μSlope: High Compression and Fast Search on Semi-Structured Logs

Rui Wang, *YScope;* Devin Gibson, *YScope and University of Toronto;* Kirk Rodrigues, *YScope;* Yu Luo, *YScope, Uber, and University of Toronto;* Yun Zhang, Kaibo Wang, Yupeng Fu, and Ting Chen, *Uber;* Ding Yuan, *YScope and University of Toronto*

https://www.usenix.org/conference/osdi24/presentation/wang-rui

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

μSlope: High Compression and Fast Search on Semi-Structured Logs

Rui Wang[†] Devin Gibson[†⋆] Kirk Rodrigues[†] Yu Luo[†‡⋆]
Yun Zhang[‡] Kaibo Wang[‡] Yupeng Fu[‡] Ting Chen[‡] Ding Yuan[†⋆]

YScope[†] *Uber* [‡] *University of Toronto*[⋆]

Abstract

Internet-scale services can produce a large amount of logs. Such logs are increasingly appearing in semi-structured formats such as JSON. At Uber, the amount of semi-structured log data can exceed 10PB/day. It is prohibitively expensive to store and analyze them. As a result, logs are only kept searchable for a few days.

This paper proposes μSlope, a system that losslessly compresses semi-structured log data, and allows search without full decompression. It concisely represents the schema structures, and only keeps this representation stored once per dataset instead of interspersing it with each record. It further "structurizes" the semi-structured data by grouping the records with the same schema structure into the same table, so that each table is also well structured. Our evaluation shows that μSlope achieves 21.9:1 to 186.8:1 compression ratio, which is at least a few times higher than any existing semi-structured data management systems (SSDMS); The compression ratio is 2.34x as high as Zstandard and the search speed is 5.77x of the other SSDMSes.

1 Introduction

In the past two decades we have witnessed an explosive growth of log data from Internet-scale systems. Conventionally, logs were only in the form of unstructured, free text (e.g., output from `printf()`). However, they increasingly appear in semi-structured format, such as JSON [12] or Protocol Buffers® [9]. [1] These data models have a tree-structure, where each node (except for the root) is a key-value pair. The value may include non-primitive data types such as nested values.

Different records may have different schema structures. This is why they are referred to as *semi*-structured. For example, JSON and YAML [15] formats are schema-less, meaning that records may have arbitrarily different schemas. This flexibility allows programmers to easily log common data types in high-level programming languages, such as C struct, class, hash table, array, etc. Although Protocol Buffer and other formats require users to declare a schema to be used on all records, they allow a field to be optional. As a result, different records may still have different structures.

The dynamic schema structure imposes challenges for data management. For example, naively extending conventional relational databases would require creating one column for each *possible* key. This results in a sparse table (i.e., each row may have many NULL values), and it is challenging to handle polymorphic typing (the value of the same key might have different types).

Some existing semi-structured data management systems (SSDMS) use custom-designed data structures to store the schema structure of each record [2, 8, 20, 26, 29, 30]. These systems were initially designed for user-generated data which is relatively small (and they primarily focused on fast search speed). For example, most of the published works on SSDMSes use Twitter® datasets (i.e., Tweets in the Twitter data feed are in JSON) as their primary evaluation target [20, 26, 29, 30]. Twitter reports that in 2023 there are a total of 500 million Tweets per day [13], which results in 140GB of Tweet data per day (assuming 280 characters for each Tweet).

In comparison, logs with machine-generated data can be orders of magnitude larger. The size of JSON logs at Uber from all services exceed 10 Petabytes on a busy day, or 60PB/week, which is more than 70,000x the size of tweets. Managing data at this scale with existing SSDMSes is prohibitively expensive. These systems need to store one schema structure for each record, resulting in a large amount of duplication when a large number of records share the same schema. As a result, even if some systems can efficiently compress the data content [20, 30], the overall storage size is still large as it is dominated by the schema structures [20].

At Uber we have repeatedly suffered from scalability issues as Uber grew. Initially we used Elasticsearch® [21] to manage our JSON logs. It builds indexes for every word in a JSON document to support full text search, hence the index size is at the same order of magnitude as the original data and needs to be stored on SSD for fast search. The resource cost, together

[1] While we do not have global data on the prevalence of the semi-structured logs versus unstructured ones, in Uber, the size of semi-structured logs is about 10x of the unstructured. Part of the reason is that, even if some third-party or legacy applications emit unstructured logs, our log aggregation tools would wrap them in JSON. This is also a common practice outside of Uber. For example, Amazon CloudWatch® [1] also wraps unstructured log outputs in JSON. Grafana Loki® [5] tags unstructured logs essentially turning them into semi-structured logs for the purposes of search, and so on.

with other issues (§2), forced us to migrate away from it around 2019. Since then we have been using ClickHouse® [4], a columnar RDBMS with features for handling JSON data. Its overall compression ratio on our setup is less than 4:1, and it also requires SSD for fast search. Therefore the resource cost is still prohibitively expensive.

CLP [27] and LogGrep [31] are able to compress unstructured logs and allow users to search compressed data without full decompression. While effective on unstructured logs [2] they are limited on semi-structured logs. Fundamentally, their storage structure is not designed for the semi-structured format. For example, CLP parses an unstructured log into three components: timestamp, log type, and variables. The entire log dataset can be stored in a table consisting of these three columns: each log message is stored in one row, and all of its variable values are stored as a list in the variables column. Even if we extend CLP's parser to recognize the key/value structure (so the schema structure can be treated as the log type), the values of all the fields are intertwined in the same variables column of a single table. This hurts compression, but more importantly, querying any field requires tedious decompression and scan of the entire variables column. Indeed, real-world users who use CLP to manage JSON logs encounter poor search performance exactly for these reasons [10]. In addition, these tools only support a wildcard substring query model (like `grep`), but do not provide boolean algebra support to filter on multiple fields.

We propose μSlope[3], an SSDMS for semi-structured log data. μSlope focuses on storage efficiency by losslessly compressing the logs. And we show that fast search can be achieved with a careful system design without having to use an index (which would increase storage overhead). μSlope does not require any user annotation as it automatically handles the dynamic schema structures.

μSlope incorporates three novel designs. The first is its concise representation of the schema structures. Unlike existing SSDMSes that store separate schema information for each record, μSlope proposes (1) a *merged parse tree* to store the schema structures generalized for patterns specific to logs, and (2) a *schema map* to concisely represent each schema with a set of leaf nodes in the tree. Each unique schema structure is stored only once in the schema map, instead of per record, and it is decoupled from the storage of the value contents.

μSlope then partitions the records into *different* tables to store their values, where the records in the same table have the same schema. Therefore each table is now *perfectly* structured: all records have the same number of keys and each value is of the same type. This allows us to apply well-studied optimizations designed for relational tables. For example, we can store and compress each table in a columnar manner that maximizes the compression ratio and search speed [16].

[2] CLP is deployed across Uber's various data and ML platforms to manage the unstructured logs.

[3] μSlope: Semi-structured LOg Processing Engine like a micro(μ)-scope.

Finally, μSlope uses a query processing algorithm that leverages the schema information and encoded tables. μSlope first builds an abstract syntax tree (AST) from the query, and systematically refines this AST by looking up the merged parse tree and schema map. This leads to early termination of queries that do not match any schema structures, and allows μSlope to decompress and scan data only when necessary.

μSlope's design was guided by a characterization study of Uber's semi-structured logs and queries (§3). We found that while records do have dynamic schema structures, there is enormous repetition as most records share a small number of common schemas. In addition, nearly one third of the queries that users performed can be terminated without table scanning because they do not match any of the schema structures.

We evaluate μSlope on a total of 21 semi-structured log datasets, and compare it with a number of widely used SSDMSes. Our result shows that μSlope achieves a compression ratio of 68.1:1 on average. This is at least 2.6x better any existing SSDMSes'. The compression ratio is even 2.34x of Zstandard's and 1.70x of LZMA's at the default compression level, even though they do not support search on compressed data. μSlope's search speed is 2.47x of ClickHouse's, 8.09x of PostgreSQL's, and 6.74x of MongoDB®'s.

This paper makes two contributions. First, it proposes μSlope, a resource efficient archival SSDMS that compresses semi-structured log data and enables fast search without full decompression. Second, we present an in-depth analysis of the characteristics of the semi-structured log data at Uber.

μSlope also has a few limitations. It is designed for semi-structured text logs, which are highly repetitive, instead of being a general-purpose SSDMS. If every record uses a different schema structure it won't work well on μSlope. Furthermore, μSlope uses an index-less design to optimize for storage efficiency; its search speed, however, may not be as fast as index-based search tools like Elasticsearch.

2 Background and Related Work

In this section we present the background information and related work. We explain (1) the semi-structured data model, (2) how prior SSDMSes manage the semi-structured data, and (3) commonly used compression algorithms.

