Difficulty Level:

Hard

Prerequisites:

System design fundamentals part 2 by Omkar

Considerations:

1. Each service is a meeting logs with log level (WARN, INFO, ERROR, etc.)
2. Sensitive information like password, auth tokens, bank details are not being published in the logs
3. There is a company-level JSON format for structured logs, including fields like:
    a. originating host
    b. datetime
    c. message body
    d. log level
    e. log type
    f. Environment
    g. trace_id -- Each log line will have a "trace_id" which allows us to follow a request's life cycle when that request travels through multiple servers in a distributed system. This unique id could be a UUID which guarantees uniqueness across machines.
    h. Etc.

Step 1:

Functional Requirements:

1. 1000s of servers generating log lines
2. Ability to search across all logs and live tailing — near real-time
3. The query format can be:
    1. Free text — Searches are performed as keyword queries against the fields of the structured records. These can be exact match queries, sub-string match queries, range queries, or full-text queries.
    2. User specified server-id

    3. Time range
4. The search results should be displayed in reverse chronological order
5. Log data typically has a stable yet steadily growing scale. The indexers should be able to handle occasional bursts; for example, due to outages or delays of the upstream log producers.
6. More recent logs are more valuable and queried more frequently than older logs. It is acceptable for the search system to provide better search performance for recent logs, and worse performance for older logs.
7. Logs can be deleted after a configurable retention threshold - 1 Year

Non-functional Requirements & Capacity Estimates:

1. Considering 1 billion requests per day. Assume 100k seconds per day(Actual 86400 seconds) —> 10^9/10^5 = 10,000 qps

   1 Commodity server = 32 cores * 8 threads per core ~= 250 requests per second
   Our servers should run at 40% CPU capacity to deal with unexpected spikes.
   10,000/250/40% = 100 servers needed to serve search requests

2. Assuming retention period 1 year, we would need to index for 400B log lines. Assuming each log line has 10 terms and each term has 6 characters that means 6 bytes.
   Total index size = 400B*10*6 bytes ~ 24TB per year
   Let's see how to calculate it rapidly:
   400 * 10^9 * 10 * 6 = 24*10^12 = 24TB
   1 Commodity server ~= 10 TB
   Servers needed at 40% capacity = 24/10/40% = 6 servers needed to store the logs

   10^3 = 1 Thousand ~= 1KB
   10^6 = 1 Million ~= 1MB
   10^9 = 1 Billion ~= 1GB
   10^12 = 1 Trillion ~= 1TB

Step 2: Find the major components(Micro Services)

From the requirements, it is quite clear that there are two major components in the system:

1. Logger service – responsible for collecting and aggregating the logs from services

2. Search service – responsible for taking user queries, parsing queries and returning the matching results

Step 3: High Level Design

Requirements:

1. **Emit Logs:**

   Application server sends the log messages to logger service where it could be stored. Let's see the possible ways of sending the log messages:

   a. **Sending log messages by HTTP calls**

   In this approach, the client library could be provided. Client service calls the library function to log the messages. Now it is the responsibility of client library to maintain a pool of HTTP connections to logger service (because opening a new HTTP connection for each login call would be very expensive. As HTTP connections use underlying TCP protocol which has high time complexity due to 3 way handshake).

   b. **Log Agents**

   The JSON-formatted logs are emitted to file or stdout depending on the environment. Per-service log agents pick up all logs and push them to a global messaging queue (like Kafka). Each service name could be a queue topic name.

We use Queue because:

1. Traffic generated by application services is high and could have unpredictable bursts also. Queue provides us a buffering mechanism that can rate limit the bursts.
2. This decouples application service from the logging mechanism. We can replace logging system by a new swanky system without impacting applications
3. By common sense, application service does not need a response whether log line was submitted or not. So asynchronous system makes sense here

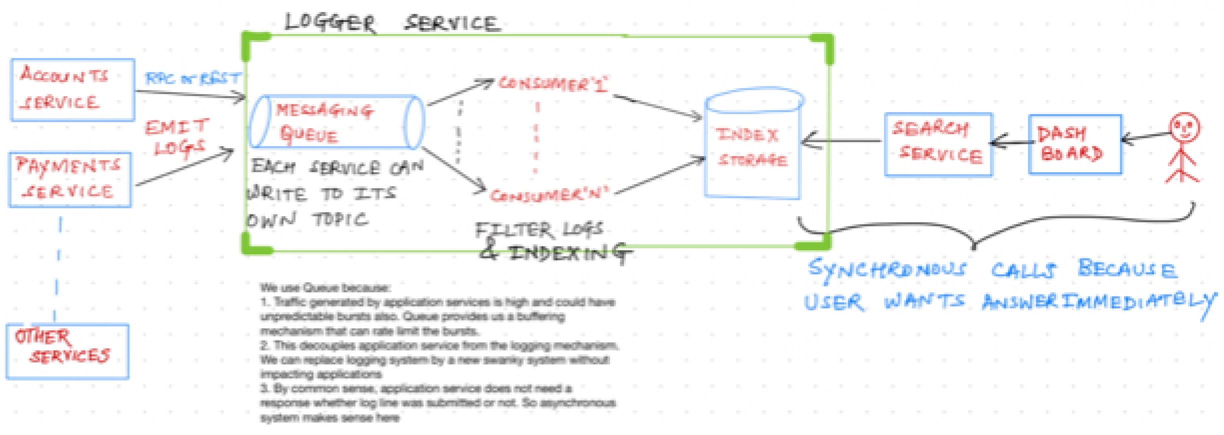
2. **Filter Logs: Remove the logs which:**

