Answer1:

```
import requests
import sys

def main(url_parameter):

   if url_parameter:
       url = url_parameter
   else:
       url = "https://s3.ap-south-1.amazonaws.com/haptikinterview/chats.txt"

   data = {}
   file = requests.get(url)
   chat_text_list = file.text.split("\n")

   for line in chat_text_list:
       if line:
           try:
               start = line.index("<") + len("<")
               end = line.index(">",start)
               name = line[start:end]
           except:
               name = ""
           wordcount = len(line.split())-1
           if name in data:
               data[name][0] += 1
               data[name][1] += wordcount
           else:
               data[name] = [1, wordcount]

   first = ["", 0]
   second = ["", 0]
   third = ["", 0]

   for name in data:
       score = data[name][0]*data[name][0] + data[name][1]
       if score >= first[1]:
           third = second
           second = first
           first = [name, score]
       elif first[1] > score >= second[1]:
           third = second
           second = [name, score]
       elif score >= 3:
           third = [name, score]
```

```
    print([first[0], second[0], third[0]])


if __name__ == "__main__":
    if len(sys.argv) > 1:
        main(int(sys.argv[1]))
    else:
        main("")
```

Answer 2:

ESTIMATION:

- Twitter does around 500 million tweets per day with 100 million daily active users.Before making selection of tech stack Let's assume similar numbers which our platform will be getting.

- The behavior will be similar to Twitter here. Each user has on average 200 followers, with certain hot users having a lot more followers. For example, users like Justin Bieber would have millions of followers.

- Our platform will have million active users with 200 followers on average. This means 100M x 200 edges = 20B edges

Design Goals:

Latency: A twitter like system needs to be fast, especially when you are competing with Twitter.

Consistency: Consistency is not that much important in, we can bear if I a person lose one or two tweets because of so much activity on the platform.

Available: It needs to be highly available.

API Skeleton:

The first 2 operations end up doing a write to the database. The last operation does a read. Following is an example of how the API might look like :

Posting new tweets : addTweet(userId, tweetContent, timestamp)

Following a user : followUser(userId, toFollowUserId)

TweetResult getUserFeed(user, pageNumber, pageSize, lastUpdatedTimestamp)

where TweetResult has the following fields :

TweetResult - {List(Tweets) tweets, boolean isDeltaUpdate}

Tweet - { userId, content, timestamp }

Application Layer:

Fault tolerance: If we have only one application server machine, our whole service would become unavailable. Machines will fail and so will network. So, we need to plan for those events. Multiple application server machines along with load balancer is the way to go.

Ans 2.2 Database Layer:

Things to consider: Are joins required? Size of the db, technology maturity

I will chose SQL here is instead of nosql.

Database Schema:

1. **Users:** id(pk), username(unique key), first_name, last_name, profile_description, password_hash, password_salt, created_at, last_updated_at, dob

2. **Dob:** id, content, created_at, user_id(fk)
3. **Connections(User A follows another user B) :** follower_id, followee_id, created_at

Now, based on the read API queries, we can look at the additional index we would need :

