# Publisher Event Service Enhancements, Automations and Rolling Stone Migration

**A PROJECT REPORT**

*Submitted in partial fulfillment of the requirements for the award of the degrees*

***of***
**BACHELOR OF TECHNOLOGY**
**in**

**Electrical ENGINEERING**

*Submitted by:*
**Kartikeya Surendra Singh**

*Guided by:*
**Dr. Trapti Jain**


**INDIAN INSTITUTE OF TECHNOLOGY INDORE**
**December 2017**

## CANDIDATE'S DECLARATION

We hereby declare that the project entitled "**Publisher Event Service Enhancements, Automations and Rolling Stone Migration"** submitted in partial fulfillment for the award of the degree of Bachelor of Technology in 'Electrical Engineering' completed under the supervision of **Dr. Trapti Jain,** IIT Indore is an authentic work.

Further, I/we declare that I/we have not submitted this work for the award of any other degree elsewhere.

**Signature and name of the student(s) with date**

---

## CERTIFICATE by BTP Guide(s)

It is certified that the above statement made by the students is correct to the best of my/our knowledge.

**Signature of BTP Guide(s) with dates and their designation**

# Preface

This report on "**Publisher Event Service Enhancements, Automations and Rolling Stone Migration**" is prepared under the guidance of Dr. Trapti Jain.

*Through this report we have tried to give a brief summary of the work done in the internship duration, the analysis is mainly on the problem statement of the goals, how they were achieved and the impact on achieving those goals.*

Kartikeya Singh
B.Tech. IV Year
Discipline of Electrical Engineering
IIT Indore

# Publisher Event Service Enhancements, Automations and Rolling Stone migration

*“Research is formalized curiosity. It is poking and prying with a purpose. ”*

# Acknowledgements

I would like to express my gratitude to my advisor Dr. Trapti Jain for invaluable guidance, patience and support

I would also like to thank my mentor at Amazon Ms. Ankita Sawlania. I have learnt a lot from her, in problem-solving. It is amazing how closely she has worked with me on problems, allowing me time to make mistakes and learn from them. I would also like to thank my manager Mr. Sumeet Bansal. He has always been approachable, friendly, responsive and enthusiastic about my work

Finally I would like to express my gratitude to my family, for providing unconditional support to pursue my academic goals. Their support and values, helped me go through some tough times. Their insistence on sincerity and dedication has been instrumental in helping me achieve my goals

**Kartikeya Singh**
B.Tech. IV Year
Discipline of Electrical Engineering
IIT Indore

# Abstract

I worked as an SDE Intern at Amazon Development Centre India, Bangalore. There I was part of the Associate Payments Team. The Amazon Affiliates program is a referral program which awards commission for any sale of Amazon Product by the affiliate store. When website owners and bloggers who are Associates create links and customers click through those links and buy products from Amazon, they earn referral fees. The Associate Payments team at Amazon is responsible for collection and calculation of referral events and two monthly payments to the store. As a part of this team I worked on Publisher Event Service. Publisher Event Service is responsible for calculation of earnings of "Bounty Events" and then storing them in database. A Bounty Event is a sale of an Amazon Exclusive product like Amazon Prime, Amazon Kindle etc. The stored earnings are then processed via Elastic Search and SNS-SQS streams to other teams like AC team which display the earnings to the store. I had to work on several enhancements required by this service. The range of enhancements included new monitors and alarms, new API's, new triggers and some additional logics. I created a self service dashboard to hasten the on-boarding process for this service. I also had to work on automation of manual tasks to improve the efficiency of the present system. As a part of Amazon's Rolling Stone initiative I had to migrate the existing Oracle database to Amazon RDS. For this I got to work on creating the schema, backfilling the data to new tables and reconciliation script for checking the consistency of data between the two databases. I ensure that all my code met the standards of Amazon and went into production.

# Table of Contents

# Abbreviations

**PES: Publisher Event Service**

**RDS: Relational Database Service**

**SQL: Structured Query Language**

**SNS: Simple Notification Service**

**SQS: Simple Queue Service**

*To my parents and grandparents.*

# Chapter 1

# Introduction

## 1.1 Background

I interned at Amazon Development Centre India, Bangalore from 24-May-2017 to 24-Nov-2017. Internships are formal programs within organizations whose primary purpose is to offer practical work experience in a particular occupation to people who are new to that field. These practice internships offer one personal real world insights and exposures to actual working life, an experiential, foundation to their career choices, and the chance to build valuable business networks. Students can serve as a valuable resource for organizations to effectively bridge the gap imposed by the requirements of changing employment needs and fill immediate needs for labor sources. These programs also enable the universities to develop a solid relationship with the industry and pave the way for joint projects of mutual benefits. Amazon is world-renowned for its celebrated University Tech Internship, and the program is known for offering diverse, intensive work experiences in an innovative, fast-paced environment. The selection was on the basis of campus interview. As far as scoring an interview, Amazon looks for many of the same traits that most big tech companies seek: motivated students and grads that are well-versed in the fundamentals of computer science and can think strategically to put together well-thought-out solutions to specific problems. Amazon is known for its unique, strong corporate culture that emphasizes creativity and autonomy. Amazon Interns are treated in the same way as employees minus the guidance of a mentor for any technical help. The Leadership Principles in Amazon are taken very seriously, of them the most important being customer obsession. Customer Obsession ensures that we work vigorously to earn and keep customer trust. All the projects/work done should have a direct impact on customer else the work is meaningless. Going by the same philosophy the work given to me was such that it could be completed completely in the internship duration and have maximum impact on customer. Completion of such work required quick grasping of computer science fundamentals like design pattern, network security and database management, industrial work practices like agile development, scrum, monthly sprints and knowledge transfer sessions, proficiency in languages like Java, C++, SQL and Amazon Web Services architecture. The internship enriched me both in terms of technical knowledge and communications skills required for working in a team.

At Amazon I was part of the Associate Payments Team. Amazon Associates is one of the first online affiliate marketing programs and was launched in 1996. The Amazon Associates program has a more than 12 year track record of developing solutions to help website owners, Web developers, and Amazon sellers make money by advertising millions of new and used products from Amazon.com and its subsidiaries, such as Endless.com and SmallParts.com. When website owners and bloggers who are Associates create links and customers click through those links and buy products from Amazon, they earn referral fees. The Associate program has a number of teams to manage its functioning. Of the teams present the Payments team is responsible for collection of referral events, calculation of earnings of these events and then two-monthly payment to the customers. The collected events are also sent to other teams for analytics and display purpose.

