Mockstagram crawler documentation

Table of Contents

Understanding problem statement

https://www.notion.so/V2-Affable-Data-Engineering-Task-5d4bec24edbf42ebbe2a285caa699b26

1) Design crawler for Mockstagram for tracking time series data of 1 million accounts.

Assumption:

- Mockstagram endpoint has no crawling protection and impossible to DDos.

Comment:

- Requires use of efficient, distributed time series database for data with high cardinality.

- Requires distributed crawler design to distribute load across multiple servers

- Strong consistency is not a priority (I assume if followers are in ranges of 100k+, daily changes will be relatively insignificant to 1) affect the performance of machine learning models used, 2) affect the decision making of the data users)

2) Provide some easily accessible aggregated metrics (eg. averageFollowerCount, most recent data) for many users

Assumption:

- The aggregated metrics are globally aggregated (over the entire time series) rather than sliding window (moving averages).

Comment:

- The averageSomeMetrics can be downsampled without too much implication on accuracy while increasing performance speed greatly (eg. Of 1,000 time series data points, sample every 100th interval for calculation)

- Recommended to have database for storing these pre-computed metrics

- Provide a distributed cache with multiple read-only slaves storing these aggregated values.

3) Design feature for updating “suspicious” status, which is computationally heavy

Assumption:

- Server for machine learning service cannot be modified

Comment:

- Updates are not time critical (once daily)

- If results can be cached, we will use a cache store for such information (eg. Same input that has been calculated before)

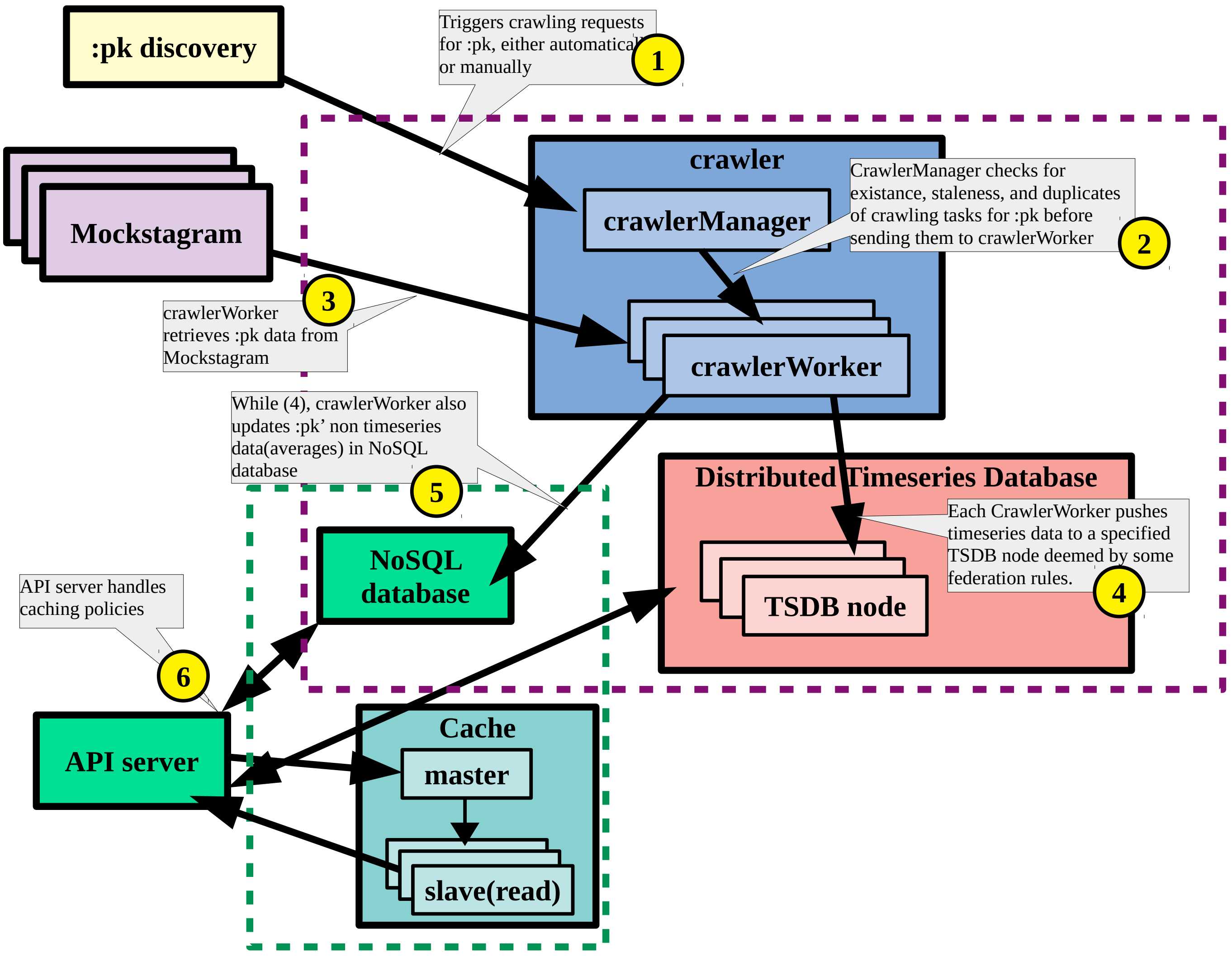
- If results cannot be cached (eg. Function is always changing), then need a buffer to queue these requests as the consumer (‘/api/v1/influencers/is_suspicious’) is much slower than the producer (it is much easier to send 1000 requests to the consumer than to receive the same amount of replies in a short time)

Proposed Solution

(Color of dotted boxes matches the problem it is solving)

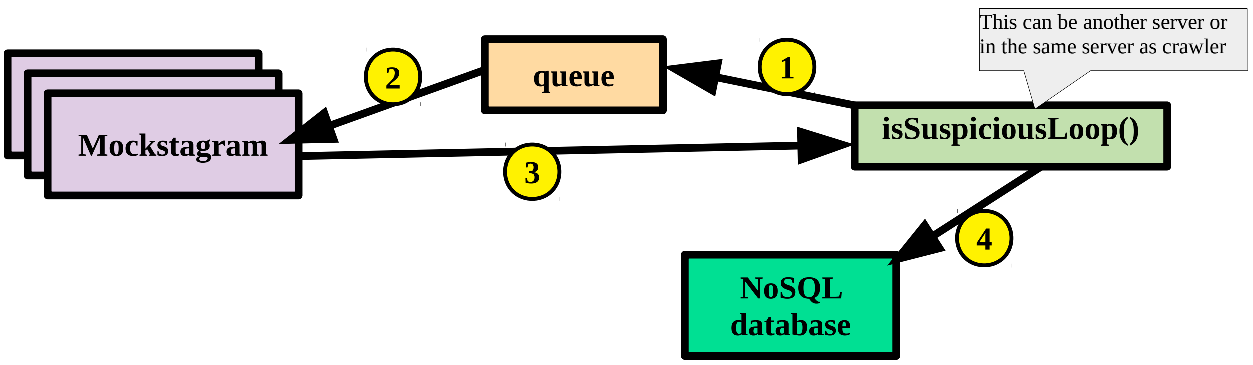
Purple box: Solution for (1), Design crawler for Mockstagram with 1 million accounts to track.

