Google Summer of Code

PROPOSAL

CORTEX: Improve Ingester Handover

By

Saurav Malani

CLOUD NATIVE
COMPUTING FOUNDATION

cortex

Table of Contents

Section I: Abstract

Cortex is an open-source project providing horizontally scalable, multi-tenant, long term storage for Prometheus. Ingester is a stateful component in the Cortex ecosystem that builds Prometheus chunks from the incoming samples. In order to distribute load, a DHT (Distributed Hash Table) is used to route requests to different Ingesters. When a new ingester is created or existing one is deleted or crashed, this process requires approximately 10 minutes of time and during that time span ingester doesn't accept any data. So, all the data at the distributer node for that time period is simply dropped. So, to minimize this, the current implementation only allows users to scale up their ingester pools by 1 ingester per 12 hour period, which is not great when load changes dramatically. So, the main aim of this project will be to improve how ingesters hand over their data when they are being created or deleted in order to easily scale.

Section II: Background

a. What is Prometheus?

Prometheus is a numerical time series based operational system monitoring & alerting toolkit. It was created to work well with dynamic systems where things float and shift all the time and you still want to be able to have the insight and be able to track where exactly a metric came from. So, it basically takes care or define ways to get metrics out of the microservices/network node, in a dimensional time series format. It point at things that have metrics, for data extraction, which might be one's own apps or it might be some kind of exporter that gets data from somewhere else.

Prometheus do not use flat or more hierarchical data model as it kind of gets a bit inflexible to work with that kind of data model. In Prometheus we have a metric name and a bunch of key-value pairs attached to it and all the key-value pairs attached to a metric name result in identifying one time series. So, given a metric name and key-value labels, the following format is used to address metrics: `<metric name>{<label name>=<label value>, ...}`.

Eg. metric name: api_http_requests_total, with path=/users, status=200, job=api-server, method=GET & instance=1.2.3.4:80 as key-value pairs and queries

could be of this type: api_http_requests_total{method=”GET”}.

So, basically it pulls the metrics, i.e. scrapes the data, then it compresses them and then store them on the disk and it also serves up queries, which come from some kind of dashboard viewer.

b. *Problems with Prometheus or what problems does Cortex solve?*

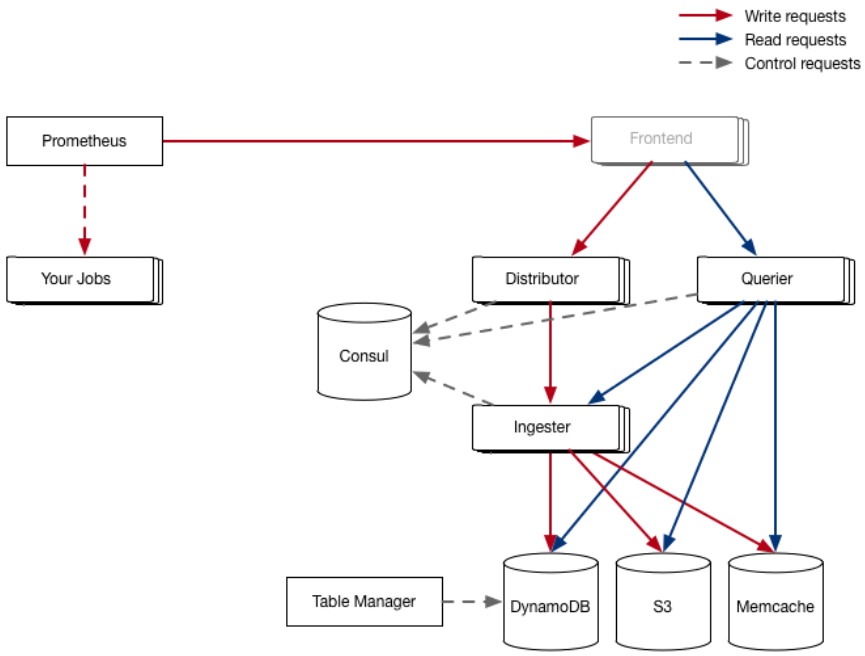
Prometheus was designed to do a small set of things very well and to work seamlessly in conjunction with other optional components rather than overburdening Prometheus with an ever-growing array of hard-coded features and integrations. Hence, Prometheus does not provide:

i. *Long-term Storage:* Each Prometheus instance store data locally and do not provide distributed data storage system. Hence, data is lost if the local storage fails.

