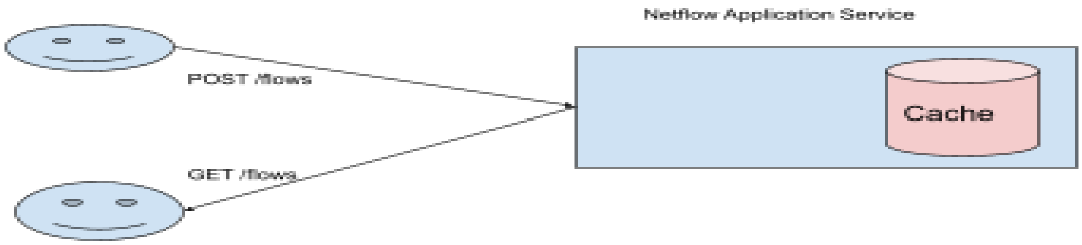
Netflow Aggregation Service

Background :

The *Netflow Aggregation Service* collects network “flows” (aggregated connection data) to understand how the network is used. The service accepts network flow data points from a set of agents on a large number of instances which monitor outbound connections and report them periodically. The service provides two API’s, one for ingesting (write) and the other for reporting (read) netflow data points .



Endpoints

Read API

- Path: ***/flows***
- Operation: GET
- Query parameters hour (int) - required
- Message/Payload: A JSON array specified below

```
curl “http://localhost:8080/flows?hour=1”
```

Write API

- Path: /flows
- Operation: POST
- Query parameters: None
- Message/Payload: A JSON array specified below

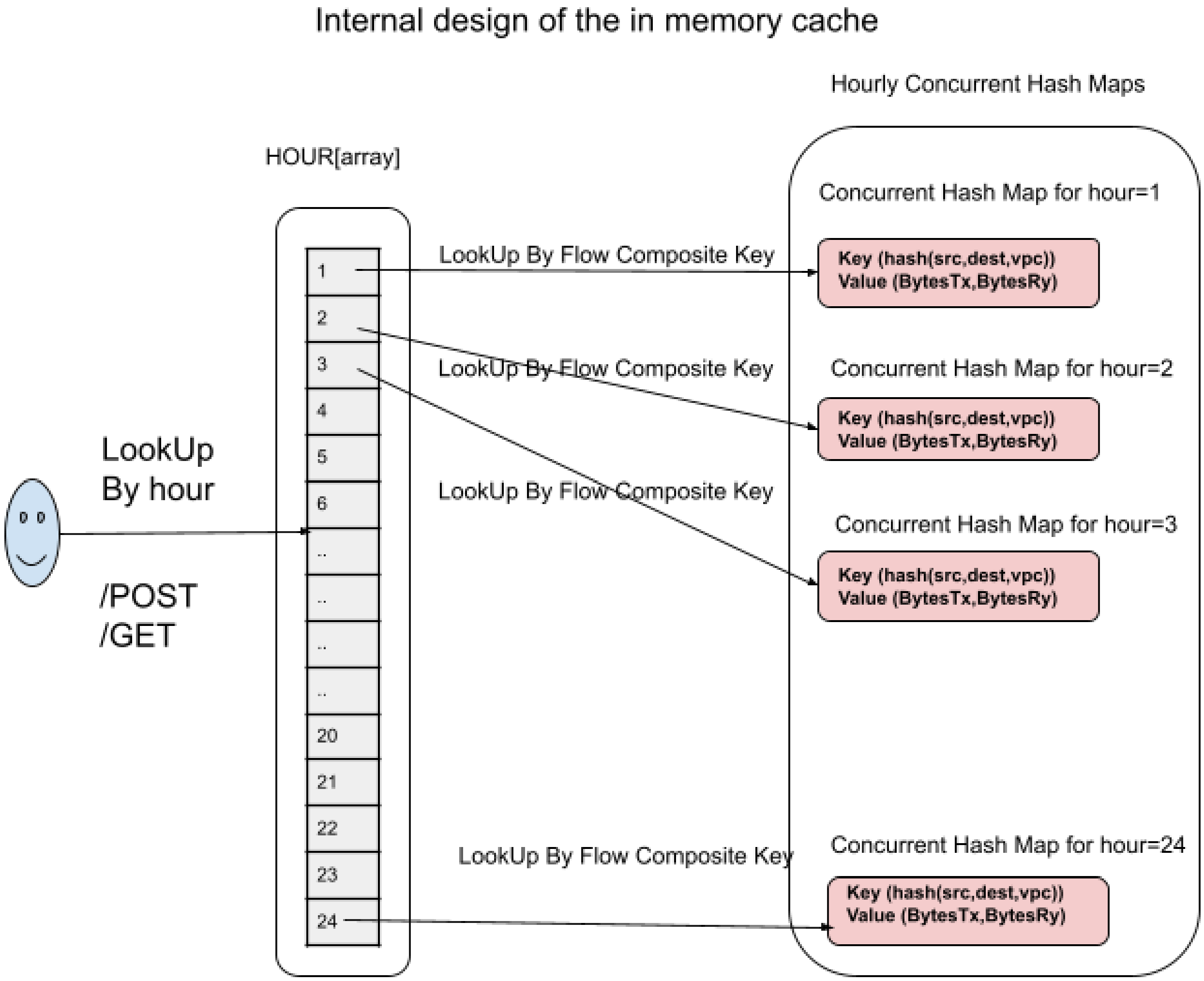
```
curl -X POST “http://localhost:8080/flows” \
  -H 'Content-Type: application/json' \
  -d '[{"src_app": "foo", "dest_app": "bar", "vpc_id": "vpc-0", "bytes_tx": 100, "bytes_rx": 500, "hour": 1}]'
```

