QuEST: Fast, Expressive, and Cheap Analytics for Distributed Traces Using Cloud Storage

Jessica Berg[1], Muhammad Haseeb[1], Haiming Chen[1], Yaojia Ju[1], Anirudh Sivaraman[1], Ravi Netravali[2] and Srinivas Narayana[3]

[1]*New York University*
[2]*Princeton University*
[3]*Rutgers University*

Abstract

Distributed tracing is the practice of tracking a user's request through a microservice application from entrance to exit. The data emitted by this practice is so large that it is often difficult to store and analyze. We propose a new system to store tracing data on top of low-cost cloud object storage, a service provided by major cloud providers. We describe indexing structures that achieve fast query times and considerable query expressiveness by adapting the design of indices to both cloud storage and the unique distributed tracing domain.

Keywords

Cloud Storage, Distributed Tracing

1. Introduction

Distributed tracing has emerged as an important debugging aid in the context of Web applications that are structured as a collection of loosely coupled microservices. In distributed tracing, a user request arriving at the frontend of a Web application is tagged with a unique request ID. This request ID is then propagated in RPCs issued by the frontend to internal microservices and recursively to RPCs issued by these internal microservices. A trace is a call graph of such RPCs for a given request. Each call within a trace (i.e., an edge in the graph) is called a span. Spans usually contain labels about the call in question, such as the execution time of a call. As Web applications increase in complexity and the user traffic to such applications increases, data management of traces becomes an increasingly important problem.

This paper presents a trace data management system, QuEST which enables cheap, expressive, and efficient data analytics. QuEST incorporates three key ideas. First, we use cloud object stores to cheaply store tracing data and tailor our database design to its limitations, such as high access latency and limited object size (§4). Second, we provide a rich query language to query tracing data. The language contains constructs for graph pattern matching that allow a user to retrieve traces that con-

tain specific subgraphs within them. Third, we develop efficient index structures that are tailored both to our language constructs as well as the challenges of cloud object stores.

To develop indices, QuEST divides a trace into its structure and attributes. The structure is the set of nodes and edges that constitute a graph, whereas attributes are labels on the nodes, such as HTTP response code or response latency. QuEST implements indices on the structure of a trace and on diverse types of attribute data. QuEST creates a *Structural Index* which uses hashes of trace structures to enable quick structure lookups. The attributes are categorized by the following characteristics: the cardinality of an attribute and whether it is searched by a point or range query. QuEST creates tailored indices for all combinations of those characteristics (Figure 3). The *Sequence Bloom Tree Index* is designed for high cardinality point queries. The *Range Index* is designed for high cardinality range queries. The *Folder Index* is designed for both point and range queries on low cardinality data. By customizing indices to the type of data, QuEST achieves fast query response times.

We evaluate QuEST and compare it with Tempo [1] and Jaeger with ElasticSearch [2] on real trace data from Alibaba's production microservices [3]. Tempo is a recent system that also uses cloud object stores to store traces, but lacks QuEST's rich query language. Jaeger with ElasticSearch supports richer queries, but uses costlier local or persistent disks attached to VM instances for trace storage. We find that QuEST's performance in terms of latency is comparable to Jaeger with ElasticSearch and significantly better than Tempo (10.51-31.07 ×, Figure 13), while only incurring a modest additional storage cost compared with Tempo. On the same data, QuEST

costs only \$6.36 per month more than Tempo, and costs \$635.33 less per month than Jaeger with Elasticsearch. Code for QuEST is available at https://github.com/dyn-tracing/trace_storage.

2. Background and Related Work

2.1. Graph Databases

In the wider database space, graph-based systems such as Kuzu have emerged [4]. Kuzu does not work on distributed tracing data in particular, but instead focus on storing many graphs in different contexts. It is a disk-based database that is able to accelerate graph-based queries through novel mechanisms for high dimensionality joins. QuEST's insight is that in the distributed tracing context, by far the most common high dimensionality join is that which allows an entire trace's data to be associated. By creating a graph structure index, QuEST targets optimizations for trace-based queries and eliminating the necessity for generalized multidimensional joins.

2.2. Observability

In the observability space, data management systems encompass metrics, logs, and traces. Metrics and logs are characterized by fixed lengths and an append-only nature. They provide aggregated quantitative measurements and textual records. In contrast, traces exhibit unbounded lengths and are constructed from temporally similar yet overlapping cross references, which can arrive in any order. Thus common techniques such as only appending to data, or compression based on standardization, do not easily transfer to the distributed tracing setting.

2.3. Trace Data Pipeline

There are usually three conceptual pipeline stages for a tracing system: collection, storage, and analysis. Collection is the practice of extracting tracing data from the microservice application. This is usually done through a library that retrieves the relevant information when a microservice executes, batches that data, and sends it to a container called the **collector**. The collector will reformat the data and send it on to storage.

2.4. Trace Storage

Once it gets to storage, the system must decide where and how to store it. There are four main models for storage.

Traditional Databases: Some tracing systems utilize traditional databases. For example, many users of Jaeger will send their data directly from the collector to a database like Cassandra or Elasticsearch [2]. These databases are often run on a cluster of VMs. The other three models deal with cloud storage in some way. The impetus for including cloud storage is primarily that cloud storage is cheap and scales to large data sets. Distributed tracing produces quantities of data that are so large that these cost and scalability improvements are important.

Tiering: In this model, the main analysis happens as in a traditional database, on a cluster of VMs, but cloud storage is used as a secondary, backup store [5]. Some prominent systems such as Edgar, Netflix's distributed tracing system, and TimescaleDB, a database tailored for time series data, take this approach [6, 7].

Cloud Storage with VMs: One can also use cloud storage in parallel with more traditional VMs; this utilizes both stores simultaneously, and speeds up queries in systems such as Honeycomb [8].

Only Cloud Storage: Some systems use only cloud storage. QuESTfits into this category. Tempo is the only other example we are aware of in this category that is made for tracing; all data in Tempo is stored in cloud storage. There are a wide range of challenges related to building on top of cloud storage (further expounded in Section 4). It is difficult to build expressive systems on top of cloud storage, so distributed tracing-specific databases tend to use both cloud and other storages, and in doing so may lose out on some of the cloud storage advantages. Other systems such as DeltaLake [9] have made models for all-purpose, generalized databases on top of cloud storage, but do not specialize for distributed tracing. Tempo currently only supports trace ID lookup, and no other analysis. In addition to using only cloud storage to store tracing data, QuEST also introduces an expressive query language on top of cloud storage.

2.5. Inputs to Trace Analysis

It is not enough only to store data; it also must be able to be queried and analyzed.

Pre-Filtering: Some systems, like Lightstep's microsatellites (containers that filter data close to the source) [10], and NewRelic's infinite tracing (a system for sampling data after it has been collected) [11], focus on selecting for interesting traces to persist to storage; this technique lowers the quantity of data that is ultimately persisted while retaining those traces that are likely indicative of problems. There are also explanation engines that try to use AI and other search techniques to automatically detect interesting anomalies in tracing data; these include Dynatrace's Davis engine, Lightstep, and Honeycomb's bubbleup mechanism [12, 13, 14].

User Query-Based Systems: More traditional systems like Jaeger with Elasticsearch, Tempo, and TimescaleDB primarily utilize user-defined queries as input. QuEST is at its heart a runtime that distills user-

defined queries to code that answers them, and remains flexible enough that it can express general, wide-ranging queries that are similar to results of an automatic analyzer.

3. Challenges of Tracing

Distributed tracing is a field that has recently had many in domain-specific storage and analysis tools. Many systems, such as TimescaleDB, Lightstep, Honeycomb, and Tempo, increasingly are *not* general-purpose databases, and instead are tailored to the tracing domain [7, 15, 16, 1]. Here we detail what challenges necessitate the push for specialization to the domain of tracing.

