computing (Beijing) Co., Ltd.
Beijing, China
zlinzlin@tencent.com

Jinan Sun
Peking University
Beijing, China
sjn@pku.edu.cn

Runyun Zhou
Tencent Cloud Computing (Beijing) Co., Ltd.
Beijing, China
runyunzhou@tencent.com

Guoqiang Jiang
Tencent Cloud Computing (Beijing) Co., Ltd.
Beijing, China
johnqjiang@tencent.com

Hua Huang
Tencent Cloud Computing (Beijing) Co., Ltd.
Beijing, China
danielhuang@tencent.com

Shikun Zhang
Peking University
Beijing, China
zhangsk@pku.edu.cn

ABSTRACT

With the trend of cloud computing, the cloud log service is becoming increasingly important, as it plays a critical role in tasks such as root cause analysis, service monitoring and security audition. To meet these needs, we provide Tencent Cloud Log Service (TencentCLS), a one-stop solution for log collection, storage, analysis and dumping. It currently hosts more than a million tenants, of which the largest ones can generate up to PB-level logs per day.

The most important challenge that TencentCLS faces is to support both low-latency and resource-efficient queries on such large quantities of log data. To address that challenge, we propose a novel search engine based upon Lucene. The system features a novel procedure for querying logs within a time range, an indexing technique for the time field, as well as optimized query algorithms dedicated to multiple critical and common query types.

As a result, the search engine at TencentCLS gains significant performance improvements against Lucene. It achieves 20x performance increase with standard queries, and 10x performance increase with histogram queries in massive log query scenarios. In addition, TencentCLS also supports storing and querying with microsecond-level time precision, as well as the microsecond-level time order preservation capability.

PVLDB Reference Format:
Muzhi Yu, Zhaoxiang Lin, Jinan Sun, Runyun Zhou, Guoqiang Jiang, Hua Huang, and Shikun Zhang. TencentCLS: The Cloud Log Service with High Query Performances. PVLDB, 15(12): 3472 - 3482, 2022.
doi:10.14778/3554821.3554837

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554837

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/aur3l14no/PVLDB2022_TencentCLS.

1 INTRODUCTION

With the trend of cloud computing, cloud log service has become increasingly popular. Cloud log services significantly simplify log collection and analysis, and lay solid foundations for applications such as root cause analysis, service monitoring and security audition.

Cloud log services also have high business values and therefore have attracted many companies. Not only there have been commercially successful enterprises dedicated in log services, such as Splunk [9] and Elastic [6], but also many cloud vendors have launched their own log service product [1, 4, 5].

Tencent Cloud Log Services (TencentCLS) [10] is the log service product provided at Tencent Cloud, and it has experienced rapid growth in the past year (500% anual growth).

In this paper, we describe some characteristics and challenges of the business scenarios faced by TencentCLS, and elaborate on its architecture and designs. We also provide detailed experimental evaluations to demonstrate the effect of some major design choices we made with TencentCLS.

The TencentCLS business scenarios have the following characteristics and challenges.

Heavy and Skewed Log Writes

The logs stored in TencentCLS are of large and skewed quantity. Currently, TencentCLS has millions of log topics, about 10% percent of which are monthly active.

The logs collected per day for the active topics are highly skewed. Concretely, the top topic has more than 100 billion logs collected per day while 90% of the active topics generate less than 10 million logs per day each.

Heavy and Skewed Log Queries

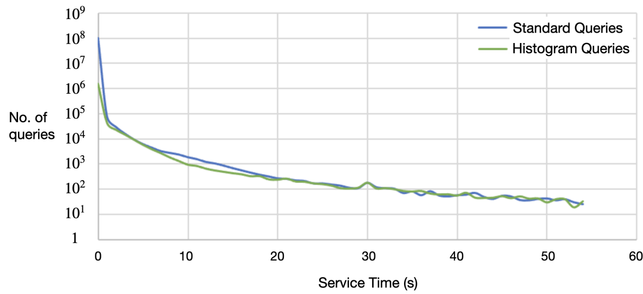


Figure 1: The latency distribution of different types of queries

Not only the quantity of logs has a skewed distribution but also the query latency. As is shown in Figure 1, although the average latency of queries is below 1 second, there is a long-tail effect where some of the queries take up to 30 seconds or even time out.

More challenging is that querying on such large data is inherently costly. To be more precise, suppose that we use Lucene [13] as the search engine of TencentCLS, the index of the timestamp field of 10 billion logs would have the size of around 30 GB. Even loading the index from a disk drive that has the speed of 150 MB/s takes up a total of 200 seconds.

According to the data of online business, around 95% of the queries are limited to the logs of the last 24h. In order for those queries to be answered in less than 30 seconds, the number of the logs written on each drive per day has to be below 1.5 billion. Therefore, for the top topic that generates 100 billion logs per day, a total of at least 67 disks (with a speed of 150 MB/s) are required, which is very costly.

Histogram Queries are Common

In addition, for each query, TencentCLS shows the distribution of logs in time that meets the conditions. We call the queries that support such visualization histogram queries, which collect the counts of hits in different time segments. Histogram queries are extremely common, but also resource demanding.

The above challenges can be worse if we use higher precision for timestamps, because the index of the timestamp will grow larger as the precision increases, and ultimately slowing down the query processes.

Therefore, it is a both necessary and challenging task to design a system that supports low-latency queries on these large-quantity, highly-skewed log data.

Our solution is a novel log search engine featuring time series index. It is based on Lucene and is optimized especially for log data. Compared with vanilla Lucene, it has the following main differences.

(1) We keep the documents sorted according to their timestamps.
(2) We design an time series index dedicated for the time field.
(3) We design a search algorithm dedicated for tail queries (queries that are expected to return the last few hits).
(4) We optimize the histogram queries (queries that are expected to return the distribution of number of logs in time).

Thanks to the design, TencentCLS significantly lowers the query latency, and supports microsecond-level precision for timestamps with little cost. In the scenario described above, we only need 3 disks instead of 67 to achieve the same query latency.

It is worth noting that higher timestamp precision not only allows storing and querying with higher precision, but also keeps the retrieved logs in microsecond-level order. This is useful when applications generate multiple logs in the same second. With TencentCLS, it is more probable that those logs are retrieved in the same order as they were written.

We have conducted detailed experiments using the open benchmark to demonstrate the superiority of our solution compared to Lucene. Generally, our solution gains 7.5x to 38x performance increases, depending on the types of queries. More detailed comparisons and analyses are shown in the experimental evaluation section. In addition to the open benchmark, we have also show the performance increases using the real world data collected by Cloud Log Service at Tencent.

