B.TECH PROJECT REPORT

On

Building a Scalable Microservice For Predicting Demand-Supply Mismatch

BY

SHASHANK GIRI, 160001054

DISCIPLINE OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE

November, 2019

Building a Scalable Microservice For Predicting Demand-Supply Mismatch

PROJECT REPORT

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

of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

Submitted by:

SHASHANK GIRI, 160001054

Discipline of Computer Science and Engineering,

Indian Institute of Technology, Indore

Guided by:

Dr. Bodhisatwa Mazumdar,

Assistant Professor,

Computer Science and Engineering,

IIT Indore

INDIAN INSTITUTE OF TECHNOLOGY INDORE

November, 2019

Declaration of Authorship

I, SHASHANK GIRI declare that this thesis titled, "Building a Scalable Microservice For Predicting Demand-Supply Mismatch " and the work presented in it is my own. I confirm that:

- This work was done wholly or mainly while in candidature for the BTP project at Dunzo,Bangalore.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

_____

Date:

_____

Certificate

This is to certify that the thesis entitled, "*Building a Scalable Microservice For Predicting Demand-Supply Mismatch* " and submitted by SHASHANK GIRI ID No 160001054 in partial fulfillment of the requirements of CS 493 B.Tech Project embodies the work done by him under my supervision.

Supervisor

DR. BODHISATWA MAZUMDAR
Assistant Professor
Computer Science And Engineering
Indian Institute of Technology, Indore
Date:

"For only in their dreams can men be truly free. 'Twas always thus, and always thus will be. "

Tom Schulman

INDIAN INSTITUTE OF TECHNOLOGY INDORE

Abstract

Department of Computer Science and Engineering

Bachelor of Technology

Building a Scalable Microservice For Predicting Demand-Supply Mismatch

Dunzo is a fast evolving startup founded back in 2015 with the main aim of becoming the logistic layer of the city. Dunzo started out as a small WhatsApp group, and transformed into a hyperlocal, app-based service currently providing delivery services in Bengaluru, Delhi, Gurgaon, Noida, Pune, Chennai, Mumbai and Hyderabad. The company also operates a bike-taxi service in NCR.

I was a part of allocation sub-team and have developed a service named **Stockout**. This service was used for finding mismatch between demand and supply .One use case was to decide whether a user should be allowed to raise a task or not depending upon whether an optimal assignment of delivery partner to that task is possible at that time. This service was also responsible for calculating and displaying the expected time of delivery if an order is placed.
This system was used extensively by the analytic team to study the factors responsible for a task to be not allowed in the system and also used as a base for implementing dynamic pricing.
This system has became so much critical that any minor abnormalities depicted by the system will have a major impact on the entire company's business.
This report consists of in depth explanation of the tech challenges faced and the research work done in designing such a critical system.

Acknowledgements

I would like to thank my B.Tech Project supervisor **Dr. Bodhisatwa Mazumdar** for his guidance and constant support during the course of this project.Though being an industrial project, his patience and continuous motivation helped me a lot in it's completion.

I am also grateful to **Mr. Kavish Dwivedi** (my mentor in the company) for his constant guidance and technical support throughout the course of the internship. Without him it would be next to impossible to complete this project.

I am really grateful to the Institute for allowing me to do industrial internship for 6 months. It was a very enthralling experience altogether.
Lastly, I offer my sincere thanks to everyone who helped me complete this project, whose name I might have forgotten to mention.

Contents

Chapter 1

Introduction

1.1 Motivation

In an e-commerce market for a company handling approximately thousands of delivery tasks per hour, the major algorithmic problem to solve is optimal assignment of delivery personnel to the tasks. Optimal assignment will have two major impacts - `Cost minimization` and `Time for one Task`.
But in a real world scenario it is not always possible to write the most optimal theoretical algorithm for solving such problems.Thus a sub-optimal solution is used at times.Allocation is a complicated business. And if a task is allowed into the queue, it is not always possible that a partner can be assigned to those tasks.
The issues in might be -

- Some business logic which forbids assignment of certain pairs of tasks and partners.
- All partners are already busy.

It will have two major impacts-

- Deteriorates user experience.
  - The user is less likely to return to the platform.
- Increases queue size of active tasks.
  - New tasks cannot be pushed in the queue if it's size is greater than some threshold.
  - It might be possible that a delivery guy can be assigned to these newer tasks.