   a. Are received from non-whitelisted services/environments

   b. Don't conform to log schema

3. **Indexing**:

   Aggregate Logs in an efficient data structure so they are searchable. Search requests could have any free text and/or the field names(like datetime, service, etc.). We index full text and fields in the log message.



We use Queue because:
1. Traffic generated by application services is high and could have unpredictable bursts also. Queue provides us a buffering mechanism that can rate limit the bursts.
2. This decouples application service from the logging mechanism. We can replace logging system by a new swanky system without impacting applications
3. By common sense, application service does not need a response whether log line was submitted or not. So asynchronous system makes sense here

Step 4:

Let's deep dive into search service.

Solution 1:

Off the shelf solution: Use Elasticsearch for indexing logs. Think of ElasticSearch as a database which supports fast text-search, as it uses "Inverted Index" under the hood.

This system starts hitting scaling limits when the data size grows high and cluster size grows >100 servers.

- **Pros**: Hit the ground running quickly with a well established solution until data size is not huge.
- **Cons**:
  - Managing the elasticsearch cluster of this size takes huge amount of time and expertise.
  - All recent logs are stored on the same server. More recent logs are queried much more frequently than older logs. Even with redundancy, this hotspot becomes a performance killer resulting into insanely high latencies.
  - Indexing and search infrastructure is tightly coupled. Indexing and search workloads share the same Elasticsearch infrastructure and can thus not be scaled independently. However, the load on both the system is very different. Indexing has constant load. Search can have frequent bursts. Also, any failure on Indexing would result into outage of Search service.

Solution 2:

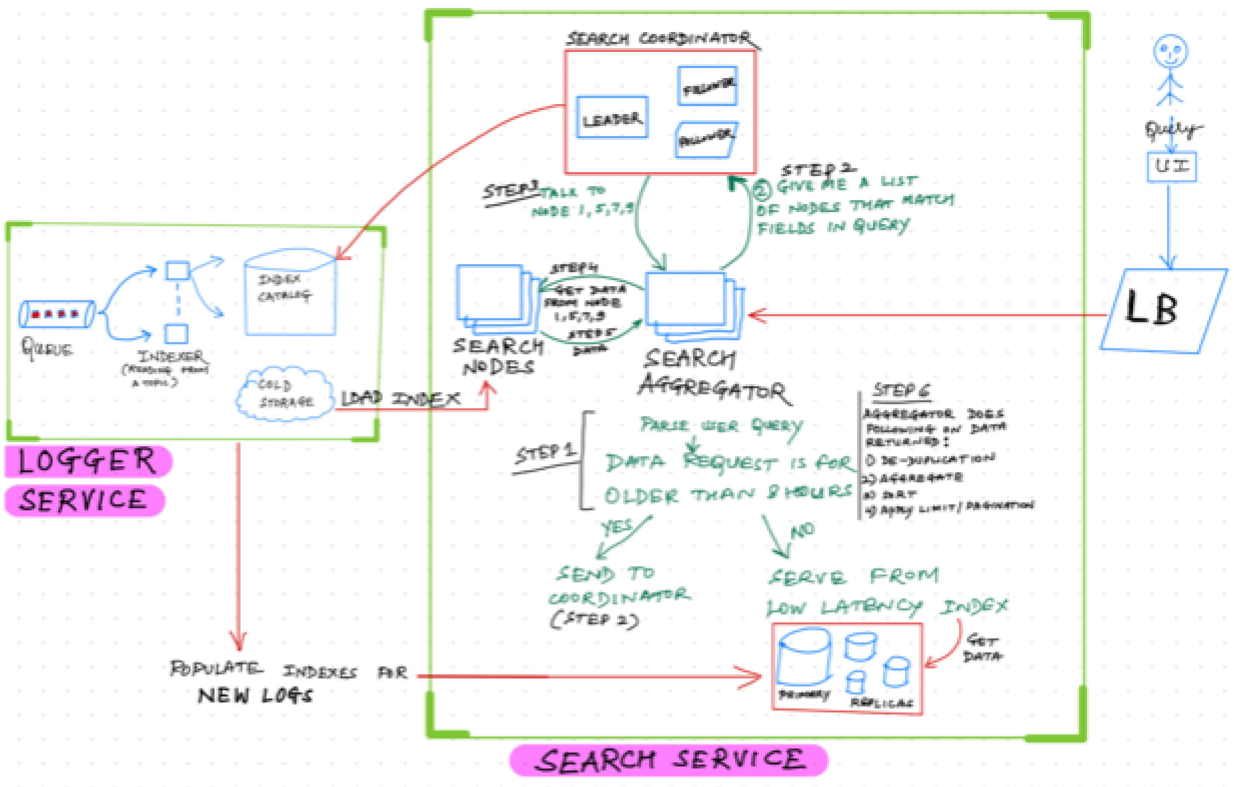
This is an improvement on solution 1 from the first principles.