The paper is organized as follows. Section 2 gives the background for the log search solutions. Section 3 describes the overall architecture of TencentCLS and its modules. Section 4 elaborates on the design of the search engine of TencentCLS. Section 5 provides both offline and online experimental evaluation of the search engine of TencentCLS.

2 BACKGROUND

Currently, there are many search engines in the industry, including Lucene [2, 13] and its variants [17, 22], Sphinx [8], Xapian [11], MG4J [7], etc. The most widely used are Lucene-based products and Sphinx. We list the characteristics of each solution below, explain why we choose to base the search engine of TencentCLS on Lucene, and describe a few weaknesses of Lucene that we need to address.

2.1 Search Engine Options

Sphinx [8] is a search engine library that features high-speed index generation and high-speed distributed search. It has a good support for MySQL. However, as a static search engine, Sphinx is neither suitable for real-time search, nor scenarios with frequent data updates. It also suffers from a high disk IO overhead.

Xapian [11] shares a similar design with Lucene, and also provides a rich and extensible set of API. It even achieves higher query performances than Lucene. However, it lacks the concepts of fields or columns, which is rather different from traditional databases.

Lucene [2], on the other hand, can generate indexes on fields and support retrieval by fields. It also supports real-time index generation, query and update. Lucene itself has excellent object-oriented system architecture and integrates a powerful search engine. Based on Lucene, Solr and ElasticSearch both support distributed storage and distributed query.

Solr [3] was the most mature and stable Lucene-based indexing component before ElasticSearch. However, **ElasticSearch** become far more popular than Solr after its launch, because ElasticSearch has more powerful real-time search capabilities as well as other advantages including easy-to-use, near-zero configuration, friendly RESTful query interface, convenient cluster deployment and management.

ElasticSearch also has the following features.

- High availability: ES supports indexing on shards and multi-node distributed query.
- Persistency: ES supports multi-machine backup, self-monitoring and balancing.
- Scalability: ES supports discovering and joining new nodes and horizontal auto scaling. Node failures do not affect the cluster.

Due to its many advantages and rapid development, ElasticSearch now enjoys a wide and vibrant community, and has attracted many well-known enterprises users as well as many startups.

After comparative analyses, we decided to use Lucene / ElasticSearch as the foundation of our distributed log storage and distributed full-text search solution.

2.2 Weaknesses of Lucene

Lucene is certainly not perfect. Lucene's support for range query was not provided at the beginning, and when it was finally introduced, the performance is not satisfactory. In practice, the search can be very slow when there are many occurrences of terms in a single document. Although Lucene is often reputed as an efficient full-text search engine, its high performance is largely limited to boolean queries.

Starting with Lucene version 6.0, a new index data structure for numeric datatype called BKD tree [20] was introduced to optimize the performance of range queries in Lucene.

The complexity of the BKD tree is linearly correlated with the index cardinality, and the number of hits. Therefore, the original BKD tree is not suitable for massive log query, whose timestamps are of high cardinality.

3 ARCHITECTURE

The architecture for TencentCLS is shown in Figure 2. The entire system is deployed on Tencent Cloud, using cloud services such as Elastic Compute Service, Cloud Object Storage, etc. Its components are described as follows.

3.1 Access Layer

The access layer receives, processes, and forwards the requests to other layers. It consists of multiple modules which takes charge of authentication, validation, centralized flow control, etc. Valid requests will be eventually forwarded to the write layer or the query layer according to their types. Modules at this layer are deployed as containers, and the resources will be automatically adjusted as the demands change.

3.2 Stateless Write Layer

The write layer processes the write requests. It writes the data to the corresponding topic in the message queue of the data store layer. The layer is designed to be stateless, and the mapping between the topics and the topics in the message queue is maintained with the Multi-Tenant Resource Manager, which is described below. The layer is deployed as containers, and supports auto scaling.

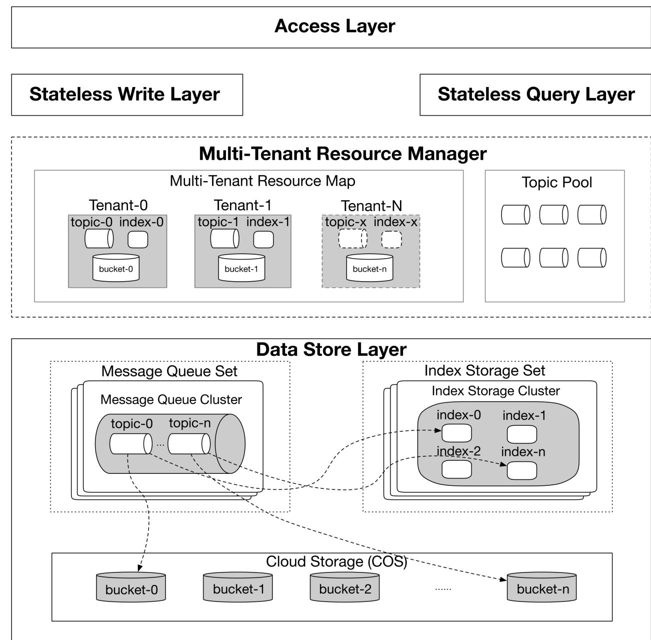


Figure 2: The TencentCLS Architecture

3.3 Stateless Query Layer

The query layer processes the query requests, and consists of functionalities such as query parsing, query translation, collection and aggregation of retrieved information, etc.

We have also customized the SQL engine to support over 300 functions to meet the needs of our users. As a comparison, built on Lucene as well, ElasticSearch only supports less than 200 functions.

In addition, TencentCLS has also introduced the *smart sampling* feature. When switched on, the engine would estimate the time required for the query based on both the current workload and the query itself, and perform on-demand sampling on the query results, therefore enabling timely response.

The layer is also designed to be stateless, thanks to the Multi-Tenant Resource Manager module. The layer is deployed as containers, and supports auto scaling.

3.4 Multi-Tenant Resource Manager

The multi-tenant [12] resource manager maintains the mappings from the topics of the tenants to three kinds of resources in the data store layer: the topics in message queue, the indexes, and the buckets. To be more precise, each topic corresponds to one topic in the message queue, many indexes and one bucket. Therefore, we achieve isolation between the data from different tenants.

We also conducted two optimizations. First, we slice the data into many indexes according to their timestamps, so that we can perform basic pre-filtering on the queries. Second, since a large proportion of the tenants never write any data, we postponed the resource allocation in the data store layer to the point where the actual data write happens. To do so, we introduce virtual storage resource (VSR), an abstraction of the storage resource. However, this design increases increases the latency of the initial write. To

mitigate the effect, we maintain a pool of resources in the data store layer so that the allocation is done beforehand, and only the data binding is required at the initial write. The size of the pool is updated daily, to keep up with the requirements of that particular day, and it is calculated mainly using the count of new users and of users that turned active.