Thus a micro-service was required which will behave as a filter and will allow tasks to enter into the queue if that task can be assigned to a delivery partner. Or else just forbid the user to raise a new task.

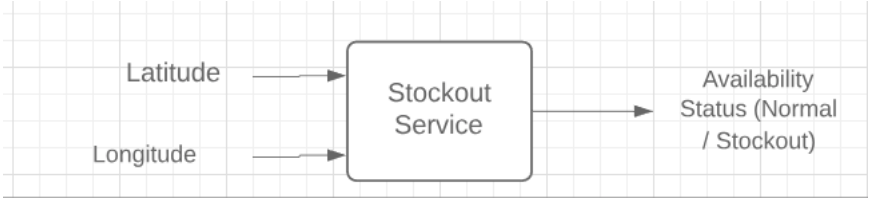

FIGURE 1.1: Architecture

1.2 Objective

Given a task's pickup location

$$\langle lat, lng \rangle$$

Decide depending upon the current distribution of active tasks and delivery partners around this location that whether this task should be allowed in the system or not. If it is allowed good, if not then do not allow the user to complete task creation.

1.3 Product Requirements

1.3.1 Functional Requirements

- The system should expose a Rest Api which accepts `GET Http` requests having parameters `Latitude` and `Longitude`.
- The output should be a `Http Response` with availability status in data.
  (Refer figure 1.1)

1.3.2 Non-Functional Requirements

- Latency should be minimum
- The system should be scalable.
- The downtime of the service should be zero seconds.(If the service is down the company can suffer huge business loss).
- Should support concurrent requests.

Chapter 2

Literature Survey

2.1 Redis Key-Value Storage

Recent beginning of ”NoSQL databases” has brought a lot of attention to key-value databases or stores. This does not however imply that RDBMSes should be avoided. The problem is that they are being used as a generic building block for every problem. Simplified **KVS** brings speed, but also provides very little storage space available.KVS are not suitable for replacing relational databases, but to provide better approach in meeting non-functional requirements especially in cloud architectures.

These databases or stores can not provide ACID style transactions. However, object persistence needs to be consistent nevertheless. This can be achieved with good design patterns that are implemented both in the architectural and component level.

Redis does not support complex queries or indexing, but has support for data structures as values. Being very simple it is also fastest KVS implementations for many basic operations.
The whole dataset is kept in-memory and therefore can not exceed the amount of physical RAM. Redis server writes entire dataset to disk at configurable intervals. This can also be configured so that each modification is always written on the disk before returning “success".

2.1.1 Redis Operations

Every Redis command is executed **atomically**. What do we mean when we say atomic? That even multiple clients issuing the same command against the same key it will never enter into a race condition. Redis supports many data structures for value corresponding to every key. Like Binary-safe strings, Lists, Sets, Sorted Sets, Hashes,etc.

Redis strings

These are used for inserting and accessing string key value pairs. Additionally a ttl can also be defined for each key. After the expiry of this interval the key is erased from the memory.

> set mykey somevalue ex 10
OK
> get mykey
"somevalue"
> ttl mykey
8

Redis Lists

Redis also allows values as a List data type.Redis Lists are implemented with linked lists because for a database system it is crucial to be able to add elements to a very long list in a very fast way.

>rpush mylist A
(integer) 1
> lpush mylist B
(integer) 2
> lrange mylist 0 -1
1) "A"
2) "B"
> lpop mylist
"A"
> rpop mylist
"B"
> brpop mylist 5
<blocking calls waits for 5 sec if operation not possible,else returns rightmost element>

Redis Hashes

Redis hashes look exactly how one might expect a "hash" to look, with field-value pairs.It can be used to store different attributes related to the same key, and then can also provide accessing any of the attribute's value corresponding to a key.

```
    > hmset user username andrew birthyear 2000
OK
   > hget user username
"andrew"
   > hgetall user
1) "username"
2) "andrew"
3) "birthyear"
4) "200"
```

Redis Sorted Sets

Sorted sets are a data type which is similar to a mix between a Set and a Hash. Like sets, sorted sets are composed of unique, non-repeating string elements, however while elements inside sets are not ordered, every element in a sorted set is associated with a floating point value, called the score.