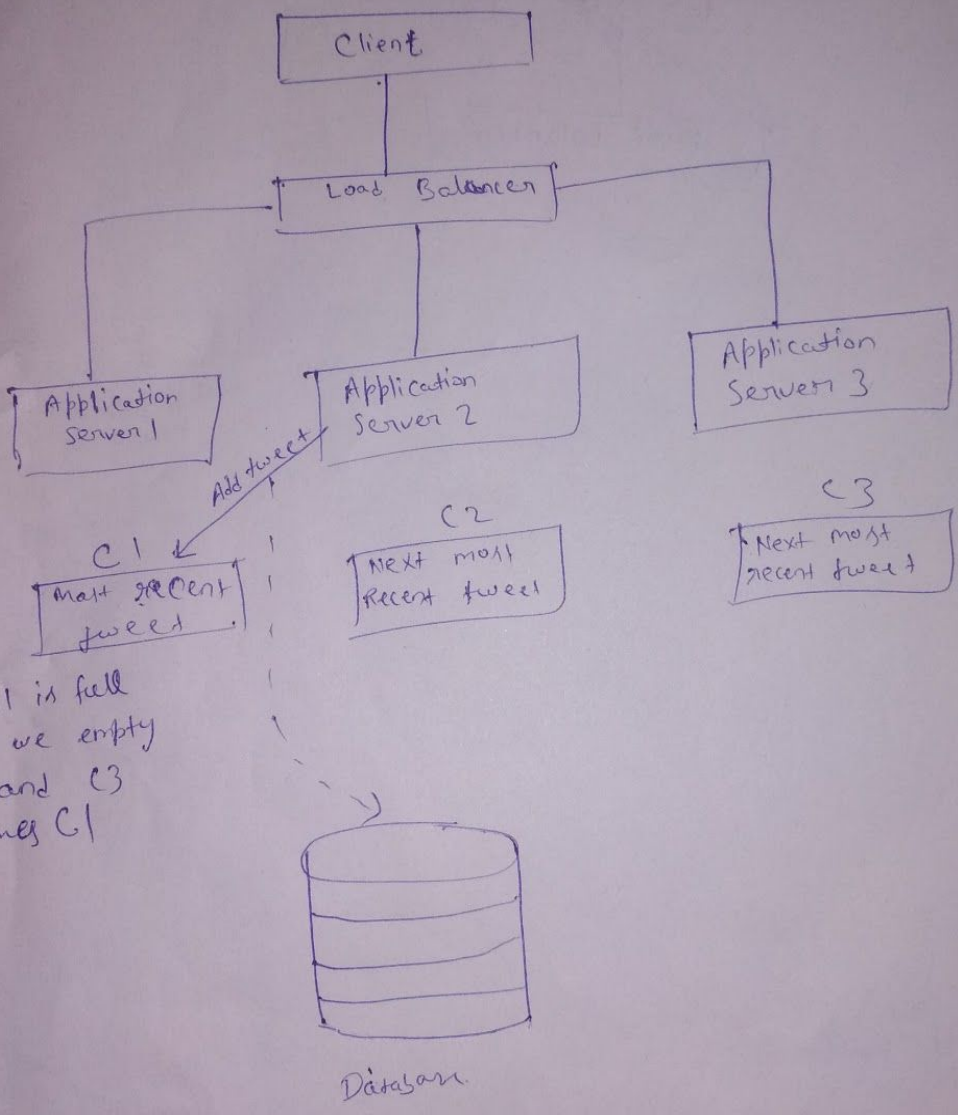
Get the feed of a user - This would require us to quickly lookup the userIds a user follows, get their top tweets and for each tweet get users who have favorited the tweet.

This means we need to following index :

