Design Aggregation Service

Problem

Build a flow aggregation service to monitor the network flows. Data about the outbound connections are sent periodically by several agents.

Service should serve :

- Write API - accepts network flow data
- Aggregate the data points per-flow
- Read API - serve the aggregated data

A flow object contains:

```
class FlowObject(BaseModel):
   src_app: str = Field(title='Source App')
   dest_app: str = Field(title='Destination App')
   vpc_id: str = Field(title='VPC ID')
   hour: int = Field(title='Hour')
   bytes_rx: int = Field(title='Bytes Received')
   bytes_tx: int = Field(title='Bytes Transmitted')
```

A unique data flow is a combination of src_app, dest_app, vpc_id for a particular hour; bytes_rx and bytes_tx are aggregated for each unique data flow.

Approach 1

Assumptions:

The total number of write and read requests is limited. Queries are expected to have limited latency. Post requests can contain out-of-order data points, delayed data points, and the aggregated result should include out-of-order requests.

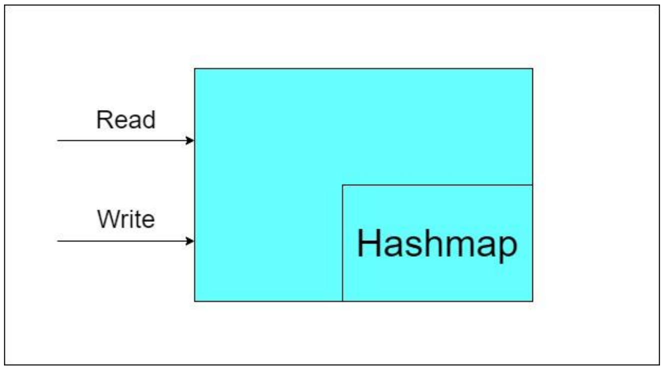
Components:

Web Server:

Handles the request.

In-Memory Data Structure :

Aggregated data is stored in an in-memory data structure like a hashmap(dict) or panda dataframe.



Work Flow:

- POST and GET calls are handled by the same web server.
- On a POST request, the appropriate method aggregates all the datapoints of the current POST request and updates the in-memory hash map. Hash Map (key: (src_app, dest_app, vpc_id, hour) and value:(bytes_rx, bytes_tx)). On every post request, the hash map is locked and the aggregated bytes are updated.
- On a GET request, all the data points containing a specific hour is identified, and the results are returned in a JSON

Advantages:

- Simple and easy to implement.
- Suitable for a single box with low load and persistence not required.

Scaling Issues:

- **Memory:** The size of the hash map keeps growing based on the number of unique data flows.
- **No Persistence**: If the webserver dies, all the in-memory data is lost and can't be recovered. If we use a panda dataframe we can store it in the disk at regular intervals(a few minutes/hours). But even in this case, some uncommitted data might be lost due to a sudden crash.
- **Locking Contention**: The entire hashmap needs to be locked during writes and reads. Hence the throughput and latency on the query will be increased if a lot of requests need to processed.
- **No Horizontal Scaling**: Since in-memory hashmap horizontal scaling is not possible

Approach 2:

Assumptions:

Write heavy system. Since data will be posted from the agents more regularly. Lesser reads are assumed, list of aggregated data is needed per hour. Since reads are limited, and aggregation is done during read calls, medium latency is expected based on the level of aggregation needed. Post requests can contain out-of-order data points, the aggregated result should handle out-of-order or delayed requests.

This is the implemented design for the POC. This is built on the MAC OS Unix environment.

Design Choice for Aggregation on Read:

Ideally, we might want to aggregate the data on write and store it so that reads are performant. This design choice was made to **handle out-of-order/delayed data.**

I rejected some *ideas because they were either non-readable(or complex) or non-performant. A notion of time would exist in real-time systems, it helps in implementing watermarking and other pieces of code to handle out-of order data.

The key problem is finding the unique combination of group keys(src_app, dest_app, vpc_id, hour) that were already registered. In a production system, we might have to implement watermarking and store the intermediate state for a period of time and then flush the results to the table (flink or spark does this well and if possible, this functionality should be delegated to these systems).