Green box: Solution for (2), Provide some easily accessible metrics (eg. averageFollowerCount, most recent data) for many users



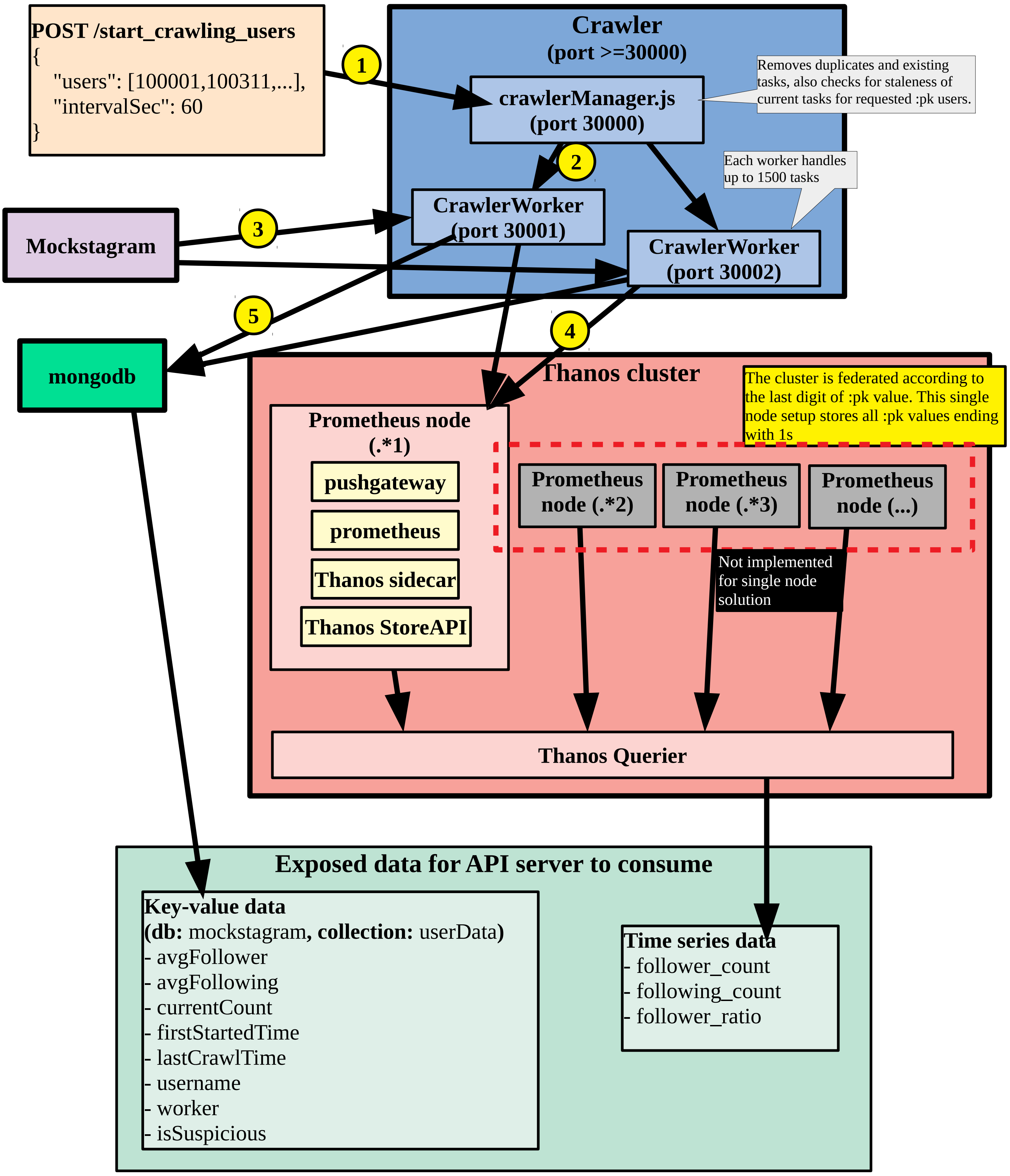
:pk discovery
Triggers crawling requests for :pk, either automatically or manually
1
crawler
crawlerManager
CrawlerManager checks for existance, staleness, and duplicates of crawling tasks for :pk before sending them to crawlerWorker
2
Mockstagram
3
crawlerWorker retrieves :pk data from Mockstagram
crawlerWorker
While (4), crawlerWorker also updates :pk' non timeseries data(averages) in NoSQL database
5
Distributed Timeseries Database
Each CrawlerWorker pushes timeseries data to a specified TSDB node deemed by some federation rules.
4
TSDB node
NoSQL database
API server handles caching policies
6
API server
Cache
master
slave(read)

For (3), Design feature for updating "suspicious" status, which is computationally heavy



Implementation design (single node)

Due to time constraints, and a desirably simplified solution for a single node setup, the implementation design is a watered down version of the proposed solution.



Choice of platform for implementation design

Distributed crawler:

NodeJS is used for the backend here because of convenience and the ease of handling asynchronous events.

Distributed Timeseries database (TSDB):

This is a difficult choice because of a few reasons.

1) Key-value store vs TSDB:

Key-value NoSQL databases are widely used and hence there is a greater variety of choices and community support for production use. However, they are not optimized for time series data hence may not perform better than TSDB for the problem statement.

TSDB, on the other hand, are only handful in terms of variety (Opentsdb, InfluxDB, TimescaledDB, Promethus/Thanos). But some of them have been used for production with good community support hence these choices are definitely preferred over a key-value database.