Quick lookup of traces is important. When creating a trace, many tracing systems generate identifiers for both traces and spans. These identifiers are universal, and often show up not just in tracing data, but in logs and other monitoring sources so that users may cross-reference data. Thus, to analyze a trace as a whole, as well as enable quick lookup for specific incidents, it is essential to quickly look up both spans and traces by their IDs.

Data is inherently graph-based. Unlike logs or metrics, traces are inherently graph-based. A trace is a directed graph of events that are linked only by causality, not by time or type. We can't simply port the metrics and logs work to this context. That work tends to operate under the assumption, in metrics, that one receives periodic updates of the same type, and in logs, that there is little need to cross-reference them. We cannot make those same assumptions.

It is important to be able to query by time. All raw tracing data has time associated with it, and time is often one of the most important attributes in a query. Users want to know when attributes of their system changed, or what the status of a system is as of an hour ago. It is also important to narrow queries by time when looking for a particular incident which was identified outside of the tracing system. Logs and people's real world experiences often have some time-based component to them. Therefore, any tracing system must be able to incorporate and emphasize time in its queries.

The data in the system scales at a different rate than the interesting data. Most systems, most of the time, work well. Thus, the vast majority of tracing data is simply the system working as intended. For debugging and analyzing unintended behavior, however, all this data is very repetitive and mostly useless. Most data is uninteresting. Any tracing system must balance the need to persist large amounts of data (we may want to reanalyze it later, it may be useful for characterizing normal behavior) with the need to pinpoint unusual behavior fast. It is essential that when creating queries about this unusual

or interesting behavior, the queries' time correlates to the quantity of data they return, not the quantity of data in the system as a whole.

4. Challenges of Cloud Storage

We now discuss challenges relating to cloud storage. These challenges are weighted differently than in traditional file systems, and are the motivation for exploring different points in the design space.

We start with some definitions. *Buckets* are the basic containers that hold objects in cloud storage. *Objects* are the basic form of data that is stored in cloud storage. Multiple objects can be grouped in a bucket.

Although both buckets and objects have names, the namespace is flat, and does not behave like a traditional file system. One interacts with cloud storage primarily through reading and writing objects through RPCs to the cloud service.

Limited Object Size: Cloud storage performance peaks at a limited object size. Cloud storage systems such as Google Cloud storage (GCS) have advertised and recommended to users that users target a 1MB size for their objects [17]. There is a latency cost associated with accessing objects, so it is best not to have too small objects. At the same time, past 1MB, GCS's performance diminishes. So we must try to maintain this object size.

Writing costs money: Cloud storage charges per I/O operation, as well as for data storage. Additionally, writes cost more than reads [18]. Thus, we may not be able to afford to write every time we receive new data. It is in our best interests, for this reason, and for the limited object concern above, to batch our data so that each object contains multiple logical pieces of data. We also may want to limit indices; each index costs money to write often proportional to the data size, and distributed tracing deals with large data.

Updates don't exist: Updates, such as overwriting old data, do not exist as a first-order concept in GCS. Rather, when one wants to update an object, one must delete and rewrite the entire object, even if only a byte was changed [19]. Thus, it is expensive to change a data structure after it is written, so the data must be written in a way that optimizes query performance the first time the data is entered into the system; rewrites are bad.

High Retrieval Latency: There is a minimum latency associated with any cloud storage operation on the order of tens of milliseconds [17]. We have to employ data structures and access patterns that are highly parallel to improve the user experience.

5. Key Design Principles

5.1. Traces as graphs

We design our system around traces as graphs to allow queries to utilize powerful and intuitive representations of the dependencies and relationships between various components and services within the application. By focusing on graphs, we address the importance of elevating trace IDs' importance in our system; if traces are first-class citizens, they must be easily identifiable.

5.2. Time as a first-class citizen

With regard to our tracing challenges, we align our system design with time, which is appropriate to the subject area, as discussed by section 3. Not only that, but by using time as a first-class citizen, we address many of the other challenges that cloud storage poses. To elevate time, we focus on creating indices and storage layouts that preserve time-based ordering.

Through using time as an arbitrary divider of data, we can limit the size of any one object in cloud storage; if each object contains a specific time range's data, we limit the quantity of data therein, and address the object size limit discussed in section 3. We will talk more concretely about how this works in the Indices section.

We can significantly reduce the data we're looking at both based on time and based on the value of the attribute. When we allow our indices and data structures to be easily divisible by time, we prevent rewriting a single byte; we don't rewrite if we only write for new data. Because time is monotonically increasing, we never have to go back and rewrite data. In addition, because doing so limits writes in general, we limit the cost of new writes, as discussed by section 3. The answer to a given query can usually be expressed as a union of queries across different time periods. For example, to find the traces with latency less than 100 ms from time 1 to 3, we can also union the results of query 1, which finds traces with latency less than 100 ms from times 1 to 2 and query 2, which finds traces with latency less than 100 ms from 2 to 3. We make query execution highly parallel by querying over disjoint data sets. This allows us to hide the somewhat high latency to access cloud storage.

In order to prevent rewriting a single byte, we index implicitly on time. In order to limit writing because writing costs money, we index implicitly on time. In order to deal with the minimum latency cost, we create time as a parallelization unit so QuEST can read salient data in parallel. These techniques allow QuEST to address all the challenges in Section 4.
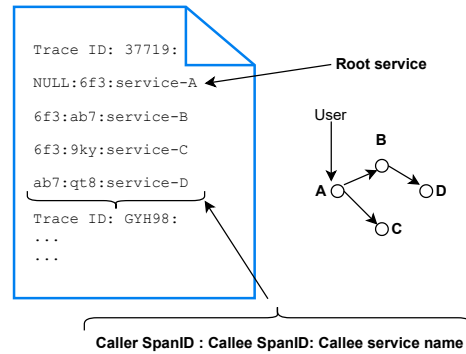

Figure 1: An object in Structural Bucket and the corresponding trace.

6. System Architecture

6.1. The QuEST query language

Data Model: QuEST models distributed tracing data as a set of graphs where nodes which represent calls to a service, may have attributes. Attributes are key value pairs, and may represent things like the time at which a request arrived at a microservice, or an HTTP response code. Each node may have an unbounded number of attributes; in practice, it is up to the developers of the application how many attributes they want to store.

Syntax: The query language is based on a subset of openCypher [20] where one can specify the structure of a subgraph one is interested in, and any attributes on that subgraph.

In particular, a query is made up of a MATCH, an optional WHERE, and RETURN clause. The MATCH clause specifies the structure on which to perform subgraph isomorphism. The WHERE clause specifies attributes that the graph must have. Finally, the RETURN clause determines what attribute should be returned from the query.

Semantics: The query

```
MATCH (a) -> (b)
WHERE a.start-time >= 50 AND
    b.name == productService
RETURN a.name
```

would match on graphs that contained at least two nodes, one called by the other, where the first node's start time was after 50, and the second was called productService. Then, it would return the name of the service that called productService in all matching traces.

6.2. Ingest

In QuEST, the application is configured to send data to OpenTelemetry collectors. As a reminder, a collector is a

container that collects spans and forwards them to storage. Often, applications do not send directly to storage, because if storage is overloaded, one does not want the critical path of the application to be affected. So the collector acts as a forwarding service, and often performs duties like reformatting or preliminary computation on the data.

In QuEST, the collector consists of one front end, and multiple autoscaling back ends. The front end load balances across the back ends according to trace ID, and the back end does trace-level computation, and then forwards data to storage.

The advantages of such a setup relate to scaling, both in terms of traces per second and in terms increasing the number of applications used. First, the back end collectors can scale with the load from the application. This allows for granular billing; you only pay for what you use. Second, in the event that an organization has multiple microservice applications, all of whom want to use distributed tracing, one is able to isolate their services. Setting up separate collectors does not come with significant replication costs because the back end collectors, who do most of the computational work (the front end is simply a load balancer) autoscale according to load. So making separate front end and back end setups for different applications comes with no significant replication costs.