The events received are of three types Orders, Shipments and Bounty. Order is an event of an item being placed for order in Amazon. Shipment is when the order is dispatched. In case of cancelling of order or return of shipment the earnings are deducted accordingly. Bounty is the sale of an Amazon exclusive product like Amazon Prime, Amazon Kindle etc. As it includes a customer in the Amazon ecosystem a direct commission is paid for bounties instead of a percentage based commission. The earnings of these events are calculated on clusters and saved in database. These earnings are paid in a two month cycle. This data is sent to the Frauds team for identifying any malpractices. They are sent to the analytics team to identify the fill-rate/ how likely the customer is to click on ads. They are also sent to the AC team for visual display of earnings in real time to the consumer. As the entire system works in real time any delay or error is taken very seriously as it leads to direct impact on customer whether it is under/over payment to customer or inaccurate display of their earnings.

In this team I mainly worked in the Bounty part of the system. The tasks were assigned to me at the beginning of the month in earnings sprints. Strict deadlines were to be set by me for the tasks assigned. I ensured to meet these deadlines by working hard and planning my schedule efficiently. A mentor was assigned to me for any technical help required. My manager also held weekly one on one to discuss the progress of my work. At the end of every task, the work was expected to be in production completely tested and deployed by me. Code reviews ensured that I maintained the code quality standard of the package. Various internal tools of Amazon like Brazil: management of dependencies, Apollo: deployment of services, Bindles: permission management tools and AWS technologies were available to me from the start itself so that I could work in my full capacity. During my internship I mainly worked on Publisher Event Service.

## 1.2 Motivation of Work

Publisher Event Service is a service used by internal teams in Amazon to send Bounty events for processing of earnings. It performs calculation of earnings and then saves them in the database. The clients were sent back a unique event Id for the inserted events through which they can track that event. There are many clients of the service. The business teams would face complaints from the customers in case their earnings were misreported or completely absent. In those cases it was difficult to pinpoint the source of problem as the service or the clients of the service. A need for monitoring was therefore felt. Monitoring would also enable the teams to find which of the clients were performing well as compared to others. This data could then be used to improve the sales of under-performing clients as well.

Real time reporting of events was done by means of Elastic search technologies. However there was a need of non-real time reporting for analyzing the data without disturbing the present workflow. Amazon SNS-SQS stream, a Pub Sub based model was therefore considered as its solution and would be required to be implemented in the service.

Client on-boarding was done manually. A basic utility with minimal UI design was needed so that this process could be automated. This automation would hasten the on-boarding process from the present 2-3 weeks to just 2-3 days. Thus creation of such a dashboard utility was considered imperative.

Payment Plan is a collection of forms of various types that can be assigned to a store. These forms have rates in them for various product groups like Electronics, Home Appliances etc. The operations team would receive requests to update the rates of these plans from the business team. The Updates were performed by them manually. The chances of human error were very high. A tool was therefore required to automatically update the plans with minimal human involvement.

Rolling Stone initiative is a move away from traditional Oracle database to new and improved databases like Dynamo DB, Amazon RDS, Amazon Red Shift etc. This initiative is a company-wide undertaking and would reportedly save millions of dollars when completed. The execution of this initiative required redesigning of present schemas and approaches, migration of data from the old databases to the new ones and using the advanced features available in the new databases.

# Chapter 2

# Publisher Event Service Enhancements

## 2.1 Workflow

The working of the service is as follows.

1. Client sends a request to the VIP (Virtual IP Address)
2. The VIP is a load balancer which assigns the request to the host with minimum load greedily
3. The Application a Bob Cat Server (modification of the popular Apache TomCat) then maps the operation requested to the operation defined in a special class extending activity.
4. The execution of the requests starts; first all the input parameters are validated.
   The input is in the JSON format and has the following attributes.
   {
   Event Type,
   Event Date,
   Original Event Date,
   Client Id,
   Client Request Id,
   Tag Id,
   Credit Amount Quantity,
   Debit Amount Quantity,
   Event Description,
   Event Data Map,
   }
   The validations performed are :-
   Both Client Id and Event Type should exist, the client should be white listed to send event of this type; Tag Id should not be invalid
   Credit and Debit Quantity should be valid positive integer; Event Date should be greater than previous months first day and less than present date.
5. It then fetches Store Id from the Tag Id; from Store Id it fetches payment Plan for the store. Then in the Payment Plan it searches for the commission to be paid for the event type to be inserted

6. It calculates the earnings as (debit-credit)* perItemCommission for this event
7. It then inserts the data in the database along with earnings and some additional parameters like Host IP Address, creation date and name of the database user.
8. While inserting it checks for uniqueness by the unique constraint on Client Request Id which acts as an idempotency key.
9. It then returns the uniquely generated key back to the client.
10. In case of error in any of the steps an exception is thrown back to the client and then the client retries the request

## 2.2 Monitors

Amazon CloudWatch is a monitoring service for AWS cloud resources and the applications you run on AWS. You can use Amazon CloudWatch to collect and track metrics, collect and monitor log files, set alarms. The following steps were to be performed for making the monitors and alarms.

1. Emission of metrics
2. Making graphs of the emitted metrics.
3. Setting of alarms on those graphs

Metrics are basic counters that count the occurrence of an event. Metrics consist of a Metric Factory. The algorithm to count is simple and is as follows. Ex.

```
Metric metric = MetricFactory.getMetric()
Try {
     metric.startMetric();
     metric.count("Counter_name")
}
Catch (Exception ex) {
    Throw new ex;
} finally {
     metric.finish();
}

}
```

The counters can be many and the metrics can be nested as well. The Metrics factory publishes the metrics in file called service log. The published metrics are then aggregated by Metric Agent. The aggregation are done per min, per 5 min, per 10 min, per day, per week, per month and per year. The Metric Agent has to be set up on the host on which the application is running. It sends the aggregated data to PMET a company wide database which has metrics published by each and every service.

The metrics published to PMET can be viewed on a monitoring dashboard. On this dashboard graphs can be made for the metrics. Following are the options available while making graph from these metrics.

On the X axis following are the parameters that can be set based on time whether per minute , per 5 minute , per hour, per day, per week, per month and per year.

1. Avg: average of the values of the aggregate metrics for the specified time period
2. Sum: sum of the values of the aggregate metrics for the specified time period
3. Tp99.9x: value of the 99.9x percentile reading when arranged in a ascending order
4. N: number of values received in the aggregate metrics.

