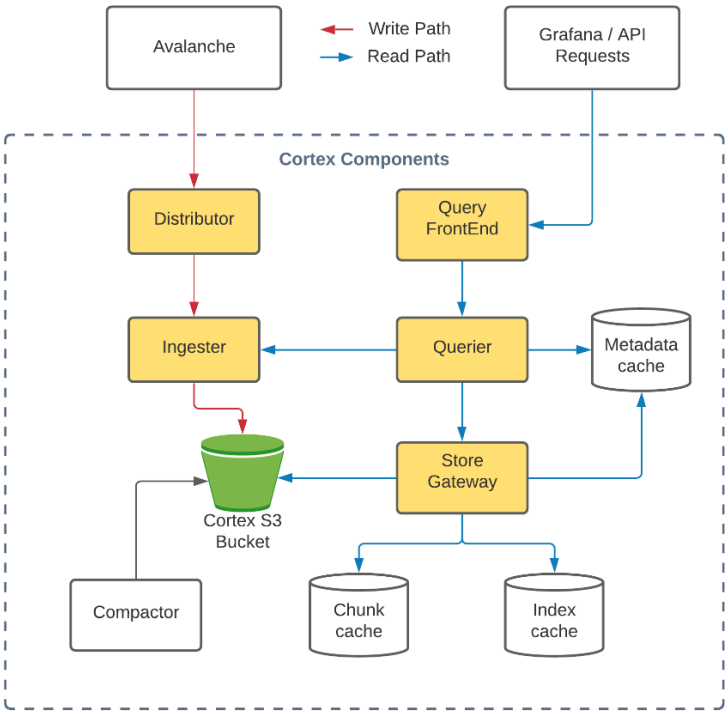
Cortex Load/Performance Testing

Scope



Write path load test requirement is to identify the minimum resources and components configuration in order to handle 110K samples/second ingestion rate. Since we have multi tenancy configuration, we configured sample ingestion as 11K samples/second from 10 different tenants. We expected to check the behavior of each Cortex components and how their utilization while such kind of load is ingested. Also, since Cortex is deployed on an EKS cluster, we planned to check the cluster behavior also.

Read path load test requirement is to identify whether the current Cortex components can handle 25 concurrent users continuously querying different metrics for few hours thru Cortex API. We planned to use different queries in order to minimize the cache usage and force more load on Cortex components.

Infrastructure

- Cortex version: 1.9.0
- Cortex helm chart version: 0.5.0

We are using Block storage architecture with a S3 bucket and Cortex cluster is deployed on an EKS cluster (*v1.19*). Since we are planning to change/fine-tune the resource configurations of Cortex components, those will be mentioned on each load/performance test respectively.

Tools

1. Avalanche

Avalanche can be used to ingest Prometheus supported samples to Cortex and it has the feature to configure tenant ids, so that we can utilize the multi tenancy feature of Cortex. But we had to bypass Cortex-gateway components which we used to authenticate remote write clients using JWT tokens. The reason for this is, Avalanche has no option to include JWT tokens (*required by Cortex-gateway*) in the API requests, but it can include the *X-Scope-OrgID* header parameter in all API requests with tenant name in plain text. So that we had to ingest data directly to Cortex-distributor.

2. Grafana

Grafana is used to visualize various important metrics in different dashboards. Since we need to observe utilization and status of the components, we have created few dashboards with important metrics. Also, Grafana can be used to query metrics from Cortex query front-end. But it is a manual task, so we had to go with another tool for querying data from Cortex. Since Grafana has the API queries, we did not need to create those manually and we retrieved the exact PromQL part from Grafana itself, but we had to change the components and time periods in each query.

3. JMeter / Taurus

To automate the API querying part for Read path load testing, we planned to use JMeter or Taurus tools to run multiple API queries for longer duration. And we can configure concurrent execution since we need to load test 25 or more concurrent users executing API calls to Cortex endpoint.

