Version: 1

Signoff

Daragh:

Clint:

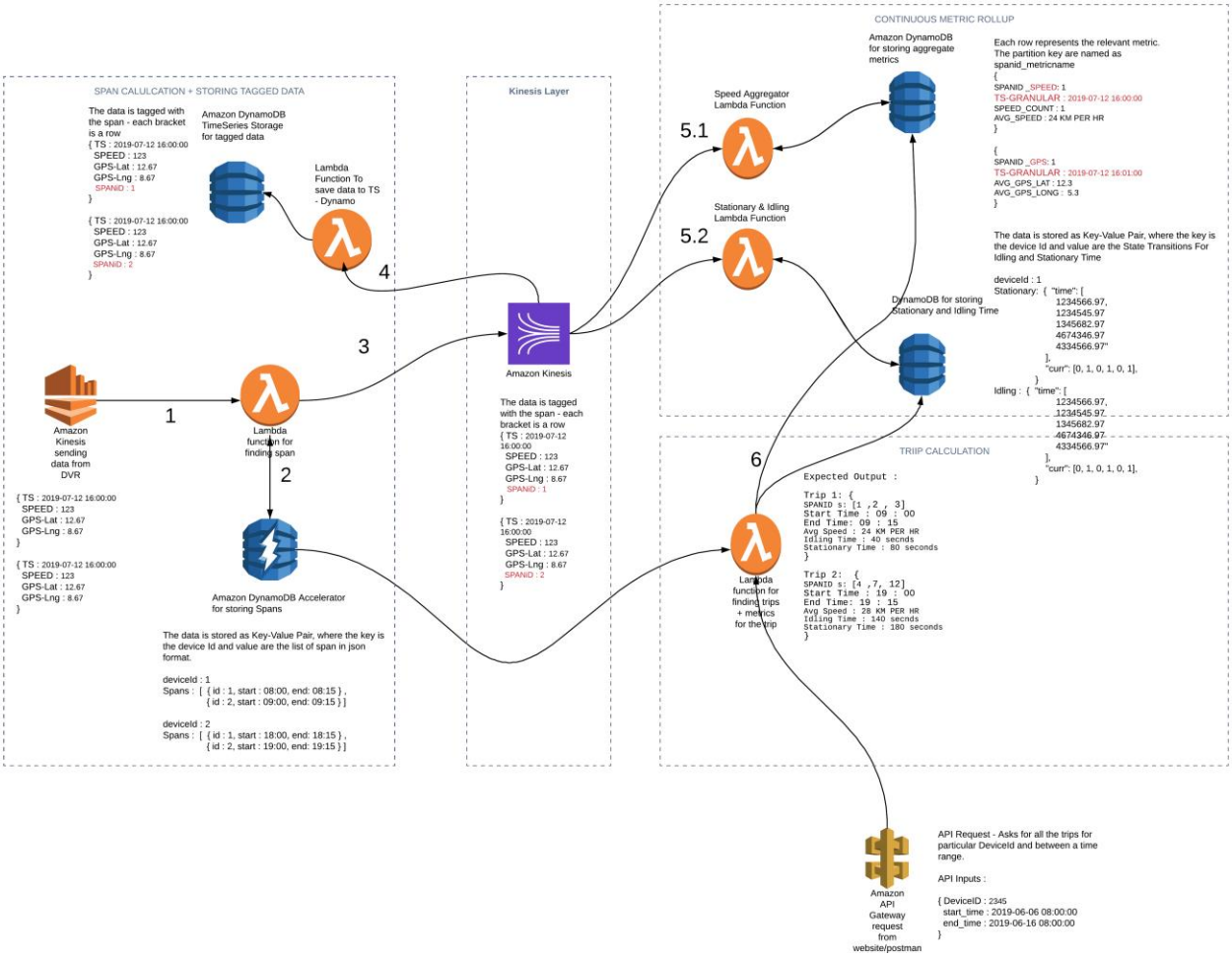
Historical Aggregation

The aim of this module is to provide aggregate values/statistics for a trip.

Expected Input: Span IDs comprising a trip.

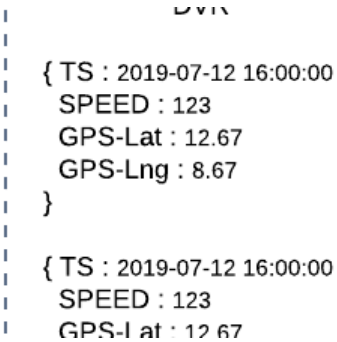
Expected Output: Average Speed, Total Idling Time, Total Stationary Time etc

Architecture



1 – Ingestion from Kinesis

The DVR sends data to Kinesis. Each data point sent by the DVR has **at least** the following attributes:



These data points are converts into records which contain 10-such data points. Our aim is to process as many data points as we can.