These can be used for making a variety of monitors

In case of measuring the error rate we can have a counter whose value is 1 in case of error and 0 otherwise. The value of Avg in this case would be thus: $number\ of\ ones/(number\ of\ ones + number\ of\ zeros)$ giving a percentage of error rate when multiplied by 100. N can be used for measuring the traffic received. Tp 99.9x is used to measure latency of the service. This is because in case of latency we are interested only in the poor performers. A distribution is symmetric if each percentile less than the median, and 100 minus that percentile, are equidistant from the median. For example, suppose the median was 2 seconds. Then, say, if the 10th percentile was 0.5 seconds, the 90th percentile should be 2.5 seconds. We'd also have to have that if the 5th percentile was 0.1 seconds, the 95th percentile should be 3.9 seconds. And so on for each other percentile. In cases like these, or when symmetry is approximately true, the **Average or Mean** latency will equal the **Median** response. Than **Standard Deviation** can then be used as a gauge of longer response times.

On Y axis the scale can be set according to the requirements. The unit is default/having no significance on its own. After making the graph monitors can be made on the graphs. The monitors can be set to throw an alarm when the y axis values cross a particular threshold or are below a particular threshold. The value of this threshold can also be set as moving average or based upon the number of times a particular value is crossed. Amazon has a utility to group this monitors together called carnaval.amazon.com. Here the monitors can be grouped together using basic logic gates rule like AND, OR, XOR etc. Complex logic can be made using them to sent an alarm only when the monitors grouped together follow the logic. Ex

$$Alarm\ Rule =$$
$$\big((Traffic < 1\ million)\ AND\ (Latency > 1\ s)\big)\ OR\ (Traffic > 10\ million) AND (Latency > 5s))$$

A ticket or email is sent when Alarm Rule is set to 1 along with the description of the alarm made. Following the entire process monitors were made on the Traffic, Latency and Error rates at an API level and per client level.

Apart from API Level monitors the host can be adjusted to emit metrics of host specific parameters like **Process Count**, **Memory Available**, **Thread count**, **CPU Usage** etc. Following the above procedure monitors were made on them as well. Load Balancer is also an important part of the system whose condition is to be monitored; by default VIP emits metrics in PMET for a range of parameters. Among them the important ones on which monitors were made

**Surge queue Length**: Surge queue length is defined by the number of requests that are queued by Elastic Load Balancing (ELB). These requests are queued when back-end systems are unable to process incoming requests as fast as the requests are being received. Some of the reasons that a load balancer can have a high max statistic for the *SurgeQueueLength* metric include:

1. Overloaded back-end instance(s) – Back-end instance resources—CPU, memory, and network—might be overloaded and unable to adequately process incoming requests.
2. Application dependency issues – Modern web applications can have multiple depencencies on external resources such as databases, S3 buckets, or other applications. If there are performance issues with an application's external dependencies, the application's performance is affected. For example, if an application is dependent on a database table that is not properly indexed, database performance can hinder the application's performance.
3. Max connections reached – Back-end web servers might have reached their maximum allowable connection limit and be unable to process new requests.

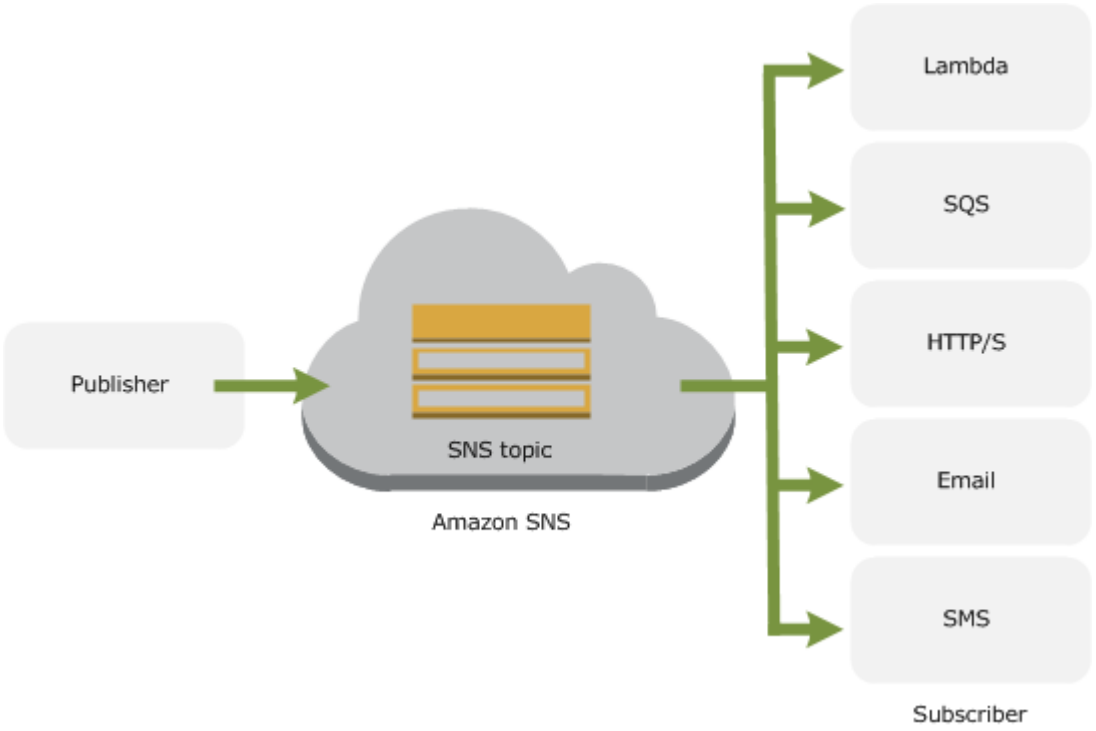
**VIP Spillover:** The total number of requests that were rejected because the surge queue is full.

**Unhealthy Host Count:** The number of unhealthy instances registered with your load balancer. An instance is considered unhealthy after it exceeds the unhealthy threshold configured for health checks. An unhealthy instance is considered healthy again after it meets the healthy threshold configured for health checks.

All the graphs and monitors were made for three regions of operation North America, Europe and Japan. The graphs were put on a wiki page for display for debugging purposes and for use by the business team at Amazon.

## 2.3 SNS-SQS Stream

Amazon Simple Notification Service (Amazon SNS) is a web service that coordinates and manages the delivery or sending of messages to subscribing endpoints or clients. In Amazon SNS, there are two types of clients—publishers and subscribers—also referred to as producers and consumers. Publishers communicate asynchronously with subscribers by producing and sending a message to a topic, which is a logical access point and communication channel. Subscribers (i.e., web servers, email addresses, Amazon SQS queues, AWS Lambda functions) consume or receive the message or notification over one of the supported protocols (i.e., Amazon SQS, HTTP/S, email, SMS, Lambda) when they are subscribed to the topic.