The elements in a sorted sets are taken in order of descending score values.

```
    >zadd key <score> <member>
(integer) 1
   >zadd user 100 "andrew"
(integer) 1
   >zadd user 200 "mathew"
(integer) 2
   >zrange user 0 -1
1) "andrew"
2) "mathew"
   >zrank user "mathew"
(integer) 1
```

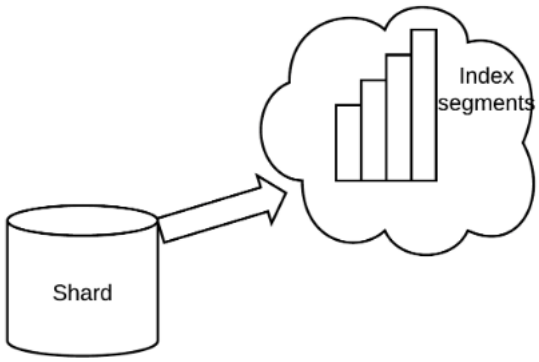

FIGURE 2.1: Lucene Index Segments

2.2 ElasticSearch

Elasticsearch is a distributed, open source search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. Elasticsearch is built on Apache Lucene and was first released in recently by Elasticsearch N.V. (now known as Elastic). Known for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the central component of the Elastic Stack.

2.2.1 Working

Elasticsearch stores data as JSON documents. Raw data flows into Elasticsearch from a variety of sources, including logs, system metrics, and web applications. Data ingestion is the process by which this raw data is parsed, normalized, and enriched before it is indexed in Elasticsearch. Once indexed in Elasticsearch, users can run complex queries against their data and use aggregations to retrieve complex summaries of their data.

The data is stored in the index segments (Refer Fig.2.1). When a new document is inserted a new segment is created internally. Then the elasticsearch might take a decision to merge existing segments into one. While, during document deletion a document is not removed from the segment, but a map is stored which marks that document to be deleted.During queries, the deleted documents are filtered out and not returned as response.
Thus inserting a new document might result in actually decrease in the total size of the index segments. Each of the index segments in made up of these two primary data structures.

- Inverted-index mapping
- Stored-fields

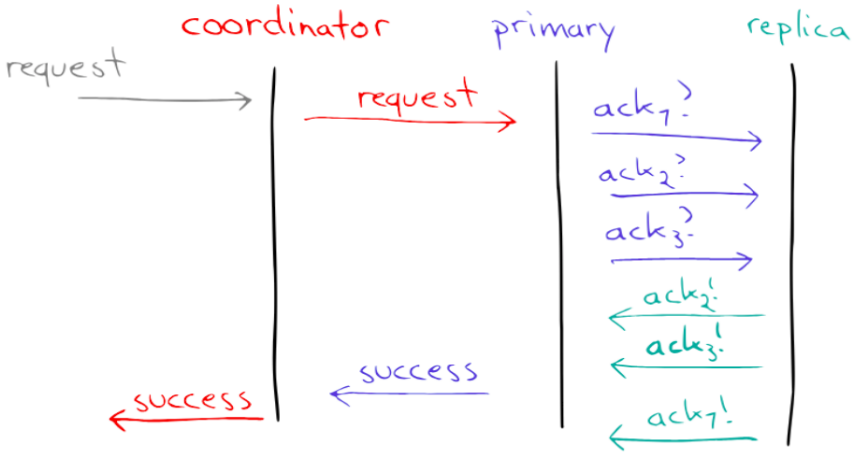

FIGURE 2.2: Indexing a new document.(credits:elastic.co)

2.2.2 Indexing a new document

Elasticsearch provides rich interfaces of rest api for inserting documents.It also created an inverted-index mapping for each of the terms in the json documents. Inverted-index map makes it very fast for performing match queries(That is accessing those documents which have a particular term present in them). Due to this inverted index mapping the queries are executed much faster. This can be used as a basis for designing a search-engine also. e.g. Suppose these documents are indexed into elasticsearch
{
"author": "Douglas Adams",
"Quote": "You live and learn. At any rate,you live"
}
The inverted-index map stored will be as shown in table 2.1.Note that the rows are sorted by term's values.