3.5 Data Store Layer

The data store layer consists of three parts: 1) the message queue, 2) the index storage, and 3) the cloud storage. The message queue is to smooth out the latency of write requests. To ensure data reliability, multiple copies of data are kept in the queue, and the write request is responded only when more than two copies have been successfully written.

3.6 Index Storage Layer

The index storage layer maintains the indexes for different tenant topics. The implementation is based on Lucene. In order to support various kinds of queries, we build various indexes such as inverted indexes [15, 23], SkipList [18] indexes and BKDTree [20] indexes. Also, column-oriented storage is adopted to support efficient analyses.

3.7 Object Storage Layer

The object storage layer takes care of the data persistence. It also supports demands such as re-indexing from objects in the event of an exception.

4 A SEARCH ENGINE OPTIMIZED FOR LOG QUERY

This section describes the search engine used in TencentCLS. The search engine is built upon Lucene and is highly optimized for log queries.

We begin with some basic examples of queries, and then we briefly describe the indexing and searching of Lucene. Next, we demonstrate the characteristics of log queries and explain why the default indexing and searching functionalities provided by native Lucene is not satisfactory. Finally, we propose our design, and elaborate on its differences from the Lucene search engine.

4.1 An Example Log Document and Log Query

A typical log document consists of a timestamp, text, and properties. Below is an example.

```
[2021-09-28T10:10:39.1234] [ip=192.168.1.1]
XXXXXXXX
```

Normally, to accelerate log query, the system will create indexes for the timestamp, text and properties respectively.

A typical log query specify a few conditions, and a time range. Below is an example.

```
SELECT * FROM xxxx_index
WHERE ip = 192.168.1.1
    and timestmap >= 2021-09-28T00:00:00
    and timestamp < 2021-09-29T00:00:00
```

4.2 Indexing and Searching in Lucene

In Lucene, every log document will be assigned a unique number called *docid*. When creating an index, an inverted index storing a mapping from contents to sets of *docid*s will be created.

For example, with the timestamp field, Lucene will create a postings list that maintains a mapping from all possible timestamps to sets of *docid*s. Based on that, Lucene can quickly response to the queries that search for a given timestamp. The algorithm complexity for the query is $O(log(n))$, where n is the number of the possible timestamps.

4.3 Characteristics and Challenges with Log Queries

Although Lucene is known to be good at full text queries thanks to the design of the inverted index [23], its performance drops dramatically when searching numeric fields [16]. The performance gets even worse when searching high-cardinality numeric fields. Unfortunately, the timestamp field of log data is a high-cardinality numeric field. In fact, a maximum of 24*60*60*1000 = 86400000 unique values can be generated every day, when using millisecond-level indexing. Therefore, searching with time conditions on massive log data with Lucene can be extremely slow.

What makes it even worse is that, most log queries do not specify a single timestamp, but instead specify a range of time, as is shown in the above example. Such queries will require even more time to finish, because Lucene has to scan through all the timestamps and retrieve the corresponding *docid*s. Therefore, the time range query on massive log data almost guarantees to time out.

Although there have been various optimizations and variants based on inverted index [14, 16, 17, 19, 21, 21], very few of them are battle tested. What we adopt here is a lightweight solution.

4.4 Our Solution: A Search Engine with Time Series Index

To achieve better performances with log queries and address the above problems, we propose a Lucene-based search engine with time series index.

A query may consist of multiple sub-queries, one of which must be a time range query. Other queries can be numeric range queries, full-text queries, etc. Sub-queries can be combined in various ways using NOTs, ANDs and ORs. A query with multiple sub-queries is typically processed in the following steps.

(1) For each sub-query, an ordered list of document IDs that meet the conditions is retrieved. Different sub-queries are handled differently. For example, high-cardinality time series range queries are processed using optimized procedure mentioned in following section, full-text queries are processed using the posting list index, and numeric range queries are processed using the BKD tree index.
(2) For the ordered list of document IDs perform intersection and merge operations, e.g. intersection using the fast multiplexing algorithm.
(3) The results are output in temporal order.

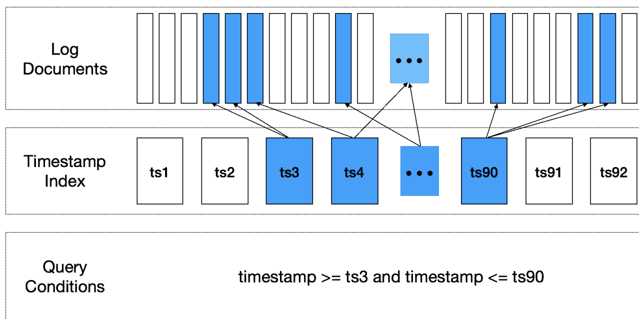


Figure 3: Range query with unordered documents. It requires visiting every timestamp index within that range in order to collect the documents.

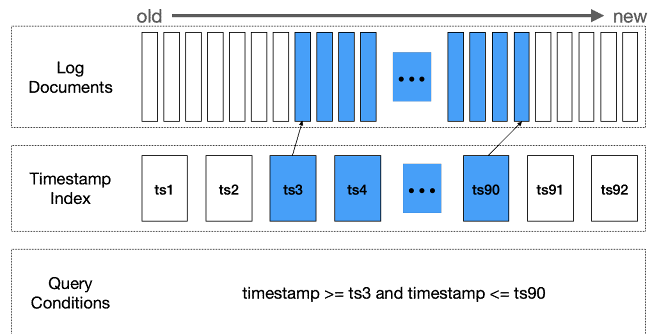


Figure 4: Range query with ordered documents. It requires visiting only two timestamp and the documents can be calculated based on the first docid and the last docid.

The core design choice we make with the TencentCLS search engine is that the log documents are always sorted by timestamps in the ascending order.

The reason why additional sorting is needed for the seemingly already time-ordered logs is that, although logs are generated in chronological order, in a distributed system, after logs generated from multiple services/servers are submitted to the logging service, their order may be disturbed.

In the following paragraphs, we first explain how this design benefits the performance of log queries, then describe the overhead of applying that functionality, and next provide some implementation details. Finally, we also describe other optimizations for specific query types.

4.4.1 Why keeping the log sorted in time.

In order to explain the reason why TencentCLS keeps the log sorted in time, we need to first describe what that changes the range query procedure into. The new procedure is provided below.