Setting up Amazon SNS has the following steps

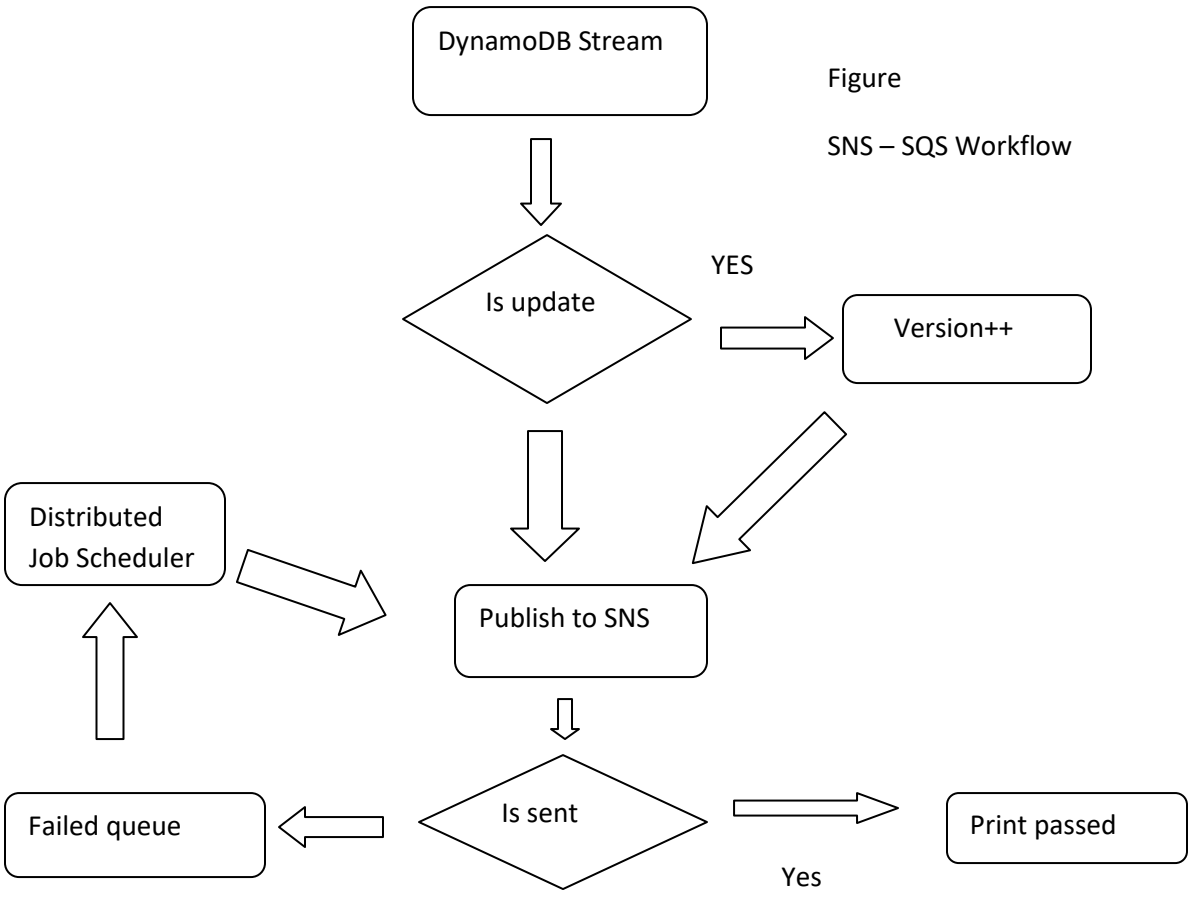
Create a topic and control access to it by defining policies that determine which publishers and subscribers can communicate with the topic. A publisher sends messages to topics that they have created or to topics they have permission to publish to. Instead of including a specific destination address in each message, a publisher sends a message to the topic. Amazon SNS matches the topic to a list of subscribers who have subscribed to that topic, and delivers the message to each of those subscribers. Each topic has a unique name that identifies the Amazon SNS endpoint for publishers to post messages and subscribers to register for notifications. Subscribers receive all messages published to the topics to which they subscribe, and all subscribers to a topic receive the same messages. The Code for this was set up in Publisher FatTireNGStream Processor. The data was fetched from the database using Dynamo DB streams.

DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time.

A DynamoDB stream is an ordered flow of information about changes to items in an Amazon DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attribute(s) of the items that were modified. A stream record contains information about a data modification to a single item in a DynamoDB table. One can configure the stream so that the stream records capture additional information, such as the "before" and "after" images of modified items. The Data from this stream was published to a SNS Topic. In case of failure in sending the data through SNS the data was sent to a failed items SQS queue. Amazon Simple Queue Service (SQS) is a fully managed message queuing service that makes it easy to decouple and scale microservices, distributed systems, and serverless applications. A Distributed Job Scheduler was set to backfill the data from SQS queue to the SNS topic. The job was set to run every day at the specified time. The SQS queues of the other team were the consumer of this SNS topic.

The following was the Algorithm for the SNS-SQS queue.




Figure

SNS – SQS Workflow

Sample code:

Create a Topic

```
//create a new SNS client and set endpoint
AmazonSNSClient snsClient = new AmazonSNSClient(new
ClasspathPropertiesFileCredentialsProvider());
snsClient.setRegion(Region.getRegion(Regions.US_EAST_1));

//create a new SNS topic
CreateTopicRequest createTopicRequest = new CreateTopicRequest("MyNewTopic");
CreateTopicResult createTopicResult = snsClient.createTopic(createTopicRequest);
//print TopicArn
System.out.println(createTopicResult);
//get request id for CreateTopicRequest from SNS metadata
System.out.println("CreateTopicRequest - " +
snsClient.getCachedResponseMetadata(createTopicRequest));
```

Subscribe to Topic

```
//subscribe to an SNS topic
SubscribeRequest subRequest = new SubscribeRequest(topicArn, "email",
"name@example.com");
snsClient.subscribe(subRequest);
//get request id for SubscribeRequest from SNS metadata
System.out.println("SubscribeRequest - " +
snsClient.getCachedResponseMetadata(subRequest));
System.out.println("Check your email and confirm subscription.");
```

Publish to topic

```
//publish to an SNS topic
String msg = "My text published to SNS topic with email endpoint";
PublishRequest publishRequest = new PublishRequest(topicArn, msg);
PublishResult publishResult = snsClient.publish(publishRequest);
//print MessageId of message published to SNS topic
System.out.println("MessageId - " + publishResult.getMessageId());
```

## 2.4 Self Service Dashboard

The process of client on-boarding was a cumbersome process done manually. Initially the manual on-boarding was feasible as new number of clients and new number of event type being registered was small. Today new clients and event types are being on-boarded almost every week. There was therefore a need for a self service tool where the clients can on-board themselves with minimum need of human involvement. In order to make such a dashboard following additional API were required

1. Insert Event Type.
2. Insert Client Id.
3. Update Client Event Map.

Event Type was the type of subscription being sent, client Id was the identity of the sender. Client Event Map was loaded at run time to check for white listing i.e. whether this particular client can send this type of event. During on boarding the client would have to insert the new event type and client Id. Then it would have to request access to use update Client event map to white list itself via a ticket. Amazon has AAA security which controls who can access a particular API. Therefore on adding AAA security to essential API one can safeguard it from use without permission.