Ideas that were rejected:

1. Implementing a vanilla watermarking solution like spark structured streaming where the intermediate state is maintained in a pandas dataframe and after a specific number of post requests for each hour(basically for each hour, we track the number of post requests they were in and a predefined watermark), and we could flush the contents to a results database. We could also be persisting on disk. This would suffer from the disadvantages identified in approach 1 and could take time to implement and test correctly. This would also mean that get requests for an hour would be empty until the watermark threshold is exceeded and it could be hard to evaluate under the given assessment conditions.
2. Have a background thread maintained for each hour, we track the number of post requests they were in and after they exceed the watermark threshold, aggregate from raw_events table and insert into results table. This would also mean that get requests for

an hour would be empty until the watermark threshold is exceeded and it could be hard to evaluate under the given assessment conditions

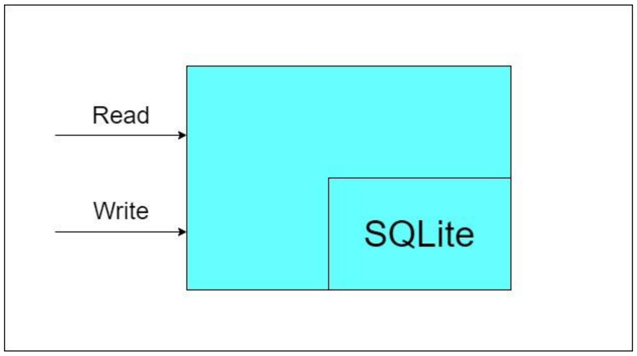
Components:

Web Server:

Handles the request.

SqlLite3:

The in-memory disk-based database serves as a persistent store. SQL Alchemy provides ORM support and https://www.encode.io/databases/ provides async support to SQLite database



Workflow:

- POST and GET calls are handled by the same web server.
- On a POST request, the appropriate method writes those values into a SQLite database.
- On a GET request, (for this initial POC) web service get requests from this database. Data aggregation is done during GET requests, as a result, latency could be higher compared to POST (Note that given the type of data, a time series database is more suited to our purpose)

Advantages:

- **Persistence**: Since data is stored to a disk-based persistent database, the data can be recovered even if the process crashed/restarted. No Data is not lost.
- No in-memory data store.

Scaling Issues:

- **Memory:** Disk volume could exceed, based on the size of the DataBase
- **Slower Reads:** Aggregation is done at reads, so latency could be increased.

- **Heavy Load:** Large read and write operations, DB could be strained.
- **No Horizontal Scaling**: Database sharing is not possible

Approach 2.1

Assumptions:

The system is a real-time system and precomputed test hour data won't work. The system does not handle out-of-order data points. Read accepts the delay for processing the current hour data. Unprocessed hours read returns no data.

WorkFlow:

- If out-of-order requests need not be handled, a better option is to have a separate thread to perform the aggregation and store the aggregated results in a second database (results database - SQLLite).
- The POST requests store the data in the event collection database. Tracks all the received requests in order. Out-of-order data is not processed.
- Background Thread runs once every hour to update the aggregated results in the results database.
- The GET requests fetch the data from the results database. Since data is already aggregated, responses are fast.
- There could be a delay in storing the aggregated results. So when a get request for an hour that is not processed yet is received, the system returns an empty result

Approach 3:

Assumptions:

Write heavy and Read heavy real-time system. Since data will be posted from the agents more regularly. Lesser reads are assumed, list of aggregated data is needed per hour. Since reads are limited aggregation is done at reading calls, medium latency is expected based on the level of aggregation needed. Post requests do not handle out-of-order data points.

This could be implemented for the production scale.

Components:

Front End Gateway:

The proxy inspects the request and forwards it to corresponding components depending on the type of request. It also takes care of load balancing.

Write Web Server:

The WS batches these events into a persistent disk-based database. We could have background threads to periodically push those batched events to kafka and also perform automatic clean up and compaction of the persistent database.

Read Web Server:

The read WS queries the Time series database (If the time series database is druid, we query the broker)

Spark/Flink:

The events are read from Kafka, aggregated, and stored in the time-series database. This is suitable to aggregate huge amounts of data. Out-of-order data and delayed data handling come for free with these systems.