(1) Suppose the timestamp range is specified to [ts_i, ts_j], we use the index to find the smallest *docid*, docid_p, that corresponds to ts_i, and the largest *docid*, docid_q, that corresponds to ts_j.
(2) The document id list is directly calculated as [docid_p, ..., docid_q]. Previously it was constructed by merging all postings lists for the timestamp within that range.
(3) Set operations might be performed on this document id list and other document id lists, in order to generate the final result.

Figure 3 and Figure 4 also demonstrate how the range query works, before and after applying the feature.

Given the above procedure, it can be concluded that once we successfully keep the documents sorted, the following merits are promised.

- In the aspect of storage, the BKD index (the data structure provided by Lucene to support range query) for timestamp is no longer required, since the column-oriented storage for timestamps is already sorted.
- The index read frequency is reduced, since we only need to locate the *docids* corresponding to the begin and the end of the timestamps.
- The CPU usage is reduced, since we can construct the postings directly from the *docids* corresponding to the begin and the end of the timestamps.
- The support for timestamps of higher precision becomes feasible.

Theoretically, keeping the documents sorted would reduce the complexity of each query from $O(n)$ to $O(log(n))$, where n is the number of the hit documents.

4.4.2 Implementation of the Sorting Mechanism.

The function that keeps the documents sorted is implemented using the existing *index-sorting* in Lucene. The native Index-sorting has two functionalities. First, the data is kept sorted by the specified field. Second, a feature called early-terminate is applied to increase the performance. The early-terminate feature is explained as follows.

By default, a search request in Lucene must visit every document that matches the query in order to return the top documents sorted by a defined sort. When the index and search sorts are the same, it is feasible to limit the number of documents that must be viewed per segment in order to obtain the top N documents globally. With early-terminate, Lucene will only compare the first N documents per segment if it detects that the top docs of each segment are already sorted in the index. The remaining documents that fit the query are gathered in order to count the overall number of results and create aggregations.

Therefore, for example, when we want the latest 10 log data, if the index-sorting is not enabled, we have to sort all the log data with timestamps and return the latest 10. With the index-sorting enabled, we only need to iterate over the latest 10 log data.

However, in practice we find that simply turning on the index-sorting function implemented in Lucene achieves little to even negative performance improvements with log queries. After some analysis, we find that there are some other issues that need to be solved before we can benefit from index-sorting when processing log queries. Those optimizations are described in the next section 4.5.1.

4.4.3 Overhead of Keeping the Log Sorted.

The overhead of keeping the log sorted is also important to consider. According to our experiments and analysis, enabling index-sorting

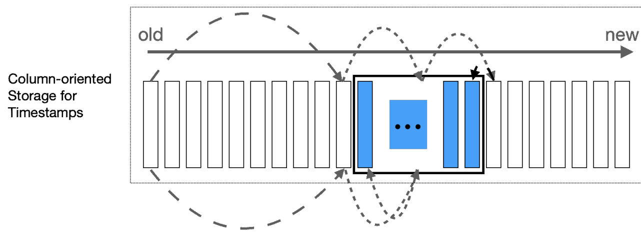


Figure 5: Binary search for timestamps endpoints directly on column-oriented storage

has only slight effect on log writes, increasing the CPU usage by approximately 6.5%, a value that is perfectly acceptable for our system. For example, if the average CPU usage was 30% before enabling *index-sorting*, now it would become 32%.

4.4.4 Microsecond-level Time Order Preservation.
We have also noticed that many commercial and open-source log service solutions do not support microsecond-level time order preservation, which is a urgent need for many time-critical log analysis scenarios. Attributed to the above design, our solution has already guaranteed the property of microsecond-level time order preservation with no additional effort, while still keeping the query latency low.

4.5 Additional Optimizations

In addition to the major design described above, we have also implement other optimizations in the search engine in TencentCLS. The motivation and the description of those optimizations are given below.

4.5.1 Optimization 1. Secondary Indexing.
We analyzed the reason why simply using index-sorting on timestamp field yields little performance gain. In Lucene, searching the sorted field is accomplished by performing binary search in the column-oriented storage of that column. The problem is, the index for the log data is too large (a few tens of gigabytes), and the binary search for the beginning and end timestamps requires a few tens of random accesses on the disks, which are slow to perform on slow storage devices.

For example, the index for 10 billion log entries would have the size of around 30 GB, and therefore even the process of loading the index data would cost 300 seconds with the speed of 100 MB/s.

To address that problem, we build a secondary index that decreases disk accesses from a few tens of times to around 3 times, as is demonstrated in Figure 5 and Figure 6.

The secondary index is implemented using the posting list and the BKD tree data structures implemented within Lucene, and does not affect the types of queries Lucene supported.

4.5.2 Optimization 2. Reverse Binary Search Algorithm for Tail Queries.
We find that the queries can be divided into two groups: head queries and tail queries, and the latter can be optimized.

We define the head queries as the queries that are to search the last few entries that satisfy the given conditions, and the tail queries as the queries that are to search the first few entries, as is shown

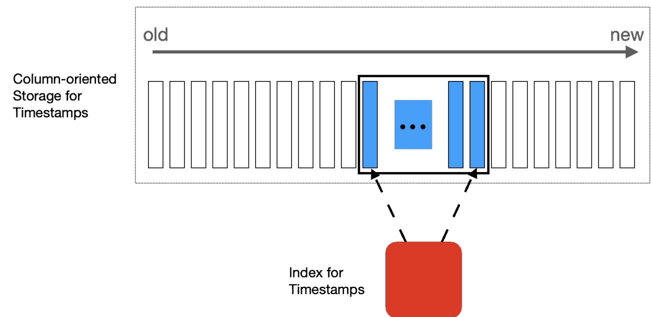


Figure 6: Binary search for timestamp endpoints with secondary index

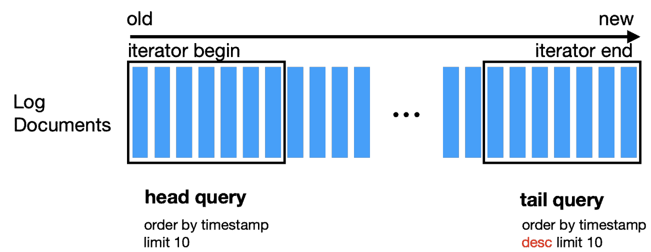


Figure 7: Head query and tail query

in Figure 7. Given that the log data are sorted in ascending order by time, head queries are to search the oldest logs that meets the conditions while the tail queries are to search the newest logs. We also provide an example of the tail query below.

```
SELECT * FROM xxx_index
WHERE ...
ORDER BY timestamp
DESC LIMIT 10;
```

Although both queries look similar, when it comes to tail queries, Lucene's implementation can be very inefficient, due to the following reasons.