After requesting access the client can white-list itself. In Amazon there is are separate network fabric called prod and beta which are exclusive of each other. The actual service runs on prod. Beta is used for testing purposes. To ensure validity of testing the conditions of beta matches that of prod. The client therefore sends a insert request to the service. The first test would be of success. In this case the client should pass his event and receive a success response. In case of failure the input provided by the client would be incorrect. Second test performed is of failure. This is done to test if the retry mechanism is working on the client side. A empty API is made for this purpose which throws a random exception. The client receives the exception and then retries. In Amazon Tier 1 service are those which guarantee 99.99999% uptime. As this is not a tier 1 service the client is supposed to retry after some time. On repeated failures the Client is expected to contact the team. The third test is of idempotency. The client sends a event with same client request id and receive a duplicate client id request received exception. This is done to prevent multiple inserts of the same event. After performing all the three tests the client is on-boarded and given access to send data to prod. All the checks and test are in the UI itself

The UI was written using java script and JSP (java server pages). Java script is a dynamic computer programming language. It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages. Java Server Pages (**JSP**) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems, JSP is similar to PHP and ASP, but it uses the Java programming language. The existing Publisher Event Service is a Coral Service which was extended to include the new API's as well.

For data retrieval and data insertion Hibernate was used. **Hibernate ORM** (Hibernate in short) is an object-relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

Sample Code:

```
    try {
       tx = session.beginTransaction();
       event_type_id = (Integer) session.save(event_type);
       tx.commit();
    } catch (HibernateException e) {
       if (tx!=null) tx.rollback();
       e.printStackTrace();
    } finally {
       session.close();
    }
    return event_type_id;
}

public void insertClientEventIdMap( ){
    Session session = factory.openSession();
    Transaction tx = null;
```

```
    try {
       tx = session.beginTransaction();
       List event_types = session.createQuery("FROM PUBLISHER_EVENT_TYPE").list();
       for (Iterator iterator = event_types.iterator(); iterator.hasNext();){
          Map<String,String> clientEventMap = new HashMap<>();
           clientEventMap[client_id] = event_type;
       }
       tx.commit();
    } catch (HibernateException e) {
       if (tx!=null) tx.rollback();
       e.printStackTrace();
    } finally {
       session.close();
    }
 }
```

## 2.5 Miscellaneous Tasks

One of the tasks involved removing the data time conversion utility in Publisher Event Service. Previously events were sent in local time and were converted to UTC using Joda Date Time utility. The time zone was fetched from the market place Id which was given in input. Clients however now were more comfortable in sending dates in UTC. It was then decided to remove this utility from the service. A bigger task was ensuring the validity of the system after removal. This required collaboration with the other teams. The system was tested end to end in beta starting from client sending event to the service to the display of earning on AC portal.

A new requirement was given to remove events which were older than previous months first day. On further digging it was found that a database trigger was set to do this however its code had a logic flaw which was causing it to give unexpected output. On making corrections the trigger was working correctly.

After deployment of SNS-SQS stream some additional fields and versioning was requested by other teams. Versioning was basically adding a field called version which would be updated whenever the record gets updated. This change required changes in Bounty Entity as well as some database changes.