Requirements

1) Handle frequent heavy write batch loads periodically .
2) Write heavy service with frequent updates . Reads are periodic .
3) The service does not have to worry about the order in which tuples are processed .
4) Aggregation of BytesTx and BytesRy per unique *src_app,dest_app,vpc_id, hour*
5) Write/Query performance should not degrade with time and load

Design

[LOCAL] In Memory Implementation



Internal design of the in memory cache
Hourly Concurrent Hash Maps
HOUR[array]
Concurrent Hash Map for hour=1
LookUp By Flow Composite Key
Key (hash(src,dest,vpc))
Value (BytesTx,BytesRy)
1
2
3
4
5
6
..
..
..
..
20
21
22
23
24
LookUp By Flow Composite Key
Concurrent Hash Map for hour=2
Key (hash(src,dest,vpc))
Value (BytesTx,BytesRy)
LookUp
By hour
LookUp By Flow Composite Key
Concurrent Hash Map for hour=3
Key (hash(src,dest,vpc))
Value (BytesTx,BytesRy)
/POST
/GET
LookUp By Flow Composite Key
Concurrent Hash Map for hour=24
Key (hash(src,dest,vpc))
Value (BytesTx,BytesRy)

On POST requests that submit a list of *netflow* data point rows , the service iterates over each of the items parsing the json row into a *NetFlowEntity .* Based on the hour field in the json row a *ConcurrentHashMap* is selected which has the key [hash(src_app,dest_app,vpc_id)] and value is an [Atomic Integer Array] . The value field is intentionally kept as an atomic array so that updates to it are thread safe . This also reduces the space complexity of the cache storage .

On GET requests since the hour param is required , the search space is one single *ConcurrentHashMap* with already aggregated values . Persistence and recovery is implemented in the write ahead log (TODO) and there are no ordering requirements of incoming request data

Storage

- Outer array of hourly data → Fast lookup based on array indexes
- Hourly Data Cache → Concurrenthashmap per hour i.e. 24 HashMaps
  Data Sample : Size of a sample JSON string : 85 Bytes

  ```
  {"src_app": "foo", "dest_app": "bar", "vpc_id": "vpc-0", "bytes_tx": 100, "bytes_rx": 300, "hour": 1}
  ```

  Entry Key :

  hash(src_app,dest_app,vpc_id)
  Size of Key : 4 bytes

  Entry Value :

  Atomic Integer Array
  Size of Value : 12 bytes for BytesTx and 12 Bytes for BytesRy

  Size per map entry : Key + Value = [4B + 12B] = 16 Bytes
  Total Storage Requirements : Unique Keys * 16 Bytes

Recovery (TODO)