What is Indexing?

Create separate "Inverted Index (refer to SS2 3 Inverted Index video)" for all the "**Queryable Metadata Fields**(like service, trace_id, datetime, etc.)" and "**each word(aka token) in message body**".

Important Observations:

1. Log messages are immutable. Once written, they won't ever change. Interesting. Does it mean that if we index a log message once, we don't need to reindex it ever? Yes! That means we can generate an "Inverted Index"(we will call it simply index henceforth) and can query it repeatedly.
2. We need to store log messages at three granularities of latency.
    a. Low Latency: Logs from the last few(say 8) hours.
        i. These will be queried at a very very high rate.
        ii. Data size is small.
        iii. Latency expectation is very low, like ~500ms.
        iv. Also this data can be used in live tailing of logs on UI
        v. That means the index should be stored on memory or SSD at least.
    b. Medium Latency: Logs older than 8 hours to a few days(say 7).
        i. These will be queried at a lower rate.
        ii. Data size is big.
        iii. Latency expectation 3-5 seconds.
        iv. Index could be stored on disk
    c. High Latency: Logs older than 7 days to 1 year(our retention cap from requirements).
        i. These will be queried rarely.
        ii. Data size is huge.
        iii. Latency expectations 3-5 minutes.
        iv. Index could be stored on cold storage solutions like `AWS S3`
3. We need to decouple Indexing and Search Service. So they can be treated as separate microservices.



SEARCH COORDINATOR
LEADER
FOLLOWER
FOLLOWER
STEP 2
② GIVE ME A LIST OF NODES THAT MATCH FIELDS IN QUERY
STEP 3 TALK TO NODE 1,5,7,9
STEP 4
GET DATA FROM NODE 1,5,7,9
STEP 5
DATA
SEARCH NODES
SEARCH AGGREGATOR
Query
UI
LB
QUEUE
INDEXER (READING FROM A TOPIC)
INDEX CATALOG
COLD STORAGE
LOAD INDEX
LOGGER SERVICE
STEP 1
PARSE USER QUERY
DATA REQUEST IS FOR OLDER THAN 8 HOURS
YES
SEND TO COORDINATOR (STEP 2)
NO
SERVE FROM LOW LATENCY INDEX
GET DATA
PRIMARY
REPLICAS
STEP 6
AGGREGATOR DOES FOLLOWING ON DATA RETURNED:
1) DE-DUPLICATION
2) AGGREGATE
3) SORT
4) APPLY LIMIT/PAGINATION
POPULATE INDEXES FOR NEW LOGS
SEARCH SERVICE

Components in detail:

Index Catalog

The Catalog is a mapping between **Queryable Metadata Fields**(like service, trace_id, datetime, etc.) and the **index shards stored on cold storage**.
Catalog storage should be durable and fast; where multiple columns can be used to query. Wide column databases are perfect for such usecase.

Indexer

1. Consumer polls and reads N messages from the queue topic to which it is bound. For a threshold of M seconds, no other consumer can read these N messages. Offset is still 0. (Note that message ordering is guaranteed - Messages are read in the same order as they were produced)

   What is this offset?
   The offset is a simple integer number that is used by Queue to maintain the current position of a consumer. That's it. The current offset is a pointer to the last

record that Queue has already sent to a consumer in the most recent poll. So, the consumer doesn't get the same record twice because of the current offset.

2. Each message is transformed into a Json document as per the log schema.
3. This document is indexed into local files on the consumer's disk. This index file is space bound and time bound. (Example 1GB and 1 hour)
4. When space bound or time bound is reached, the index file is compressed and transferred to the cold storage.
5. Cold storage location and the metadata for the index file is stored into the index catalog database. Catalog storage should be durable and fast
6. Then the consumer commits the offset in the queue topic. That means any consumer that reads messages from this particular Queue topic, will read from N+1 position now.
7. Consumer repeats the whole process from the new position and new index file.