2.1 Semi-structured Data Model

Semi-structured data have a tree structure, and its data model can be defined as follows:

$$
\begin{aligned}
T_{root} &= T_{object} \\
T_{object} &= \{key_1 : T_{value1}, ..., key_n : T_{value_n}\} \\
T_{value} &= T_{object} | T_{array} | T_{primitive} \\
T_{array} &= [T_{value}, ..., T_{value}] \\
T_{primitive} &= string | number | boolean | null \\
key &= string
\end{aligned}
$$

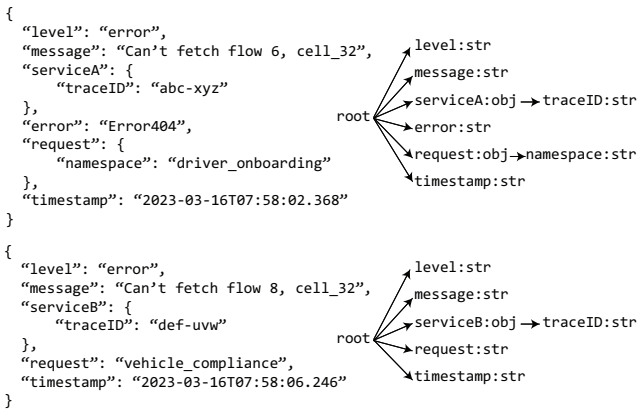

Figure 1: Two example log records and their schema trees.

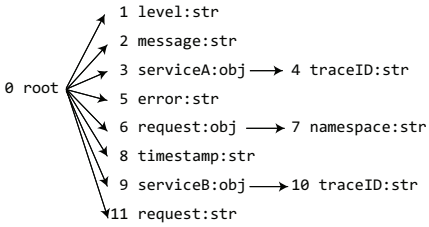

Figure 2: The merged schema tree of the two records.

Each semi-structured *log record*, or record for short, corresponds to a T_{root}. Figure 1 shows two sample records. A log dataset contains a number of such records. The *schema* of each record can be represented as a tree [30]. Figure 1 shows the two schema trees. Each node records a *field*, which consists of the name of the key and the type of its value.

Only the leaf nodes can have primitive value types in the schema tree. Any non-leaf nodes would have non-primitive value types, that is, either object or array.

If two records have the same schema tree, we say they have the same schema. The two nodes in their respective trees are considered to be the same if and only if both the key and the value type are the same. In a schema-less data format like JSON, a key could have "polymorphic" values, i.e., different value types in different records. For example, the "request" field in the two schemas in Figure 1 have different value types.

The schema trees of multiple records can be merged into a single tree [30]. We call it the *merged schema tree*, or MST. Specifically, given node N_1 and N_2 from two schema trees, we can merge them if and only if: (1) N_1 and N_2 have the same key name; (2) the value type are the same; and (3) all the predecessor nodes of N_1 and N_2 in their respective schema trees can be merged. Figure 2 shows the MST of the two schema trees in Figure 1.

2.2 Existing SSDMSes

Different semi-structured records may have different schemas, therefore we cannot naively store them in the RDMS table.

```
\x16\x00\x00\x00          // size (32-bit): 22 bytes
\x02                      // 0x02 = value type String
hello\x00                 // key name
\x06\x00\x00\x00world\x00 // size of value (6 bytes), value
\x00                      // 0x00 = 'end of object'
```

Figure 3: The BSON representation of {"hello": "world"}.

As a result, existing SSDMSes either use natively designed storage format or extend RDBMS in sophisticated ways.

Native Support for Semi-structured Data. These SSDMSes use custom data structures to represent the schema structure *of each record*. MongoDB uses a concise binary format called BSON (Binary JSON) [2]. Figure 3 shows the BSON representation of a simple JSON record {"hello": "world"} [3]. It stores the schema structure, including the type and size of each key/value pair, interspersed with the record content. The BSON records can only be stored in a row-oriented manner, which limits both the compression and search speed because it has to scan the entire record even when the user only searches for a specific key. Fast search can only be achieved via creating external indexes.

PostgreSQL®'s jsonb [8] and Oracle®'s OSON [26] are two other examples of custom schema structures. The former is similar to BSON, while the latter stores more metadata information, such as the number and offsets of nested keys.

Steed [30] proposes both row and column oriented storage methods. It operates on the merged schema tree (MST). Instead of storing the key name with each record, Steed only needs to store the node ID of the key in the MST. However, it still keeps one schema structure for each record in its row-oriented method. Keeping track of the schema structure is even more complex in its columnar method, as it splits the datasets into N independent columns where N is the number of keys, each column stores all the values of that key. Because the key/value pairs of a single record are now split, assembling the original record's schema structure requires even more complex data structures (2 additional columns for each key) and algorithms (finite state machine).

Scuba [17] is an in-memory SSDMS that keeps records in a compressed row-oriented format. Records are stored one after another in a table. Therefore it needs to scan all records in a table (and all the fields) during search. It stores the string values in a dictionary, and the dictionary is used as an index during search. However, unlike CLP, the entire string is stored as a single entry instead of being parsed into timestamp, log type, and variable values. It also has some restrictions on the data model, most notably it prohibits nested keys. Note that Scuba is designed for general-purpose data storage instead of narrowly focusing on logs.

Extending RDBMS. Sinew [29] uses a hybrid design that materializes a subset of the keys as separate columns, and stores the remainder using a binary format similar to BSON in a single RDBMS column. JSON Tiles [20] extends the idea of Sinew, reordering the records to group those with

similar schemas into disjunct tiles. However, it still needs to keep a binary schema structure for each record. Although JSON Tiles applies compression to those separate columns, the compression ratio is around 1 because the large size of the per-record schema structure.

Some RDBMSes map semi-structured data into relational tables. For instance, Argo [19] proposed two methods. The first stores fields in a single table, while the second splits it to three where each is used for a primitive value type. However, the first method will result in sparse tables (many fields with NULL values). Both methods take up too much storage space due to repetitive key/values. Liu *et al.* proposed to store the entire JSON record in a single RDBMS column [25].

Elasticsearch and ClickHouse: Experiences at Uber. Elasticsearch [21] is a JSON document store that supports search. It assumes the type of the field will never change by default and cannot handle fields with dynamic types easily. (By default, Elasticsearch will drop records if a field has a different type than in a previous record.) It also struggles to handle the case where every record has new unique keys (e.g., using a UUID as a key) [18]. At Uber we used to use Elasticsearch, but migrated away (to ClickHouse) due to these issues as well as its excessive resource usage.

ClickHouse is a columnar RDBMS with features for handling JSON data. At Uber we built a layer to transform records before ingestion into ClickHouse. After experimenting with different storage formats, we settled on storing a handful of common fields in dedicated columns, and storing all other unique key-value pairs that are less than or equal to 3 levels deep in a pair of arrays per type. For instance, string-type fields were stored in two arrays, "`String.names`" and "`String.values`". To query a field, we first find the key in the `String.names` array, use the index of that key to index the `String.values` array, and finally compare the value against the query. We also keep a `_source` field that contains raw JSON and build inverted index on top of it. Finally, to improve query performance for frequently accessed fields, our abstraction layer temporarily extracts these fields into temporary dedicated columns (materialized columns). This setup results in a compression ratio lower than 4:1.

2.3 Compressors

General-purpose compressors like Gzip [23] and Zstandard [22] use a sliding window approach proposed by Lempel and Ziv [33]. They locate duplicates within a fixed-size sliding window, so they cannot detect duplications if the distance between the two duplicated patterns is larger than the size of the sliding window. Therefore storing duplicated patterns close to each other would maximize the compression ratio.

Searching, unfortunately, requires decompressing the entire data. These compressors typically encode duplicates in length-distance pairs [28, 33]. Starting from the current position, if the next L (length) characters are the same as the ones starting at D (distance) behind, the next L characters can be encoded with (D,L). This (D,L) pair is directly embedded in the compressed data, hence search requires decompression from the beginning.

Log compression and search. Existing log compression techniques focus only on unstructured (i.e., free text) logs. CLP [27] uses a custom-designed log parser to split each message into three components: timestamp, static text (i.e., log type), and variable values, structurizes logs into a table of three columns. CLP stores the static text and repetitive variable values into respective dictionaries, and the dictionaries also serve as coarse-grained index during search to minimize decompression and scan. It then compresses the three-column table in columnar order.

LogGrep [31] also compresses unstructured logs and allows search. It uses a training phase to identify the common patterns of messages. LogGrep identifies patterns in much finer granularity (e.g., a variable can be further divided into subcomponents if a different subcomponent is repetitive). Therefore a message is split into a larger number of subcomponents without clear mapping to program semantics, and it uses tables to store the complex mapping to assemble these subcomponents into the original log message.

3 Characterizing Semi-structured Log Data

We first provide a characterization study on semi-structured log data before describing the design of μSlope. While prior works have characterized semi-structured *user-generated* data [30], we are the first to provide an understanding of *machine-generated* semi-structured log data.