ii. *Global View of Data:* Each Prometheus instance act as an isolated data storage unit, it wasn’t built as a distributed database. Hence, it is pretty complicated to get a “global” view of the time series data.

iii. *Multi-tenant:* Prometheus has no built-in concept of tenant. So, it is impossible to get tenant-specific data access and resource usage stats.

c. *Cortex Architecture*



Cortex is fundamentally a service-based design, with its essential functions split up into single-purpose components that can independently be scaled. Cortex has the following main components:

i. Distributor: It automatically replicates and shard the incoming time series data from Prometheus and send it to multiple Cortex ingesters in parallel using distributed hash table (DHT).

ii. Ingester: It compresses the received time series data for a few hours using Gorilla Compression and then writes that data to long-term storage backends.

iii. Ruler: The ruler execute rules and generates alerts and sends them to Alertmanager.

iv. Querier: It query both in ingester and in long-term storage and display the result in Grafana dashboard.

d. How Cortex solve Prometheus problems?

Cortex, as a Prometheus-as-a-Service, solves all of the above mentioned problems (that Prometheus faces) and provides an out of the box solution to even the most demanding monitoring and observability use cases.

i. It supports long-term storage systems like AWS DynamoDB, Google Cloud Bigtable.

ii. It also offers a global view of Prometheus time series data including data in long-term storage.

iii. Every Prometheus metric that passes through Cortex is associated with a specific tenant & the data can only be queried by the same tenant. Hence, Cortex has a built-in concept of multi-tenancy.

So, in short, Cortex is a time-series store build on Prometheus which is horizontally scalable, highly available, provides long-term storage & it is a multi-tenant system.

e. *Problems with current Cortex & Where my project comes in?*

In the current scenario, when we do a rolling update, each ingester does a hand-over of all the data from the “leaving” one to the “joining” one. This process of copying data takes about 1 minute per million timeseries, which means each ingester will take about 3-4 minutes to hand-over all it’s data to the joining ingester, in a typical installation. But during this process neither the leaving or joining ingester accepts any data.

So, the data that maps (via the DHT) to the leaving ingester is spilled to another ingester (apart from leaving and joining ingester), creating a 3-minute overflow time series that must be flushed to the data store. This process of spilling will create millions of 3-minute chunks in the database (as for this 3 minute there won’t be any compression because Gorilla compression need hours of data for better compression). This creating of millions of 3-minute chunks in the database is quite inefficient and will result in memory overhead.

So, to minimize spilling, the current implementation only allows users to scale up their ingester pools by 1 ingester per 12 hour period, which may create problems, when load changes dramatically.

This is where my project comes in, i.e. the main aim of this project is to solve this problem & improve how ingesters hand over their data when they are being created or deleted in order to easily scale.

Section III: Possible Solutions

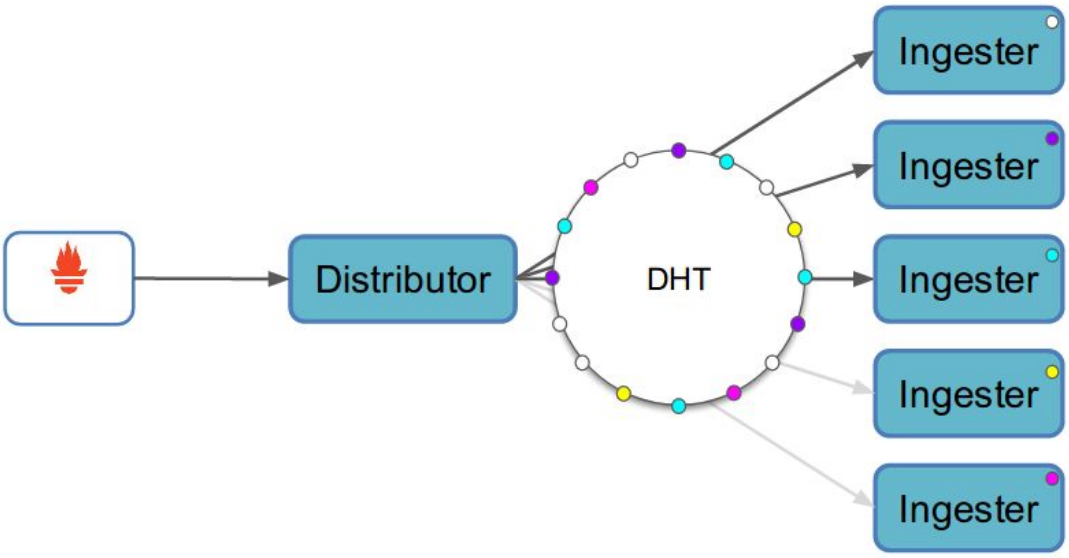
Note: The problem that we are addressing in this project is pareto optimal. Afters hours of discussion over weeks with the mentor over slack and hangout call, where we discussed several possible solutions, some of them involved making some architecture level changes, whereas some involved change in the basic idea. But, each of ideas has its own merit and demerit. And as optimality will depend on the factors like efficiency, speed, impact on latency, elegance & testability. Hence, we came to the conclusion that at this point of time we can’t really say which is the most optimal solution. Hence, as a part of this project, over the summer, for the first 6 weeks I will be prototyping the various possible solutions (discussed later) and based on the following parameters we will be deciding which is the most suitable solution in our scenario:-