User reviews:

"Cassandra/MongoDB (NoSQL) has been disqualified, since it is much slower than Timescale(TSDB)" [5]

2) Choice of TSDB

This is an even more difficult choice to make. Between the various TSDBs, each of them have their pros and cons in a way that doesn't make any of them stood out as an obvious choice (although there are easy rejects).

The pros/cons are summarized briefly below:

Opentsdb:

Pros:

- Based on Hadoop and Hbase

Cons: ?

InfluxDB:

Pros:

- Developed from PostgresDB

Cons:

- Expensive distributed solution

- Performance issues

- “Influxdb suck for collecting large point in time metrics. Also the memory it uses is huge.”[1]
- “If you want clustering for HA or for horizontal scaling, you need the enterprise version of InfluxDB. “[1]
- “InfluxDB performance dropped ... This is significant performance loss comparing to other competitors(VictoriaMetrics, TimescaleDB)” [6]
- “InfluxDB: didn’t finish because it required more than 60GB of RAM.”[6]

TimescaledDB:

Pros:

- Developed from PostgresDB

Cons:

- Inefficient storage, “Timescale data occupies **whopping 29GB** on HDD. That’s 50x more than InfluxDB and 75x more than VictoriaMetrics”[5]

Prometheus/Thanos:

Thanos as a platform to manage Prometheus clusters.

Pros:

- More scalable than some other platforms,

  - “Yea, the Prometheus + Thanos combo for larger deployments is crazy awesome. It allows for nearly infinite storage, while keeping the deployment simple and robust against failure.”[1]
  - “Prometheus open source is more scalable than influxdb”[1]

Cons:

- Not a conventional choice for TSDB, mainly for monitoring

- No strong consistency, data stored may not be accurate.

- Higher number of moving parts (more complex to setup)

- By nature, ingests data by ‘pulling’. But a *pushgateway* can be used to change the ingestion to a ‘push’ one.

- Backfilling is a pain to implement

Prometheus/VictoriaMetrics

VictoriaMetrics as a platform to manage Prometheus clusters.

Pros:

- Better query speed, “VictoriaMetrics wins InfluxDB and Timescale in all the queries by a margin of up to 20x. It especially excels at heavy queries, which scan many millions of datapoints across thousands of distinct timeseries.”[5]

- Better insert speed and performance on low cardinality, “VictoriaMetrics wins in insert performance and in compression ratio(with respect to InfluxDB, TimescaleDB)” [6]

Cons:

- Same as Prometheus/Thanos

Conclusion:

The obvious choice is to use a TSDB, but which one? Unfortunately, time does not permit for setting up a test environment to benchmark these various TSDBs. So judgement can only be based on the respective platform features and user reviews (which may be biased). The reasons for deciding on the choice of TSDB are as follows:

- InfluxDB is a strong reject based on multiple sources on performance issues
- TimescaleDB may have some performance issues
- Not enough user review on Opentsdb
- Prometheus/Thanos and Prometheus/VictoriaMetrics seems comparable
- There are seemingly more users/community support for Prometheus/Thanos
- I am already familiar with Prometheus

Hence, Prometheus/Thanos became the final choice for distributed TSDB.

References

[1] https://www.reddit.com/r/devops/comments/8qvpz7/prometheus_or_influxdbs_tick/

[2] https://medium.com/faun/comparing-thanos-to-victoriametrics-cluster-b193bea1683?

[3] https://www.reddit.com/r/devops/comments/941n2k/tsdbs_at_scale_part_one/

[4] https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/

[5] https://medium.com/@valyala/when-size-matters-benchmarking-victoriametrics-vs-timescale-and-influxdb-6035811952d4

[6] https://medium.com/@valyala/high-cardinality-tsdb-benchmarks-victoriametrics-vs-timescaledb-vs-influxdb-13e6ee64dd6b

NoSQL

This is used for storing meta-data for :pk users. No nested structure or other complications involved so a simple key-value NoSQL database is enough. MongoDB is an obvious choice.

Distributed design may not be needed as

1) data is presumed to grow very slowly, or since :pk users is capped at 1 million, it may not grow at all. Only involves updating the existing 1million data points.

2) Taking fully saturated storage into account, each field storing 64 bytes (worst case) , total storage requirements : 1,000,000 users x 7 fields(see implementation design) x 64 bytes < **0.5Gb** which is quite managable.