6.3. Data Layout

In QuEST, we incorporate some of the most important indices into the structure of the stored data. That is, rather than store raw data, and index over that data, we organize the data to maximize locality at both a span and trace level. We also design the system to achieve good query performance under the constraints of cloud storage, which are shared by both the ingest path and the query system. We characterized the cloud storage challenges in Section 4. These challenges guide how data should be laid out in the storage so that searching through the database does not amount to high query latencies. Now, we describe the design of our database. Prior to that let's review the following definitions that apply to cloud storage:

Bucket: Buckets are the basic containers that hold data stored in Cloud Storage.

Object: Objects are the basic form of data that is stored in Cloud Storage. Multiple objects can in a bucket.

There are three types of buckets in our architecture:

1. Structural Bucket
2. Microservice Buckets
3. Index Buckets

There are three **Structural buckets** (each plays a different role as explained in Section 7.1), several **Microser-**
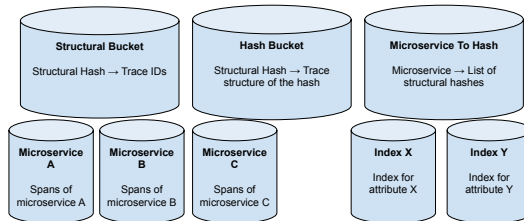

Figure 2: Database Architecture

vices buckets (one for each microservice), and several **Index buckets** (one for each indexed attribute). We first review the Structural and Microservices buckets. Figure 2 shows the basic architecture of our database system. The objects in all these buckets are batched according to the time as exhibited by the bold object names in Figure 2. The structural bucket contains trace IDs and the structures of corresponding traces. The structure list bucket contains a list of all hashes of trace structures and their corresponding structures. The hashes of trace structures play an important role in this index, and will be referred to as **structural hashes**. The structures by microservice bucket provides a mapping from microservice to trace structure. In the microservices buckets, the spans are stored i.e., the bucket Microservice A will contain all the spans corresponding to the microservice named "A". Spans have their own unique IDs, just like traces have Trace IDs, and these IDs are also present in the objects of Microservices buckets.

Figure 1 shows the anatomy of an object in the structural bucket. A single object can have information about several traces (Trace IDs 37719 and GYH98 are visible in the figure), that have been ingested into the system at a similar time. Each trace has several spans that are all linked together by the caller-callee relationship. The example trace in the figure 1 has trace ID 37719 and a root span with span ID 6f3 that originated at service-A. Then the span 6f3 called service-B and created a new span with ID ab7. In parallel, the span 6f3 also called service-C and generated span 9ky. At the end, the span ab7 called service-D forming a new span with span ID qt8. This bucket categorizes traces based on time and structure. However, this is not necessarily enough information to efficiently sort through data based on structure. For this, we also have two auxiliary buckets, one which provides a mapping from microservice to structural hashes of structures that include that microservice, and one which simply lists all structural hashes and their corresponding structures.

The information in the structural object is a serialization of a trace graph. These span IDs present in the object of a structural bucket link to the corresponding spans present in the Microservices buckets. To fetch a

complete trace, one would first read the trace structure from the Structural bucket and then fetch all the spans of that trace from each of the Microservices buckets corresponding to microservices represented in the trace. All the fetch requests for the spans can occur in parallel. The object names for all buckets are the timestamps denoting the duration of the data contained in an object. This naming scheme can introduce numerous objects with a similar prefix in their names. Cloud storage performs badly when creating many objects with a shared prefix because on the backend of the cloud provider, requests are load balanced across replicas according to their prefixes. To avoid any decrease in performance, we prepend the object names with a small random hash and the final object names are written as `<smallHash>-startTime-endTime`. Even if objects are written close together in time, prefixes become relatively diverse due to the hash. We have found this to be a good compromise between the need for disparate prefixes on the ingress side (prefixes are disparate enough that we have good throughput), and fast querying on the query side (the object name still contains metadata to the data within the object).

The third type of bucket, the Index bucket, stores indices. The next section presents the indices in detail. The important point to mention here is that we have two indices that are built by default: (i) Trace Structures Index (a Folders Index) and, (ii) Trace IDs index (a Sequence Bloom Trees Index). We have observed that these are substantially helpful in answering common queries; as explained in the next section, they cover most of the kinds of tracing data attributes users find useful. For further indices, developers can choose what attributes to index and QuEST will build them.

7. Indices

The structural index is a special index specifically used for graph-based queries. With regard to queries based on attributes of the trace, we use Table 3 to make a taxonomy of indices with example queries that they will be useful for. The terminology used in the table is as follows: (i) Low Cardinality: data points whose total number of possible values are very low, for example all HTTP errors (202, 402, 500, etc.), (ii) High Cardinality: data points whose total number of possible values are very high, for example, total trace IDs, (iii) Point Query: querying on some point value like fetching traces with HTTP error 500, (iv) Range Query: querying on some range of values like fetching traces with HTTP error between 200 and 500.

For low cardinality data, both the point queries and range queries can be handled by a Folders Index (Section 7.2). High cardinality point queries are served by Sequence Bloom Trees, (Section 7.3). High cardinality

range queries are served by the Range index (Section 7.4).

7.1. Structural Index

Data Layout: By default, QuEST creates a strucural index. This index stores information which is used for finding traces of some specific graph structure. For example, a user might be interested in finding traces where service X calls service Y, which in turn, calls some other service Z. As discussed separately, user could provide any kind of structure to search for in the traces. In this paper, we describe the index as solving these sorts of graph questions, formally called subgraph isomorphism, and we use a well-known algorithm for determining subgraph isomorphism as a proof-of-concept in our prototype. However the index design is fundamentally agnostic to the kind of graph processing. Theoretically it can be used for a wide range of graph computations, such as matching graphs exactly or determining properties such as branching factors.

Ingest: While sending the traces to the cloud storage, hash of each trace structure is computed using an algorithm pioneered by eBay's distributed tracing system[21]. The algorithm hashes the structure of a trace (that is, the combination of named nodes and edges). When several traces have same hash, it shows that all of them represent the same graph, and they may only differ in the information stored in the individual spans, such as latency or HTTP response code. This index uses the hashes in three ways: first, it creates a mapping from a given hash and time period to all the trace IDs of that hash and time period (contained in the Structural Bucket), second, it creates a list of all structures that currently exist in the system (in the Hash Bucket), and finally, it creates a mapping from a microservice to all the structures it is a part of (in the Microservice To Hash bucket). The buckets of these mappings are shown in Figure 2.

Querying: The reasoning for making three different structures to answer one structural question is simple: listing objects takes time. By first retrieving all the hashes that match a query's microservices, then doing the more expensive isomorphism operation on just the structures, then doing the most expensive job (due to scale) of retrieving all matching trace IDs, we remove large amounts of data from consideration quickly, before doing the most expensive operations computationally. Then, we are left with a list of trace IDs that match the graph in the query.

Analysis: Each of these buckets' objects also address challenges from Section 4. The structural index also allows for parallelism within all three stages: first, by looking up each microservice's hashes in parallel, then, by retrieving structures and computing subgraph isomorphism in parallel, and finally by retrieving the relevant trace IDs in parallel. In this way, although we may do many calls to cloud storage, we are able to hide the relatively expensive latency of retrieving from cloud storage.

	Low Cardinality	High Cardinality
Range Queries	Give me all 5xx errors (Folder Index)	Get traces with latency 100 to 200ms (Range Index)
Point Queries	Give me all 503 errors (Folder Index)	Give me trace with ID 1234 (Sequence Bloom Trees)

Figure 3: Taxonomy of indices used within QuEST.