Execution

Setup for Write Path Load Test

To simulate real scenario, we deployed Avalanche in a different EKS cluster and configured Avalanche to send 11K samples per second to Cortex Distributor. Below is one avalanche deployment manifest,

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: avalanche-01
 namespace: load-test
 labels:
  name: avalanche-01
spec:
 selector:
  matchLabels:
   app: avalanche-01
 replicas: 1
 template:
  metadata:
   labels:
    app: avalanche-01
  spec:
   containers:
   - name: pg-avalanche
    image: quay.io/freshtracks.io/avalanche:latest
    args:
    - "--metric-count=16000"
    - "--label-count=20"
    - "--series-count=1"
    - "--metricname-length=90"
    - "--value-interval=30"
    - "--remote-batch-size=2000"
    - "--remote-requests-count=2500"
    - "--remote-write-interval=1500ms"
    - "--port=9001"
    - "--remote-url=http://<distributor-endpoint>:<port>/api/prom/push"
    - "--remote-tenant=cortex-loadtest-01"
    - "--const-label=TheLabelsNameFIeld01=TestFIeldNumberis-01"
    - "--const-label=TheLabelsNameFIeld02=TestFIeldNumberis-02"
    - "--const-label=TheLabelsNameFIeld03=TestFIeldNumberis-03"
    - "--const-label=TheLabelsNameFIeld04=TestFIeldNumberis-04"
    - "--const-label=TheLabelsNameFIeld05=TestFIeldNumberis-05"
    - "--const-label=TheLabelsNameFIeld06=TestFIeldNumberis-06"
    - "--const-label=TheLabelsNameFIeld07=TestFIeldNumberis-07"
    ports:
    - containerPort: 9001
```

Setup for Read Path Load Test

Installed the JMeter in an EC2 and created a JMeter test plan with API queries which we retrieved from Grafana graphs and other panels. JMeter can execute Postman API queries, so that we have created a JMeter test plan with the API requests to query data from Cortex query front-end. These API calls differ from one another because component names and time periods are different in each execution, this is handled by inputting a list of components and time periods into JMeter test plan as variables. Below is an example API query,