We collected 16 frequently used log datasets (LogA-LogP) from Uber and 5 log datasets from open-source software (Apache Spark™, MongoDB, CockroachDB®, Elasticsearch, and PostgreSQL). All of these datasets are in JSON format. We limit each dataset to 1,000,000 log records. We also provide a characterization of real-world queries on semi-structured log data by analyzing a total of 23,091 queries spanning twenty days from Uber; 7665 of them are unique.

Schema Variation. We first study the schema variation. JSON's schema-less nature means that the variation of schemas between records can range from zero (i.e. all records have the same schema) to 100% (i.e. all schemas are different). Recall from §2.1 that the schemas of two log records are considered the same if and only if their schema trees are identical. The degree of variation is a critical consideration to the design of μSlope and prior systems. If there is no variation, then one can easily store logs in an RDBMS by materializing one column for every leaf-node node.

Figure 4 shows the unique schemas for each dataset. All except two datasets have more than 1 unique schemas, with LogE having the largest variation (6,176 unique schemas). The median number of unique schemas of all datasets is 40.

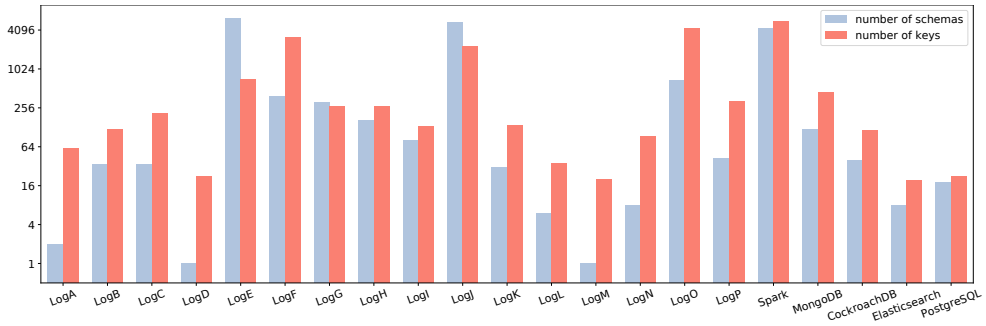

Figure 4: Number of unique schemas and keys for each dataset.

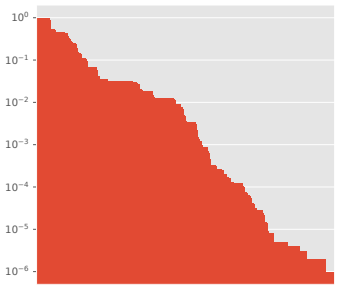

Figure 5: KF distribution of LogE

Despite the relatively large degree of variation, there also exists a large degree of *repetition*. On average, there are 25,000 records in each dataset with the *same* schema; if we increase the sample size, this repetition will be larger. Even for the most noisy dataset, LogE, there are still an average of 162 records per schema.

To understand the schema variation, we further measure the variation of individual keys. Figure 4 also shows the number of unique keys for each dataset. Two keys are considered the same if and only if their full name and value types are the same. A key's full includes all the nested keys (i.e., predecessor nodes in the schema tree). For example, "serviceA.traceID" is the full name whereas "traceID" is not. The median number of unique keys is 138, and it varies greatly. LogM has only 20 unique keys (and it has only 1 schema), whereas Spark logs have 5,627 unique keys.

This result suggests that the variation in schemas is likely due to the variation of individual keys instead of their combinatorial effect. In theory, a large variation in schemas could be the result of a small number of keys: n unique keys could result in 2^n different combinations, hence 2^n schemas. However, this is not the case with logs. In fact, in 18 out of 21 datasets the number of unique keys is larger than the number of schemas, showing the opposite of a combinatorial effect.

To get better understand how keys are distributed within a dataset, we measure the key frequency (KF). It is defined as

$$KF(x) = \frac{number\ of\ records\ that\ contain\ key\ x}{total\ number\ of\ records}$$

Figure 5 takes LogE as an example to show the distribution of KF. Each bar represents a key. LogE has 6176 schemas and 704 keys. There are a small number (21) of keys that have KF = 1.0, indicating that they appear in every record. These are the keys uniformly added by the logging library, such as "timestamp" and "level".

On the other hand, there is a long-tail in the KF distribution of LogE. 83.0% of the keys has a KF <0.1. Most of them are different data structures in the program that documents the program state relevant to a specific event. There are also cases where the variation in schemas is caused by a large variation in the name of a key, like Spark using the pathname as the key name, or some datasets using the UUID as the key name.

Type Composition. Next, we break down the value types. Recall from §2.1, the value type can be an object, array, or one of the primitive types. We further refine the types as follows. First, we break down the number type into integer and float. For strings, we separate single-word values from multi-word ones (using white space as word delimiter), because the former is likely a variable (e.g., an identifier) whereas the latter is free-text log. We call the former *variable* and the latter *log-text*.

Figure 6 shows the breakdown of the value types in each dataset. On average, 70.8% of the values in each dataset are variables (i.e. single-word strings). 10.8% are objects, i.e. non-leaf nodes in the schema tree leading to nested keys. In comparison, the percentage of boolean, float, null are low, averaging 1.74%, 1.21%, 0.22%.

The percentage of **array** fields are also low at 0.79%. In addition, only 28 of the 7,665 (0.4%) of the unique queries explicitly search on an array field. Furthermore, these explicit array searches only match 0.05% of the data on average.

Each log record contains an average of 1.6 log-text keys. In addition, 41% of the unique queries contain filters on log-text, so efficient search on log-text is important. In addition, a record contains 4.0 integer fields on average.

Repetitiveness of Variables. Next, we zoom into variable fields (i.e., single-word strings), because they dominate the composition of logs. The question we care about is: Are these values repetitive? We use the repetition ratio to measure the repetition of variable fields. It is defined as

$$repetition\ ratio = \frac{number\ of\ all\ variable\ values}{number\ of unique\ variable\ values}$$

A high repetition ratio means that unique variable values are much fewer and these fields are repetitive. Figure 6 shows the repetition ratio. The median repetition ratio is 37.8 and the average is 58.2 across all datasets. It can be as high as 433.4 (LogD) and even the minimum is still 9.29 (LogC). This means dictionary deduplication can be effective.

The variable fields are also frequently queried. The average query contains 3.450 filters with 2.663 of those being variable filters. For example, `level:"warn" OR level:"error"` is a query that has two string filters on the `level` key. The largest query has 93 string filters. Furthermore, filters that implicitly

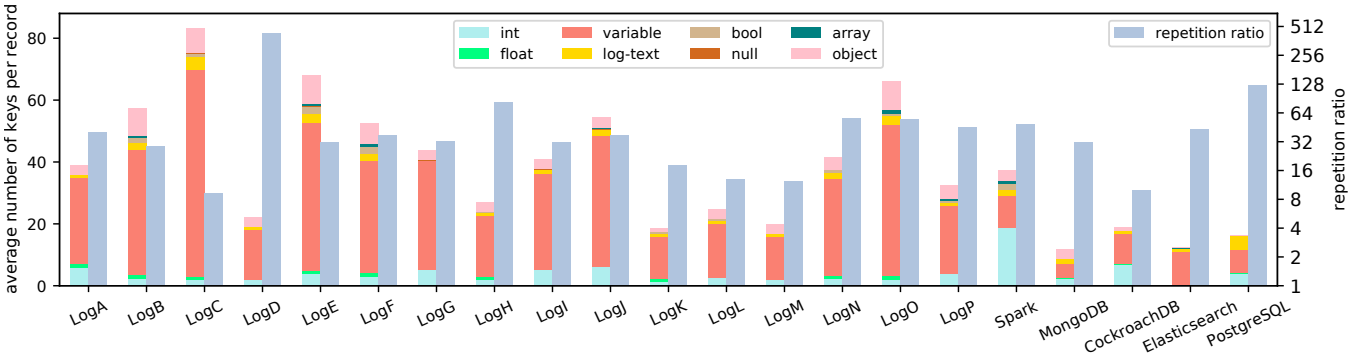

Figure 6: Average number of keys per record, broken down by different value types and repetition ratio of variables.

match all keys – "wildcard keys" – are common in practice. For example, `*:"aUUID"` matches any keys as long as its value is "aUUID". Of the 7,665 unique queries 2,271 of them contain a wildcard key. These filters are easier to express for users of search, but pose a significant performance hazard. In the worst case such filters can impose scans over the entire record, eliminating much of the benefit of querying structured data. However, using a dictionary to store the variables would significantly speed up such queries as we only need to search the dictionary for matching values.

Importance of Schema Search. Nearly one third (29%) of the unique queries do not match any of the schema structures. That is, they can be returned without scanning the values. Examples include searching for a nonexistent key, the key exists but the value type does not match, or the conjunction of keys/value types do not exist. For example, engineers performs such search to regularly verify the nonexistence of certain error events. However, for existing SSDMSes (such as MongoDB and PostgreSQL) this opportunity is wasted, because the schema structure is interspersed with the values.

Summary and Takeaways. Schemas are dynamic, and those unstructured keys are frequently queried. Therefore simple extension of RDBMS to only materialize those structured keys as columns is insufficient, and we need to precisely track the schema of semi-structured data. On the other hand, a large number of records have the same schema, presenting opportunities to group them by the same schema so that each group is well-structured.

