Allezon Analytics Platform

Overview

We are going to create a data-collection and analytics platform for a big online retailer Allezon. Users' actions on the Allezon site are sent to us as VIEW and BUY events containing information about the user and the product they interacted with. In order to solve Allezon's business use cases our platform has to collect these data and provide endpoints answering specific queries.

Use Cases

Input events we are going to work with represent users' actions on the Allezon site and are called user tags. There are two types of actions, VIEW when the user visits some product page and BUY when they decide to make a purchase. Our primary goal is to collect those input events and then respond to the queries according to their specification. All communication with our platform should happen via HTTP requests.

Data Format

Let's look first at
User actions such as VIEWs and BUYs are sent as user tag events in the following format:

<user_tag>

```
{
  "time": int64,
  "cookie": string,
  "country": string,
  "device": PC | MOBILE | TV,
  "action": VIEW | BUY,
  "origin": string,
  "product_info": {
    "product_id": string,
    "brand_id": string,
    "category_id": string,
    "price": int32
  }
}
```

Time ranges in request parameters are of the following format:

<time_range>

- type: String
- format: <1m_datetime>_<1m_datetime>
- example: 2022-03-22T12:15:00_2022-03-22T12:30:00
  - it represents 15 minute time range

<1m_datetime>

- type: String
- format: 2022-03-22T12:15:00
- info
  - UTC
  - 1m granularity (multiples of 1m)

Use Case 1: Adding User Tags

We should provide an endpoint that allows adding user tag events, one at a single HTTP request. Storing these data in an efficient and reliable manner makes a foundation for the further use cases.

Request

- POST /user_tags
    - Parameters
        - None
    - Body (type: application/json)
        - <user_tag>

Response

- 204 (No Content)
    - when a user tag has been added successfully.
- Other status codes are interpreted as errors.

Requirements

- max throughput: 1000 req/s
- request timeout: 200 ms
- losing events may cause producing inaccurate answers to the queries

Data Characteristics

- number of unique cookies ~ 1M
- ...

Recommendations

- Start from a dummy server which responds always with expected 204 status code, but ignores incoming events.
    - Queries about events and their aggregates can be run in a debug mode (more on that in Testing Platform section), so you should be able to bootstrap a "working system" quite easily ;)
        - In short: debug mode means that the expected answer is sent in the queries' body, so you can just send it back.

Use Case 2: Getting User Profiles

Business Context

Allezon plans to build a recommendation engine in the near future. At the start of this initiative they want to have quick access to the user profiles (history of the user actions).

Request

- POST /user_profiles/{cookie}?time_range=<time_range>
    - Parameters (query-string)
        -

	type	format	required	default	preconditions	info
[qs] time_range	String	<time_range>	Y	N/A		seconds granularity

- Why POST (not GET)?
    - All queries can be run in a debug mode causing there is an expected answer sent in their body.
    - Find more on that in the Testing Platform section.

Response

- 200 OK (body: application/json)
    -

```
{
  "cookie": string,
  "last_views": [<user_tag>],
  "last_buys": [<user_tag>]
}
```

Requirements

- max throughput: 1000 req/s
- request timeout: 200 ms
- data latency: 10 s
    - i.e. time between accepting an input event and serving it in the query responses should be less than 10 s.
- We have to keep 200 most recent VIEWs and 200 most recent BUYs for each cookie (older events can be discarded).
    - There is a separate limit for each action type.

Recommendations

- Start with an in-memory solution.
- Use a key-value store.

Use Case 3: Aggregated User Actions (Statistics)

Business Context

Executives at Allezon want to view historical data from different perspectives. There is a broad category of queries for which aggregating results in 1 minute buckets is satisfactory. Such queries will allow us to present various metrics over longer periods of time enabling us to visualize trends such as: users' buying patterns, ad campaigns' performance, etc. Aggregates should become available in query results soon after their time buckets are closed (in a matter of minutes). There is no predefined list of metrics which are going to be visualized.

Examples

Example 1: We want to draw a chart showing the number of Nike products bought over time.

- Query parameters
    - time_range = <time_from>_<time_to>
        - <time_from> and <time_to> have to be proper 1m bucket ends.
        - Let's assume there are N buckets inside this range: bucket1 ... bucketN.
    - action = BUY
    - brand_id = Nike
    - aggregates = [COUNT]
        - list of aggregates to retrieve for each bucket

- HTTP query
    - POST /aggregates?time_range=<time_from>_<time_to>&action=BUY&brand_id=Nike&aggregates=COUNT
- Result (presented here as rows)
    -

1m_bucket	action	brand_id	count
bucket1	BUY	Nike	49
bucket2	BUY	Nike	52
...	...	...	...
bucketN	BUY	Nike	77

        - Note that values in the first 3 columns derive directly from query parameters, i.e. they are known without running the query. Only aggregates' values have to be fetched from the system.

Example 2: We want to draw a chart showing the number of Adidas women shoes viewed over time.