TABLE 2.1: Inverted-index

TABLE 2.2: key-author

Term	Frequency	Documents
Adams	1	1
Douglas	1	1

TABLE 2.3: key-Quote

Term	Frequency	Documents
and	1	1
any	1	1
at	1	1
learn	1	1
live	2	1
rate	1	1
you	2	1

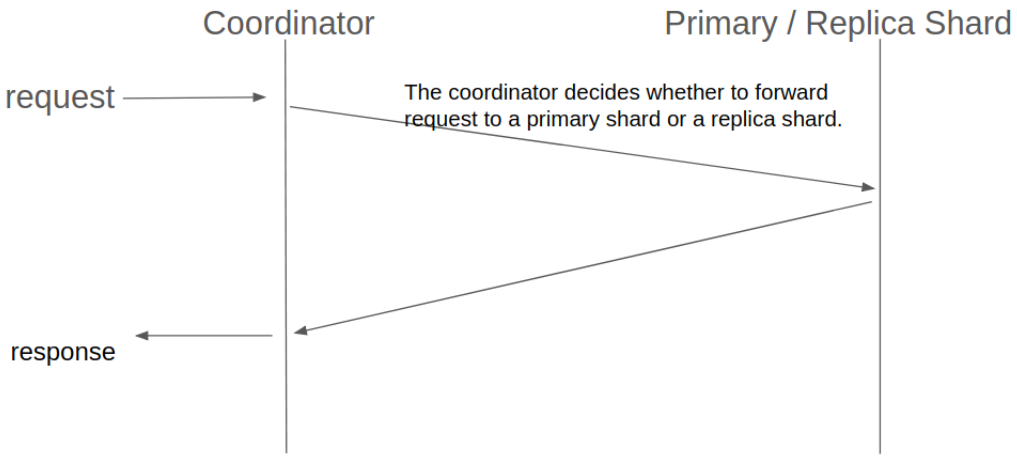

FIGURE 2.3: Query-FLow

2.2.3 Searching index segments

The client making the query makes a get request to the coordinator node with the desired query's json. The coordinator then forwards this request to any of the primary or replica shard. The coordinator node uses a hashing method to decide whether to forward a query to a primary shard or to a secondary shard (Refer Fig2.3. It tries to balance the load on each of the shards as both the shards are having the same data and querying any will return the same result.

e.g

GET <index_name>/_search

{ "query" :

{ "match":

{ "author" : "Douglas"}

}

}

This query will perform a lookup at the inverted-index map stored corresponding to term author and search for "Douglas" is it.It will then return the document numbers in which this term is present.

Chapter 3

Design And Implementation

3.1 System Overview

For a task's pickup location $\langle lat, lon \rangle$. The system will consider a radial area of $\pi * (1.5Km)^2$ around this point and finds-

- The number queued tasks [1] having pickup location within that area (Denoted By **Demand**).
- The count of delivery partners present in that area who can be assigned to this task assuming this is the only task in the system (Denoted by **Supply**).