70.8% of the keys are single-word strings. They are highly repetitive, and commonly queried on. This indicates that storing them in a variable dictionary would effectively deduplicate them, and at the same time, speedup the search.

Finally, efficiently storing the schema structure and decouple it from the record value data would significantly speedup the 29% of the queries that can be completed only by querying the schema structure.

4 Overview of μSlope

μSlope is a resource-efficient SSDMS that we designed from the ground-up. μSlope has three major designs that are novel compared to other SSDMSes: (1) Decoupling the storage of schema metadata from the storage of each record; (2) Grouping records by schema to store them into well-structured tables, and apply efficient encoding; and (3) Optimized schema metadata lookup to speedup search.

Figure 7 shows μSlope's architecture. When data is ingested, μSlope parses each record and extracts its schema. It uses two core data structures to track the schema structure: the merged parse tree (MPT) and the schema map. The MPT and the schema map are collectively referred as *schema metadata*. It is critical to keep the schema metadata as small as possible, yet still captures the highly repetitive structure of the log.

The MPT is a more general form of the merged schema tree (MST) as described in §2.1. It has four differences when compared to MST. First, the MPT can contain special unnamed nodes that mark the truncation of some key value pairs. Multiple rare keys can be mapped to the same unnamed node when the key names contain random data, such as UUID or file path. Including such non-repetitive key names would bloat the metadata. In contrast, sometimes the value of a key could be highly repetitive. For example, all the records from the same application would have the same value under the application-name key. Therefore, the MPT also allows a node to include the value (only if the value is highly repetitive).

The third difference is that MPT can encode the structure of strings with key-value pairs. This allows μSlope to capture more structural information from strings, improving compression and search. For instance, given a record `{.. "message": ".. latency=35, status=OK, type=READ, ..."}`, μSlope would create three nodes for "latency", "status", and "type" respectively as the children of the "message" key. This requires that schemas contain an *ordered* region where we maintain some leaf nodes in a specific order, because the order of the keys in a string needs to be preserved.

Another difference is that MPT stores more fine-grained string types. A string value could either be a timestamp, a single-word string, or a log-text. Storing fine-grained types

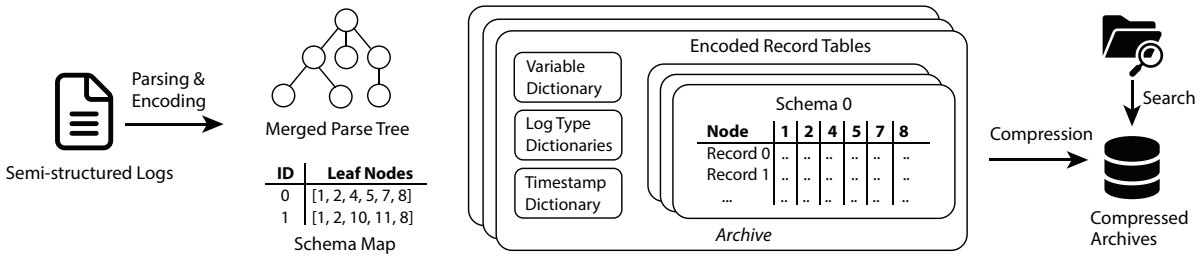

Figure 7: Architecture of μSlope.

enables more filtering opportunities on MPT, resulting in better search performance.

The *schema map* stores each unique schema in the log dataset in a hashmap. We make the observation that a schema can be unambiguously identified with the list of *leaf nodes* in the MPT. For example, the schema map in Figure 7 shows the two schemas of the two records in Figure 1, where the node IDs corresponds to the MST shown in Figure 2. (In this example, the MPT and the MST are the same.)

The MPT and schema map deduplicate the highly repetitive schema structures. Unlike prior SSDMSes §2.2 that store a schema structure for every record, each unique schema is stored only once. In practice, the schema metadata size is typically less than 0.0001% of the total compressed data size. This design also enables fast search: the succinct metadata can be efficiently searched, providing powerful filtering capability.

μSlope uses one table for each schema to store the values. Therefore each table only stores the values of the records that have the same schema. As a result, there can be thousands of tables, one for each schema. The advantage is that each table is now perfectly structured, as all records have the exact same keys and value types. μSlope essentially structurizes those semi-structured data.

The tables are called *Encoded Record Tables* (ERT), because instead of storing their raw value, μSlope performs type-specific encoding. For example, single-word string will be stored in a *variable dictionary*, so only an ID is stored in the ERT. The dictionaries serve two purposes at the same time: it effectively deduplicates the highly repetitive patterns, and it serves as coarse-grained index for search so μSlope only needs to scan the ERT that contains the matching record.

Each ERT is stored and compressed in a columnar order. This significantly improves both the compression ratio and search performance [16], because a column groups the semantically similar values together so it maximizes general-purpose compressors' ability to find repetitions, and during search we only need to decompress and scan the columns whose keys were searched for.

μSlope leverages the efficiency of metadata and dictionary lookup to optimize the search performance. It uses Kibana Query Language (KQL) as its query language, which is both concise and powerful. μSlope transforms a query into an abstract syntax tree (AST) to perform a number of optimizations, including determining if the query matches any schema and if the filter pattern matches any dictionary values. If not, μSlope will terminate query processing early. Otherwise, only relevant ERTs are finally decompressed and searched through.

Log Type Dictionary

ID	Log Type
L0	Can't fetch flow \INT, cell \DICTVAR

Timestamp Dictionary

ID	Format
T0	yyyy-MM-dd'T'HH:mm:ss'.'SSS

Variable Dictionary

ID	Variable Value
V0	error
V1	cell_32
V2	abc-xyz
V3	Error404
V4	driver_onboarding
V5	def-uvw
V6	vehicle_compliance

Schema 0 Encoded Record Table

Node	1	2	4	5	7	8
Values	V0	L0	6,V1	V2	V3	V4	T0	1..8

Schema 1 Encoded Record Table

Node	1	2	10	11	8
Values	V0	L0	8,V1	V5	V6	T0	1..6

Figure 8: How μSlope encodes log records.

Both compression and search are embarrassingly parallel. During compression, when parsing a new record μSlope extracts the key-value pairs of each log record. Corresponding key nodes are added to the MPT, with leaf node IDs collectively representing a schema. Values are encoded by various methods and are stored in the ERTs. Upon identifying a new schema, μSlope dynamically creates a new ERT to store the encoded values. All the ERTs, dictionaries and schema metadata will be buffered in the memory. Once the buffer reaches a certain size, they are compressed using Zstandard before stored to disk, creating what we call an *archive*. μSlope then clears the buffer and dictionaries before compressing newly arrived records. Therefore different archives can be compressed, searched, and decompressed independently in parallel.

5 Compression

We use simdjson [11] parser to parse the JSON structures. Other log formats representable by the data model in §2.1 can also be supported by integrating a parser to extract the key-value pairs from the records.

μSlope uses different encoding techniques on different value types. Next we describe them in turn.

Strings. μSlope treats string values differently depending on whether it is (1) a timestamp, and if not, (2) a variable, i.e. a single-word string, or (3) log-text. μSlope uses CLP's parser to parse a string value. It uses heuristics to detect if the string is a timestamp. If so, μSlope encodes it into a Unix® epoch time and stores the format pattern in the timestamp dictionary. As a result, each timestamp field is encoded into *two* columns in the ERT: a timestamp dictionary ID and a Unix epoch time. The ID column consumes negligible space after compression because most of the time there is only one timestamp format.

For a log-text, the CLP parser extracts the log type and variables. The log type is stored in the log type dictionary. Different types of variable values are encoded differently. Integers and floating point numbers are directly encoded in binary format. Other variables are encoded as a variable dictionary ID after storing them in the variable dictionary. Therefore a log-text also has two columns: a log type ID, and a list of encoded variables stored in a single column. We extend CLP's log parser [7] to allow users to specify rules to extract key/value pairs from log-text fields and store them as JSON fields. Therefore, if the log is in unstructured format (instead of JSON), or dominated by unstructured text log (i.e. majority of a record is an unstructured text message field with a few additional fields containing metadata like hostname, verbosity, etc.), μSlope essentially falls back to CLP encoding.

μSlope directly stores a single-word string in the variable dictionary and stores the dictionary ID in the ERT. The dictionary effectively deduplicates the repetitions in variables as we shown in §3. We use the same variable dictionary used for the log-text for maximum deduplication.