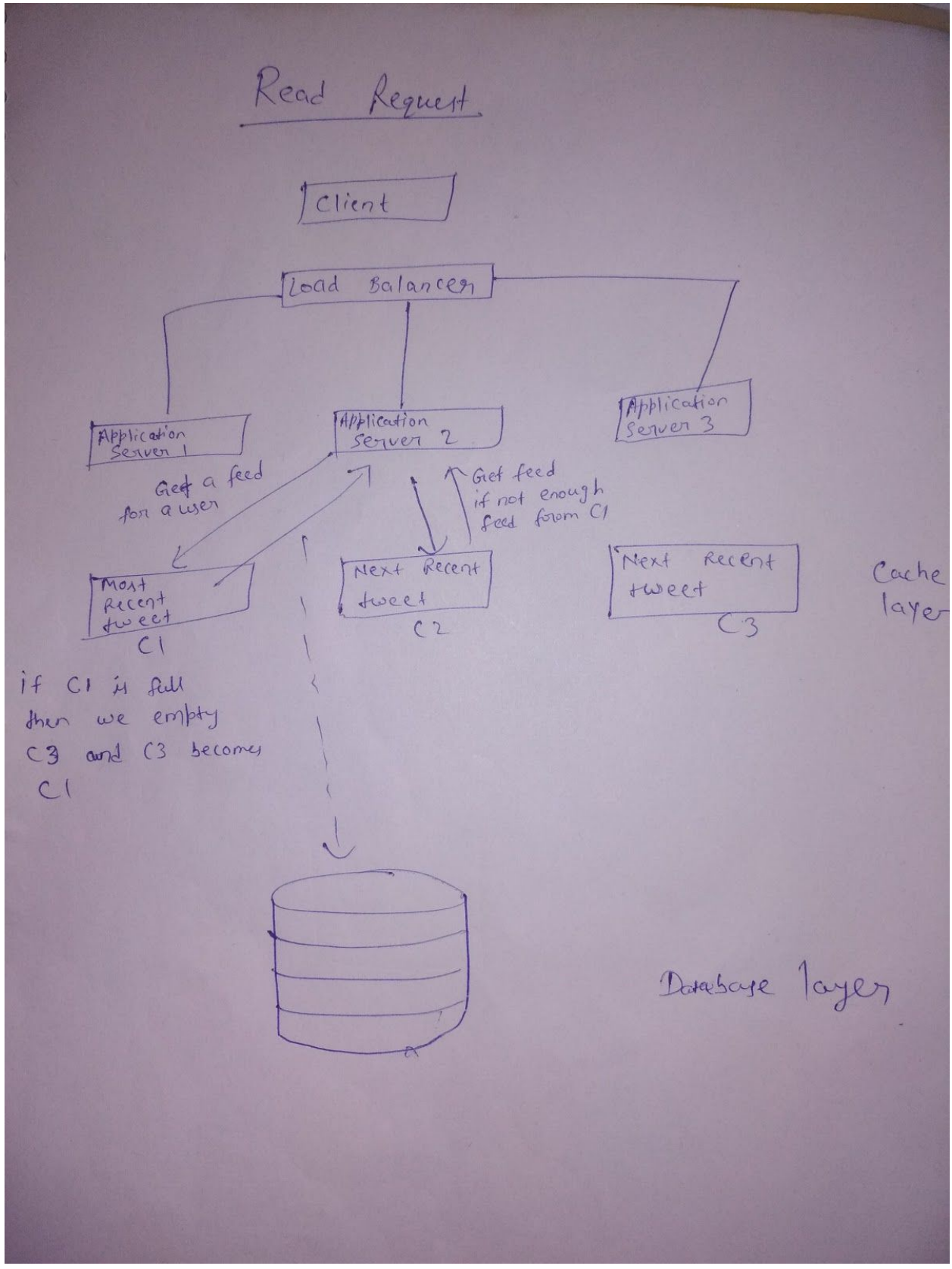
- An index on follower_id in connections to quickly lookup the userIds a user follows
- An index on user_id, created_at in tweets to get the top tweets for a user (where user_id = x sort by created_at desc)

Write Request

Write Request



Read Request



Ans 2.4 Breaking Point, When system will start failing:

Lets look at both cases one by one.
Let's say Shahrukh Khan tweets. Following is what happens :

We write the tweet to the table. Not a problem.
We add it to the recent tweets cache. Again, not a problem.

As all followers get their feed update by specifically requesting for an update, the resultant change is that a lot of followees will get an update when they request for it. This should manifest as an uptick in the upload bandwidth. In the worst case, assuming that 30% of the followers are online at a time.

Lets look at really popular tweets now. They'll have an unusually high rate of being favorited (The highest being 3M total favorites). This means a really high rate of write to the database which can cause deadlocks.

We can add some optimizations here if required in terms of batching the updates to favorites table in a queue before flushing them.

Because of deadlock scenario system will become unresponsive and break.

High Availability:

As stated in design goals, we need to make sure our system is more available at all times.
Solution:

Database sharding - database partitioning that separates very large databases the into smaller, faster, more easily managed parts called data shards.

Master slave database structure - When the master goes down, the slave can take over. Now there is a problem here. What if there were some updates which the slave had not caught up to yet. Do we lose that information? We can take a call either way. If we are particular about getting the data back, we know that we can get that information from the cache layer and resolve stuff on the DB layer.

Ans 2.1 Tech Stack(to start, new tools can be used as traffic grows) :

Languages and framework - Python(server side), django(web application), C++(for alogrithms)

Front end - Web techonlogies like html, css, bootstrap with any Javascript mvc framework

Caching - memcache or redis

Database - mysql or postgres

Tools - Webpack, celery, unittest, elasticsearch, rabbitmq, logstash, Kibana

Microservices based architecture

Blobstore - For storing images and videos

Cloud - AWS or azure with load balancers and auto scaling, along with multiple replica of databases.