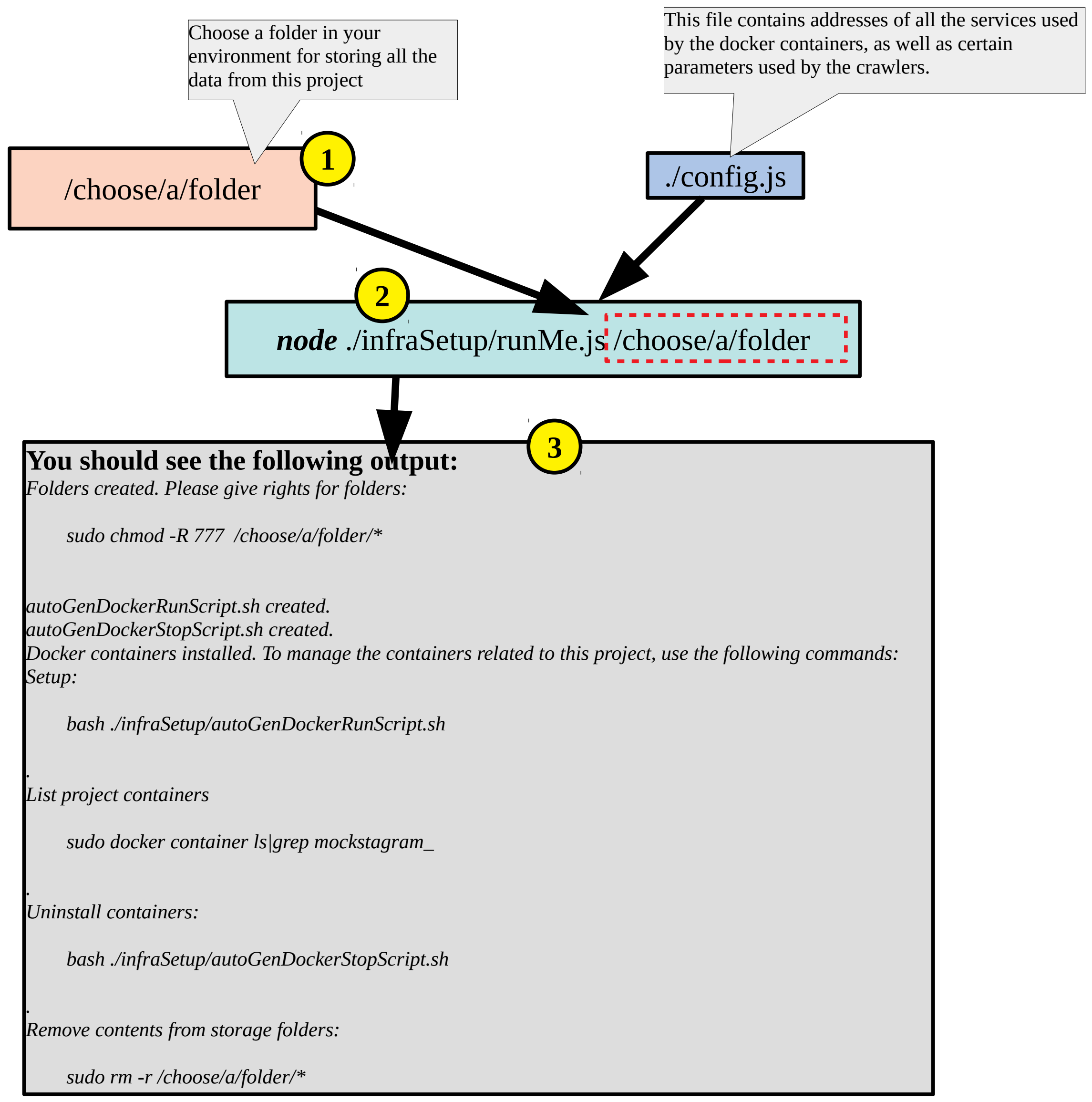
Infrastructure installation

All infrastructure platforms are ran in docker containers to avoid polluting and potentially conflicting the test environment.

The setup steps are as below:

- cd into repository

- Run “node ./infraSetup/runMe.js /choose/a/folder”


Choose a folder in your environment for storing all the data from this project
This file contains addresses of all the services used by the docker containers, as well as certain parameters used by the crawlers.
1
/choose/a/folder
./config.js
2
node ./infraSetup/runMe.js /choose/a/folder
3
You should see the following output:

```
Folders created. Please give rights for folders:

      sudo chmod -R 777  /choose/a/folder/*


autoGenDockerRunScript.sh created.
autoGenDockerStopScript.sh created.
Docker containers installed. To manage the containers related to this project, use the following commands:
Setup:

      bash ./infraSetup/autoGenDockerRunScript.sh

.
List project containers

      sudo docker container ls|grep mockstagram_

.
Uninstall containers:

      bash ./infraSetup/autoGenDockerStopScript.sh

.
Remove contents from storage folders:

      sudo rm -r /choose/a/folder/*
```

There are 7 containers used for this project. They are

- mockstagram_pushgateway
- mockstagram_prometheus
- mockstagram_thanos_sidecar
- mockstagram_thanos_store

- mockstagram_thanos_querier
- mockstagram_mongodb
- mockstagram_pushwiper

You should ***sudo docker container ls|grep mockstagram_*** make sure they are running successfully.

Optional

You may find some Docker/Kubernetes files in ***/mockstagram_scalable*** folder for increasing Mockstagram instances so as to handle crawling load.

Installing dependencies

- run *‘npm i --save’*

Running the code

- cd into the repo

1) Starting crawler manager

- run *‘node ./crawler/crawlerManager.js’*

Output:

crawlerManager listening on port 30000

To start crawling requests for :pk values, send HTTP request with the following format

```
POST localhost:30000/start_crawling_users
{
   'users': [1996161,1787381],
   'intervalSec': 5
}
```

2) Sending some requests to crawler with auto-generated :pk values

This is to help with auto-generating large number of random :pk values to help with load testing.

Pass a number argument, N, to the script to send crawling requests for N randomly generated :pk values to the crawlerManager. Eg: the following example sends crawling requests for 20 random :pk values

- run *‘node ./crawler/crawlerStressTest.js 20’*

3) Tests

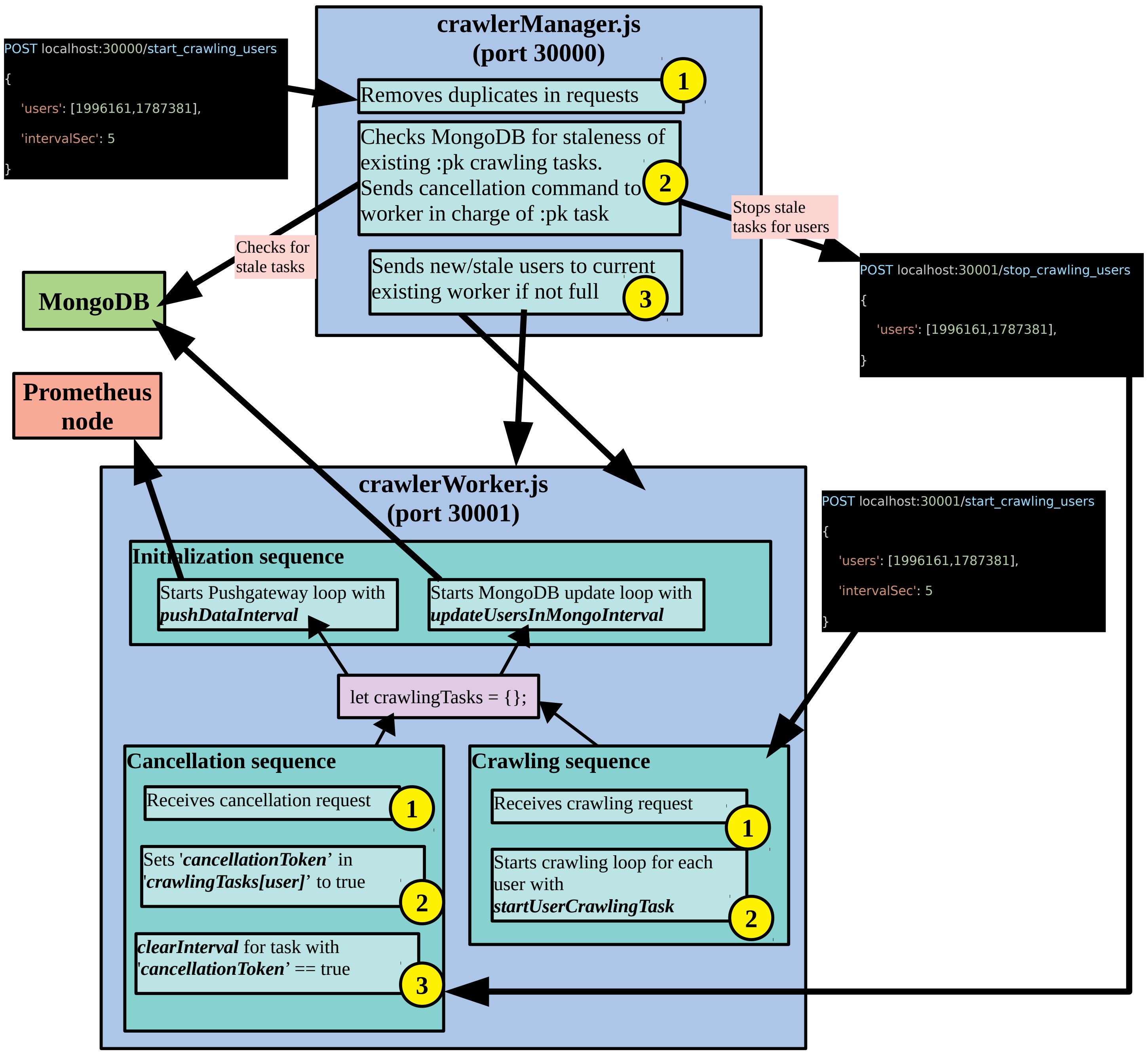
Some basic unit tests are setup in ./crawler/crawlerSpec.js. To use them,