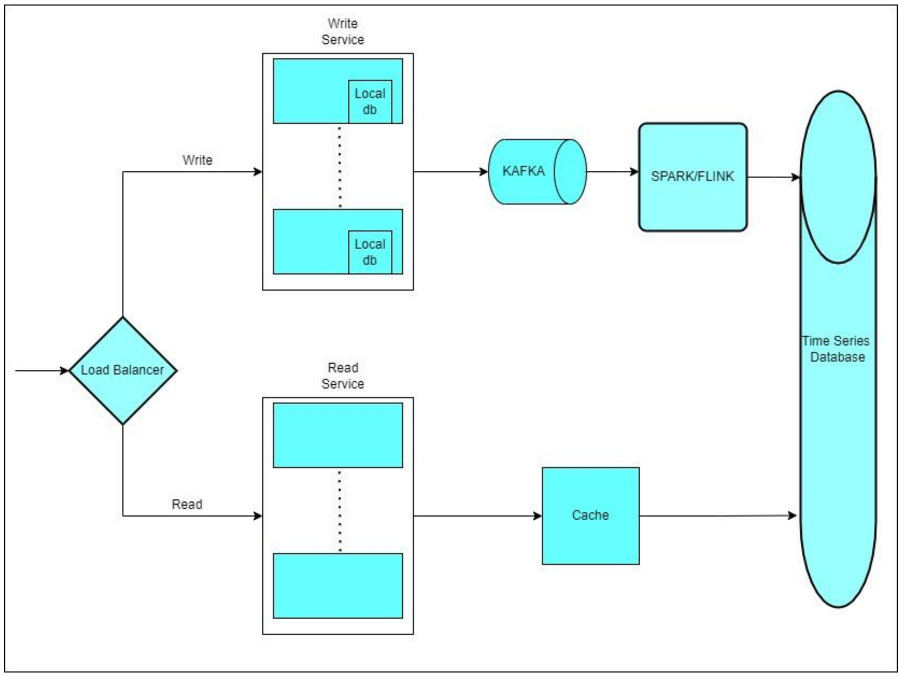
Time Series Database:

We are dealing with time-series data and we mostly have aggregation queries. NoSQL, columnar databases are a great fit for this use case and to name a few- Apache Druid, Vertica is a well-known time-series analytics database that is extremely performant for aggregation queries. Assuming we choose Apache Druid, we can have ingestion from Kafka with dimension columns src_app, dest_app, hour, vpc_id, and metrics columns bytes_rx, bytes_tx. Then we can just do a sum of the bytes_rx and bytes_tx.

One more advantage is when we run into scaling problems, we can scale each of the components independently. If we identify that the ingestion process is a bottleneck, we can add more druid middle managers and overlord component nodes. If the Query side is the bottleneck, we can add more druid brokers and to scale database size, we can add more historical nodes.

Note:

Each of the components could be scaled independently based on the load.



Workflow:

- POST and GET are handled by respective web servers.
- On a POST request, the write web server ingests the data to a time-series database through Kafka and Spark.
- On a GET request, the data is first looked into the cache like Redis/Memcache and if data is not available it looks into the time series database.
- For the current hour, data is not processed completely and only partial data will be available.
- Retention of events will be stored for up to 1 week and can be altered as per need. The system follows eventual consistency.

Advantages:

- **Persistence**: All incoming events are recorded locally.
- **Heavy Load:** Large read and write operations are performed by individual services. The time series database handles aggregation for such data and provides the results efficiently. The read is connected to a cache to provide faster results
- **Horizontal Scaling**: Each component can be scaled individually, to support large traffic.
- **Retention:** Event data are stored for up to a week. And can be increased based on the scale and use of the application.
- **Latency**: The overall latency is minimum as the data is aggregated and stored separately.

Maintainability:

- A monitoring /health check process will capture the heart beats of each service at regular intervals to ensure the status of each service.
- Different logs like access and application logs ll be supported and error logging messages will be implemented at different levels.
- Run books will be recorded to bring the service up in need of any emergency failure.

Availability:

- Each pod will have a cluster of web service. In case of failure of one node, traffic will be handled by others. Also, this will distribute the load on each node.
- For larger amounts of data, these clusters can be distributed geographically in different locations.

Proof of Concept Implementation:

- Application and access level logging is implemented.
- Exceptions are handled to throw more information on the error.
- Ideally, I would have separate unit tests, functional tests, and integration tests. However, I merged the functional and integration tests into a single file. So basically, the fixture spawns a test database on the fly and closes it once done.
- I was not able to make a few test runtime exceptions in my testing.
- For API Versioning, I used a package fastapi_versioning which works fine but causes issues with dependency overrides during test time. So they are commented and the given solution is not API versioned.

References:

- https://fastapi.tiangolo.com/
- https://pytest-cov.readthedocs.io/en/latest/reporting.html
- https://aws.amazon.com/big-data/what-is-spark/