In addition, each object is relatively limited in size, relating information from a single batch. Each microservice's hashes are represented in limited size objects. Each possible trace structure exists within its own object, and is of a small size. Finally, the mapping from time period to a list of trace IDs is limited because generally only a certain number of traces are created within a given time period. Finally, the structural index never rewrites any data; if the data relevant to an incoming trace already exists within the system, we add nothing. Otherwise, we either add a microservice's information once, a hash to its corresponding exemplar structure once, or trace ID data in batches that are linked to time periods. Thus we never rewrite data.

7.2. Folders Index

Data Layout: Figure 4 depicts an example index of a folders index. In a folder index, there is a bucket for the attribute on which a folders index has been created; this is not a native data structure to cloud storage, but rather something we built on top of it. For example, if we index HTTP Error, then we will see a bucket named "index-http-error". There will be several folders inside this bucket, one for each distinct value of the indexed attribute. In our example of HTTP errors, we will see folders like "Error-500", "Error-402", etc,. In each folder, QuEST inserts objects containing Trace IDs which are batched according to time. The name of objects reflect the time range for which batched data is present in a particular object.

Ingest: The folders index update occurs at scheduled intervals during the execution of QuEST. It reads raw trace data from the structural buckets, and constructs the index as described above.

Querying: When it comes to querying a folders index, we first choose the folder, or several folders if it is a range query. Inside the folder, we have several batched objects that are processed in parallel. We read trace IDs from these objects, and then consult the Structural bucket to get structure and span IDs of our trace IDs.

Analysis: Using the folder index, we avoid reading all the trace and spans data from microservices buckets and *then* filtering the results based on some attribute. The index directly give us trace IDs that have some particular attribute of our interest. In addition, they are well suited to the cloud storage. Objects with common prefixes can be easily obtained using a LIST operation in the cloud
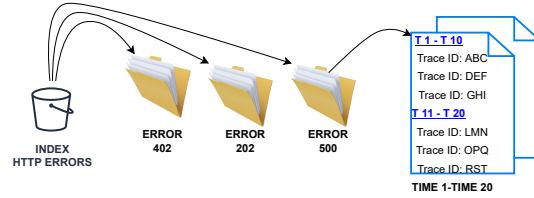

Figure 4: HTTP Errors Index: An example of Folders Index

storage. As the cloud storage namespace is flat, we can simultaneously search by category (folder name) and time (object prefix) in the same LIST operation, precisely the data we need. Upon querying a folders index, we get the Trace IDs of all the traces corresponding to a user query and then the actual traces data (i.e., spans) can be fetched from Microservices Buckets against the found Trace IDs using parallelism that significantly improves the system's performance.

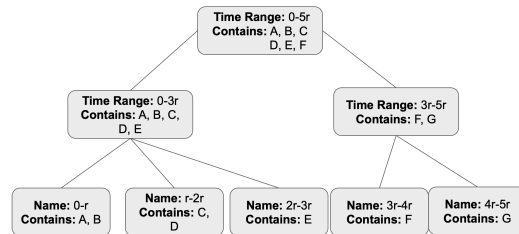## 7.3. Sequence Bloom Tree (SBT) Index

Figure 5: A Sequence Bloom Tree containing trace IDs. Each Bloom filter is the union of its children.

Trace ID and span ID queries are among the most common types of queries in a tracing system. QuEST must have a fast ID lookup. This data consists of uniformly distributed identifiers of 128 bits for trace IDs. At configurable intervals, an index is updated with the newest ID data. Then the data is available to be queried faster than the linear lookup of every ID to exist.
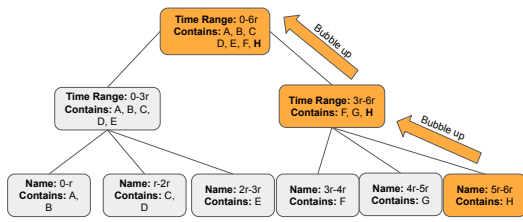
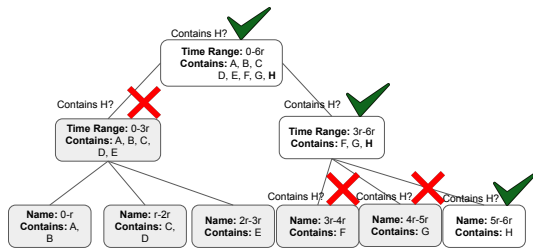Figure 6: Insertion of a new batch from time period 5r to 6r.
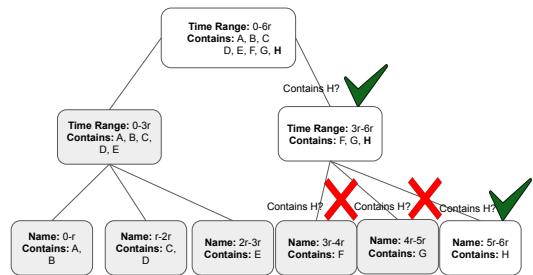

Figure 7: Querying for the trace ID H.

Figure 8: Query for the trace ID H, using a shortcut to avoid the root node.

To address this need, we use a data structure that has gone by multiple names; here, for simplicity, we call them Sequence Bloom Trees (SBTs) after their use in gene sequencing problems [22]. The basic data structure has been used in diverse contexts and excels at problems with the following attributes [23]:

1. There are many small batches of data added in a streaming fashion, and data is not often repeated across or within batches.
2. A batch is static once added.
3. The primary query is: which (if any) of these many batches is the target, small piece of data in?

Data Layout: The SBT has two parameters: a fan-out factor, *f*, and a smallest range size, *r*. From this, the SBT is primarily a *tree of Bloom filters*. Each parent Bloom filter is the union of its children, and is named such that the Bloom filter represents the set of trace or span IDs that were created within a named time frame. The leaves' time frame is precisely that of the smallest range size, *r*, and contain the data from multiple batches, which is a group of traces ingested into the system. The parents' range is *f* times the range of its children. Each parent has at most *f* children. This tree is shown in Figure 5. The basic data structure consists of nodes that are arranged into an *n*-ary tree, where *n* is a constant.

Ingest: When one inserts a batch, one creates a Bloom filter on all the data in the batch. A node consists of this Bloom filter and a pointer to the ground truth data. When the node is inserted, it does not only get inserted in the tree; the data in it must be bubbled up to the root. The Bloom filter in the new leaf node is unioned with its parent's Bloom filter, and its parent is updated to the new value. That parent will then update its parent to the union of the parent and grandparent's Bloom trees. In this way, the root will contain the union of all Bloom filters in the tree as shown in Figure 6.

Querying: Querying the tree happens as follows: if the Bloom filter at the root returns that the value does not exist in that Bloom filter, we return that the value is not found. While Bloom filters can return false positives, they never return false negatives, so this is a valid operation. Otherwise, we look to see which of its children's Bloom filters return that they contain the value. At each level, we check Bloom filters, and if the Bloom filter returns that it contains the value, we check its children. We include false positives in this exploration; note, however, that it becomes more and more unlikely with each level that we will get multiple false positives in a row. In the case of a negative result, we stop exploring that subtree, because that parent node contains all its children, thus the subtree does not contain our value. Eventually, we get to a set of leaf nodes. Note that in the case of the values being trace IDs or span IDs, this would likely be a single leaf node, because trace and span IDs are unique; the only way it would be multiple is in the event of a collision in the Bloom filter itself. If the leaf nodes' Bloom filters report that they contain the queried value, we verify that with the lookup to the ground truth data (remember, false positives are possible). We can then conclude the data belongs to the batch represented by the aforementioned leaf node(s). This operation, in the average case, is $O(\log n)$, where n is the number of nodes in the tree. The number of nodes in the tree is on the order of the number of batches. In the worst case, we'd get false positives for every batch and explore all nodes in the tree. Note that this worst case is extraordinarily unlikely; each Bloom filter often has a false positive rate

of about 1-3 percent. To get that false positive of 1-3 percent repeatedly, for every node, would be 0.01^n. An example of querying an SBT in the regular case is shown in Figure 7.