The iterators implemented in Lucene only support one-way iterations. Therefore, for tail queries, we have to iterate through all data till the end, as is shown in Figure 7. The complexity of this process is $O(n)$, where n is the number of the documents that meet the condition.

Even if we add support for reverse iteration on top of Lucene, tail queries would still be inefficient. The reason is that the reverse access to disks would render the file cache provided by operating systems ineffective.

Therefore, to address the inefficiency of tail queries, we propose the Reverse Binary Search algorithm. The algorithm is implemented on top of the existing iterators of Lucene. In effect, the algorithm reduces the complexity of tail queries from $O(n)$ to $O(log(n))$.

The execution of the algorithm consists of two steps. The first step is using binary search algorithm to locate the second to last document that meets the given conditions, and store every middle point during the search. The second step is to iterate over the collection of the middle points. For every middle point, we examine

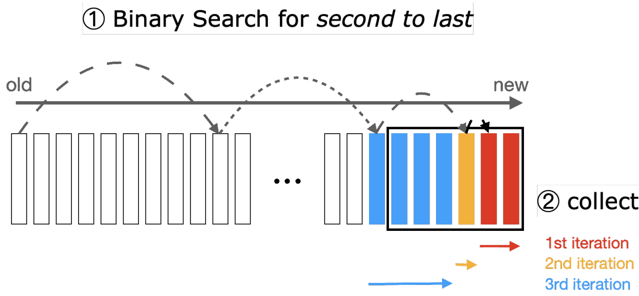


Figure 8: Demonstration of the Reverse Binary Search algorithm for tail queries.

if there exists K documents that meet the conditions. If there are K documents, the execution is finished and K documents are returned. If not, we continue that process and examine the next middle point.

The algorithm is demonstrated in Figure 8, as well as Algorithm 1.

Algorithm 1 The Reverse Binary Search algorithm.

```
MiddlePoints ⟵ BinarySearch (Hits)
▹ Here BinarySearch refers to a modified algorithm that returns
a series of middle points instead of the found document
for each MiddlePoint ∈ MiddlePoints do
    iterator ⟵ Iterator(MiddlePoint)
    count ⟵ 0
    documents ⟵ {}
    for each document ∈ iterator do
        documents.add(document)
        count ⟵ count + 1
    end for
    if count >= K then
        return The last K elements of documents
    end if
end for
```

4.5.3 Optimization 3. The Histogram Query.

Lastly, the histogram queries are also optimized. A histogram query asks for the distribution of logs in time that satisfy certain conditions.

The histogram query is an extremely common type of query in TencentCLS. Because TencentCLS almost always provide a histogram view for every normal query, to give the user an overall sense of the log distribution.

By default, to handle such queries, Lucene first filter the logs by the conditions, then check the timestamps of the remaining logs. However, this process may cause tens of thousands of look-ups in the table, which in turn causes huge latency.

To address that, we optimize the histogram queries so that they no longer require looking up the table. The details are as follows.

First, the doc_ids that correspond to the edges of the bins are retrieved, using the time series index. Second, we iterate over the logs that satisfy the given conditions and check which bin each log belongs to, by comparing the doc_id of the log and the doc_id edges retrieved in the first step. The counts of the corresponding bins are increased in this process.

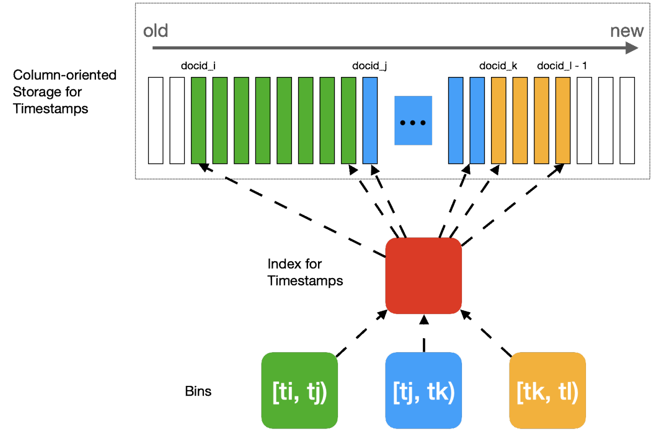


Figure 9: The optimized procedure for histogram queries consists of two steps: 1) calculating the *docids* that corresponds to the bin edges, and 2) iterating over the logs that satisfy the conditions and increase the count of the corresponding bin, which is recognized by comparing the *docids* of the log and the edge *docids* derived in the first step.

With such technique, we managed to reduce the cost of tens of thousands of lookups into the table to a few lookups into the time series index.

The process is shown in Figure 9.

4.5.4 Optimization 4. IO Optimizations.

In the development process, we observe spikes of disk write.

We investigated this issue and found that it was related to the operating system's page cache. When the file systems perform write operations, the data are first put into the page cache, and are later put onto disks only when either of the conditions are met.

1. The data inside page cache reached certain amount (controllded by *vm.dirty_background_ratio* in Linux).
2. The data inside page cache reached certain time limit (controlled by *vm.dirty_expire_centisecs* in Linux).

To avoid occasional high query latency, we smooth the disk write and eliminate the write spike by increasing the two aforementioned parameters. As a result, the long tail latency is significantly reduced, as well as the size of the query queue.

5 EXPERIMENTAL EVALUATION

The experimental evaluations are mainly to demonstrate the effectiveness of the design of the search engine in TencentCLS.

Overall, the experiments consist of two parts: offline experiments with open benchmarks and online experiments with real world data. The first part of experiments is relatively cheap to perform, we use them to analyze the performance gains of our methods under various scenarios. The second part, on the other hand, provide more convincing evidences of the effectiveness of our solution, since it utilizes real-world data at TencentCLS.

5.1 Open Benchmark Evaluation

In the open dataset experiments, we quantitatively investigate the effectiveness of the query optimizations in a single-machine setup.

The experiment is performed on Tencent Cloud machines, each with a 16-core vCPU and 64 GB of ram. The storage devices are local NVMe SSD drives (IT3.4XLARGE64), local SATA HDD drives (D3.4XLARGE64) and Tencent Premium Cloud Storage.

Table 1: Statistics of the NYC Taxi Benchmark

Name	Value
No. of documents	~12 b
No. of shards	6
average Lucene segment size	~5 GB
No. of documents per Lucene segment	~24 m
average No. of hits per query	~40 m

The benchmark we use is the NYC taxi benchmark provided by esrally. The dataset consists of taxi rides information in New York in 2015, and contains up to a total of 12 billion documents. Some important statistics for this benchmark are listed in Table 1.