# Chapter 3

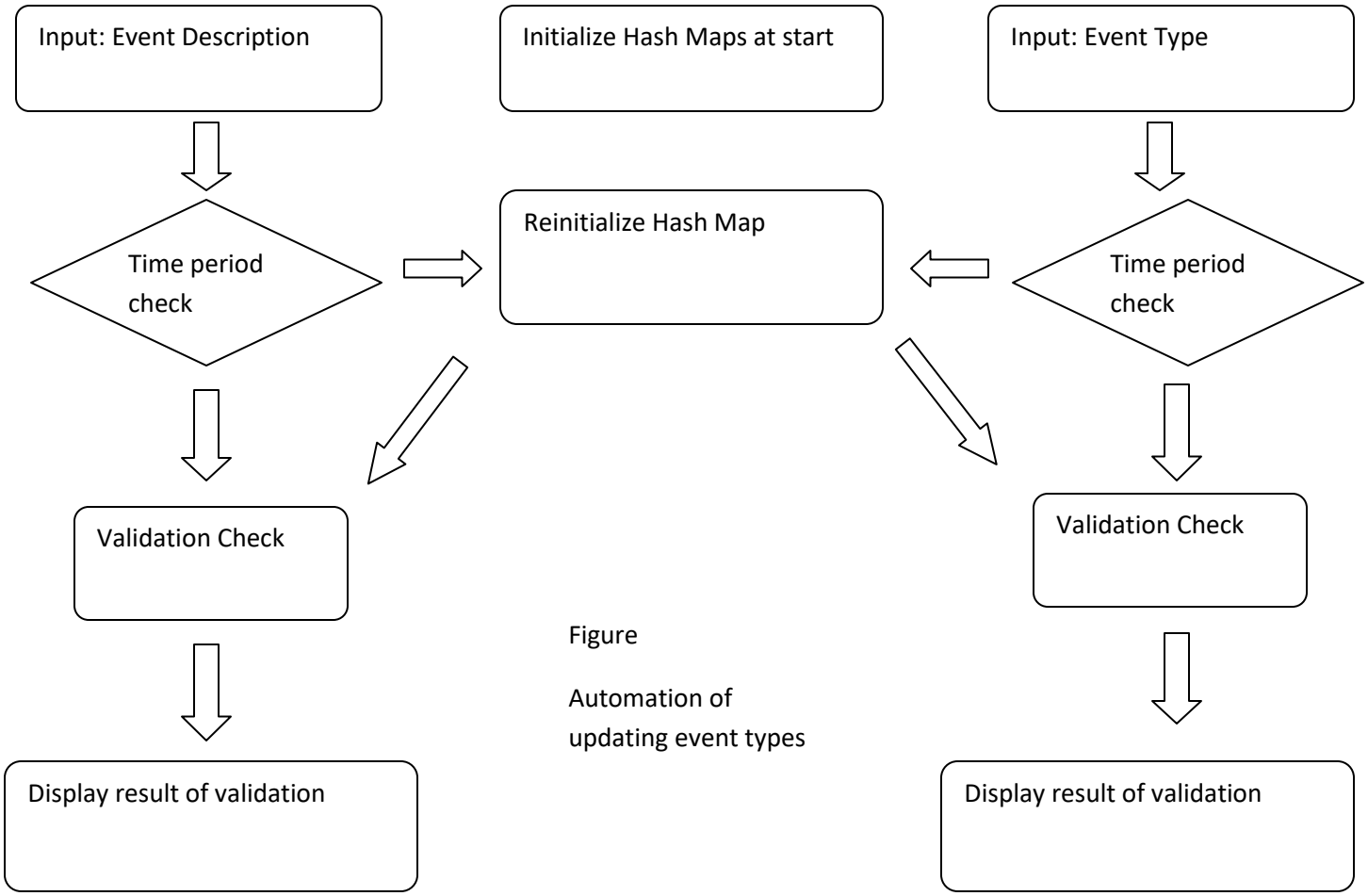
# Automations

## 3.1 Dynamic Loading of Payment Plan

Payment Plans are created by either the operations or business team at Amazon. The Payment Plan has rates for various Product Group and Event Types. Associate Payment Plan Service, a legacy C++ code would validate the existence of the parameters in the payment plan. Event Type would be checked for its existence. For this a map of Event type was hard coded in the C++ code so whenever a new event type was added, code changes would be required as well as a deployment after making the code change. This would take up to 1-2 weeks of time. A better approach was, instead of hard coding the values the map should be generated at run time. This would be done by fetching values from the database and then putting them in a map. The map would have to be periodically updated to check for new values in database. Thus through this the deployment and code review steps could be skipped. As written above an API to insert a new event type in database was already made. This could be used for adding the event type to the database. Thus the final algorithm for this would be as follows.

A Hash Map Event_Type_To_Description <Event_Type, Event_Decription> would be used to map event type to event description and similarly a reverse map Event_Description_To_Type<Event_Description,Event_type> would be initialized at startup. These maps would then be needed to updated at regular intervals of time. Whenever a request is received it would check if it needs to be updated or not. This would be done by means of a timer. If the current time is past timer then the map needs to be updated. The input would be event description. It will first check map 2 to get event type for this event description. This step is necessary for storing in database. If the input it event type it will then instead use map 1 to get event description. It will also need to check for consistency of values in the two maps. In order to fetch the data from database, Amazon equivalent of C++ MySql was used. It would return the data in the form of a linked list. This linked list could then be traversed to make a map out of it.

The final flow diagram would therefore be as follows :-

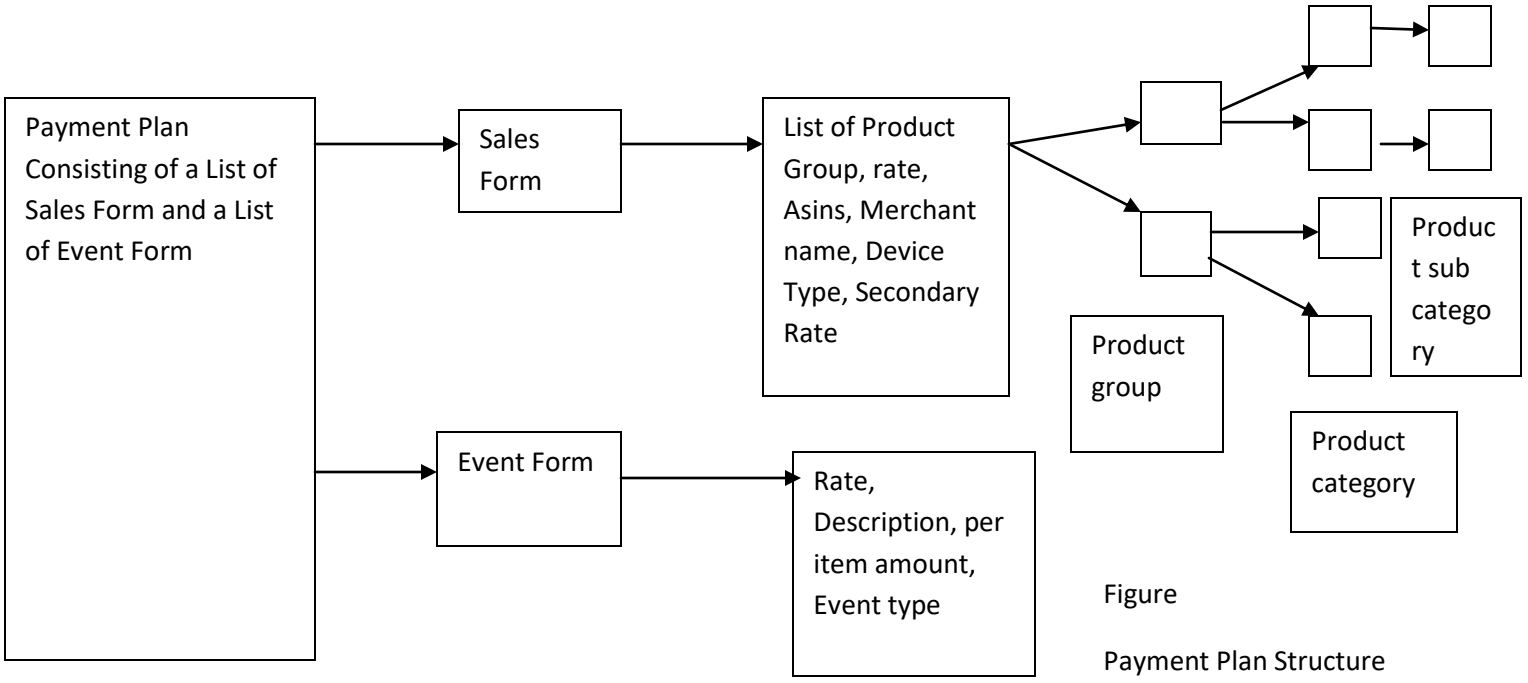



Figure

Automation of updating event types

## 3.2 Automatic Updates

The operations team would receive requests to update the various parameters of the payment plan like rate, GL name etc. These updates were done manually. This was again inefficient and it was decided to make a utility which could perform this work in a generic way. A payment plan consists of sales and event forms. These forms themselves further contained information. This would yield a complex data structure on which updates were to be performed.