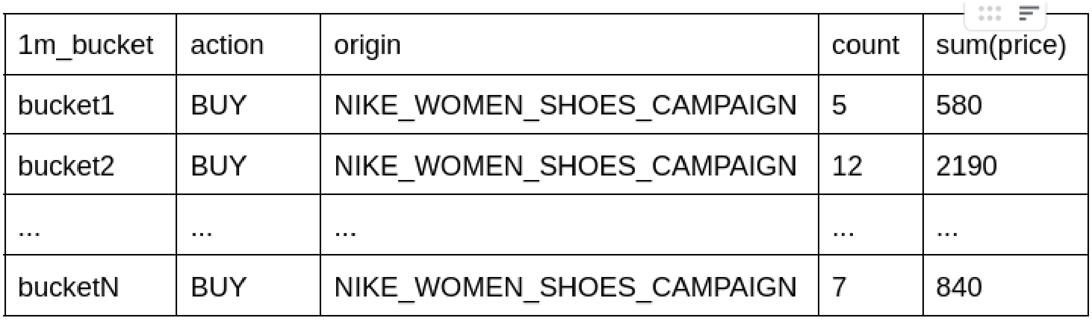
- Query parameters
    - time_range = <time_from>_<time_to>
        - <time_from> and <time_to> have to be proper 1m bucket ends.
        - Let's assume there are N buckets inside this range: bucket1 ... bucketN.
    - action = VIEW
    - brand_id = Adidas
    - category_id = WOMEN_SHOES
    - aggregates = [COUNT]
        - list of aggregates to retrieve for each bucket
- HTTP query
    - POST /aggregates?time_range=<time_from>_<time_to>&action=VIEW&brand_id=Adidas&category_id=WOMEN_SHOES&aggregates=COUNT
- Result (presented here as rows)
    -

1m_bucket	action	brand_id	category_id	count
bucket1	VIEW	Adidas	WOMEN_SHOES	510
bucket2	VIEW	Adidas	WOMEN_SHOES	490
...	...	...	...	...
bucketN	VIEW	Adidas	WOMEN_SHOES	770

        - Note that values in the first 4 columns derive directly from query parameters, i.e. they are known without running the query. Only aggregates' values have to be fetched from the system.

Example 3: We want to draw a chart showing two metrics: the number of BUY actions and the revenue, generated by the given origin (ad campaign) over time.

- Query parameters:
    - time_range = <time_from>_<time_to>
        - <time_from> and <time_to> have to be proper 1m bucket ends.
        - Let's assume there are N buckets inside this range: bucket1 ... bucketN.
    - action = BUY

    - origin = NIKE_WOMEN_SHOES_CAMPAIGN
    - aggregates = [COUNT, SUM_PRICE]
        - list of aggregates to retrieve for each bucket
        - Note that here we have more than one aggregate.
- HTTP query
    - POST /aggregates?time_range=<time_from>_<time_to>&action=BUY&origin=NIKE_WOMEN_SHOES_CAMPAIGN&aggregates=COUNT&aggregates=SUM_PRICE
        - Different aggregates are represented by repeating "aggregates" query params.
- Result (presented here as rows)
    - 

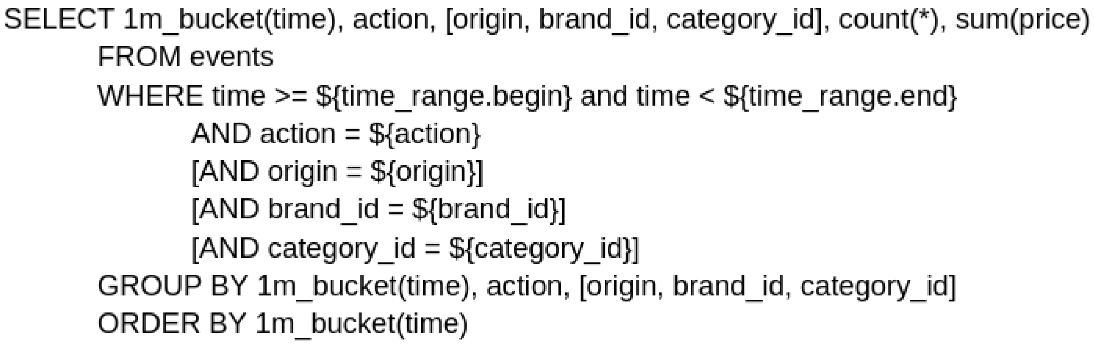
1m_bucket	action	origin	count	sum(price)
bucket1	BUY	NIKE_WOMEN_SHOES_CAMPAIGN	5	580
bucket2	BUY	NIKE_WOMEN_SHOES_CAMPAIGN	12	2190
...	...	...	...	...
bucketN	BUY	NIKE_WOMEN_SHOES_CAMPAIGN	7	840

        - Note that values in the first 3 columns derive directly from query parameters, i.e. they are known without running the query. Only aggregates' values have to be fetched from the system.