If we know from the start that we are querying within only a specified time range, we can shorten our search. We can query the root to determine the data structure's parameters and its the overall time range, calculate a starting node, and start our search midway down the tree. This significantly reduces the expected search complexity from $O(logn)$. For example, for the tree in Figure 7, if we know that we only want to search within the 4r-6r time range, we find the node with the smallest range that contains that entire range, and search below, as shown in Figure 8.

Analysis: This data structure is appropriate for high cardinality point queries for a number of reasons. First, insertion happens in batches, and data enters our system in batches. Unlike disk-based solutions like LSM-trees, once data is inserted, it never needs to be edited again. This is convenient because we need to batch our data, and each edit to an object in cloud storage is costly. Second, the data structure is very amenable to highly parallel computation; the n−ary tree has a configurable number of children, and by setting n to a relatively large number, one gets a fairly shallow tree, which makes the update time, bubbling up, less expensive, and allows for fast computation due to parallelism. Third, high cardinality data by definition has a lot of possible values. For the data structure in general, the less common a value is in the data set, the more efficient its lookup is. For trace and span IDs, this is very efficient because one has to follow only one path from root to leaf (excluding false positives), due to the fact that these IDs are unique. The shallow tree necessitated by high parallelism keeps this path relatively short.

7.4. Range Index

The range index is used to compute range queries over high and low cardinality data. The key challenge in the range index is that there must be some ordering on the data to efficiently do a range query. However, placing an ordering on the data means that some data might be rewritten; one must adapt if data should be ordered between already existing data points. The range index strikes a compromise between these two. There are two parameters to the range index: the number of time ranges per summary, s, and the time range, t.

Data Layout: We completely order the data within a specified time range, determined by the parameters. We make this time range configurable, so a user may anticipate their most frequent queries. Then, within a time range, we split the data into fixed sized blobs, and record the first and last items in a blob.

Then, we can store highly compressed representations of the data within a single time period through measuring just their first and last items. So when querying a given range, we can narrow down our search considerably by simply ignoring blobs whose specified range is not of interest to us. This is the general intuition behind the index. We give a more formal definition below.

Ingest: To insert into the range index, one first finds the last time a index was updated. If the index does not yet exist, this is the time of the earliest data in the system.

Then, for each subsequent time period t, we retrieve and sort the data that is indexed. This happens automatically in the background. For example, if the index is on latency, latency for each trace within the time period is retrieved and sorted, and annotated with pointers to the original trace data. Then, the data is split into 1MB batches, with 1MB being the object size at which cloud storage performs best. Finally, that data is sent to storage, and summary objects, containing only pointers to the data, plus their first and last values, is placed in a summary object every s time ranges.

Querying: To search the range index, one first retrieves all summary objects within the specified time range. All queries contain time ranges by default. Then, within each summary object, one looks through each time range, and sees if any of the 1MB objects contain the range of interest. If they do, those 1MB objects are retrieved, and for the data strictly within a range, the pointers to the rest of the trace are retrieved and used.
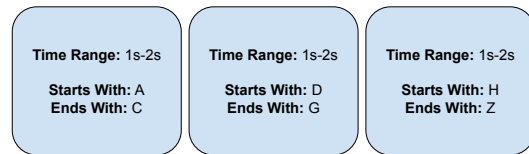
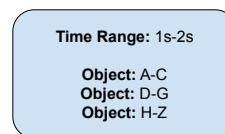

Figure 9: A range index's time range objects.

Figure 10: The corresponding range index's summary object.

Analysis: The range index's primary speedup comes from increasing the speed with which one can search through large quantities of data. The choice of the two parameters, the time range, and the number of time ranges per summary object, make the most difference in evaluating how efficient it will be for a given query.

It also adapts to the distribution of the data. If some change is made to the underlying data, such as a certain

error code becoming much more popular, or latencies increasing greatly, the size of the objects within the index stay the same, thus they are still appropriately sized for optimal retrieval from cloud storage.

8. Query Execution

We now describe how our runtime is designed to process the user-provided query. The runtime operates by taking the user's query as input, generating an execution plan, and then executing that plan to obtain the desired output. The generation of the execution plan and its execution are interleaved at several stages because the results of part of the execution plan often help in optimizing the rest. It allows QuEST to retrieve the query's answer efficiently.

Next, we present the runtime design in detail. It would be helpful to take an example query and refer to it throughout the design for better understanding. Let's use the following example query; it is intended to fetch the total number of traces where product-service gets called by some service (say the caller) and the start time of the span of the caller service is X and the start-time of the span of the product-service is at least Y:

```
MATCH (a) -> (product-service)
WHERE a.start-time == X AND
product-service.start-time >= Y
RETURN count-of-traces
```

There are three main stages in the runtime: (i) Indices Consultation, (ii) Brute Force search, and, (iii) Processing of the RETURN clause of the query. Within each stage, the runtime utilizes parallelism to its full extent as there are significant opportunities to do so. Below is the overview of each stage and then we present explanation of each item in further subsections.

- *Indices Consultation Stage*
  - *Structural Filter Sub-stage*: Processes MATCH clause of query by consulting Trace Hashes Index
  - *Attribute Filter Sub-stage*: Processes the parts of WHERE clause that contain indexed attributes
- *Brute Force Search Stage*
  - Processes the parts of WHERE clause that contain non-indexed attributes
- *Processing of RETURN Clause*
  - Once the previous stages have completed, it uses their results to form the result requested in the query

8.1. Indices Consultation

The indices consultation stage has two main goals: (i) get all the trace IDs corresponding to the traces that match the query structure provided in the MATCH clause and, (ii) if any indices can help in processing some parts of WHERE clause, get the trace IDs from them. Each goal is achieved in a separate sub-stage described below. The two sub-stages, structural filter and attribute filter, can be processed in parallel; they do not depend on one another.

8.1.1. Structural Filter Sub-stage

This stage outputs the trace IDs corresponding to the traces whose structure matches the query structure. The runtime consults the Trace Hashes Index described in 7.2 to retrieve the relevant trace IDs.

First, QuEST determines which microservices are specified by name in the query; here, it is just product-service. Then, it refers to the Microservice to Hash bucket, and retrieves all trace hashes that include the named microservice(s). Then, QuEST retrieves all those structural exemplars from the Hash bucket. The structural exemplars contain the nodes and edges of the graph. QuEST performs subgraph isomorphism to determine which exemplars conform to the query. The expensive subgraph isomorphism is only performed once we have already eliminated some hashes from consideration, and is performed once per hash. Because trace structures are very likely to be repeated, we limit the amount of data on which we do the expensive isomorphism operation by a large margin. In addition, the retrieval and computation of structural hashes is highly parallel, and thus masks the latency inherent to a call to cloud storage.

Once the list of trace hashes has been narrowed down by isomorphism, QuEST consults the Structural Bucket. The Structural Bucket contains folders of hashes, and within each folder are batches of lists of trace IDs by time. QuEST looks within each folder in its hash list and retrieves trace IDs that match the query's time (if one is given in the query).

The query structure could contain placeholder services, i.e., the service represented by "a" in our example query. In this case, we need to map the placeholder to a real service name so that the further stages know which service is referred to by the user query. This knowledge is necessary if the WHERE or the RETURN clause refers to such a service as in our example query. Therefore, this stage also outputs mappings from the query structure to an exemplar trace from each folder. This sub-stage's output consists of all the trace IDs that satisfy the structural query and the isomorphism maps (mapping the target structure to the structure of traces).

8.1.2. Attribute Filter Sub-stage

This stage helps in processing the WHERE clause. The WHERE clause instructs the runtime to filter query results based on the values of attributes. In our example query, the WHERE clause restricts the query results to only those traces where the span of the service calling the product-service has start-time X and the span of the product service has start-time at least Y. If we have indexed the start-time then both of the sub-clauses of our example WHERE clause could be resolved in this stage. Otherwise only a part of WHERE clause is resolved depending on what has been indexed. Note that the where clause can be divided into multiple sub-clauses each concerning one attribute. And then these sub-clauses can be processed in parallel; we can retrieve which traces satisfy each part of the WHERE clause independently, and later filter those which satisfy all of them.