Figure

Payment Plan Structure

Updating this complex data structure involved splitting of maps and lot of conditional checks. The following were the type of update utilities that were made

1. Add new Sales Form at x% rate
2. Add new Event Form at x% rate
3. Update all forms with split GL at % rate of original GL
4. Update existing Sales Form with x% rate if rate is less than y%
5. Update existing Event Form with x% rate if rate is less than y%
6. Add new Sales and Event form at x% and y% rate
7. Update existing Sales and Event form at x% rate if less than y%

The code was to be done in a generic way. Spring rule design pattern was used to achieve this

First all the payment plan would be fetched from the database using Get All Payment Plan API. The Payment plan being fetched would only be the ones that are currently assigned to stores. These plans would then be mapped with the store that they correspond to. An initial validation of plan was to be added. This was because some of the currently assigned plans were incorrect. These incorrect plans were logged and corrected manually. After collecting all the plans the desired function based on the input parameters were called. The action rule determined which activity would be called. The input parameters would then be validated and the query would be processed.

The updates required covering of multiple cases. Payment plan forms maintained a strict ordering of masking. Any Sales form list should always follow the following priority

Asin > Merchant > Product Sub Category > Product Category > Product Group

This means that form having Asin always comes first before any form having Asin as null. Validation code was written in order to check for this masking. A corresponding 5 bit number was made for every form with Asin being the Most Significant Bit and Product Group being Least Significant Bit. 1 specified Field present and 0 as field absent. Simple bit comparison would enable us to determine which form is of higher priority. Along with masking check there where many other checks also present. These included whether the rates were valid percentage values, whether there were any duplicate forms present. This validation was done before updating and then after the updated form had been created.

The Update logic had to cover many complex cases. A sales form had a list of product group. Each Product group in turn mapped to many Product Category which in turn mapped to many Product Sub Category leading to the figure shown in the diagram. The update parameters were generic so it could be that only a single product group with some of its product category and product sub category would be updated. This would require splitting of the original form into two forms, the new form only having the product group with only the product category and product sub category to be updated and the old form with only the product group and product category and product sub category which are not to be updated.

It may be possible that all the product category and product sub category may be updated so the old sales form would have to delete that product group. It may also be possible that the form didn’t have any other product group apart from the one that was updated, in this case the entire old form would be deleted. After handling all such cases the new payment plan would be generated, validated then saved in the database.

Another kind of update was split GL update. In some cases a product group was split into two. Ex. Electronics was split into a new GL Home Electronic Appliances. In such cases it is necessary for the customer to be paid the same rate as that of the original GL for the split GL. Split GL update would therefore add the split GL to all the forms having the original GL at the same rate as that of the original GL.

After the new plan was created the new plan would then be mapped to the stores. The old plan would be left as it is. Any plans which failed to get updated would be logged. These plans would then be updated manually.

# Chapter 4

# Rolling Stone Migration

## 4.1 Removing dependency on Tags Table

Publisher Event Service collected Tag data from tables in Oracle. Associate Data Service team had made a new API which would collect Tag Data from RDS instead of Oracle. It was decided to use their API for collecting the data. For this apart from required code changes TPS (Throughput per second) testing would also be required to verify whether the ADS team's API could handle the traffic. A gamma environment was made for TPS testing. The TPS for the service was sufficient in handling the calls. After this the appropriate client connection changes were made to use their API

## 4.2 Generation of Event Id

For all the events inserted a unique alpha-numeric key is generated which is returned to the client. This field is the Event Id. Previously it was generated in the following way

$Event\ Id =$
$(\max numeric\ value\ of\ existing\ Event\ ID + 1)\ numeric\ part\ +$
$(Host\ IP\ hashed\ into\ hexadecimal\ using\ base\ of\ 256)$ Character part

The host hashing was done in the following way

$$xxx.yyy.zzz.www = (xxx) * 256^3 + (yyy) * 256^2 + (zzz) * 256 + www$$

This was then converted to hexadecimal value.

In the new implementation the step of finding the max numeric value of Existing Event Id step would require a full table scan. This was slow and database dependent. In order to make it database independent other numeric key generators were looked upon. The property desired was that the key generators should follow a unique numeric monotonically increasing sequence. Sagan Sequence an internally used key satisfied all the design constraints. It is a Tier 1 service, also used for generating Order Id at Amazon.

The working of the sequence is as follows. After on boarding the Sagan team assigns a private database of long sequence to its user. In this long sequence the user can determine the offset per region. Ex if different sequences are desired for two different regions the sequence of one region could be 10, 20, 30 and the other region could be 11, 21, 31.When the get next sequence is called by user it loads a default number of sequences in the host cache. Whenever a sequence is used it is deleted from the cache. In order to ensure uniqueness the sequence numbers are always removed irrespective if the transaction was a success or failure. As it is a long sequence (10^18) it can last for a very long time.

## 4.3 Creating Amazon RDS Schema

Migration in the context of enterprise and web-based applications means moving from one platform to another. Database Migrations are particularly complicated as you have all the challenges of changing your software platform, where some old features are missing, or behave differently and some new features are available and you'd like to take advantage of those.

Amazon Relational Database Service (Amazon RDS) makes it easy to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backups. It frees you to focus on your applications so you can give them the fast performance, high availability, security and compatibility they need.

Amazon RDS is available on several database instance types - optimized for memory, performance or I/O - and provides you with six familiar database engines to choose from, including Amazon Aurora, PostGreSql, MySql, MariaDB, Oracle, and Microsoft SQL Server.

Of the choices available Aurora PostGresSql was chosen as the new database engine. Amazon Aurora is a MySQL and PostgreSQL compatible relational database built for the cloud, that combines the performance and availability of high-end commercial databases with the simplicity and cost-effectiveness of open source databases. Aurora is up to five times faster than standard MySQL databases and three times faster than standard PostgreSQL databases. It provides the security, availability, and reliability of commercial-grade databases at 1/10th the cost. Aurora is fully managed by Amazon Relational Database Service (RDS), which automates time-consuming administration tasks like hardware provisioning, database setup, patching, and backups.

Aurora features a distributed, fault-tolerant, self-healing storage system that auto-scales up to 64TB per database instance. Aurora delivers high performance and availability with up to 15 low-latency read replicas, point-in-time recovery, continuous backup to Amazon S3, and replication across three Availability Zones.

Of the choice available between PostGreSql and MySql, PostGre was selected. **PostgreSQL**, often simply **Postgres**, is an object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance. As a database server, its primary functions are to store data securely and return that data in response to requests from other software applications. It can handle workloads ranging from small single-machine applications to large Internet-facing applications (or for data warehousing) with many concurrent users; on mac OS Server , PostgreSQL is the default database; and it is also available for Microsoft Windows and Linux (supplied in most distributions).

PostgreSQL is ACID-compliant and transactional. PostgreSQL has updatable views and materialized views, triggers, foreign keys; supports functions and stored procedures, and other expandability. PostGres was chosen over MySQl for the following reasons.