Figure 8 shows the contents of the dictionaries and encoded record tables for the example log records in Figure 1. Their MPT is the same as the MST shown Figure 2, except that the MPT keeps a fine-grained string type on each string field. The schemas are shown in Figure 7. We use different prefix and colors for the different types of dictionary IDs: log type ('L'), timestamp ('T'), and variable ('V'). For example, consider the first log record, which is stored in the ERT of schema 0. Four of its keys ("level", "serviceA.traceID", "error", "request.namespace") have single-word strings; they have the MPT node IDs 1, 4, 5, 7 respectively (Figure 2), and their values are encoded as V0, V2, V3, and V4 which are IDs into the variable dictionary. The "message" field (node ID 2) is a log-text. Its first column stores L0, which is the ID into the log type dictionary, and the second column stores the two encoded variables. Note that '\INT' and '\DICTVAR' in the log type are special placeholder bytes for variables (of integer and dictionary variable types respectively). The "timestamp" value is stored in the last 2 column (node 8); the value "1..8" is the encoded timestamp in Unix epoch time.

Integers, floating point, and boolean values are directly encoded in binary form in the same way as in CLP [27].

Arrays are stored as log-text strings by default, i.e., using CLP to parse it into a log type and a list of variable values.

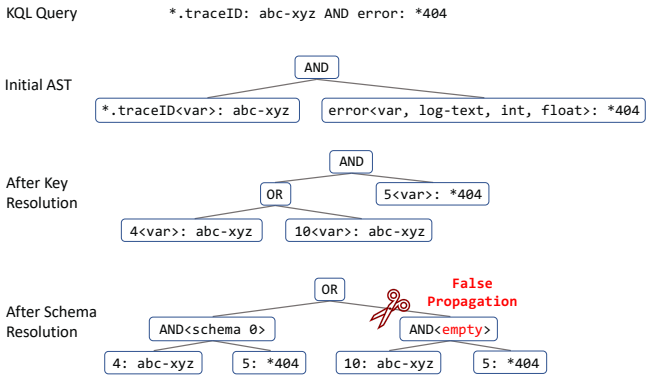

Figure 9: Example of query processing.

This means that arrays are typically searched by decoding and parsing their string representation. This is acceptable as §3 shows that arrays occur rarely and are seldom searched. Note that the log type parsed from an array string is stored in a separate log type dictionary to avoid polluting the regular log type dictionary for log-text. We also support fully parsing arrays and recording their structure in the MPT under a non-default configuration. This approach offers performance benefits for array searches, but typically results in growth in schema size due to the diverse internal structures of arrays.

Preserving record order. Splitting the records into different tables means that we lose the order between records of different schemas. Using timestamp to order them is unreliable because records may have the same timestamp. To preserve the order, μSlope adds a column in each ERT to store the order of the record in the original log stream.

Random keys and invariant values. Recall that μSlope truncates the key name from the MPT if it is not repetitive, and includes values that don't change into the MPT. The heuristic we use is that if a key does not appear in more than 1% of the records of the archive, it will be truncated; whereas a value will be included in the MPT only if it never changes. We implement them by keeping counters for the keys and values as records are parsed and stored in the memory buffer; the decision to truncate a field or include a value in the MPT is made when we write the buffered data to disk (into the archive format). The structures of a truncated field and its successors (a truncated field may be a non-primitive type, in which case all the subfields will be recursively truncated) are encoded in a row-oriented format similar to BSON, and stored in a column that is mapped to a (special) MPT node located at the place of the truncated node.

6 Search

μSlope search accepts queries which combine filters on one or more keys. The key names in the query may contain wildcards and be of ambiguous underlying type. Search takes advantage of the MPT, *schema map*, and dictionaries to evaluate queries

efficiently. This is a multi-step process, each step performs refinements and optimizations on a custom Abstract Syntax Tree (AST) μSlope built from the query. Next, we explain each step using an example shown in Figure 9.

Step 1: Constructing the initial search AST. Given a query μSlope transforms it into an AST where each leaf node specifies a filter on some key, and each non-leaf node is a logical AND or OR. Figure 9 shows an example KQL query. The query consists of two filters joined by AND. The key name in first filter contains a wildcard, meaning it can match any hierarchy of keys that end with "traceID". The second filter contains a wildcard in the value pattern.

μSlope parses this query into a search AST shown in Figure 9. Initially this AST contains two nodes, each maps to a filter. Each node also stores the possible value types inferred from the query. The value type of the first filter is unambiguous; it must be single-word string type (i.e., a variable) given the pattern is surrounded by '"'. The second filter, however, has ambiguous type; it can either be a variable, a log-text, an integer, or a floating point number.

Step 2: Key resolution further turns each key name into zero or more MPT leaf nodes by resolving ambiguities. Ambiguities come from two sources: (1) ambiguous key name, i.e. a key with wildcard can match more than one leaf of the MPT, and (2) ambiguous value type, when it is polymorphic. In both cases we replace the corresponding filter with an OR node where every child of the OR is the same filter with the key replaced by each of the matching MPT leaf nodes in turn. The "*.traceID<var>" in Figure 9 is an example of ambiguous key name. It is resolved into two MPT leaf nodes, node 4 and 10, which corresponds to "serviceA.traceID" and "serviceB.traceID" respectively. These two nodes are connected by OR in the refined AST after the key resolution.

The "error<var,int,float>:*404" node in the initial AST has potentially ambiguous value type. In our example, because the only possible type for the "error" key is a variable (i.e., single-word string), we replace it with a single node "5<var>:*404". However, if "error" has polymorphic type, say an integer, in the dataset, then we need to consider both possibilities and connect them by an OR.

When a key name matches no leaf nodes in the MPT, that AST node is eliminated by replacing it as false and propagating this false up the AST, a process known as *false propagation*. This can eliminate part or all of the query.

After key resolution each key in the search AST refers to a leaf node from the MPT. The one exception is if the searched key name is a single wildcard '*'. Expanding such keys would result in a bloated AST because it can match any key name, adding combinatorial overhead to the later steps (particularly if the query specifies multiple filters on '*' key). Wildcard keys are expanded dynamically only at the last step, before search on an ERT.

Step 3: Schema resolution looks up the *schema map* to find a set of schemas that match the record structure implied by the query. It first transforms the AST into OR of ANDs form (i.e. sum of products). The key insight is that for an AND to ever be true, all of its children must exist together in a schema. We implement this check by performing an intersection between the set of MPT node IDs of the children of an AND node with each schema. If the intersection is empty, the entire AND sub-tree is removed by treating it as false, and we propagate this false along the AST.

The last AST in Figure 9 shows the AST after schema resolution. The rightmost AND expression matches no schemas and can be removed since MST node 5 and 10 never appear in the same schema. In this case we are able to narrow down the ambiguous query to a single schema, schema 0, by only searching the schema metadata.

Step 4: Search on strings. Next μSlope searches the dictionaries on relevant string filters. Search needs to be performed over the log type dictionary, the variable dictionary, or the timestamp dictionary. Searches on log-text are handled the same way as CLP would. The ability of the dictionaries to reject string queries is critically important for performance. Consider the query *:<uuid>, which is commonly issued at Uber. In archives that do not contain this uuid, this query can be terminated early after searching the variable dictionary, avoiding any column scan. In general, an empty dictionary search would result in the AST node being eliminated, and the false value gets propagated to further simplify the AST.

Step 5: Column decompression and scan is guided by the remaining nodes in the AST. Specifically, the remaining AST tells us exactly which ERT(s), and which column(s), should be scanned. This minimizes the decompression and scan.

Note that we also add a simple timestamp range index at the archive level. This is used to avoid having to decompress and scan any data in the archive when there is no overlap with the time range specified by the query.

7 Implementation

The implementation of μSlope closely follows the design specified in the previous sections. However, some details not called out in the design are critical to the overall performance of the system, so we highlight them here.

We have written a custom JSON serializer in order to improve decompression and search speed. With each schema precisely defining the structure of a log record, μSlope is able to generate a bytecode that describes how to reconstruct records in terms of the columns they have been split into. Unlike JSON serializers designed for dynamic objects, our serializer doesn't require the creation or traversal of mutable in-memory data structures during serialization. Instead, it uses the bytecode generated at the table granularity to directly append the values to the serialization buffer. This approach has proven several times faster than conventional JSON serializers based on our experience.

Given that μSlope can sometimes produce archives containing many small ERTs, minimizing the overhead of storing and loading ERTs is crucial. In μSlope, ERTs are concatenated together into a single file, with a metadata file that describes the location of each ERT within the file and the number of log records within each ERT, which reduces I/O overhead. During search, μSlope scans the AST to determine which ERTs need loading, and then loads them following the storage order, thus eliminating random I/O. Additionally, it optimizes bytecode generation by only producing bytecode for serializing records from an ERT after finding at least one matching record.

8 Evaluation

We implement μSlope with about 18k lines of C++ code. We evaluate the performance of μSlope on both Uber logs and public logs. Specifically, we focus on the following aspects: (1) the compression ratio and speed; (2) the query performance; (3) worst case performance on synthetic logs; (4) the scalability of μSlope on large-scale logs.