```
{       "name": "01. Sample Receive Rate",
        "request": {
                "method": "GET",
                "header": [{
                                "key": "X-Scope-OrgID",
                                "value": "<tenant-id>",
                                "type": "text"       }],
                "url": {        "raw": "http://<query-frontend-dns-
endpoint>:<port>/api/prom/api/v1/query_range?
query=sum(rate(cortex_distributor_received_samples_total%7Bservice%3D%
22cortex-distributor%22%7D%5B1m%5D))&start=1625458260&end=
1625461860&step=60",
                        "protocol": "http",
                        "host": [],
                        "path": [
                                "api", "prom", "api", "v1", "query_range"      ],
                        "query": [{
                                        "key": "query",
                                        "value":
"sum(rate(cortex_distributor_received_samples_total%7Bservice%3D%
22cortex-distributor%22%7D%5B1m%5D))"
                                },{     "key": "start",
                                        "value": "1625458260"
                                },{     "key": "end",
                                        "value": "1625461860"
                                },{     "key": "step",
                                        "value": "60"
                                }]}},   "response": []
}
```


Results

Write Path

We carried out over 50 load tests for Write Path, but this document only contains the important tests which can be highlighted. Below table contains the resource configuration and limits of each component (*pod*) which is used in Write Path,

Component	Memory	CPU	Persistent Disk
Ingester	Request: 5GiB Limit: 6GiB	Request: 0.1 Limit: 1.5	10GiB
Distributor	Request: 100MiB Limit: 3GiB	Request: 0.1 Limit: 2	N/A
Compactor	Request: 100MiB Limit: 2GiB	Request: 0.1 Limit: 1	50GiB

Most of the time we preferred to scale components horizontally, since it is easy to increase the replicas (*pods*) count rather than increasing the resources of the pods because Cortex cluster is running on an EKS and we will have to increase the EC2 resources accordingly if we were to go with vertical scaling. Plan was to run each test at least 10-12 hours to capture at least 3-5 block uploads to S3 bucket.

Total Samples per Second	Duration	Avalanche Replica Count	Test Type	Status	Component Replica Count	Comments / Observations
27,000	24hrs	1	Load	Pass	Ingester: 4 Distributor: 3 Compactor: 1	Cortex handled the ingested samples without any issues.
30,000	12hrs	1	Load	Fail	Ingester: 4 Distributor: 3 Compactor: 1	Successfully uploaded data to S3 bucket few times, but after some time the ingesters restarted continuously due to memory over utilization.
54,000 (27Kx2)	40mins	**2**	Load	Fail	Ingester: 4 Distributor: 3 Compactor: 1	27K samples/sec from each Avalanche replica, Ingesters failed after 40mins.
50,000	4.5hrs	1	Load	Fail	Ingester: **7** Distributor: 3 Compactor: 1	The ingesters restarted continuously due to memory over utilization when uploading data to S3 bucket.
46,000	13hrs	1	Load	Pass	Ingester: **8** Distributor: 3 Compactor: 1	Cortex handled the ingested samples without any issues.
110,000 (11Kx10)	45mins	10	Load	Fail	Ingester: **9** Distributor: 3 Compactor: 1	Ingesters failed after 45mins.
66,000 (11Kx6)	4hrs	**6**	Load	Pass	Ingester: 9 Distributor: 3 Compactor: 1	Cortex handled the ingested samples without any issues.
110,000 (11Kx10)	5.5hrs	10	Load	Fail	Ingester: **17** Distributor: 3 Compactor: 1	Successfully uploaded data to S3 bucket few times, but after some time all the ingesters restarted continuously due to memory over utilization.
110,000 (11Kx10)	14hrs	10	Load	Pass	Ingester: **18** Distributor: 3 Compactor: 1	Few ingesters restarted during several S3 uploads, but after those it handled the ingested samples without any issues. No data loss.