3.1.1 Request Format

The system exposes a GET Rest Api with endpoint as,

$$/getSupply?latitude = lat\&longitude = long$$

3.1.2 Response Format

If

$$\textbf{Demand} \leq \textbf{Supply},$$

The response returned is

```
{
'availability_status':'Normal'
}
```

else response is

[1] Tasks which are active but not yet assigned to a partner

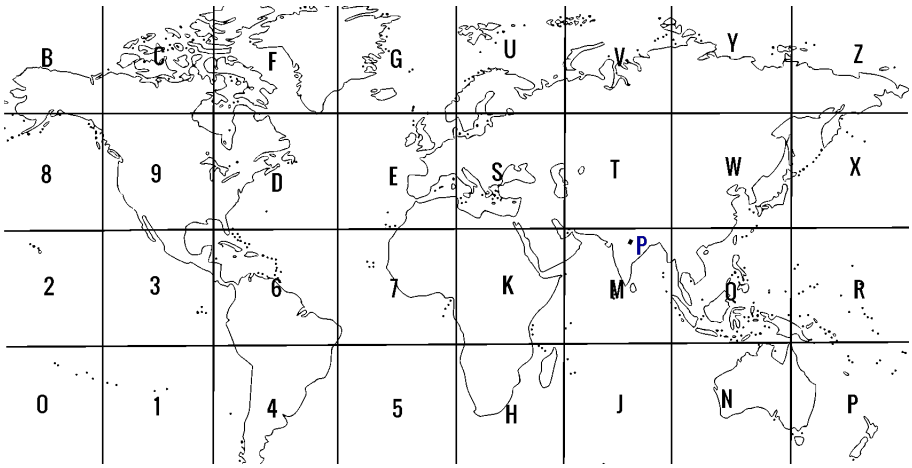

FIGURE 3.1: Geo Hash

```
{
    'availability_status':'Stockout'
}
```

3.2 Calculating Demand

The problem is to find the number of queued tasks around a new task's location in the most efficient possible way.
The information about the queued tasks were stored in a redis cache keyed by geohashes. Redis was decided to be used for storage because the information retrieval time should be as minimum as possible. Redis has data stored in the RAM and hence performing I/O operations corresponding to some key takes comparatively very less time as compared to performing those operations on a disk-based storage.

3.2.1 Geo-Hashes

The Geo-Hashes are the hashes corresponding to a particular geographical bounding box. All the geo-coordinates in that box will be mapped to the same geo-hash. Hence, this method is a very sane approximation for grouping near by geospatial points on the earth's surface.
Figure 3.1 shows the earth map divided into 32 blocks. Hence the hash is stored as a base32 variant.
The Hash of Point P is the letter corresponding to the block in which it is present. For increasing precision recursively sub-divide each block and consider the letter corresponding to those smaller blocks in which point P lies.

Algorithm 1 Encode <lat,lng,precision>

```
const base32 = '0123456789bcdefghjkmnpqrstuvwxyz'
latMin=-90, latMax=90,
lonMin = -180, lonMax=180,
idx=0, count=0, evenbit= false
geohash = ''
while geohash.length < precision do
    if evenBit then
        latMid = (latMax + latMin)/2
        if lat < latMid then
            idx = 2 * idx
            latMax = latMid
        else
            idx = 2 * idx + 1
            latMin = latMid
    else
        lonMid = (lonMax + lonMin)/2
        if lon < lonMid then
            idx = 2 * idx
            lonMax = lonMid
        else
            idx = 2 * idx+1
            lonMin = lonMid
    evenbit = !evenbit
    count ++
    if count is 5 then
        geohash.append(base32.charAt(idx))
        idx = 0, count =0
end
return geohash
```

Following is the pseudo code for finding the geographical bounding box represented by a geohash.

Algorithm 2 Decode <geohash>

```
const base32 = '0123456789bcdefghjkmnpqrstuvwxyz'
latMin=-90, latMax=90,
lonMin = -180, lonMax=180,
evenbit = false
for each character c in the hash do
    idx = base32.indexOf(c)
    for iter = 4 downto 0 do
        parity = idx » iter & 1
        if evenbit then
            latMid = (latMax + latMin) / 2
            if parity then
                latMax = latMid
            else
                latMin = latMid
        else
            lonMid = (lonMax + lonMin) / 2
            if parity then
                lonMax = lonMid
            else
                lonMin = lonMid
        evenbit = !evenbit
return latMin, latMax, lonMin, lonMax
```

3.2.2 Maintaining data sanity

Whenever a new task with a certain pickup location <lat,lon> has been created by a user, just before publishing this information in task queue stream, the geohash of this task's pickup location of precision 6 will be calculated. Refer Algorithm 1, which was designed for calculating the desired geohash of precision(p) of a geo-point. The precision 6 hash approximates a bounding box of about 600 * 500 metres around a geo-coordinate. Suppose the geohash comes to be something like **MSTK75**.

Increment the value in redis stored at key <“MSTK75"> by one.
Redis Provides an incr command for incrementing value of 64 bit signed integers keyed by a string key.
INCRBY <geo_hash> 1
e.g INCRBY “MSTK75" 1
Additionally, when a task is assigned a delivery partner, that is task is no longer in the queue, decrement the value in redis corresponding to the task’s pickup location.
DECRBY <geo_hash> 1
e.g. DECRBY “MSTK75" 1

3.2.3 Algorithm

The Algorithm 3 was used for calculating the value of demand around 1.5 Km radial distance around a task’s pickup location. It takes as input a geo-coordinate (latitude, longitude) and the radial distance(r)in Km which is 1.5 here.