Depending upon the type of attribute, we find out the type of corresponding index (folders index, sequence bloom tree index, etc.,). Figure 3 shows which attributes have which indices.

Once we know the bucket corresponding to the index of an attribute, it is straightforward to fetch trace IDs from that bucket. For folders index, the bucket contains multiple objects batched by time and each object lists trace IDs. We read the objects and get all the trace IDs. For Sequence Bloom Tree Index, we query starting at the root, and find the batches that have the relevant value as described in Section 7.3. Then from those batches, we can look up the relevant trace or span IDs in the relevant structural bucket.

At the end of the Structural filter stage and Attribute filter stage, we take the intersection of the results. The resulting trace IDs are passed on to the next stage, brute force search. To aid further stages, we retain two key pieces of information along with the resultant trace IDs: (i) the object names, (ii) the isomorphism maps. In further subsections, we will detail how this data is used.

8.2. Brute Force Search

This stage processes the parts of WHERE clause that have not been processed by the Attribute Filter Sub-stage of Indices Consultation Stage. It deals with the attributes of the WHERE clause that are not indexed and must be processed by a brute force search. In our example query, if the start-time attribute is not indexed then this stage will search all traces present in the system and determine which have a start-time $>= Y$. Note that this stage only needs to search the traces that are results of the previous stage, the Indices Consultation Stage.

The results of the previous stage have trace IDs along with their corresponding object names. The runtime uses these object names for reading the corresponding

objects from the Microservices Buckets. These objects are used for finding spans of traces which has trace IDs present in the result of previous stage. The attribute values of the spans are checked for the conditions of WHERE clause and this way, the trace IDs get filtered; if the span passes the check, the corresponding trace ID is retained otherwise excluded from the result. Note that the runtime needs to know which Microservices Bucket to read for finding the span of a trace and filtering it based on the attribute present in the WHERE clause. The bucket name is determined based on what microservice is being referred in the WHERE clause. For our example query, the runtime would read the Microservice Bucket for the product-service. It also needs to know which service is referred by the placeholder service "a". The isomorphism maps that are included in the results of the previous stage, help in solving this problem. The isomorphism map tells what is the service name for a place holder service for a given trace ID. This way we know what Microservice Bucket to read for finding objects corresponding to a placeholder service.

Parallelism is easily incorporated in this stage. The results of previous stage has object names and corresponding trace IDs. In this stage, the runtime processes each object name in parallel so that there is only one GCS fetch per thread and there are no redundant fetch calls; we retrieve each object precisely once. Each thread receives an object name and corresponding trace IDs to filter. It then fetches the corresponding object and processes each trace ID.

8.3. Processing the RETURN Clause

This is the last stage of the runtime wherein QuEST processes the RETURN clause of the query. It receives the results of the previous stage as input (the trace IDs which conform to the Structural and WHERE clauses). The RETURN clause can be as simple as asking the count of traces or it can ask for some attribute values of the traces that satisfy the previous clauses. In our example query, the return clause is the count of results. However consider an alternative RETURN clause for our example query: a.end-time. This clause is asking for returning the end-times of the spans of services that calls the product-service. The runtime will read the corresponding spans from Microservices Buckets for the trace IDs this stage was given as input and extracts end-times for the final output. If the attributes referred to in the RETURN clause were also present in the WHERE clause then the runtime will not need to fetch the data from the Cloud Storage because they would already be present in the VM running the query. This is because the WHERE clause retrieves and stores data related to the attributes it filters.

Query	Meaning	Use Case	Execution Time (s)
MATCH (x)->(a), (x)->(b), (x)->(c), (x)->(d) WHERE x.service.name == "adservice" RETURN a.span.id	Match on traces that have experienced a fan out of at least 4 from adservice and return their span IDS.	Determine what traces have wide fan outs from adservice to multiple replicas.	1.4
MATCH $(a) -> (b)$ WHERE a.service.name == "frontend" RETURN trace.id	Return traces where front-end calls at least one other service.	To list all traces, with at-least two spans.	1.3

MATCH $(a) -> (b) -> (c)$
WHERE a.service.name == "frontend" AND
    b.service.name == "recommendation" AND
    b.duration > 50ms
RETURN trace.id
	Return trace IDs where recommendation service is called by the frontend and the duration of the recommendation span is greater than 50ms.	Investigate abnormal latency.	1.2

MATCH $(a) -> (b) -> (c) -> (d)$ RETURN trace.id	Get trace IDs of all traces that are of height at least 4.	Investigate unusually large traces.	1.3

Figure 11: Example queries in QuEST and execution times based on 10 minutes' worth of data at a rate of 1000 traces per second.

9. Evaluation

We evaluate our own system in isolation, and we also evaluate two other systems: (i) Tempo and, (ii) Jaeger with Elasticsearch, which we will refer to for brevity as just Jaeger. Both are open source systems for managing distributed trace data. Tempo and QuEST use cloud storage for storing the traces. Jaeger uses a database, Elasticsearch, for storing the traces, usually on a cluster of VMs. It is expected that any query latency for Jaeger will be less than any cloud storage-based system because all data is kept locally. However, cloud storage-based systems will save in terms of both dollar and operational cost. We provide cost analysis of different systems to better understand the latency vs cost trade-off.

9.1. Questions

We evaluate our system through a series of questions:

1. How expressive is QuEST? *§9.2*
2. How long does it take to answer a query in QuEST? *§9.3*
3. How does query latency change as data accumulates? *§9.4*
4. How does query performance depend on whether indices are available or not? *§9.4.2*
5. How much does it cost QuEST to ingest and store data? *§9.5.1*
6. How much does it cost for QuEST to answer queries? *§9.5.2*
7. How many traces per second can QuEST support? *§9.6*
8. How does QuEST compare to other distributed tracing systems (Tempo and ElasticSearch)? *§9.2, 9.3, 9.5*

9.2. Expressiveness

In Figure 11, we show examples of queries QuEST supports. These queries can be used to debug errors, gain insight from normal behavior, or find anomalous behavior. We focus on graph-based queries as a first-order concern in constructing queries. Another distributed tracing system, Grafana Tempo, is also working towards a query language for traces that enables graph-based queries. As of writing of this manuscript, their query language, TraceQL is in its infancy and does not support general graph queries [1].

9.3. Latency

Cloud-storage systems incur more latency to fetch data than local storage based systems because they transfer data over the network. We therefore expect that our system will have higher latency than disk systems (i.e., Jaeger), but due to our indices, will have lower latency than other cloud storage-based systems (i.e., Tempo).

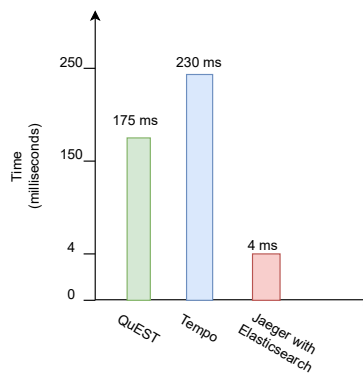### 9.3.1. Plain Query Latency

Figure 12: Median latency of a plain (i.e., Fetch By Trace ID) query

A plain query is a fetch of a trace given a trace ID. Support for the plain queries is ubiquitous in the existing

distributed tracing systems; some systems are even limited to only this type of query [1]. Figure 12 shows the time it takes to execute a plain query on all three systems. QuEST performs better than Tempo (another cloud storage based system) as QuEST has lower latency. QuEST performs worse than Elasticsearch as Elasticsearch is able to retrieve data locally and does not need to fetch anything over the network. In 9.5, we will see that Elasticsearch shows this superiority at the cost of dollars.