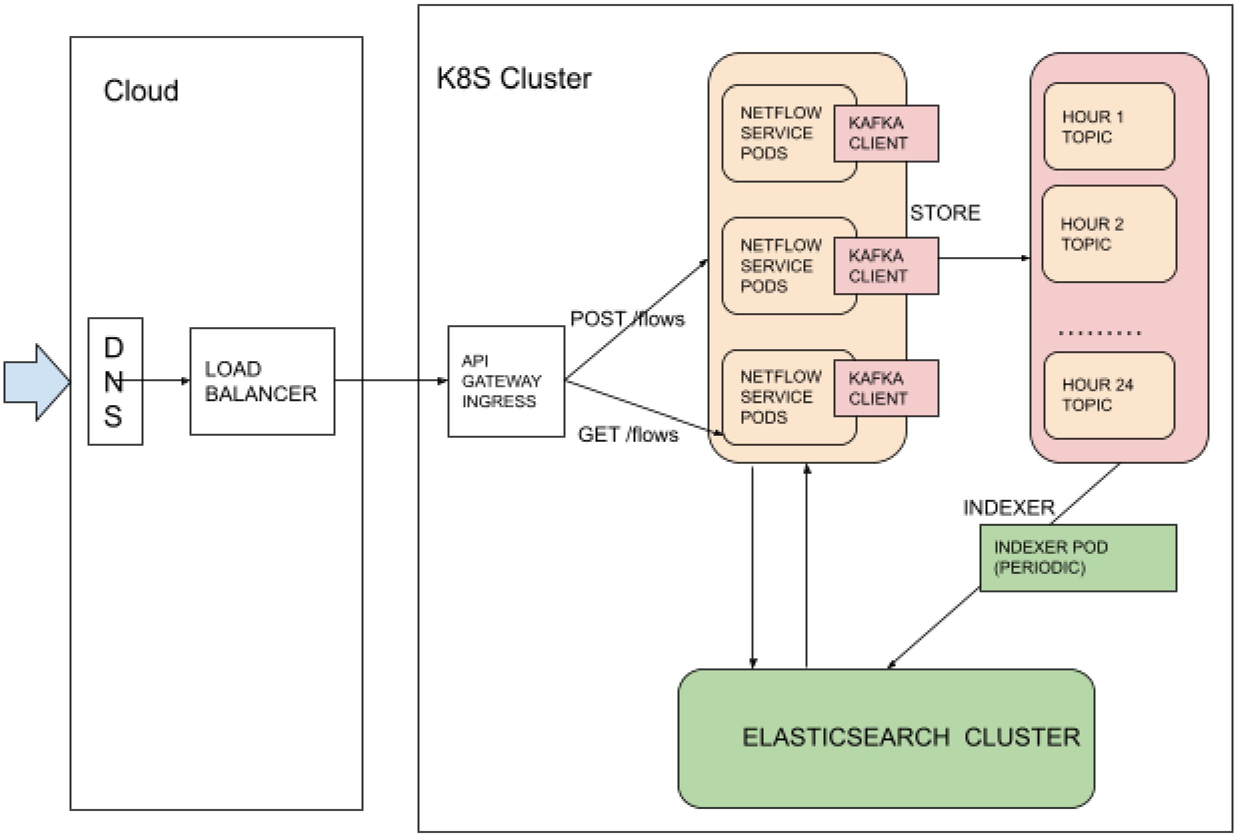
Write ahead log with periodic flushing . Before setting an entry or updating an entry in cache the raw json string can be appended to a list or blocking data structure which can then be flushed at periodic intervals to persist the data to disk . On restarting of the application due to failure this WAL can be replayed to reconstruct the storage structures

Alternatives (Considered)

Store every tuple without aggregating BytesTx and BytesRy and then on query iterate over the result set and compute/aggregate BytesTx and BytesRy during query processing .

- Advantages : Improves ingestion performance .
- Disadvantages : Degrades query performance .

[PRODUCTION] In Real World Implementation



- DNS is geo based , hence login from US-WEST1 would hit US-WEST1 Load balancer and same for all regions . This will also enable regional failover using Regional load balancers.
- The Load Balancer is a set of forwarding rules to API proxy.
  - Can be used to terminate HTTPS connections at this layer .
  - Rate Limiting/Request Throttling can be configured at this layer
  - Request Transformation , Request Tracing Headers can be configured here
- Kafka will serve as the source of truth (WRITE AHEAD LOG)
  - Although no ordering is required one TOPIC for writing and reading would suffice
  - Otherwise Kafka can have hourly topics partitioned into tuple composite keys
- Scaling the *service pods* horizontally will enable it to take more load based on traffic .
- Elastic sharded clusters to serve GET requests aggregating data during query phase
- **Redis/Postgres** clusters can also be experimented to see if they can be used instead of Kafka/Elasticsearch