8.1 Experiment Setup

Overall we conduct the experiments in two setups: (1) single-thread, single-process experiments on smaller datasets (referred as single-thread experiments), and (2) parallel setups on larger scale Uber's logs. For (1) we compare μSlope with a number of other SSDMSes. The logs are from the same services as described in §3, except that here we increased the data size. In addition, we include logs from five public software, which are generated by running HiBench [24] and YCSB [14] benchmarks. Table 1 shows the size and the number of records of each log dataset. These datasets are relatively small because (1) we had problems to ingest larger data to some of the tools we compare with (for example the ingestion throughput for Elasticsearch is 5MB/s), and (2) μSlope is embarrassingly parallel, therefore its single-threaded performance is the most critical. We also evaluate μSlope on larger Uber's datasets in §8.5.

Single-thread experiments were performed on a Linux server with Intel Xeon E5-2630v3 processor and 128GB of DDR4 memory. Both the uncompressed and compressed logs are stored in a distributed file system (MooseFS) running on multiple 7200RPM SATA HDDs.

We compare μSlope with SSDMSes including CLP 0.0.2, MongoDB 6.0.5, PostgreSQL 15.2, ClickHouse 23.3.1.2823, Elasticsearch 8.6.2, Zstandard 1.4.9 and XZ Utils (for LZMA compression) 5.2.2. (we were informed that Steed [30]'s artifact isn't yet available upon contacting the authors). MongoDB and PostgreSQL have native JSON support (i.e. BSON and `jsonb` data type respectively). For ClickHouse, we explore three setups to store JSON records: (1) in pair-wise arrays which was described in §2.2. Here we only use two arrays and do not differentiate types of the values. (2) in a single string field, which can be parsed by ClickHouse functions. Both setups are commonly used in practice for JSON management. (3) in a single JSON field, which is a new experimental data type introduced in v22.3. It can infer the schema of a JSON record and store every field in a separate file automatically.

	Name	Uncompressed Size	Number of Records
Uber Logs	LogA	30.0GB	22,996,492
	LogB	47.1GB	16,606,964
	LogC	60.4GB	15,306,125
	LogD	50.7GB	58,309,754
	LogE	91.8GB	22,345,071
	LogF	102.9GB	17,251,752
	LogG	30.9GB	3,046,845
	LogH	30.8GB	11,461,221
	LogI	39.7GB	27,209,375
	LogJ	36.0GB	13,605,274
	LogK	30.2GB	57,919,224
	LogL	37.1GB	45,827,554
	LogM	36.5GB	42,206,452
	LogN	38.0GB	22,307,407
	LogO	38.6GB	4,438,786
	LogP	38.3GB	34,840,347
Public Logs	Spark	2.0GB	1,011,651
	MongoDB	64.8GB	186,287,600
	CockroachDB	9.8GB	16,520,377
	elasticsearch	8.0GB	140,012,234
	PostgreSQL	392.8MB	1,000,000

Table 1: Log datasets used in our experiments.

For these systems we do not create any index for a fair comparison with μSlope, because μSlope is designed to be an archival SSDMS and does not have any external index. MongoDB automatically builds an index on the default key `_id` and we exclude the size of the index when calculating compression ratio.

We also compare μSlope with two general-purpose compressors Zstandard and LZMA. Zstandard is the underlying compression method for μSlope and LZMA is known for its high compression ratio.

We do not evaluate search on CLP because wildcard queries (which CLP supports) are incompatible with semi-structured data model. For example, a KQL query `error: "*404"` searches for the "error" field whose value ends with "404"; but CLP could return records like {"error": "0", "keyx": "404"}, because '*' could match an arbitrary amount of text. We do not evaluate search on Elasticsearch because we cannot ingest the three datasets where we designed query benchmarks on into Elasticsearch. Nevertheless, as we will show that Elasticsearch consumes too much storage space that cannot be used as an archival SSDMS.

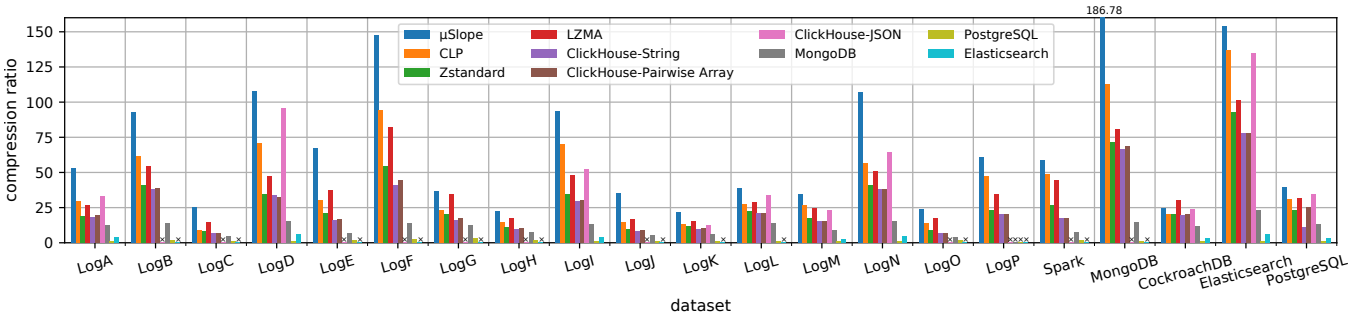

Figure 10: Compression ratio of μSlope and other tools on different JSON datasets. A "×" means the dataset cannot be ingested by the tool.

8.2 Compression Ratio and Speed

Figure 10 shows the compression ratio of μSlope and other tools on different datasets. The compression level of Zstandard is set to 3 (default) for μSlope. For a fair comparison, we use the same compression method and level for CLP, ClickHouse and MongoDB. PostgreSQL and Elasticsearch do not support Zstandard compression, so we use lz4 [32] instead. For LZMA, we use the default compression level 6.

Note that out of 21 datasets, ClickHouse-JSON can only ingest 10 and Elasticsearch can only ingest 9. The most common reason is that they cannot accept fields with the same key name but different types. Additionally, MongoDB and PostgreSQL cannot ingest one dataset because of the escape character. For the average compression ratio and speed comparison, we only include the datasets that are successfully ingested by these systems.

μSlope achieves the highest compression ratio on all JSON datasets. The average compression ratio of μSlope is 68.1:1, ranging from 21.9:1 (LogK) to 186.8:1 (MongoDB). On average, μSlope's compression ratio is 2.75x of ClickHouse-String's, 2.62x of ClickHouse-Pairwise Array's, 1.34x of ClickHouse-JSON's, 6.10x of MongoDB's, 16.50x of PosgreSQL's, and 15.71x of Elasticsearch's. It surpasses Zstandard's, LZMA's, and CLP's compression ratios by factors of 2.34x, 1.70x, and 1.50x respectively. but it is only 4.8% on average, which still makes μSlope's compression ratio the best among all.

We delve into the breakdown of compressed data size in μSlope, using LogP as an example. Out of the 710 MB total compressed data, the MPT and schema map only occupy 3.6KB and 1.9KB, respectively. Dictionaries account for 26.3% of the storage space, with the remaining 73.7% attributed to compressed columns of ERTs.

Figure 11 shows the average ingestion speed on all datasets. μSlope's ingestion speed is slower than ClickHouse-String and ClickHouse-Pairwise Array because ClickHouse-String directly store the raw JSON string and does not parse it, while ClickHouse-Pairwise Array only parses the top-level fields. In comparison, μSlope parses every field of the entire JSON record. μSlope's ingestion is slighter slower than CLP and faster than all other fully-parsed JSON tools, outperforming

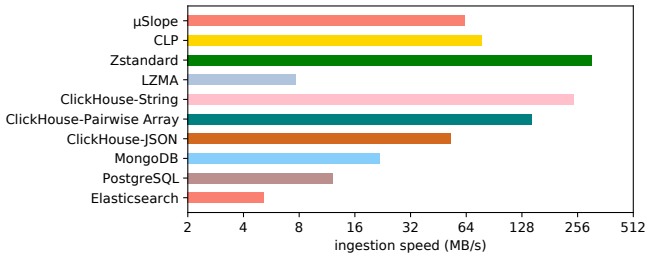

Figure 11: Average ingestion speed (log scale).

ClickHouse-JSON, MongoDB, PostgreSQL and Elasticsearch by 19.3%, 186.7%, 419.8%, 1127.3% respectively. Additionally, it exceeds LZMA's performance by 814.8%.

8.3 Search Performance

We use 15 queries to evaluate the search performance of μSlope and other tools on Uber LogF, LogO and MongoDB logs. For queries on LogF and LogO, they are the top queries performed in Uber (with repetitive patterns removed). For queries on MongoDB, we try to cover different possible patterns. Table 2 shows the queries in KQL [6]. For ClickHouse and PostgreSQL, we convert KQL queries to SQL queries with their built-in functions and operators. For MongoDB, we use their own query language.

Query B is a special case. It does not specify any search key, but searches for all fields for the matched UUID. Other tools does not natively support this kind of query so we have to convert it to a full-text search instead. It works for Query B, but may get incorrect results for other queries that span across keys and values. MongoDB is required to have a text index on that table to do a full-text search. However, after running for 196 seconds, it reports an error.