- run *‘./node_modules/jasmine/bin/jasmine.js ./crawler/crawlerSpec.js’*

Crawling logic

The crawlerManager does a series of checks before sending these :pk values to workers.

If there are existing workers with available slots, manager will assigns these :pk tasks to them. If not, the manager creates new workers with ports >30000 and sends these tasks to them.



```
POST localhost:30000/start_crawling_users
{
    'users': [1996161,1787381],
    'intervalSec': 5
}
```

```
POST localhost:30001/stop_crawling_users
{
    'users': [1996161,1787381],
}
```

```
POST localhost:30001/start_crawling_users
{
    'users': [1996161,1787381],
    'intervalSec': 5
}
```

crawlerManager.js
(port 30000)
Removes duplicates in requests
1
Checks MongoDB for staleness of existing :pk crawling tasks. Sends cancellation command to worker in charge of :pk task
2
Sends new/stale users to current existing worker if not full
3
Checks for stale tasks
Stops stale tasks for users
MongoDB
Prometheus node
crawlerWorker.js
(port 30001)
Initialization sequence
Starts Pushgateway loop with *pushDataInterval*
Starts MongoDB update loop with *updateUsersInMongoInterval*
let crawlingTasks = {};
Cancellation sequence
Receives cancellation request
1
Sets '*cancellationToken*' in '*crawlingTasks[user]*' to true
2
clearInterval for task with '*cancellationToken*' == true
3
Crawling sequence
Receives crawling request
1
Starts crawling loop for each user with *startUserCrawlingTask*
2

Updating aggregated metrics

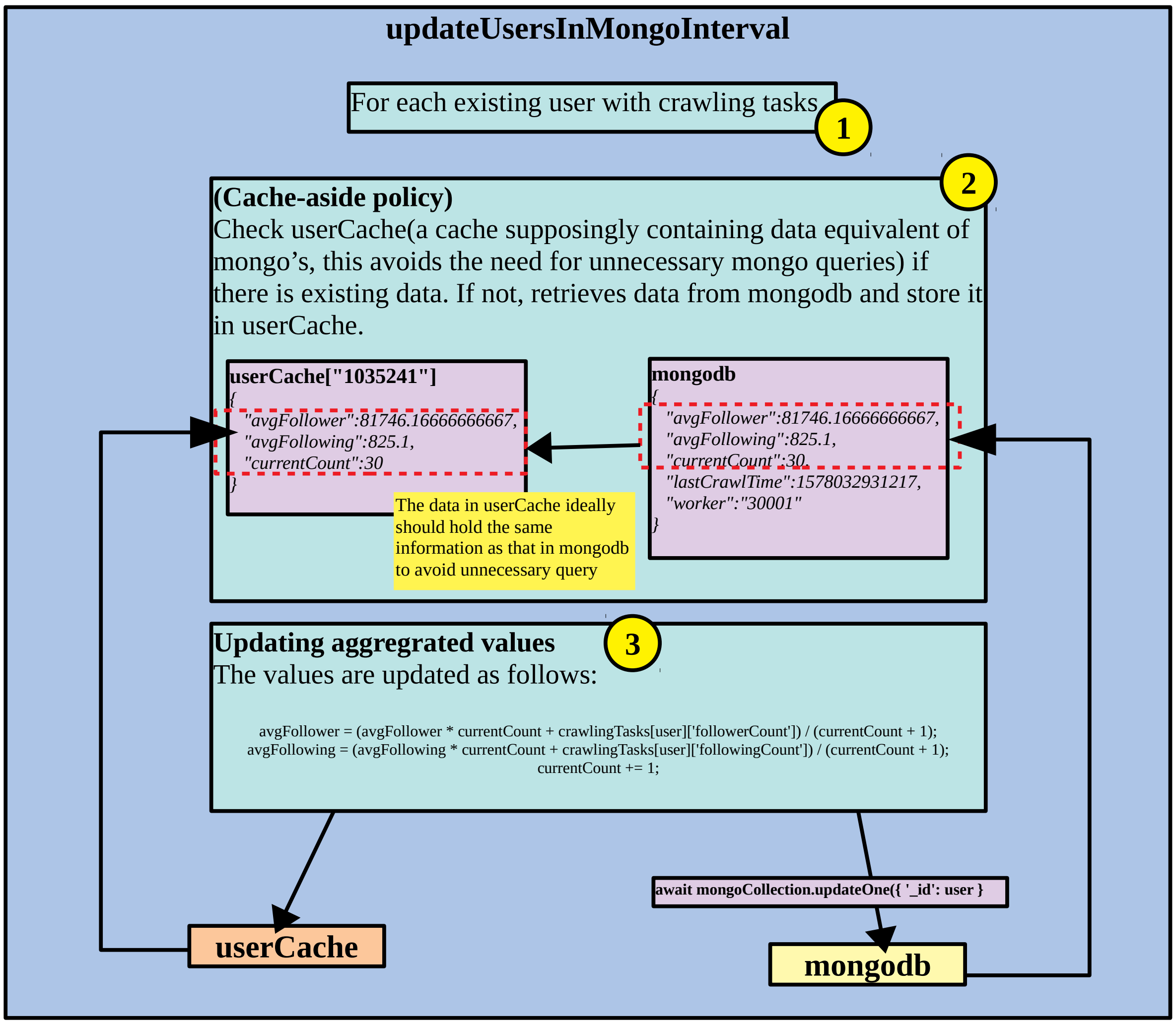
Aggregrated metrics refer to metrics like global average follower/following counts.