110,000 (11Kx10)	**25hrs**	10	Endurance	Pass	Ingester: 18 Distributor: 3 Compactor: 1	Few ingesters restarted during several S3 uploads, but after those it handled the ingested samples without any issues. No data loss.

110,000 (11Kx10)	**6hrs**	10	Spike	Pass	Ingester: 18 Distributor: 3 Compactor: 1	Sent additional 11K samples/sec spike for 5mins and another 11Kx2 samples/sec spike for 5mins.
110,000 (11Kx20)	18mins	**20**	Stress	Fail	Ingester: 18 Distributor: 3 Compactor: 1	Distributors restarted due to CPU over utilization and CPU limit increased to 2. Then ingesters restarted continuously.
110,000 (11Kx20)	18hrs	20	Stress	Pass	Ingester: **36** Distributor: 3 Compactor: 1	Few ingesters restarted during several S3 uploads, but after those it handled the ingested samples without any issues. No data loss.

Read Path

We ran several type of load tests like Write Path. Below table contains the resource configuration and limits of each component which is used in Read Path,

Component	Memory	CPU	Persistent Disk	Replicas Count
Store Gateway	Request: 1GiB Limit: 5GiB	Request: 0.1 Limit: 1	75GiB	1
Querier	Request: 512MiB Limit: 2GiB	Request: 0.2 Limit: 1	N/A	3
Query Frontend	Request: 512MiB Limit: 1GiB	Request: 0.1 Limit: 1	N/A	3

JMeter tool is used to execute API requests to Cortex query front end. Component names and query periods are inputted as variables to JMeter test plan, then execute 18 different API calls for 80 different components (*Total number of different API calls = 18 x 80 = 1440*) with different time periods in each execution, so that cache usage is minimized. JMeter has the option to execute API calls concurrently and main requirement is to check whether if Cortex can handle long duration API queries from concurrent 25 users. Below are the test results for each type of tests,

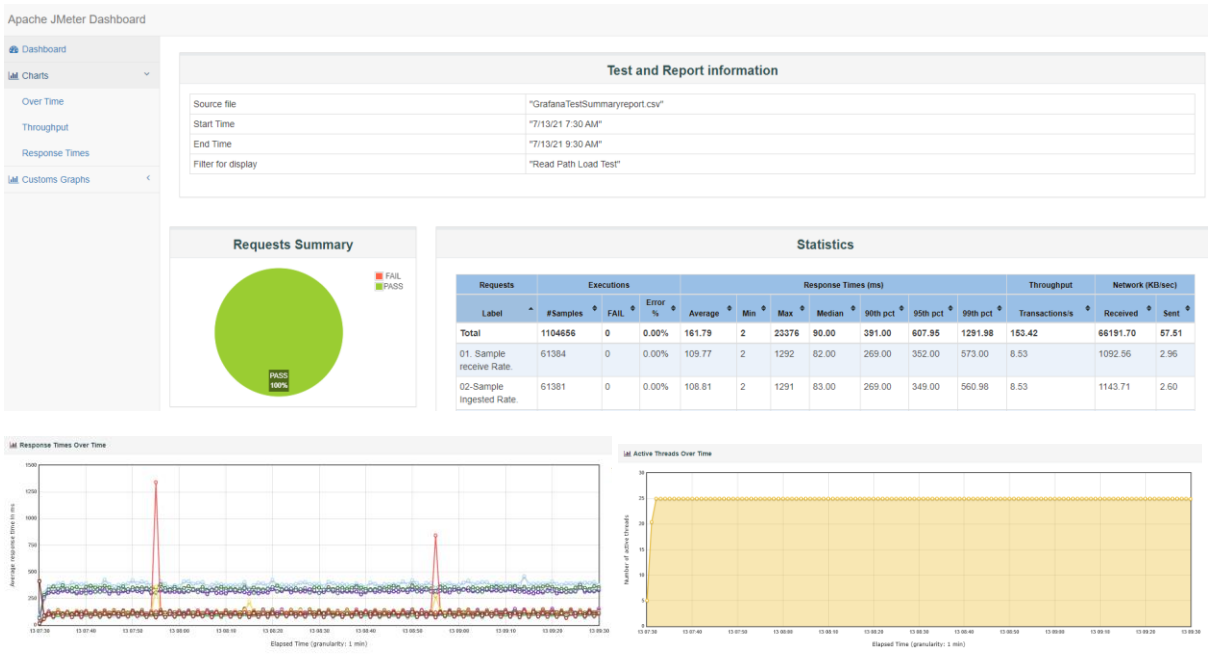
Total Queries per Second	Duration	Users Count	Test Type	API Call Time Period	Query Pass Rate	Status	Comments / Observations
~230-250	2hrs	25	Load	1hr, 24hrs, 7d, 15d	100%	Pass	Cortex handled the queries without any issues.
~230-250	2.5hrs	**40**	Stress	1hr, 24hrs, 7d, 15d	99.03%	Pass	Few failures found with 11K data point in one API call limit error

~1500	2.5hrs	40	Stress	15mins	99.99%	Pass	Cortex handled the queries without any issues.
~230-250	8hrs	**25**	Endurance	1hr, 24hrs, 7d, 15d	91.95%	Pass	1st error spike: Store Gateway restarted due to memory over utilization, but it got stable after that. 2nd error spike: Few failures found with 11K data point in one API call limit error
~230-250	1hr	25	Spike	1hr, 24hrs, 7d, 15d	100%	Pass	Sent additional 25 users spike for 10mins during the spike, QPS increased to ~630-650.

Utilization and Result Dashboards

Grafana Dashboards were used to visualize the utilizations of important metrics during the Load tests,

JMeter publishes a result dashboard when the tests are finished and that is used to get the error details, pass rate, response time, etc;



Avalanche does not have such kind of result dashboards, so that we used Grafana dashboards.

Conclusion

To handle 110K samples per second in Write path, we require 18 ingesters and 3 distributors with the mentioned resource limits. Also, we observed that Cortex can handle more ingestion rate when we increase the components count, so that we can say that there is no limitation on ingestion rate within our requirements. And Cortex community members have clusters which handles millions of ingested data per second. But since this is the minimum configuration for Write path and we will have to have a free buffer for utilizations, because an event where Cortex is failed, then to recover, ingesters need bit more resources. Regarding Read path, with the current resource configurations, Cortex can handle the queries and we do not need to increase the resources or components. But we will be checking on Store gateway memory and persistent volume utilization since those are at the peak of the utilization.