2 – Span Calculation

Aim: Tag the data points with SpanID and save them to Timeseries Database.

This module starts with ingesting datapoints from Kinesis. Each data point is the data inside the {}. The data points contain a timestamp, GPS coordinates, Speed etc. The lambda function ingests the data from the Kinesis and queries the Dynamo Table which stores the Span for devices. The algorithm then finds the spanID and tags the data point with it. This tagged Data is shown in Red colour, which is sent to a separate Kinesis Stream. The implementation of the algorithm is not in scope of this document. (Refer to the Span Calculation Module/GitHub Repo for implementation detail)

3 – Sending tagged data to Kinesis Stream

Once we tag the data with the span, we send this data to a Kinesis Stream which will persist it for later use and also perform other aggregations!

4 - Persisting tagged data to timeseries DynamoDB

Here we have a lambda function which takes the tagged data from Kinesis and does a Batch Write to Dynamo DB to persist. The TS DynamoDB database looks like below :

spanId	timestamp	latitude	longitude	speed
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:00:01	51.612198	-0.043751	180
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:00:11	51.611962	-0.043996	0
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:00:21	51.611962	-0.044000	0
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:00:31	51.611962	-0.044001	0
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:00:41	51.611962	-0.044001	0
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:00:51	51.611962	-0.044001	0
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:01:01	51.611962	-0.044001	0
3b9b08f9-1298-44fc-9cbe-4409c4900df6	2019-09-04 00:01:11	51.611962	-0.044001	0

spanId	timeStamp	count	idling	speed	stationary
52dc662f-2dac-47ab-9b84-5360250ce066	2019-09-10 08:09:00	1.0	1.0	0.0	1.0
52dc662f-2dac-47ab-9b84-5360250ce066	2019-09-10 08:10:00	4.0	2.0	20.0	2.0
52dc662f-2dac-47ab-9b84-5360250ce066	2019-09-10 08:11:00	6.0	0.0	3.3333333333333335	5.0
52dc662f-2dac-47ab-9b84-5360250ce066	2019-09-10 08:12:00	6.0	0.0	0.0	6.0
52dc662f-2dac-47ab-9b84-5360250ce066	2019-09-10 08:13:00	5.0	0.0	0.0	5.0

5 – Continuous Metric Rollup

This section is intended to be a modular layer, which computes different metrics as needed. Each Lambda Function takes data from the Kinesis Stream and updates the relevant aggregate in the DynamoDB. In the following sub-section, we will discuss how each metric is aggregated.

5.1 - Speed Aggregation Function

Implementation

When new data comes in from DBStream object, we extract the data from the batch and convert the timestamp from string to datetime type and round the data to the minute.

```
423
424     # extract the data from event
425     extracted_data = list()
426     for record in event["Records"]:
427         extracted_data.append(extract_data_from_record(record, schema))
428     logging.info(
429             "Extracteed : {} Datapoints from event".format(len(extracted_data))
430     )
431
432     if(event['Records'] == []):
433         logging.info('No Records to Process')
434         return 'No records to process'
435
436     # get the event data frame
437     event_df = format_event_data(extracted_data)
438
```

Then we find the spanID and timestamps from the Kinesis Event, as we need to get data for those specific spans and timestamps from the Dynamo to update them.

```
439
440     # find unique span-timestamp combos
441     # returned value is a list of dict
442     # [ {'spanId': '...' , 'timeStamp' : '...'},
443     #   {'spanId': '...' , 'timeStamp' : '...'},
444     #   {'spanId': '...' , 'timeStamp' : '...'},
445     # ]
446     unique_span_timestamps = find_unique_span_timestamp_pairs_from_event_df(
447         event_df
448     )
449
450     # get the metrics for these unique span-timestamp combos - get in batch - convert to dataframe
451     metrics_form_dynamo = get_data_from_dynamo_batch(unique_span_timestamps)
452     metrics_from_dynamo_df = pd.DataFrame(metrics_form_dynamo)
453
```

After that we update the metrics for those spans and timestamp

```
456
457     # Update the metrics
458     aggregate_values = update_average_metrics(combined_df)
459     logging.info("Updated metric values are : \n{}".format(aggregate_values))
460
```