Ans 2.3 Function that will return all the tweets to show on the dashboard of a particular user.

```
from .serializers import TweetModelSerializer
from .pagination import StandardResultsPagination
from tweets.models import Tweet

class TweetListAPIView(generics.ListAPIView):
   serializer_class = TweetModelSerializer
   pagination_class = StandardResultsPagination

   def get_serializer_context(self, *args, **kwargs):
       context = super(TweetListAPIView, self).get_serializer_context(*args,
**kwargs)
       context['request'] = self.request
       return context

   def get_queryset(self, *args, **kwargs):
       requested_user = self.kwargs.get("username")
```

```
        if requested_user:
            qs =
Tweet.objects.filter(user__username=requested_user).order_by("-timestamp")
        else:
            im_following = self.request.user.profile.get_following() # none
            qs1 = Tweet.objects.filter(user__in=im_following)
            qs2 = Tweet.objects.filter(user=self.request.user)
            qs = (qs1 | qs2).distinct().order_by("-timestamp")

        query = self.request.GET.get("q", None)
        if query is not None:
            qs = qs.filter(
                    Q(content__icontains=query) |
                    Q(user__username__icontains=query)
                    )
        return qs
```

Ans 2.5 : Here is the sample app which I have worked on. You can see models, api, and some front end code here. https://github.com/parvez301/Twitter-clone-with-django

Answer 3:

can_create.py

```
def check_if_word_is_formed(others, inputLen, nextStart):
  """ This function is recursive and eventually returns
  True or False depending on whether the word is formed or not
  """
  for r in others:
     if r[0] == nextStart:
        if inputLen == r[1]:
           return True
        else:
           others.remove(r)
           nextStart = r[1]
           return check_if_word_is_formed(others, inputLen, nextStart+1)
  return False

def can_create(listOfStrings, input):
  """ This function is detects the first word and
```

```
    uses check_id_word_is_formed function to
    determine boolean output for can_create
    """
    possibleStarts = []
    others = []
    inputLen = len(input)-1
    if inputLen != -1:
        for word in listOfStrings:
            i = input.find(word)
            if i >= 0:
                endIndex = i+len(word)-1
                if i != 0:
                    others.append([i, endIndex])
                else:
                    possibleStarts = [i, endIndex]
        nextStart = possibleStarts[1]+1
        if others:
            if check_if_word_is_formed(others, inputLen, nextStart):
                return True
            return False
        elif possibleStarts:
            return True
        else:
            return False
    else:
        return False


def main():
    listOfDataStrings = ['back', 'end', 'front', 'tree']
    inputToCheck = 'backend'
    print(can_create(listOfDataStrings, inputToCheck))

if __name__ == '__main__':
        main()
```

can_create_test.py

```
import unittest
from can_create import can_create, check_if_word_is_formed

class TestCanCreate(unittest.TestCase):
    """
    basic unit tests for cancreate.py
    """
    def test_can_create(self):
        """
```

```
        unit tests for can_create function
        """
        dataStrings = ['back', 'end', 'front', 'tree', 'sparta']
        stringToCheck = 'back'
        actual = can_create(dataStrings, stringToCheck)
        self.assertTrue(actual)

        dataStrings = ['back', 'end', 'front', 'tree', 'sparta']
        stringToCheck = 'backend'
        actual = can_create(dataStrings, stringToCheck)
        self.assertTrue(actual)

        dataStrings = ['back', 'end', 'front', 'tree', 'sparta']
        stringToCheck = 'backendtree'
        actual = can_create(dataStrings, stringToCheck)
        self.assertTrue(actual)

        dataStrings = ['back', 'end', 'front', 'tree', 'sparta']
        stringToCheck = 'spartaback'
        actual = can_create(dataStrings, stringToCheck)
        self.assertTrue(actual)

        dataStrings = ['back', 'end', 'front', 'tree', 'sparta']
        stringToCheck = ''
        actual = can_create(dataStrings, stringToCheck)
        self.assertFalse(actual)

    def test_check_if_word_is_formed(self):
        """
        unit tests for check_if_word_is_formed function
        """
        others = [[4, 6]]
        inputLen = 6
        nextStart = 4
        actual = check_if_word_is_formed(others, inputLen, nextStart)
        self.assertTrue(actual)

        others = [[10, 14], [0, 9]]
        inputLen = 14
        nextStart = 0
        actual = check_if_word_is_formed(others, inputLen, nextStart)
        self.assertTrue(actual)

if __name__ == '__main__':
    unittest.main()
```