Description

Having seen the above examples we are ready to formulate a generic description of the query for this use case:
We want to get aggregated stats about user actions matching some criteria and grouped by 1 minute buckets from the given time range.

- Required parameters
    - time_range = <time_from>_<time_to>
        - <time_from> and <time_to> have to be proper 1m bucket ends.
            - example: 2022-03-22T12:15:00_2022-03-22T12:30:00
                - the above string represents 15 minute time range [12:15, 12:30)
        - time_range consists of non-overlapping, consecutive 1m buckets
    - action
        - allowed values: {VIEW, BUY}
    - aggregates
        - list of values from {count, sum_price}
- Optional parameters
    - origin
    - brand_id
    - category_id
- Semantics of the query corresponds to the following SQL



```
SELECT 1m_bucket(time), action, [origin, brand_id, category_id], count(*), sum(price)
        FROM events
        WHERE time >= ${time_range.begin} and time < ${time_range.end}
                AND action = ${action}
                [AND origin = ${origin}]
                [AND brand_id = ${brand_id}]
                [AND category_id = ${category_id}]
        GROUP BY 1m_bucket(time), action, [origin, brand_id, category_id]
        ORDER BY 1m_bucket(time)
```

Request

- POST /aggregates?time_range=<time_range>&action=<action>&aggregates=[<aggregate>]
    - Parameters (query-string)
        -

	type	format	required	default	preconditions
[qs] time_range	String	<time_range>	Y	N/A	proper 1m bucket ends
[qs] action	enum	{VIEW, BUY}	Y	N/A	
[qs] aggregates	enum	{COUNT, SUM_PRICE}	Y	N/A	
[qs] origin	String		N	null	
[qs] brand_id	String		N	null	
[qs] category_id	String		N	null	

Response

- 200 OK (body: application/json)
    -

```
{
  "columns": ["1m_bucket", <filter_columns>, <aggregate_columns>],
  "rows": [
    [<bucket1>, <filter_col_values>, <aggr_col_values>],
    ...
    [<bucketN>, <filter_col_values>, <aggr_col_values>]]
}
```

Requirements

- max throughput: 1 req/s
- request timeout: 60 s
- data latency: 5 m
    - i.e. we can query about 5 minute old buckets

Recommendations

- Keep data in Kafka in such a way you can query them quite easily.
    - Read as little as possible data from Kafka.
    - Start from sth easy (may be not very efficient).
    - You should be able to implement it using Kafka consumer API.
- If you have time you can experiment with some Kafka libraries/frameworks.

Testing Platform

Nothing seems to be more attractive for a developer than being able to work with a well designed testing platform. Allezon tries to provide such an environment and hopes you will find it useful.
Testing Platform enables you to subscribe to a stream of events (user tags and queries) which are then emitted to the host:port of your choice. The easiest way to start a subscription is through Web-Panel, another option is using HTTP Subscription API.

Web-Panel

Web-Panel is available here: http://rtb101vm.rtb-lab.pl:8082/
You can authenticate to the panel with a token generated from your rtb-lab password, for example:

```
echo "your_passwd_here" | md5sum | awk '{print $1}'
```

gives:

```
d5b08b2caf9d6272065f0ddac8ca0da6
```

There is a single page where you can manage your subscriptions. You can have one subscription at a time.
After specifying host, port, events_per_second, seed and clicking "Start" your app running on host:port should start receiving HTTP requests (events and queries) according to the above specs.
User tags (events) will come with the specified throughput, whereas queries (use cases 2 and 3) will be sent with the following frequency:

- UserProfileQuery: one query for every 10 user tags (100 queries/s for the maximum throughput)
- AggregatesQuery: one query for every 1000 user tags (1 query/s for the maximum throughput)

At the beginning of the subscription some queries may return "empty" results (meaning there are no user tags matching criteria). "Empty" does not mean "with empty response body", but e.g. for UserProfileQuery with empty views and buys lists.
The subscription can be paused, resumed (multiple times) and then finally closed (cannot be resumed after closing it).

Stream of Events and Queries

More detailed description will be put here shortly. For the time being please go through the below list of the most important subscription features:

- Stream of events and queries is deterministic and repeatable for the given seed.
- Field "time" in user tags represents event-time and starts always from the same point in time.
- Time delta between consecutive user tags is always 1 ms which means that the logical throughput of events is 1000 events/s.
  - Think of it that it is the prerecorded stream of events which can be replayed with the given speed (the throughput you define in your subscription).
  - Setting lower values of throughput in the subscription does not change "time" values in the events. Your host:port will just get the events at a lower rate.
- The only parameter changing events in the stream is seed.
  - i.e. Starting a subscription with the same seed should result in producing exactly the same events and queries.

Subscription API

For managing your subscription please use Web-Panel - you are able to run all actions on your subscription through Web-Panel (for now this is the recommended way of working with subscriptions).
Allezon promises to put HTTP specs here soon ;)