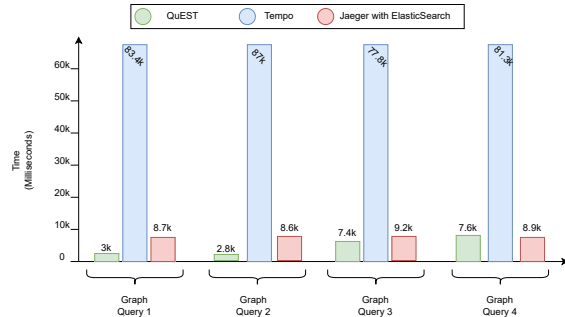### 9.3.2. Graph Query Latency

Figure 13: Median latency of various graph queries on QuEST, Tempo and Jaeger

We also evaluate all three systems using several graph queries and show that QuEST is faster than the others due to its indices for graph structures. Other systems answer the graph query via a scan of all the stored traces. Note that Tempo and Jaeger do not support graph queries so we write a wrapper around their APIs that enables them to answer graph queries. Figure 13 shows the time it takes to run the following four graph queries:

```
MATCH (frontend)->(adservice)->(a)
RETURN trace-id
```

```
MATCH (frontend)->(adservice)->(currencyservice)
RETURN trace-id
```

```
MATCH (frontend)->(adservice)->(checkoutservice)
WHERE frontend.duration > 100ms
RETURN trace-id
```

```
MATCH (frontend)->(adservice)->
    (checkoutservice)
WHERE checkoutservice.duration > 50ms
RETURN trace-id
```

Figure 13 shows that QuEST outperforms the existing systems by a significant margin in terms of query latency.

The above queries exhibit a variety of trace structures and span attributes. We constructed the evaluation queries to include a mixture of indexed (trace structures, trace IDs) and unindexed (span durations) attributes. It is pertinent to mention that QuEST's latencies would further improve if we index all the queries' attributes.

9.4. Graph Query Latency On Larger Data Sets

We also evaluated our system on a larger data set to see how it scaled. The results are as follows:

9.4.1. Realistic and Increasing Data Set Size

Figure 14 shows the results of running four different queries on the published AliBaba tracing data set. The queries are as follows:

duration: Get all the traces where a parent service calls at least two other children services and the response latency of the parent is atleast 300 milliseconds.

fanout: Get all the traces where a parent service calls at-least four other services.

onecall: Get all the traces where a parent service calls at one other service.

height: Get all the traces where a chain of service calls has a length of at least four.

We convert the AliBaba set into OpenTelemetry format and load it into QuEST. We measure query latency as the data is continuously being ingested by the system. We show the number of traces (and the number of bytes) on the horizontal axis and query latency on the vertical axis. We see that query latency is a linear function of the amount of data stored in the system. The onecall query is the most general query which resolves into the most number of traces. This is depicted in the figure as its latency grows faster than other queries.

9.4.2. Impact of Indices and Data Volume on Query Latency

Time (Minutes)	Latency with index (ms)	Latency without index (ms)
10	775	2847
20	851.5	2990.5
30	946.5	3070.5
40	1046.5	3207.5
50	1212	3307.5
90	1681	3745.5

Table 1
Median latency with a SBT index as data accumulates

Table 1 shows as more data accumulates, the query latency increases. The first column represents the time for
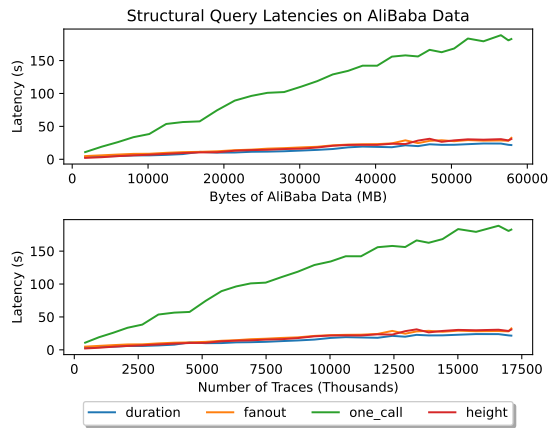
Figure 14: Median latency of various graph queries on AliBaba Data Set

which the application sends traces to QuEST at the rate of 12k spans per second. The query we use to perform this experiment is:

```
MATCH (frontend)->(adservice)->(*)
WHERE adservice.span-id = XYZ
RETURN trace-id
```

This query has span-id attribute in the WHERE clause. We perform the evaluation with and without indexing on the Span ID attribute and present the numbers in Table 1. The table shows that although the query latency increases as more data is accumulated, having an index helps in keeping the overall latency low.

9.5. Cost Analysis

9.5.1. Operational And Storage Costs

	QuEST	Tempo	Jaeger + ES
Operational Cost	$8.70	$8.70	$6.40
Storage Cost	$3.31	$0.007	$32.30
Data Retention Cost	$ 123.26	$120.20	$732
Total Cost	$135.27	$128.9	$770.70

Table 2
Cost breakdown of tracing systems

The cost to ingest traces at a collector and export them to storage is defined as **operational cost**. The cost to store and retain data for a configurable retention window is defined as **storage cost**. We present the operational and storage cost of a day with the retention window
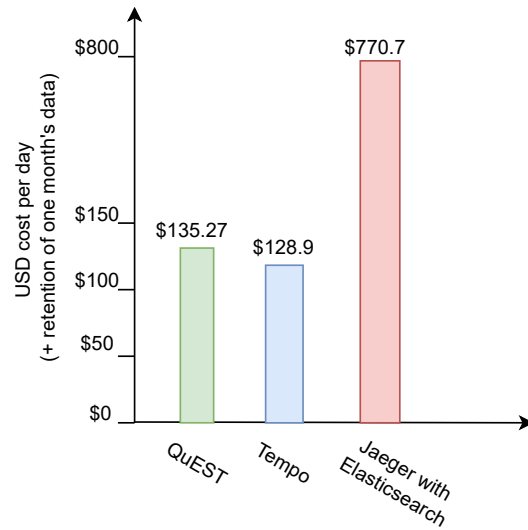

Figure 15: Cost incurred in a day by tracing systems

set to 30 days, i.e., the system ran for 31 days and we present the cost of the 31st day including the cost of retaining the previous 30 days' data. We present this analysis for QuEST, Tempo, and Jaeger. To obtain these numbers, we run each system for 30 minutes to get the operational and storage costs and then extrapolate the cost of a day along with the retention cost for 30 days' worth of data. In our setup, the application generates 12k spans per second. All systems collect these spans, do pre-processing if required, and export the data to some storage. Figure 15 summarizes the cost incurred by the three systems in a day. Table 2 presents the cost breakdown.

Operational Cost: Here we analyze the operational cost for QuEST, Tempo and Jaeger. The cost includes running an OpenTelemetry collector that receives spans data from application and exports it to a storage backend. The storage backend for QuEST and Tempo is Google Cloud Storage (GCS), whereas for Jaeger uses a virtual machine that runs the Elasticsearch database. QuEST and Tempo perform significnat pre-processing at this step, and less processing on data within store, whereas ElasticSearch does little preprocessing in the collector, but significant processing at storage. Thus, QuEST and Tempo require a VM that has enough memory and compute to pre-process the data. We found the e2-highmem-8 VM, which has 8 vCPUs and 64GB of memory, suffices for QuEST's and Tempo's pre-processing requirements. The VM e2-standard-8, which has 8 vCPUs and 32 GB of memory, suffices for running the ingestion and exporting for Jaeger. The VM e2-standard-8 costs $6.4 per day

while e2-highmem-8 costs $8.7 per day.

Storage Cost: QuEST and Tempo both use GCS while Jaeger uses VMs' disks. We only run one VM with a persistent disk mounted for the Elasticsearch database. In production operations, Elasticsearch is often run on a cluster of VMs, but we only assume a single VM to provide a *lower bound* on cost.