3.3 Calculating Supply

After demand is correctly calculated around a new task’s pickup location, the final thing which remains is finding the count of assignable delivery executives around that location. For delivery executives, the delivery executive need to be free (obviously) but the partners keep changing their GPS locations, as they are always moving. Thus the older idea of storing data in redis, grouped by geo-hashes will not work for delivery executives, as for every update of a delivery executive we have to consider it’s initial state, reset the information inferred from initial state and then make updates corresponding to its newer state.

3.3.1 Maintaining Data Sanity

The delivery executive data is stored in a **real-time data store**. The data store currently used is Firebase. A real-time database is based on publisher-subscriber model. For any change in any field of the data stores, the database takes care of publishing that update to all devices which have subscribed to that field with the updated data. Thus the device can now take necessary actions required at it’s end with this updated newer data. The delivery executives data have already been continuously pushed to runner stream. Runner Stream was a kinesis data stream having two shards. Each shard signifies a horizontal partition of data.

Algorithm 3 FindDemand(lat, lon, radius)

```
redisClient = redis.NewCLient()
H = geo-hash of (lat,lon,6) using alorithm 1
Suppose Q represents a queue.
Q.push(H)
visited = {}
demand = 0
while Q is not empty do
    F = Q.pop
    demand = demand + integer(redisClient.Get(F))
    for each 8 neighbouring hash N around F do
        if N not in visited then
            bounding_box = geohash.decode(N) // Refer algorithm 2
            box_lat = (bounding_box.latMin + bounding_box.latMax) / 2
            box_lon = (bounding_box.lonMin + bounding_box.lonMax) / 2
            haversine = sin^2(box_lat − lat) + cos(box_lat) cos(lat) sin^2(box_lon − lon)
            distance = 2 * 6400 * arcsin √haversine
            if distance ≤ 1.5 then
                Q.push(N)

end
return demand
```

Further an another consumer service was built ,which was responsible for reading the data in this stream by iterating over it's elements. KDS provides a **shard iterator** for iterating over the element stored in them.

This delivery executive metadata was then indexed into **elasticsearch**. The id of the inserted documents was <runner_id>. Hence for each delivery executive there would be only one document present in the elasticsearch segments.
Each updation in elasticsearch is made up of

- deletion
- followed by insertion

A deletion operation does not delete the data directly from the index segments, but rather makes a mark that a particular document is deleted. During insertion of data, the new document is inserted into an another segment. The elasticsearch then might trigger the merge operation. Hence, during this merge operation the marked documents which were deleted can actually be removed from the storage.

The sample runner data indexed into the Elasticsearch from runner consumer was something like

{
“Id”: <runner_id>,
“location”: [<lat>,<lon>],
“tsMillis” : <server_unix_timestamp>,
“runnerstate”: “FREE”, “ON_TASK”,
“cohortAreas” :<list of area ids>,
“partnerType” : < ‘commerce’ or ‘cycle’>
}
e.g.Sample Insertion request
POST <index_name>/_doc/<runner_id>
{
“Id”: <runner_id>,
“location”: [12.73,77.48],
“tsMillis” : 1575235890,
“runnerstate”: “FREE”,
“cohortAreas” : nil,
“partnerType” : ’commerce’
}

3.3.2 Cluster setup

Elasticsearch, like any other data store can be configured to run on a number of primary shards and a fixed number of replicas of each primary shard.
Sharding is a means for horizontally partitioning the data. Thus the entire data to be indexed is divided into buckets. Each of these buckets is what is referred to as a 'shard'. Each primary shard has also a replica shard (which is the exact copy of the primary shard). The reason for increasing the overhead by nearly twice is **Fault Tolerancy** and **Load Balancing** for every query opearation.
The setup configuration currently used for this particular problem had

- 5 primary shards
- 1 replica shard for each primary shards (Thus total 10 shards, out of which half are only replicas)

This is also the default configuration of a elasticsearch cluster.
The entire cluster was deployed on 2-nodes , that is two individual machines.
Elasticsearch provides itself manages the shards and it's replicas. But the problem was to decide the number of such shards depending upon the size of the dataset and it's scalability. Use following for defining the configurations of each index.
PUT <index_name>

```
{
        "settings" : {
            "number_of_shards" : 5,
            "number_of_replicas" : 1
        }
}
```