The experiments are designed with the goal of analyzing the performance increases in the following scenarios.

(1) Different types of queries: head queries, tail queries, and histogram queries (defined in Section 4.5.2 and Section 4.5.3).
(2) Different types of storage devices: Tencent Premium Cloud Storage, NVMe SSD drive, and SATA HDD drives.
(3) Different number of users: 1, 2, 4, 6, 8, 10, 15, 20, 50, 100, 150, 200.
(4) Different timestamp precisions: second-level, and millisecond-level.

The most important scenario is ones that use Tencent Premium Cloud Storage as storage devices, and adopt second-level timestamp precision. The reason for prioritizing the Tencent Premium Cloud Storage is that TencentCLS is built on Tencent Cloud. And the reason for using second-level timestamp precision is that the timestamp in the benchmark dataset has the second-level precision.

Based on the results, we have been able to answer the following research questions.

5.1.1 *RQ1. What is the overall performance increase compared with Lucene?*

Under the most important scenario described above (Tencent Premium Cloud Storage + second-level time precision), using the inverse of the service time (in milliseconds) as the indicator for the performance, we observe that the performances increase by 38x for head queries, 26x for tail queries and 7.5x for the histogram queries.

Figure 10 shows a more detailed result, distinguishing the performances under different user counts. Generally, the performance steadily increases more, as the user counts get higher. The reason is that when the user counts are low, the workload is low, and the strength of the system design is not fully displayed. Therefore, we always use the results from the heaviest workload as the arguments for our analyses.

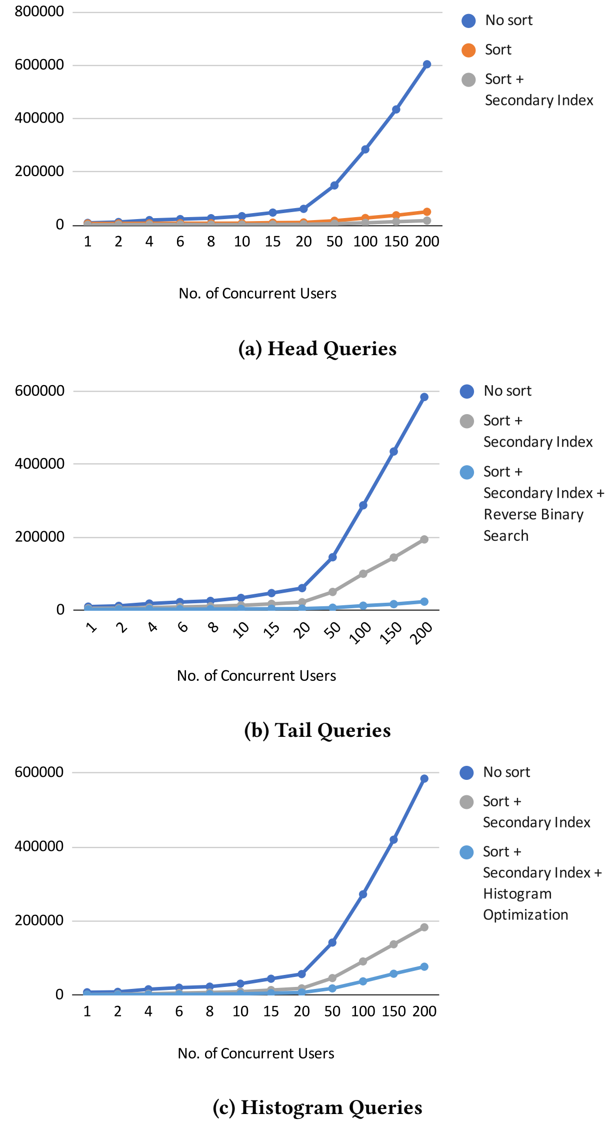


Figure 10: Performances for three types of queries with different optimization options

5.1.2 *RQ2. How much does each of the optimization techniques contribute to the performance improvements?*

There are four optimization techniques described in this paper:

- O0: Keeping the documents sorted.
- O1: Constructing the secondary index for the timestamp field.
- O2: Reverse binary search algorithm for tail queries.
- O3: Optimizing histogram queries.

We turn them on and off individually (while we can) in order to understand the contribution of each technique to the overall performance increase. Other experimental setup is the same as the one used in **RQ1**.

Results show that turning on **O0** alone increases the head query performances by 12x, increases the head query performances by 3x and increases the histogram query performances by 3x.

On top of that, the turning on the secondary index (**O1**) further increases the head query performances by 3x, but has little effect on the performances of other types of queries.

Furthermore, the Reverse Binary Search Optimization technique (**O2**) increases the tail query performances by 3.5x, while the Histogram Optimization technique (**O3**) increases the histogram query performances by 1.6x.

The results are shown in Figure 10, distinguishing the performances under different user counts, as well as in Table 2.

5.1.3 *RQ3. How does the choice of the storage option affect the query performance, before and after the optimization?*

Tencent Cloud provides a series of customizable storage options, among which Tencent Premium Cloud Storage, SATA HDD drives, and NVMe SSD drives are the most representative ones.

All the above analyses (**RQ1** and **RQ2**) are based on the experiments using Tencent Premium Cloud Storage as the storage option. However, experimental results with other storage options are also important, because they not only show the comparison of effectiveness of the optimization techniques, but also serve as a guidance for choosing the storage option.

Tencent Cloud Premium Cloud Storage is a hybrid storage option. It adopts the Cache mechanism to provide a high-performance SSD-like storage, and employs a three-copy distributed mechanism to ensure data reliability.

SATA HDD is the most economical option suitable for scenarios that involve sequential reading and writing of large files, but its random access performance is relatively low.

NVMe SSD has the highest performance. But its low cost performance ratio restricts its strength in the log service scenarios.

Table 3 shows the comparison of the specifications of the three storage options.

The experimental results with different storage options are shown in Table 4. We can draw the following conclusions. First, the NVMe SSD option consistently outperform other storage options, while the Tencent Premium Cloud Storage option is less than an order of magnitude behind. Second, compared with the NVMe SSD, the Tencent Premium Cloud Storage consistently enjoys more benefits from the query optimization techniques.

5.1.4 *RQ4. Will the increase of timestamp precision level impact the query performances?*

It is also the goal of Cloud Log Service to support storing and querying higher-precision timestamps. Therefore, it is important to check how does the increase of the timestamp precision level impact the query performance. To this end, we change the timestamp from second to millisecond, and analyze the query performance. The data also comes from the experiments using Tencent Premium Cloud Storage.