As seen in the crawling logic above, aggregrated metrics is handled by ***updateUsersInMongoInterval*** in ***crawlerWorker***.

Updating of aggregrated metrics is separated from the crawling tasks because the decoupling means that we can control the frequency of these 2 different tasks separately.

This means that we can downsample aggregrated metrics easily (crawling may be at 1 minute interval, but updating aggregrated metrics can be done at hourly or daily interval instead). This helps with improving the performance of overall system.

The sequence are as follows:



For each existing user with crawling tasks

(Cache-aside policy)
Check userCache(a cache supposingly containing data equivalent of mongo’s, this avoids the need for unnecessary mongo queries) if there is existing data. If not, retrieves data from mongodb and store it in userCache.

userCache["1035241"]
```
{
  "avgFollower":81746.16666666667,
  "avgFollowing":825.1,
  "currentCount":30
}
```

mongodb
```
{
  "avgFollower":81746.16666666667,
  "avgFollowing":825.1,
  "currentCount":30,
  "lastCrawlTime":1578032931217,
  "worker":"30001"
}
```

The data in userCache ideally should hold the same information as that in mongodb to avoid unnecessary query

Updating aggregrated values
The values are updated as follows:

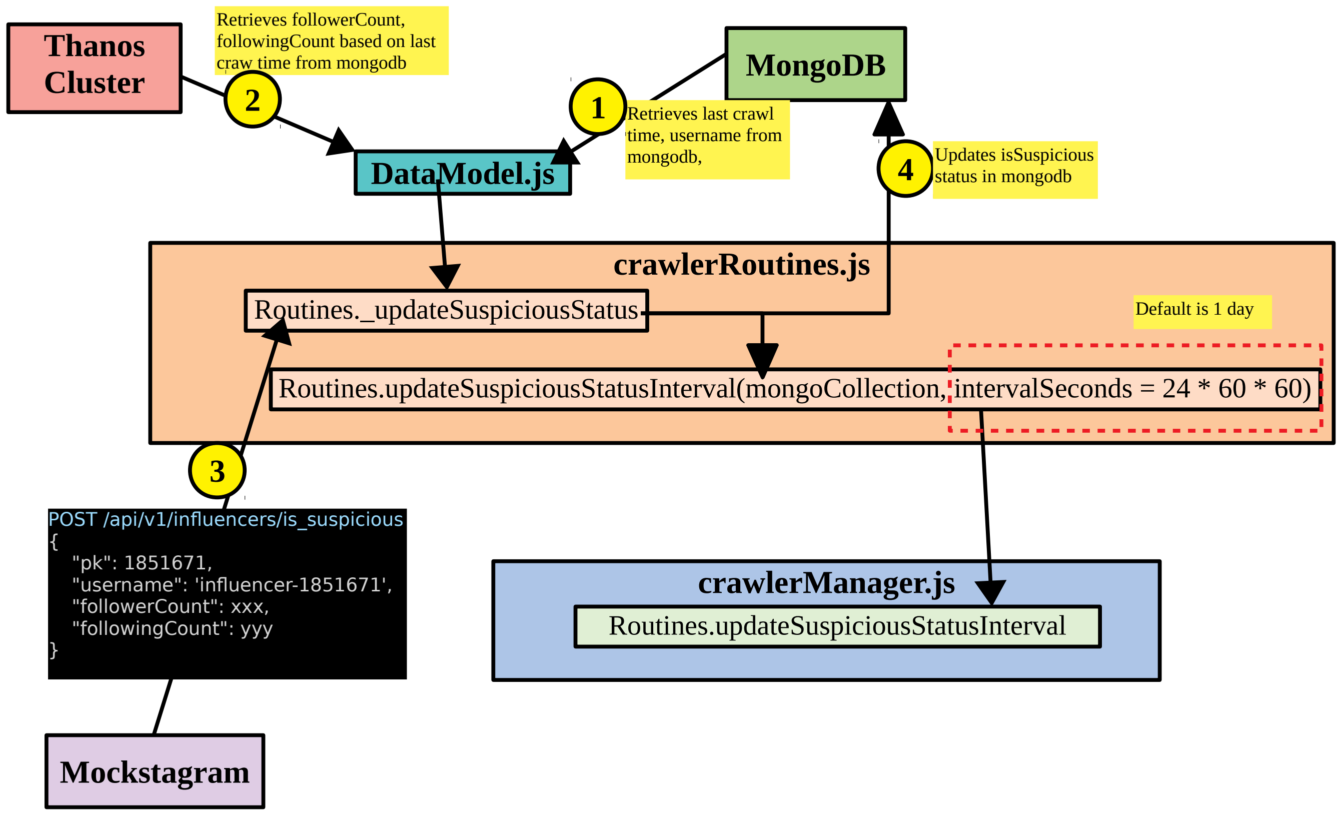
```
avgFollower = (avgFollower * currentCount + crawlingTasks[user]['followerCount']) / (currentCount + 1);
avgFollowing = (avgFollowing * currentCount + crawlingTasks[user]['followingCount']) / (currentCount + 1);
currentCount += 1;
```

```
await mongoCollection.updateOne({ '_id': user }
```

Updating suspicious status

This is implemented minimally due to time constraints. There is no buffer used, so the function will send requests for /is_suspicious for all users concurrently(albeit in a synchronized manner so there is some “queuing” to avoid DDOSing the server) once the interval is triggered.