For QuEST and Tempo, we observe the I/O operations in GCS to calculate the cost. QuEST performs 661,296 operations (47,136 reads and 614,160 writes) per day and Tempo performs 1440 operations (all writes) per day. The Tempo's write operations include writing large objects that GCS may consider multiple writes, but we consider one write. We err on the side of generosity toward Tempo by not counting write operations' amplification from writing large objects. GCS charges $0.05 per 10k operations. Exporting one day of data to GCS costs $3.31 for QuEST and $0.007 for Tempo. We calculate an upper bound on the cost for QuEST and a lower bound on the cost for Tempo. For Jaeger, we record no extra payment because persistent disks cost in terms of storage volume, not I/O.

An Elasticsearch database needs a VM that can process the data, calculate indices and then store data in the persistent disk. The overall database cost (i.e., storage cost) comprises of the cost of the VM and persistent disk utilized. We use e2-highmem-8 VM that costs $8.7 for a day. Elasticsearch stores 12.3 GB of data for a 30 minute run of the system, which includes the original tracing data and all the indices that Elasticsearch creates by default. Per day, the data amounts to 590.4 GB. The cheapest persistent disk offered by GCP costs $0.04 per GB. To store one day's data, Elasticsearch database costs $23.6 for the persistent disk and $8.7 for the VM that runs the database, which sums to $32.30.

To calculate the data retention cost for a month, we find the data stored by each system: QuEST stores 6,163.18 GB, Tempo stores 6011.52 GB and, Jaeger stores 18,302.4 GB. The retention cost provided by GCS is $0.02 per GB and the cost of persistent disk is $0.04 per GB. Therefore the overall retention cost comes out to be $123.26 for QuEST, $120.2 for Tempo and $732 for Jaeger.

9.5.2. Cost To Query With QuEST

We ran 10,000 queries on 30 minutes' worth of data. A lookup by trace ID costs $0.92 in cloud storage charges, whereas a structural request costs $20.33.

9.6. Throughput

QuEST's throughput is defined by the throughput of the collector. We originally had a throughput bottleneck with one collector replica at about 5K traces per second, which was replicated by Hindsight [24]. However, to scale is not as straightforward as horizontally autoscaling, as spans from the same trace must be sent to the same collector, since we do trace-level analysis at the collector.

We thus enabled a load balancer OpenTelemetry collector that load balanced based on the trace ID of a span. However, that only increased throughput slightly to about 8K traces per second. We determined that the load balancer sent spans one at a time to each backend; the throughput was improved by batching spans.

The collector now adapts to changes in load by having a frontend loadbalancer which forwards spans to backend collectors, which send data to GCS. If there are not sufficient backend collectors for the given load, they will refuse to accept spans, and build up a queue at the load balancer. This ensures no data is lost during temporary spikes in workload. Then, horizontal autoscaling is enabled through GCP, and kicks in when the backend collectors are overwhelmed. Using this horizontal autoscaling, we scale our system to up to about 34K traces per second, which uses 25 backend collector replicas.

While our fix to the collector wasn't trivial, it also was not a huge project. We describe it to illustrate the bleeding edge, unstable character of the OpenTelemetry project. We believe there may be other similar low-hanging fruit within the OpenTelemetry collector, and that throughput may be improved even further.

10. Conclusion

We presented QuEST, a system that enables fast and cheap distributed tracing storage and analysis. We have found that QuEST improves upon traditional database systems in terms of cost, and provides more expressibility in query language than current cloud storage-based systems. In particular, QuEST excels at graph-based queries through building data structures that emphasize both the graph-based nature of distributed tracing and time.

11. Acknowledgements

References

[1] G. Labs, Grafana tempo, 2023. URL: https://grafana.com/oss/tempo/.

[2] Jaeger, Frequently asked questions, 2021. URL: https://www.jaegertracing.io/docs/1.18/faq/#what-is-the-recommended-storage-backend.

[3] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, C. Xu, Characterizing microservice dependency and performance: Alibaba trace analysis, in: Proceedings of the ACM Symposium on Cloud Computing, Association for Computing Machinery, New York, NY, USA, 2021, pp. 412–426.

[4] X. Feng, G. Jin, Z. Chen, C. Liu, S. Salihoğlu, Kuzu graph database management system, 2023.

[5] S. Yang, K. Srinivasan, K. Udayashankar, S. Krishnan, J. Feng, Y. Zhang, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Tombolo: Performance enhancements for cloud storage gateways, in: 2016 32nd Symposium on Mass Storage Systems and Technologies (MSST), IEEE, Santa Clara, CA, USA, 2016, pp. 1–14. doi:10.1109/MSST.2016.7897076.

[6] R. Guiu, What are traces, and how sql (yes, sql) and opentelemetry can help us get more value out of traces to build better software, 2022. URL: https://www.timescale.com/blog/what-are-traces-and-how-sql-yes-sql-and-opentelemetry-can-help-us-get-more-value-out-of-traces-to-build-better-software/.

[7] M. Pandey, Building netflix's distributed tracing infrastructure, 2020. URL: https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304.

[8] I. Wilkes, From "secondary storage" to just "storage": A tale of lambdas, lz4, and garbage collection, 2021. URL: https://www.honeycomb.io/blog/secondary-storage-to-just-storage.

[9] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. undefinedwitakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, M. Zaharia, Delta lake: High-performance acid table storage over cloud object stores, Proc. VLDB Endow. 13 (2020) 3411–3424. doi:10.14778/3415478.3415560.

[10] R. Whitmore, Learn about microsatellites, 2021. URL: https://docs.lightstep.com/docs/learn-about-micro-satellites.

[11] N. Relic, Introduction to infinite tracing, 2023. URL: https://docs.newrelic.com/docs/distributed-tracing/infinite-tracing/introduction-infinite-tracing/.

[12] dynatrace, Meet davis, our powerful ai-engine, 2023. URL: https://www.dynatrace.com/platform/artificial-intelligence/.

[13] Lightstep, Lightstep features and product overview, 2023. URL: https://lightstep.com/change-intelligence.

[14] Honeycomb, Quickly identify and investigate anomalies with bubbleup, 2022. URL: https://www.honeycomb.io/bubbleup.

[15] R. Whitmore, Get started with lightstep observability, 2022. URL: https://docs.lightstep.com/docs/welcome-to-lightstep.

[16] A. Vondrak, Why observability requires a distributed column store, 2022. URL: https://www.honeycomb.io/blog/why-observability-requires-distributed-column-store.

[17] C. McAnlis, Optimizing your cloud storage performance: Google cloud performance atlas | google cloud blog, 2018. URL: https://cloud.google.com/blog/products/gcp/optimizing-your-cloud-storage-performance-google-cloud-performance-atlas.

[18] G. C. Platform, Pricing cloud storage, 2023. URL: https://cloud.google.com/storage/pricing.

[19] G. C. Platform, About cloud storage objects | google cloud, 2023. URL: https://cloud.google.com/storage/docs/objects#immutability.

[20] Neo4j, Inc, Cypher Query Language Reference, Version 9, 2019.

[21] X. Guo, X. Peng, H. Wang, W. Li, H. Jiang, D. Ding, T. Xie, L. Su, Graph-based trace analysis for microservice architecture understanding and problem diagnosis, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, 2020, p. 1387–1397.

[22] B. Solomon, C. Kingsford, Fast search of thousands of short-read sequencing experiments, Nature Biotechnology 34 (2016) 300–302. doi:10.1038/nbt.3442.

[23] D. Lillis, F. Breitinger, M. Scanlon, Hierarchical bloom filter trees for approximate matching, The Journal of Digital Forensics, Security and Law 13 (2018) 81–96. doi:10.15394/jdfsl.2018.1489.

[24] L. Zhang, V. Anand, Z. Xie, Y. Vigfusson, J. Mace, The benefit of hindsight: Tracing edge-cases in distributed systems, 2022. URL: https://arxiv.org/abs/2202.05769. doi:10.48550/ARXIV.2202.05769.