Interestingly, as is shown in Figure 11, increasing the timestamp precision has almost no impact on the query performance, thanks to the search engine design in TencentCLS.

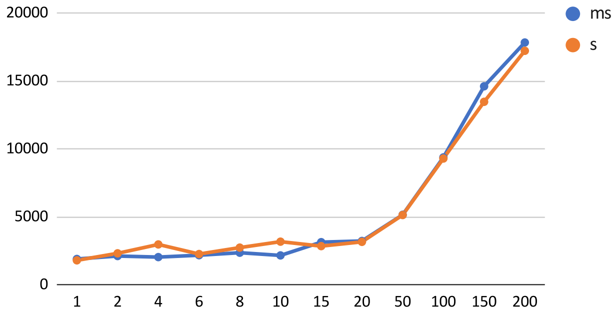
The reason is that although the precision increases, the frequency of the log writes stays the same. Although theoretically some operations such as locating the endpoints will get slower, after applying the secondary index optimization, the difference in costs is significantly reduced. Also, those precision-sensitive operations do not take up a large proportion of the total service time. Therefore, generally speaking, the performance is virtually unaffected by the time precision.

Table 2: Performances when turning on and off different optimization techniques. Multiplier refers to the boost multiplier of current optimization config, compared with the previous one. Accumulative Multiplier refers to the accumulative boost multiplier of the current optimization config. CPU / query refers to the CPU usage per query (CPU usage percentage * time). rMB refers to the disk read per query.

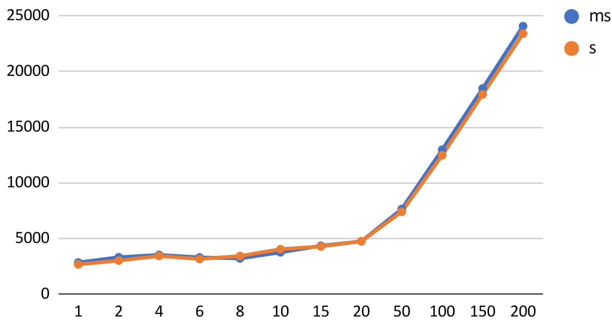
	Head Query Service Time	CPU / query	rMB / query
No Optimizations	604124.0	200.5	452.7
O0	50318.2	7.3	37.3
Multiplier	12.0	27.6	12.1
Acc. Multiplier	12.0	27.6	12.1
O0 + O1	17224.8	5.5	12.5
Multiplier	2.9	1.3	3.0
Acc. Multiplier	35.1	36.5	36.2
O0 + O1 + O2 + O3	15904.2	5.2	12.1
Multiplier	1.1	1.1	1.0
Acc. Multiplier	38.0	38.9	37.3
	Tail Query Service Time	CPU / query	rMB / query
No Optimizations	585014.0	196.0	438.4
O0	193487.0	831.7	144.3
Multiplier	3.0	0.2	3.0
Acc. Multiplier	3.0	0.2	3.0
O0 + O1	194551.0	821.8	82.2
Multiplier	1.0	1.0	1.8
Acc. Multiplier	3.0	0.2	5.3
O0 + O1 + O2 + O3	23931.0	34.4	17.1
Multiplier	8.1	23.9	4.8
Acc. Multiplier	24.4	5.7	25.6
	Histogram Query Service Time	CPU / query	rMB / query
No Optimizations	584511.0	116.4	438.0
O0	179252.0	66.6	134.0
Multiplier	3.3	1.7	3.3
Acc. Multiplier	3.3	1.7	3.3
O0 + O1	183304.0	69.2	137.7
Multiplier	1.0	1.0	1.0
Acc. Multiplier	3.2	1.7	3.2
O0 + O1 + O2 + O3	76893.0	39.8	57.0
Multiplier	2.4	1.7	2.4
Acc. Multiplier	7.6	2.9	7.7

Table 3: The specifications of different storage solutions at Tencent Cloud. IOPS is tested with 4 KiB IO, and throughput is tested with 256 KiB IO.

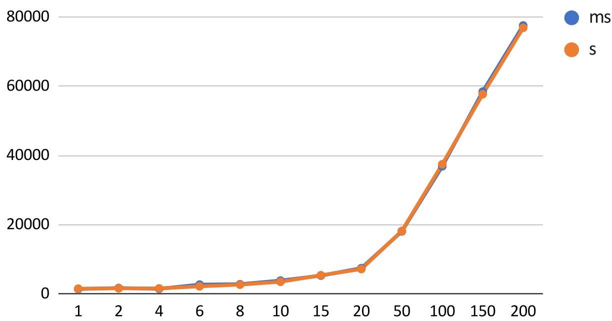
Disk Type	IOPS	Throughput
Premium Cloud Storage	6,000	150 MB/s
NVMe SSD	650,000	2.8 GB/s
SATA HDD	200	190 MB/s



(a) Head query performance



(b) Tail query performance



(c) Histogram query performance

Figure 11: Performances with second-level timestamp precision and millisecond-level timestamp precision, evaluated using the total service time (in milliseconds).

Table 4: Comparison of performance improvements among different storage solutions. For each storage solution, three rows list the native performances, the performances after optimizations, and the multipliers for performance improvements, respectively. The results are tested under 200 concurrent users for Premium Cloud Storage and NVMe SSD, and under 150 concurrent users for SATA HDD because of the its limited performance.

	Head Query Service Time	CPU / query	rMB / query
Premium Cloud Storage	604124.0	200.5	452.7
	15904.2	5.2	12.1
	38.0	**38.9**	**37.3**
NVMe SSD	84986.6	405.6	459.4
	2704.1	9.0	9.6
	31.4	**45.3**	**47.6**
SATA HDD	1426810.0	215.7	423.9
	108863.0	8.6	14.0
	13.1	**25.1**	**30.2**
	Tail Query Service Time	CPU / query	rMB / query
Premium Cloud Storage	585014.0	196.0	438.4
	23931.0	34.4	17.1
	24.4	**5.7**	**25.6**
NVMe SSD	77402.1	370.8	449.6
	13134.5	61.1	17.3
	5.9	**6.1**	**26.0**
SATA HDD	1448450.0	211.7	433.2
	183195.0	35.7	17.7
	7.9	**5.9**	**24.5**
	Histogram Query Service Time	CPU / query	rMB / query
Premium Cloud Storage	584511.0	116.4	438.0
	76893.0	39.8	57.0
	7.6	**2.9**	**7.7**
NVMe SSD	53759.4	237.7	425.5
	17333.5	77.4	48.9
	3.1	**3.1**	**8.7**
SATA HDD	1326030.0	130.9	411.9
	465770.0	42.4	58.1
	2.8	**3.1**	**7.1**

This conclusion also theoretically applies to higher time accuracy. In fact, the online version of TencentCLS is running with microsecond-level time precision with no additional optimization, which is much higher than many vendors that are providing second-level time precisions.