1. It implements the SQL standard very well
2. It includes support for advanced SQL like window functions or common table expressions
3. It is very innovative in terms of how plpgsql interacts with SQL
4. It supports lots of advanced data types, such as (multi-dimensional) arrays, user-defined types, etc.

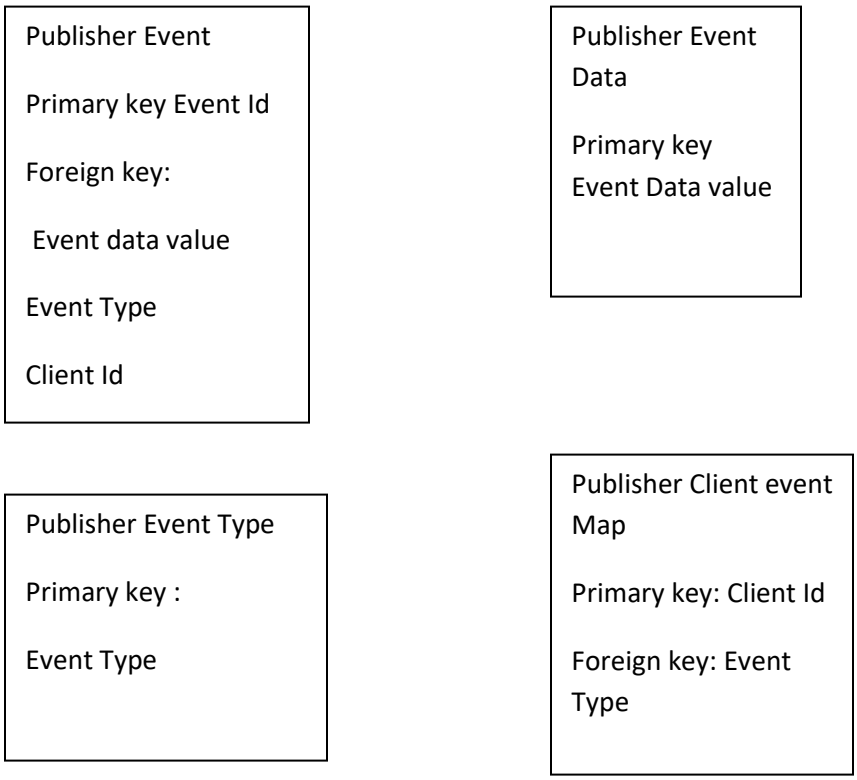
The database creation was followed by creation of read only replicas. These facilitate faster reads and in case of failure in the primary database engine they can be promoted as the primary database engine. Multi A-Z deployment enables one to create replicas in different region than the primary source. Thus in case of failures in a particular region the data would remain safe in the other region.

After creation of the database engine, the database and schema were created.

The created database schema is as follows

Figure

RDS Database Schema




Apart from the tables several constraints were made on the data. Triggers were also added to modify the data on update or deletion to change last updated date and last user who updated.

Sample writes to RDS

Create a connection

```
private static Connection getRemoteConnection() {
    if (System.getProperty("RDS_HOSTNAME") != null) {
      try {
      Class.forName("org.postgresql.Driver");
      String dbName = System.getProperty("RDS_DB_NAME");
      String userName = System.getProperty("RDS_USERNAME");
      String password = System.getProperty("RDS_PASSWORD");
      String hostname = System.getProperty("RDS_HOSTNAME");
      String port = System.getProperty("RDS_PORT");
      String jdbcUrl = "jdbc:postgresql://" + hostname + ":" + port + "/" + dbName
+ "?user=" + userName + "&password=" + password;
      logger.trace("Getting remote connection with connection string from
environment variables.");
      Connection con = DriverManager.getConnection(jdbcUrl);
      logger.info("Remote connection successful.");
```

```
      return con;
    }
    catch (ClassNotFoundException e) { logger.warn(e.toString());}
    catch (SQLException e) { logger.warn(e.toString());}
    }
    return null;
  }
```

Connection to postGres

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.PostgreSQLDialect</property>
    <property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
    <property name="hibernate.connection.username">postgres</property>
    <property name="hibernate.connection.password">password</property>
    <property name="hibernate.connection.url">jdbc:postgresql://localhost:5432/hibernatedb</property>



    <property name="connection_pool_size">1</property>

    <property name="hbm2ddl.auto">create</property>

    <property name="show_sql">true</property>




  </session-factory>
</hibernate-configuration>
```

Writing using hibernate

```
private insertEvent() {
Session session = SessionFactory.getSession();
Try {
     session.save(event_obj);
}
Catch (Exception ex) {
    Throw new ex;
} finally {
     session.close();
}

}
```

It was also necessary to backfill all the data from Oracle to Amazon RDS. For this a utility was written which would take all data from Oracle from a particular date and dump it to Amazon RDS database.

Besides a backfilling script, reconciliation script was also written for validating the consistency of the Oracle and Amazon RDS database. The script would run again as a DJS job. It would scan for each entry in the Oracle database whether the corresponding key is present in RDS database. If the key is not present it will log the key which is absent. After the check is done it generates a list of events which are missing. These events can then be inserted back into the RDS database.

# Chapter 5

# Conclusion

## 5.1 Impact of work

Wrote 20000 lines of code during my internship duration.

All work done is in production effecting nearly 20000 stores having more than 10 lakh insertion of events in a single day.

During my course of internship 15 anomalies were detected by the monitors made by me during the course of internship.

Improved the on boarding time of Publisher Event Service from 2-3 weeks to one day.

Improved the on boarding time of new events from 1-2 weeks to one day.

Rolling Stone initiative when implemented on a company wide level would save millions of dollar and I was grateful to be a part of this initiative.

Moving from Oracle to Amazon RDS reduced latency from around 600-700 ms per query to 100-200 ms per query, at the same time it provided advanced security and features which can be used in further developments.

The future work on this internship include additional monitors and update utilities. Some other service dependent on PES still use Oracle database which needs to be moved to Non-Oracle database. My mentor would carry on the remaining work after the end of my internship.

# Bibliography

*[1] https://aws.amazon.com/ Amazon AWS*

*[2] https://aws.amazon.com/documentation/rds/ Amazon RDS documentation*

*[3] https://aws.amazon.com/documentation/sns/ Amazon SNS documentation*

*[4] https://aws.amazon.com/documentation/sqs/ Amazon SQS documentation*

*[5] https://affiliate-program.amazon.com Amazon Associate Program*

*[6] http://hibernate.org/*

*[7] https://aws.amazon.com/rds/aurora Amazon Aurora*

*[8] https://www.postgresql.org/*