3.3.3 Query Optimizations

Elasticsearch index segments are made up of a term-document map as shown in table 2.1. The string values are converted into this inverted map. Similarly numeric data types, or tuples are stored as **K-D trees**.
The query used to implement the desired logic was a bit complicated. But the primitive queries used were

- Match Query on a term
- Geo-distance filter

- Range Filter

Then it required a aggregation of the results returned by these primitive queries for finding the final answer.
The geo-distance filter on location field.(Rember location field's data is stored as a K-D tree , hence finding neighbouring documents are much faster). Also the result of the filter is cached as a **BITMAP**. Hence, successive queries with same parameters will return results in lesser time as compared to the initial first query time.
GET <index_name>/_search

```
{
   “query": {
      “filter" : {
         “geo_distance" : {
            “distance" : “1.5km",
            “location" : [<lon>, <lat>]
         }
      }
   }
}
```

Match query on “runnerstate" field to consider only those documents in which the value of this field is “FREE".
GET <index_name>/_search

```
{
   "query": {
      "match" : {
       "runnerstate" : “FREE"
      }
   }
}
```

Chapter 4

Results and Conclusion

The system has two major objectives

- Forbid user to complete task creation if supply is not enough.
- Develop a metric for finding demand-supply mismatch to implement dynamic delivery pricing.

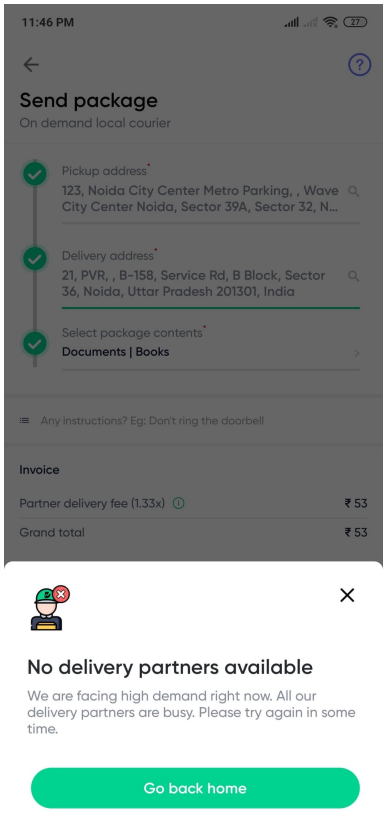

FIGURE 4.1: App Blocker

The System was deployed in production and it actually began experiencing traffic.Depending upon the response of this service the app shows a blocker just before checkout page and forbids user to continue creation of a new task (Refer Figure 4.1).
The service was able to reduce the user cancellation rate by about 20 percent.Not only

this, the service was also able to appropriately keep track of number of queued tasks in a certain area and the distribution of delivery executives around that area. Thus it was also helpful in analysing the peak time (w.r.t. the number of orders in a certain area).

There are some improvements still possible in the system. Currently the radius of 1.5Km is constant and used in every situation. The idea was to make radius configurable depending upon time and geographical areas.

- During high peak time (when the number of tasks are more than some threshold) the radius can be increased a bit to say about 2-2.5 Km.
- Similarly, during low peak time the radius can be decreased a bit.

Bibliography

[1] *Matti Paksula*, “Persisting Objects in Key-Value Datastores"

[2] *Hang Li, Wanlong Li, Guochun Wang, Xinyi Peng*, “Information Retrieval Services Based on Lucene Architecture"

[3] *Yong Zhang and Jian-lin Li*, “Research and Improvement of Search Engine Based on Lucene"

[4] *Alex Brasetvik*, “Elasticsearch a top-down view" (“https://www.elastic.co/blog/found-elasticsearch-top-down#a-cluster-of-nodes")

[5] *Alex Brasetvik*, “Elasticsearch from bottom-up" (“https://www.elastic.co/blog/found-elasticsearch-from-the-bottom-up/")

[6] *Alex Brasetvik*, “Optimizing Elasticsearch Queries" (”https://www.elastic.co/blog/found-optimizing-elasticsearch-searches/")