We clear the OS buffer cache before every run. This is to simulate search on archival storage. However, by default MongoDB uses about half of the memory (63.5G in our machine) to cache uncompressed data and the cache cannot be cleared, while others use only a minimum amount (or even no) or the cache can be cleared. For a fair comparison, we test MongoDB with the minimum cache size.

Figure 12 shows the query latency of those 15 queries on

Queries for LogF	
A	zone:... AND NOT @reserved.collector.filename:stdout AND runtime_env:staging
B	*: "d...-...-...-...-...9"
C	level: error AND message: d...*
D	timestamp >date("2022-04-14T08:00:00.000") AND timestamp <date("2022-04-14T08:15:00.000")
Queries for LogO	
E	headers.x-tenancy:testing* AND NOT headers.x-tenancy: testing/.../4...-...-...-...6d AND headers.caller-procedure:"fareEstimateV2" AND headers.x-source:public
F	headers.x-region-name:... AND headers.x-tenancy:"production" AND caller:*create*
G	level: error AND NOT @reserved.collector.filename: executor AND runtime_env:production AND partition: compute-... AND instance: 3...5 AND mesos_executor_id: t...5-6
H	level: error AND message: "Error handling inbound request."
I	glue.handler.method: get_ranked_products AND env: production AND level: error
Queries for MongoDB logs (public dataset)	
J	attr.tickets: *
K	id: 22419
L	attr.message.msg: log_release* AND attr.message.session_name: connection
M	ctx: initandlisten AND (NOT msg: "WiredTiger message" OR attr.message.msg: log_remove*)
N	c: WTWRTLOG AND attr.message.ts_sec >1679490000
O	ctx: FlowControlRefresher and attr.numTrimmed: 0

Table 2: Queries used in our experiments. "..." is used to anonymize (part) of the actual values.

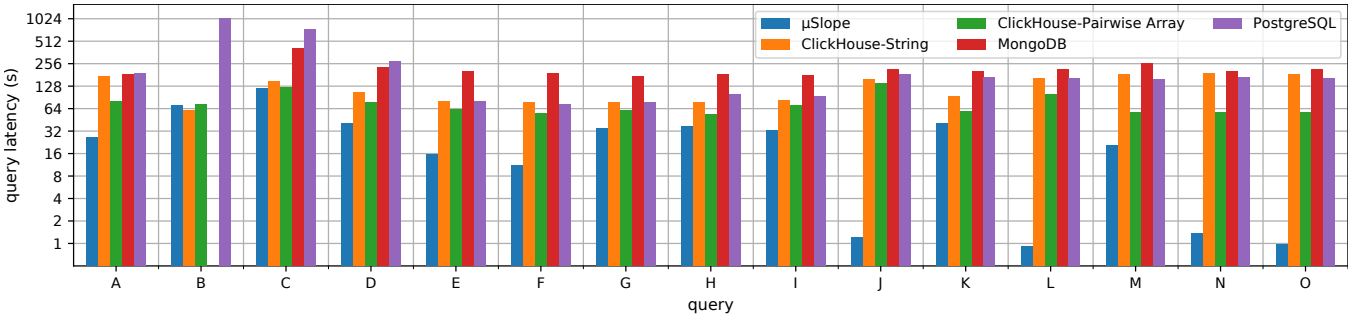

Figure 12: Query latency of μSlope and other tools (log scale).

different tools. It includes both the search time and the time to write results to the disk. On average, the speed of μSlope is 2.47x of the fastest setup of ClickHouse (ClickHouse-Pair Array), 8.09x of PostgreSQL's, and 6.74x of MongoDB's.

μSlope outperforms all other tools on 14 queries. The fast search performance comes from its use of metadata (MPT and schema map). For example, μSlope outperforms all other tools by at least 116x on Query J. This query checks the existence of a key and returns all the records containing that key. In this case, there are only a small number of schemas that contain this key, so after μSlope checks its MPT and schema map, it only needs to decompress a small number of ERTs. For other tools, they will have to scan nearly the entire dataset. Query L, N, O are also similar as there are only a small number of matching schemas and μSlope only needs to decompress small ERTs. Note that even for these queries, the schema metadata lookup is not the bottleneck. For Query J, for example, searching the MPT, schema map and ERTs only accounts for 5.5% of the total query time and loading dictionaries accounts for 73.0%. An even more extreme case is that μSlope can return no-match right after the MPT and schema map search, because, say, the query searches for a key that doesn't exist. In fact, our query benchmark does not even contain such best-case scenario for μSlope.

For Query B, μSlope is slower than ClickHouse-String and ClickHouse-Pairwise Array because μSlope needs to scan all the ERTs and decode them. , while the two ClickHouse setups can perform a full-text search on the raw JSON string values, without the need to decode them.

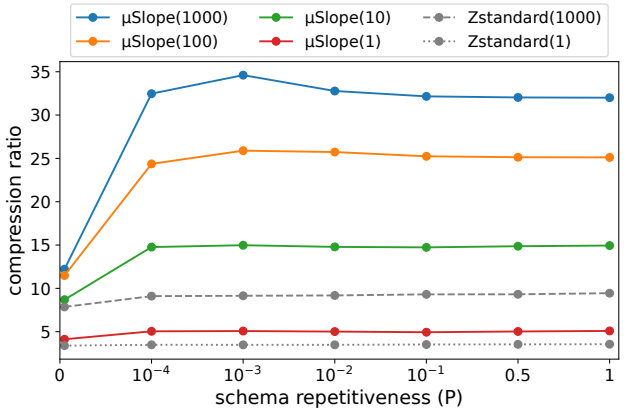

Figure 13: Compression ratio of μSlope against Zstandard on synthetic logs. The number enclosed in brackets within the legend represents the repetition ratio of variable values.

8.4 Synthetic Evaluation

The efficiency of μSlope relies on the repetitiveness of schemas and variable values. To demonstrate the boundaries of μSlope's capabilities, we evaluate its compression and search performance on a corpus of synthetic log data, which

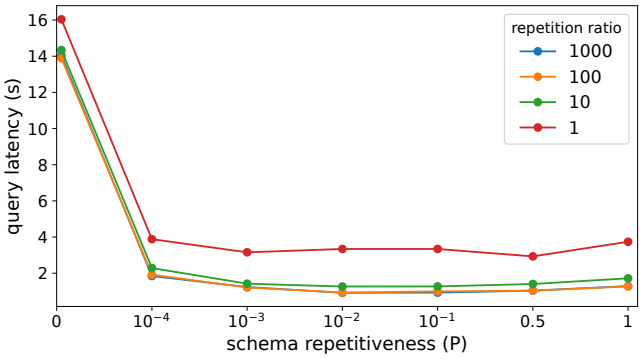

Figure 14: Query latency of μSlope for needle-in-haystack wildcard queries on synthetic logs.

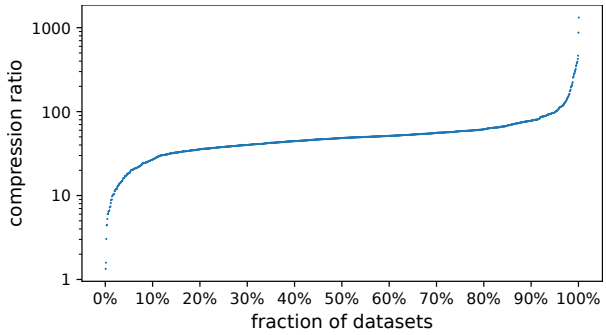

Figure 15: Compression ratio distribution of 1378 datasets in Uber's production logs.

varies in both repetitiveness of schemas and variable values.

Each synthetic dataset contains 1GB of data, consisting of 670K records. Each record has 20 fields. Every key and value is a UUID. We use UUIDs because they are one of the most common source of noises (i.e. non-repetitiveness) in the real logs. To vary the repetitiveness of variable values, the logs are generated with repetition ratios (§3) of variable values ranging from 1 to 1000, achieved by randomly drawing values from a uniform distribution. Specifically, we use four repetition ratios: 1, 10, 100, and 1000.

To vary the repetitiveness of schemas, we draw schemas from a power law distribution. Specifically, the n-th most frequent schema appears in $P \times (1-P)^n$ of the records (where n starts from 0). P is a value within the range of $(0,1]$, and it is a constant within one dataset. For example, the most frequent schema ($n = 0$) appears in P of the records, the next most frequent schema ($n = 1$) appears on $P \times (1-P)$ of the records, and so on. We use a total of 7 different P values as shown in Figure 13. The degree of schema repetitiveness increases with P. When P is the smallest every log record has a unique schema; when it increases to 1, all records have the same schema.

In total, we generate 28 (4 different repetition ratios combined with 7 different P values) synthetic datasets each with a different combination of schema repetitiveness and repetition ratio of the variable values. Figure 13 illustrates the interplay among compression ratio, schema repetitiveness, and repetition ratio of variable values. In all scenarios, μSlope outperforms Zstandard, with the compression ratio increasing as the repetition ratio of variable values increases. In the extreme case where P approaches 0, the compression ratio drops notably. This is attributed to each log record having a unique schema. The small tables and extra metadata overheads lead to a significant reduction in the compression ratio. However, the compression ratio quickly increases as P increases to the next smallest value (10^{-4}) and remains relatively stable.