Search data node

Responsible for serving indexes stored on its memory and disk. Serves for both medium-latency and high-latency requests. This layer could be implemented with consistent hashing and at least 3 replicas for each index shard.

Search coordinator

The coordinator keeps track of the available search nodes and manages the allocation of relevant index shards to nodes.

When a search data node comes alive, it queries the Coordinator - "What Indexes should I get from cold storage and store them on my memory and disk?" - then fetches the corresponding indices from cold storage

Search aggregator node

- Accepts queries and parses them.
- Depending on the type of query, it aggregates the answers and responds to the requestor.

Low latency Index

Our users would be highly interested in searching for logs which happened in last 8 hours. So we need to serve those queries with very low latency. Explained in Point 2 of important observations.
These indexes are simply inverted index data structures stored on memory for very fast access. Similar to web search problem where ***`key = keyword to be searched`*** and

`*value = sorted list of log line ids*` so that we could find the intersection of log lines for given keyword search

Use Case 1: Low latency Index Requests

- If the query is for last 8 hours data, then talks to low-latency indexes and fetches the intermediate data.
- Low-latency indexes are short-lived indexes. So data size is small.
- Since the data is stored on memory mostly, these requests are blazing fast.
- applies limits to the aggregated query results, de-duplicates, and sorts results
- serves the result back

Use Case 2: Medium latency Requests

- If query is for data older than 8 hours, forwards request to Coordinator - "Tell me which data nodes serve index relevant for this query"
- Coordinator finds that the query is for medium latency data(8 hours to 7 days). This data is always stored on the disk of search nodes. Search nodes can have LRU cache in memory to serve recurring queries faster.
- Coordinator returns a list of Nodes where index shards related to query are there - say <1,5,7,8>
- Aggregator sends requests to above nodes. indices held by the search data nodes on disk.
- Fetches the intermediate data
- Applies limits to the aggregated query results, de-duplicates, and sorts results
- Serves the result back

Use Case 3: High latency Requests

- If query is for data older than 8 hours, forwards request to Coordinator - "Tell me which data nodes serve index relevant for this query"
- Coordinator finds that the query is for high latency data(>7 days). This data is NOT available on search nodes readily.
- Coordinator instructs few nodes to load the index from cold storage
- Responds to search UI which responds to the user that the relevant data is being prepared before the query is dispatched to the backend.
- The relevant shards are cached by the data nodes so that additional queries against the same time range can be dispatched instantaneously.
- Coordinator returns a list of Nodes where index shards related to query are there - say <1,5,7,8>

- Aggregator sends requests to above nodes. indices held by the search data nodes on disk.
- Fetches the intermediate data
- Applies limits to the aggregated query results, de-duplicates, and sorts results
- Serves the result back

Deduplication

Duplicate log messages can appear on indexes. Take this:
- Indexer successfully pushes an index shard to cold storage and registers it with the Catalog
- but the queue commit fails.
- Then another Indexer node may pick up the same log records from the queue and push them as an additional, partially duplicative index shard.

Sharding

A typical user query to fetch matching logs could look like: (Using a sample easy-to-understand query syntax)

(keyword="503 ERROR" OR keyword="500 ERROR") AND env="PROD" and time >= "07/07/2021 09:00:00" AND time <= NOW()

Since logs can be attributed to multiple **Queryable Metadata Fields**(like service, trace_id, datetime, environment, etc.)**,** we need to shard indexes by such fields individually. User need to specify fields in the query so that relevant shards could be accessed for finding matching log lines. All the returned log lines are de-duplicated, sorted by timestamp, and returned back as answer.

Future Growth and Sudden Spikes:

Refer capacity calculations.
We are running our servers at 40% capacity. So even sudden 2x traffic could be handled.

For Application Servers, if requests increase more than 2x then we can have an autoscaling setup for the system. Autoscaling could be triggered :

1. when the CPU crosses a threshold like 70% for more than 5 minutes OR
2. when Requests per second increase by more than 2x for 5 minutes

Since we have a Load Balancer, new servers will be added automatically. We will also have service discovery in place, so that new servers start receiving requests as soon as they are registered and pass health checks(simple TCP pings and more complex Application level health checks).

For storage servers(index metadata DB and cold storage), our servers are running at 40% capacity and we have sharding enabled. With consistent hashing strategy, shard servers in the storage layer could be scaled up and down automatically.

Bonus:

- Inverted Indexes that we talked about above - could be created with open source Apache Lucene.
- For messaging queue, we can use Kafka, AWS Kinesis, GCP PubSub, etc.
- For storing index catalog, we can use wide column NoSQL databases like cassandra, BigTable, etc.
- For cold storage, we can use AWS S3, Google Cloud Storage, etc.