The sequence are as follows:


Thanos Cluster
2
Retrieves followerCount, followingCount based on last craw time from mongodb
MongoDB
1
Retrieves last crawl time, username from mongodb,
DataModel.js
4
Updates isSuspicious status in mongodb
crawlerRoutines.js
Routines._updateSuspiciousStatus
Default is 1 day
Routines.updateSuspiciousStatusInterval(mongoCollection, intervalSeconds = 24 * 60 * 60)
3
crawlerManager.js
Routines.updateSuspiciousStatusInterval
Mockstagram

```
POST /api/v1/influencers/is_suspicious
{
   "pk": 1851671,
   "username": 'influencer-1851671',
   "followerCount": xxx,
   "followingCount": yyy
}
```

Accessing data for API server

The data from each database can be access as follows:

Key-value data
(db: mockstagram,
collection: userData)
- avgFollower
- avgFollowing
- currentCount
- firstStartedTime
- lastCrawlTime
- username
- worker
- isSuspicious

Using terminal

This assumes you have mongo installed

```
mongo --host localhost:27018
use mockstagram
db.userData.findOne({'_id':"1841861"})
```

Output

```
{
    "_id" : "1841861",
    "avgFollower" : 241803.07142857142,
    "avgFollowing" : 5940.952234881691,
    "currentCount" : 4564,
    "firstStartedTime" : 1578279363088,
    "lastCrawlTime" : 1578288491399,
    "username" : "influencer-1841861",
    "worker" : "30001"
}
```

Time series data
- follower_count
- following_count
- follower_ratio

Using terminal

Query a specific range

```
curl -G \
--data-urlencode 'query=follower_count{userid="1841861"}' \
--data-urlencode 'start=1578288291' \
--data-urlencode 'end=1578288491' \
--data-urlencode 'step=20' \
--data-urlencode 'dedup=true' \
--data-urlencode 'partial_response=true' \
http://localhost:19192/api/v1/query_range
```

Output

```
{
  "status":"success",
  "data":{
    "resultType":"matrix",
    "result":[
      {
        "metric":{
          "__name__":"follower_count", "env":"prod",
          "exported_job":"30001", "instance":"localhost:9091",
          "job":"users1", "userRange":"1",
          "userid":"1841861"
        },
        "values":[
          [ 1578288291, "278163"], [ 1578288311, "262199"],
          [ 1578288331, "215374"], [ 1578288351, "231446"],
          [ 1578288371, "237648"], [ 1578288391, "178951"],
          [ 1578288411, "110037"], [ 1578288431, "26435"],
          [ 1578288451, "22995"], [ 1578288471, "126131"],
          [ 1578288491, "51744"]
        ]
      }
    ]
  }
}
```

To query multiple labels, change "query" to:

```
'query={__name__=~"follower_count|following_count",userid="1841861"}[3m:30s]'
```

Query latest 3minute data with 30s resolution

```
curl -G --data-urlencode \
'query=follower_count{userid="1841861"}[3m:30s]' \
http://localhost:19192/api/v1/query
```

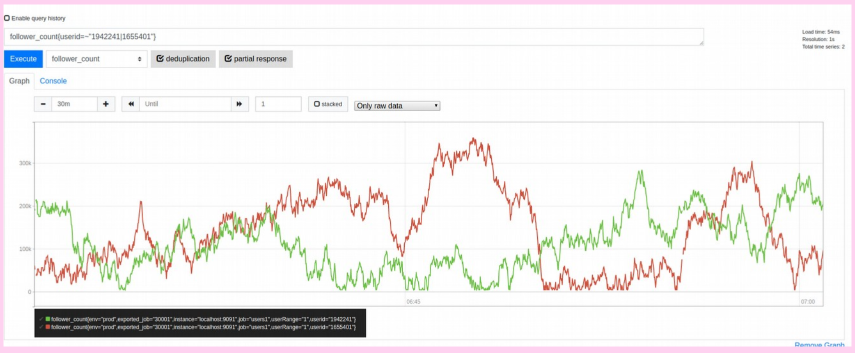
Output

```
{
  "status":"success",
  "data":{
    "resultType":"matrix",
    "result":[
      {
        "metric":{
          "__name__":"follower_count", "env":"prod",
          "exported_job":"30001", "instance":"localhost:9091",
          "job":"users1", "userRange":"1",
          "userid":"1841861"
        },
        "values":[
          [ 1578290010, "147682"], [ 1578290040, "133289"],
          [ 1578290070, "154656"], [ 1578290100, "253461"],
          [ 1578290130, "276340"], [ 1578290160, "354658"]
        ]
      }
    ]
  }
}
```

Using web ui

Web UI

http://localhost:19192/graph



Enable query history

```
follower_count{userid=~"1942241|1655401"}
```

Load time: 54ms
Resolution: 1s
Total time series: 2

Execute | follower_count | ✓ deduplication | ✓ partial response

Graph | Console

30m | Until | 1 | stacked | Only raw data

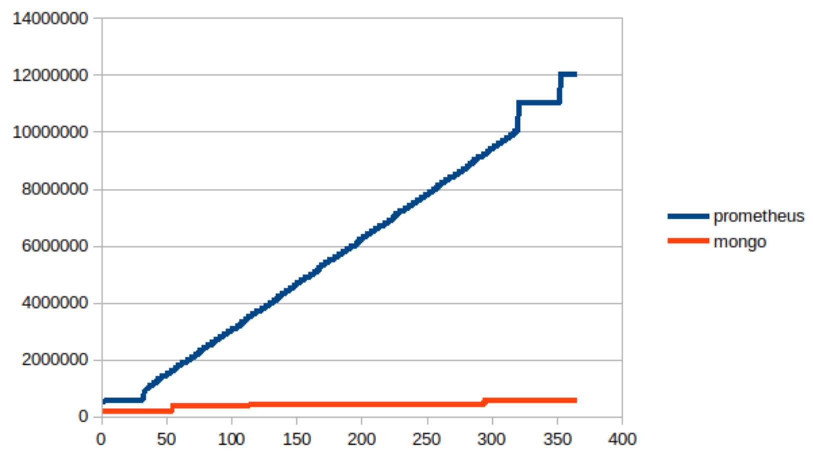
🟩 follower_count{env="prod",exported_job="30001",instance="localhost:9091",job="users1",userRange="1",userid="1942241"}
🟥 follower_count{env="prod",exported_job="30001",instance="localhost:9091",job="users1",userRange="1",userid="1655401"}

Remove Graph

Performance Evaluation

Storage performance

Testing was done with 1000 writes per second. Storage was profiled using `sudo du -cha –max-depth=1` on user-defined storage folder (as determined when setting up infrastructure) at 1 second interval for a period of ~ 5 minutes.