Write the updated metrics back to Dynamo DB in batches of 25 items.

```
460
461     # convert to dynamo Put Item type - as we want to do batch write to dynamo
462     dynamo_putItems = convert_to_dynamo_put_item_format(aggregate_values)
463
464     # send to dynamo in batches-batch size is 25 (Max value by Dynamo)
465     for batch in chunks(dynamo_putItems, 25):
466         logging.info("Dynamo batch to write : {}".format(pformat(batch)))
467         response = dynamo_client.batch_write_item(
468             RequestItems={env_vars["MetricDynamoDBTableName"]: batch}
469         )
470         logging.info(
471             "Response from sending data in batch : {}".format(
472                 response["ResponseMetadata"]["HTTPStatusCode"]
473             )
474         )
475     return "Hello"
```

Updating Speed Example

Dynamo Aggregate Table

SpanID	Granular Timestamp	count	speed
123	00:01	6	25

	00:02	6	25.2
	00:03	5	27
	00:04	1	22
124	19:36	1	28
125	16:04	5	24
	16:05	6	24
	16:06	6	24.8
	16:07	6	28
	16:08	6	32

DBStream Event: {

TS: 2019-07-12 16:04:36

SPEED: 123

GPS-Lat: 12.67

GPS-Lng : 8.67

SPANID: 125

EVENTNAME : 'INSERT"

}

Consider the above DynamoDB table, and the DBStream Event. We can see that the spanID is 25 and the timestamp's minute is **04**. So, we first go to dynamo and retrieve the row with spanID 125 and Granular Timestamp as 16:04.

SpanID	Granular Timestamp	count	speed
125	16:04	5	24

Then we do the following calculation:

$$\text{New Speed} = \frac{\text{Count} * \text{speed} + \text{New Event Speed}}{\text{Count} + 1} = \frac{24 * 5 + 123}{5 + 1} = 40.5$$

New Count = Count + 1 = 5 + 1 = 6

We then write these new values for speed and count to our Dynamo Table, which is shown in red

SpanID	Granular Timestamp	count	speed
123	00:01	6	25

	00:02	6	25.2
	00:03	5	27
	00:04	1	22
124	19:36	1	28
125	16:04	6	40.5
	16:05	6	24
	16:06	6	24.8
	16:07	6	28
	16:08	6	32

5.2 - Stationary and Idling Time Aggregation

Implementation

Incoming Data

Green = first invocation for lambda

Red = second invocation for lambda

Black = third invocation for lambda

We seperate the incoming data for simulating out of order data points received from DVR.

S = Stationary, if the speed was zero at that data point

Timestamp	
10:10	
10:11	S
10:12	S
10:13	S
10:14	
10:15	S
10:16	S
10:17	
10:18	
10:19	
10:20	S
10:21	S
10:22	S

Step 1:

Get State Transition table from Dynamo, which will not exist for the device at first. So we create an empty table.

Timestamp	State

Step 2:

The data comes in random order (no data point is ordered). (First Batch)

Insert the data into the table as it comes

Timestamp	State
10:18	
10:19	
10:20	S
10:21	S
10:22	S

Step 3:

Order the data in the table according to the timestamp in ascending order

Timestamp	State
10:18	
10:19	
10:20	S
10:21	S
10:22	S

Step 4:

Remove rows which are consecutive in terms of time and which also have the same state.

Timestamp	State
10:18	
10:19	Consecutive Row + Same state as prev row (Remove This row)
10:20	S
10:21	S <- (delete this)
10:22	S <- (delete this)

Timestamp	State
10:18	
10:20	S

Step 5:

Save this above table in DynamoDB.

Step 1-5 for second invocation (red batch)

Step 1: Get table

Timestamp	State
10:18	
10:20	S

Step 2: Insert Data as it comes

Timestamp	State
10:18	
10:20	S
10:10	
10:11	S
10:12	S
10:13	S
10:14	

Step 3: Order Data according to timestamp

Timestamp	State
10:10	
10:11	S
10:12	S
10:13	S
10:14	
10:18	
10:20	S

Step 4: Remove Consecutive (in terms of timestamp) rows with same state.

Timestamp	State
10:10	
10:11	S
10:14	
10:20	S

Step 5: Save the above table to DynamoDB

Follow Step 1 –5 again for the third invocation to get the following Table after step 4

Timestamp	State
10:10	
10:11	S
10:14	
10:15	S
10:17	
10:20	S

6 - Trip Calculation and metric for trips

Aim: Provide the user with their trips for specific device and relevant metrics/stats for the trip.

In this module we calculate the trips and find the average metrics for those trips.