We evaluate the search performance using a needle in the haystack query. One variable value is replaced with a fixed UUID value known as the "needle", so we can benchmark the query `*: "needle-value"`. The performance of this query is influenced by both the repetition ratio of variable values and schema repetitiveness. Lower repetition ratio leads to larger variable dictionaries, which are required to be loaded before decompression and scan, introducing a constant overhead before any results can be returned. This explains the consistent gap between each curve in Figure 14. As a result μSlope sometimes struggles to achieve low response time for datasets with a low repetition ratio, although this challenge can be partially addressed in practice by generating smaller archives.

Figure 14 shows a significant decline in query performance as P approaches 0. This is because in this extreme case where each record has a unique schema, we have a large number of small Encoded Record Tables where each has only one record. This significantly slows down the decompression and scan as we need to load a large number of small tables, and each table is decompressed using a different Zstandard stream.

8.5 Scalability Evaluation

We evaluate μSlope on 434TB of production logs from Uber representing 1,378 datasets, and select a 26.2TB subset from Uber's LogF to evaluate search scalability. This production dataset achieves an average compression ratio of 30.5:1, Figure 15 shows the compression ratio for each of the 1378 dataset in sorted order. The outliers with low compression ratio typically contain large amounts of random non-repeating binary data such as base64 encoded binary data and UUIDs. For example the index with the worst compression ratio has a column which appears to contain several megabytes of base64 encoded binary data in each log message.

To evaluate the scalability of search we run queries A-G from Table 2 on 26.2TB of Uber's LogF data with increasing amounts of parallelism. Values in the queries have been changed to match this dataset where appropriate. Results for query F have been omitted because its characteristics are identical to query E on this dataset.

Experiments are run across 8 containers, each has access to 96 cores, 2TB of network attached SSD, and 32GB of RAM.

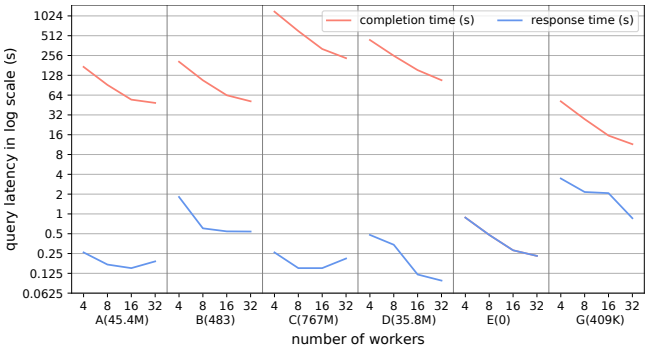

Figure 16: Query completion time and response time of μSlope (log scale) using 4, 8, 16 and 32 workers per container. The number of matching records for each query is indicated at the bottom.

There are a total of 2,155 archives after compression. These archives are evenly distributed across the 8 containers. The maximum skew between any two containers is 11% in both data size and in number of archives.

The searches are run using 4, 8, 16, and 32 workers per container. Each archive is processed by a single worker process. We show two results: (1) end-to-end completion time including marshalling all matching records, and (2) time to get the first matching record, i.e. response time. This experiment was conducted before we implemented our optimizations for JSON marshalling, so archives which contain many matching results can have an outsized impact on overall query performance. Figure 16 shows how the search performance scales with the increase of number of workers per container.

All of the queries scale well to 16 workers but have limited scalability to 32 workers. This limit is imposed by different kinds of skew in the dataset. For example in Query A a single archive becomes a bottleneck for completion time because it returns 5.9x more results than average, and in Query D all 19 archives with matching results happen to be allocated to the same container. Query E is extremely fast because it only needs to consult the MPT before terminating.

In practice, we manage this sort of skew within a dataset by producing smaller archives.

9 Limitations and Future Work

μSlope is a system designed for storing and searching archival semi-structured log data. It is not suitable for data that can be updated or deleted. Since μSlope leverages the repetitive nature of logs to achieve a high compression ratio, if the data has too many different schema structures or values are unique overall, μSlope may not be able to achieve a high compression.

As for search, μSlope provides support for basic queries, including term search, field search, wildcard search, and range search. However, currently it lacks support for more complex queries like joins. Besides, μSlope may struggle with queries that necessitate scanning the entire dataset and generating a large number of results.

The current implementation of μSlope compresses each table into its own Zstandard stream. We plan to implement optimizations to combine small tables into fewer streams (to improve compression ratio and amortize the cost of decompressing each small table), and split large tables into several streams by columns (to avoid decompressing columns in large tables not being searched on unless necessary). We also plan to improve scan performance and support more aggregation operators in the future.

10 Concluding Remarks

This paper presents μSlope, a resource efficient system for semi-structured log management that losslessly compresses the log data, and enables search without full decompression. Its design is guided by a careful analysis on the characteristics of real-world semi-structured logs and their query patterns. μSlope does not require any user annotation, and can automatically handle the dynamic schema structures. Our evaluation shows that μSlope achieves unprecedented compression ratio of up to 186.8:1, and its search speed is at least 2.47x of the fastest existing SSDMSes. μSlope is available at https://github.com/y-scope/clp.

Acknowledgements

We thank our shepherd, Andrew Warfield, and the anonymous reviewers for their insightful feedback and comments. Michael Stumm and Ashvin Goel have provided valuable feedback on an early draft of the paper. Devin Gibson has been partially supported by an NSERC Alliance Missions grant and a QEII-GSST scholarship.

References

[1] Amazon CloudWatch: Observe and monitor resources and applications on AWS, on premises, and on other clouds. https://aws.amazon.com/cloudwatch/, 2024.

[2] BSON. https://bsonspec.org, 2024.

[3] BSON example: How BSON is stored in MongoDB database. https://www.mongodb.com/basics/bson, 2024.

[4] Clickhouse. https://clickhouse.com, 2024.

[5] Grafana Loki. https://grafana.com/oss/loki/, 2024.

[6] KQL: Kibana Query Language. https://www.elastic.co/guide/en/kibana/current/kuery-query.html, 2024.

[7] Log-surgeon: a performant log parsing library. https://github.com/y-scope/log-surgeon, 2024.

[8] PostgrSQL JSON Types. https://www.postgresql.org/docs/current/datatype-json.html, 2024.

[9] Protocol Buffers - Google's data interchange format. https://github.com/protocolbuffers/protobuf, 2024.

[10] Search taking a lot of time using CLP. https://github.com/y-scope/clp/issues/154, 2024.

[11] simdjson. https://github.com/simdjson/simdjson, 2024.

[12] The JSON Data Interchange Standard. https://www.json.org/json-en.html, 2024.

[13] The Number of tweets per day in 2022. https://www.dsayce.com/social-media/tweets-day/, 2024.

[14] Yahoo! Cloud Serving Benchmark. https://ycsb.site/, 2024.

[15] YAML: YAML Ain't Markup Language. https://yaml.org/, 2024.

[16] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682. ACM, 2006.

[17] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[18] Konrad Beiske. Six Ways to Crash Elasticsearch, September 2014. https://www.elastic.co/blog/found-crash-elasticsearch#mapping-explosion.

[19] Craig Chasseur, Yinan Li, and Jignesh M Patel. Enabling json document stores in relational systems. In *WebDB*, volume 13, pages 14–15, 2013.

[20] Dominik Durner, Viktor Leis, and Thomas Neumann. Json tiles: Fast analytics on semi-structured data. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 445–458. ACM, 2021.

[21] Elasticsearch B.V. Elasticsearch, 2024. https://www.elastic.co/guide/en/elasticsearch/reference/8.7/index.html.

[22] Facebook, Inc. Zstandard, 2024. https://facebook.github.io/zstd/.

[23] Free Software Foundation, Inc. GNU Gzip, August 2024. https://www.gnu.org/software/gzip/.

[24] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International conference on data engineering workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[25] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. Json data management: Supporting schema-less development in rdbms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1247–1258. ACM, 2014.

[26] Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Ying Liu, and Hui Joe Chang. Closing the functional and performance gap between sql and nosql. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, SIGMOD '16, page 227–238. ACM, 2016.

[27] Kirk Rodrigues, Yu Luo, and Ding Yuan. CLP: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 21)*, pages 183–198. USENIX, July 2021.

[28] James A Storer and Thomas G Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, 1982.

[29] Daniel Tahara, Thaddeus Diamond, and Daniel J Abadi. Sinew: a SQL system for multi-structured data. In *Proceedings of the 2014 International Conference on Management of Data*, SIGMOD '14, pages 815–826. ACM, 2014.

[30] Zhiyi Wang and Shimin Chen. Exploiting common patterns for tree-structured data. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 883–896. ACM, 2017.

[31] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys' 23)*. ACM, 2024.

[32] Yann Collet. LZ4, 2024. http://lz4.github.io/lz4/.

[33] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.