The increment in storage space of Prometheus worked out to be 25kb per second.

To scale this up to 1 million users:

25Kb/1k Ops/second * 16.7k Ops/second

~ 0.4175 Mb/second

~ 36Gb/day

Running performance

The number of crawler tasks is increased gradually in steps of 500(ops/sec) at 60 seconds interval using the command below:

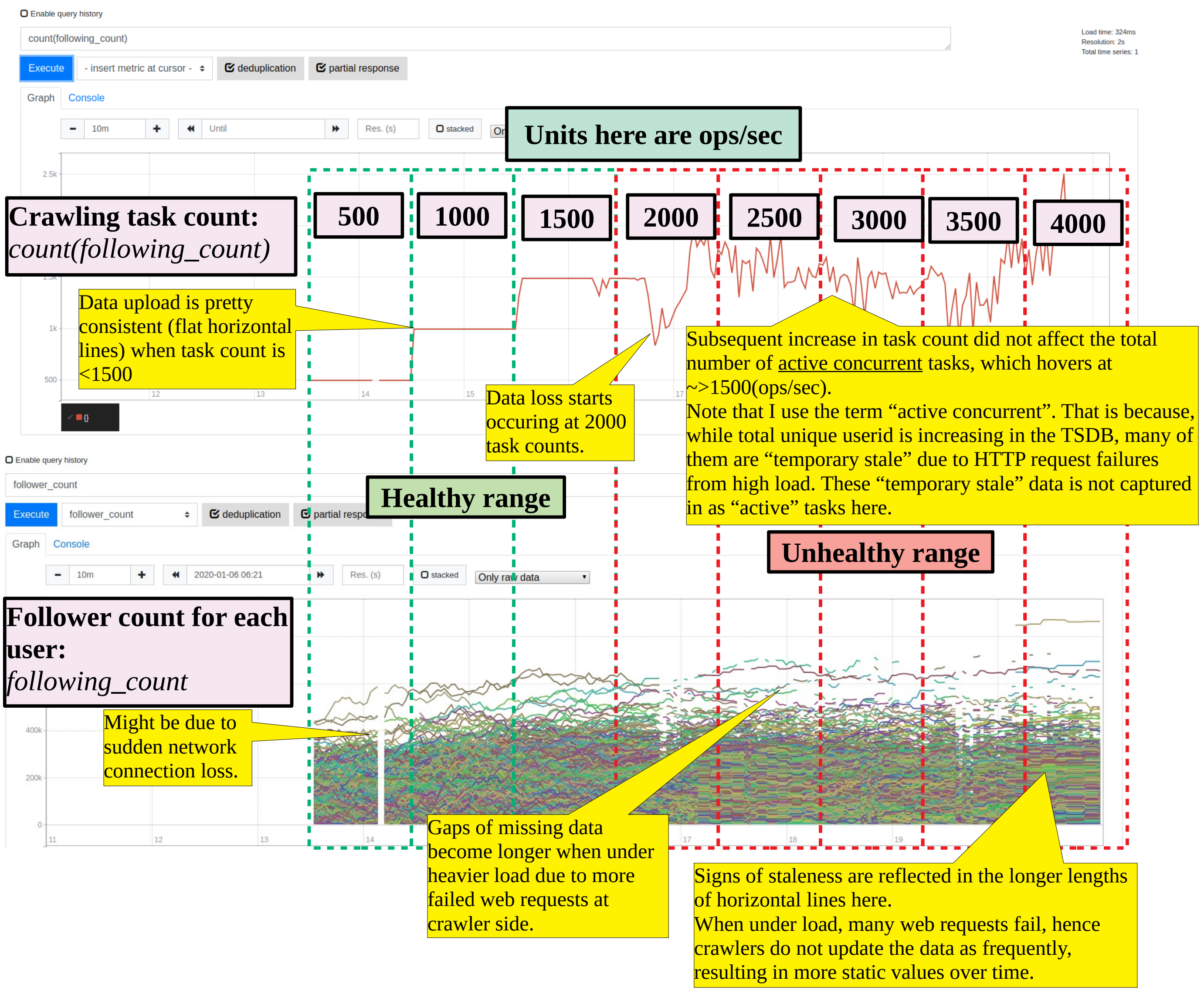
```
while true; do node ./crawler/crawlerStressTest.js 500; sleep 60; done
```

The specs of the hardware is as follows:

CPU: Intel® Core™ i7-6900K CPU @ 3.20GHz × 16

RAM: 64Gb

However, there are other unrelated processes running in the same environment so testing may not be accurate.



Enable query history: count(following_count)

Enable query history: follower_count

Data upload is pretty consistent (flat horizontal lines) when task count is <1500

Data loss starts occuring at 2000 task counts.

Subsequent increase in task count did not affect the total number of active concurrent tasks, which hovers at ~>1500(ops/sec).
Note that I use the term "active concurrent". That is because, while total unique userid is increasing in the TSDB, many of them are "temporary stale" due to HTTP request failures from high load. These "temporary stale" data is not captured in as "active" tasks here.

Might be due to sudden network connection loss.

Gaps of missing data become longer when under heavier load due to more failed web requests at crawler side.

Signs of staleness are reflected in the longer lengths of horizontal lines here.
When under load, many web requests fail, hence crawlers do not update the data as frequently, resulting in more static values over time.

One major bottleneck is the number of concurrent HTTP requests that can handle by a single node. This limitation of HTTP requests is due to 2 major factors

1) Fully utilized CPU time (Crawler side)

2) Suspected fully utilized sockets in either NodeJS or machine itself since the network traffic is not saturated at all. Need more time with this.

Hence, it it highly desirable to operate the crawlers within healthy limits, in this case <1500(ops/sec) per node to avoid data loss.

Summary

This assignment proposes a possible pipeline for crawling Mockstagram. The choice of TSDB(Prometheus) and handling of computational values(pre-computation, downsampling) means that the solution favours higher availability (AP) over stronger consistency (weak CP).