Table 5: Results of the online experiment.

Head Query				
# Log	10^9	10^{10}		
Original (ms)	12882	16904		
Ours (ms)	399	780		
Boost Multiplier	32x	21x		
Tail Query				
# Log	10^9	10^{10}		
Original (ms)	10577	17483		
Ours (ms)	391	1299		
Boost Multiplier	27x	13x		
Histogram Query				
# Log	10^9	10^{10}	$5 * 10^{10}$	10^{11}
Original (ms)	16623	>42764	TIMEOUT	TIMEOUT
Ours (ms)	1144	4253	10300	17920
Boost Multiplier	15x	>10x	N/A	N/A

5.1.5 RQ5. What is the bottleneck of our system?

We have also investigated the bottlenecks of our system by analyzing the CPU usage and the disk IO during the above experiments.

As is shown in Table 4, the bottlenecks for Premium-Cloud-Storage-based solutions and NVMe-SSD-based solutions are IO bandwidth and CPU, respectively. For SATA-HDD-based solutions, the bottleneck is IOPS from our experience, although it is not explicitly reflected in Table 4.

5.2 Online Test

In addition to the offline experiments with open benchmarks, we have also tested the system with real world data.

The experiments involve two clusters, one equipped with ElasticSearch (version 7.10.1), and the other equipped with the search engine of TencentCLS. Each cluster consists of 3 master nodes as well as 40 data nodes. We select a single large log topic as input, and its data is written to those clusters at the same time.

The results are shown in Table 5. Generally, the head/tail query performances increase by 20x, while the histogram query performances increase by 10x. Moreover, the new system supports histogram queries on 100 billion log documents, and can process the queries within 20 seconds, while the original system has started to time out on only 10 billion log documents.

6 CONCLUSION

In this paper, we introduce the motivation of TencentCLS, and propose the architecture of TencentCLS. Then we elaborate on the design and optimizations of the search engine in TencentCLS, a system that supports low-latency queries with massive high-cardinality data. Finally, we evaluate and analyze the performance of our search engine, both with open benchmarks and with online data in TencentCLS.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their valuable comments and helpful suggestions. We must also thank the R&D Team and the PM Team of the TencentCLS. Especially helpful during this time were Wenshuang Ma, Jueling Li, Jian Wang, Xianbin Wu. And special thanks to TencentES OTeam for their technical support. Jinan Sun is the corresponding author.

REFERENCES

[1] 2022. Amazon CloudWatch - Application and Infrastructure Monitoring. https://aws.amazon.com/cloudwatch/.
[2] 2022. Apache Lucene. https://lucene.apache.org/.
[3] 2022. Apache Solr. https://solr.apache.org/.
[4] 2022. Azure Monitor | Microsoft Azure. https://azure.microsoft.com/en-us/services/monitor/.
[5] 2022. Cloud Logging | Google Cloud. https://cloud.google.com/logging.
[6] 2022. Elastic. https://www.elastic.co/.
[7] 2022. MG4J: High-Performance Text Indexing for Java™. https://mg4j.di.unimi.it/.
[8] 2022. Sphinx: Open Source Search Engine. http://sphinxsearch.com/.
[9] 2022. Splunk. https://www.splunk.com.
[10] 2022. Tencent Cloud Log Service. https://intl.cloud.tencent.com/products/cls.
[11] 2022. The Xapian Project. https://xapian.org/.
[12] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. 2008. Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1195–1206. https://doi.org/10.1145/1376616.1376736
[13] Andrzej Białecki, Robert Muir, and Grant Ingersoll. 2012. *Apache Lucene 4*. 24 pages.
[14] Matteo Catena, Craig Macdonald, and Iadh Ounis. 2014. On Inverted Index Compression for Search Engine Efficiency. In *Advances in Information Retrieval (Lecture Notes in Computer Science)*, Maarten de Rijke, Tom Kenter, Arjen P. de Vries, ChengXiang Zhai, Franciska de Jong, Kira Radinsky, and Katja Hofmann (Eds.). Springer International Publishing, Cham, 359–371. https://doi.org/10.1007/978-3-319-06028-6_30
[15] D. Cutting and J. Pedersen. 1990. Optimization for Dynamic Inverted Index Maintenance. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval - SIGIR '90*. ACM Press, Brussels, Belgium, 405–411. https://doi.org/10.1145/96749.98245
[16] Marcus Fontoura, Ronny Lempel, Runping Qi, and Jason Zien. 2005. Inverted Index Support for Numeric Search.
[17] Xiaoming Gao, Vaibhav Nachankar, and Judy Qiu. 2011. Experimenting Lucene Index on HBase in an HPC Environment. In *Proceedings of the First Annual Workshop on High Performance Computing Meets Databases - HPCDB '11*. ACM Press, Seattle, Washington, USA, 25. https://doi.org/10.1145/2125636.2125646
[18] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *Structural Information and Communication Complexity*, Giuseppe Prencipe and Shmuel Zaks (Eds.). Vol. 4474. Springer Berlin Heidelberg, Berlin, Heidelberg, 124–138. https://doi.org/10.1007/978-3-540-72951-8_11
[19] Giulio Ermanno Pibiri and Rossano Venturini. 2021. Techniques for Inverted Index Compression. *Comput. Surveys* 53, 6 (Nov. 2021), 1–36. https://doi.org/10.1145/3415148 arXiv:1908.10598
[20] 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*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Thanasis Hadzilacos, Yannis Manolopoulos, John Roddick, and Yannis Theodoridis (Eds.). Vol. 2750. Springer Berlin Heidelberg, Berlin, Heidelberg, 46–65. https://doi.org/10.1007/978-3-540-45072-6_4
[21] Hao Yan, Shuai Ding, and Torsten Suel. 2009. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proceedings of the 18th International Conference on World Wide Web - WWW '09*. ACM Press, Madrid, Spain, 401. https://doi.org/10.1145/1526709.1526764
[22] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, Shinjuku Tokyo Japan, 1253–1256. https://doi.org/10.1145/3077136.3080721
[23] Justin Zobel and Alistair Moffat. 2006. Inverted Files for Text Search Engines. *Comput. Surveys* 38, 2 (July 2006), 6. https://doi.org/10.1145/1132956.1132959