1. Efficiency, e.g. amount of CPU, RAM, disk I/O.
2. Speed - overall time to complete handover
3. Impact on latency of push and query operations.
4. Elegance / ease of understanding the code.
5. Testability.

In short, in this project, for the first 6 weeks I will prototype the below discussed solutions. And, based on their performance, the best one will be selected and remaining summer will be spent fully in implementing it.

Here are some of the possible solutions among which the best one will be decided by prototyping:-

1. **Transfer ownership of tokens one at a time:**



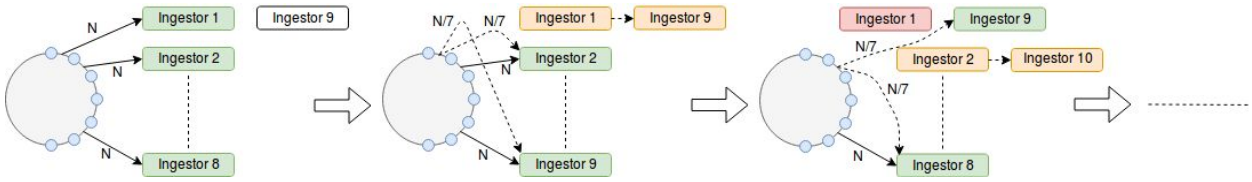
As each ingester has the ownership of some ‘n’ number of tokens. And, the time series corresponding to a token is transferred to the respective owner ingester.

So here, in this method during handover of one ingester, say ingester-A, to another ingester, say ingester-B, instead of transferring the ownership of all ‘n’ ingester at a time. We transfer the ownership of tokens in the DHT one at a time to minimise the time. And, also to be able to merge data between the leaving and joining ingesters so there is only one stream of data.
In this method, there will be really small about of data spilling because the

ownership of ingester of transferred one at a time. So, the time required to transfer corresponding data from old ingester to new ingester will be really small, around 1 second. Hence, re-merging of the spilled data won't be difficult.

Yet another possibility to avoid spilling in this scenario is to reject the call in the distributor, which return a 500 error code, and then the sender will retry. This basically creates queue inside the sender. This would be workable if the window of rejection is really small, e.g. 1 second.

2. **Redistribute all the tokens during the handover.**



One of other possible solutions is, redistribute all the tokens of the leaving ingester to the remaining ingester, before changing the "state" to leaving. And, repeat the same with the remaining ingester one by one.

So let's say, initially there are 8 ingester named 1-8 respectively and we want to perform handover. So, in place of these old 8 ingester new 8 ingester will be introduced, let them be numbered 9-16. Now, as per the above algorithm the following sequence of steps will happen:

- **a.** Ownership of all the tokens, whose owner is ingester-1, are redistributed among the Ingester 2-8. So, after redistribution of the tokens, the time series will go to the new owner Ingester.
- **b.** Once the redistribution of token is done. The corresponding ingester i.e Ingester-1 is marked as leaving. And, now all the data in ingester-1 in transferred to the Ingester-9.
- **c.** Once, the above two steps (a & b) are performed. Now, we do the same will be done for the remaining 2-8 ingester. But, the only difference will be, now

the tokens corresponding to Ingester-2 will be redistributed among Ingester 3-9.

So, in general if there are initially ‘n’ number of ingester, then for each ‘i’ we will redistribute it’s tokens among ‘i+1’ to ‘i+n-1’ ingester. And after the redistribution of the tokens, ingester ‘i’ will be marked leaving and it’s time series will be transferred to the ‘i+n’ th ingester. And, the incoming time series will now go to the new owner.

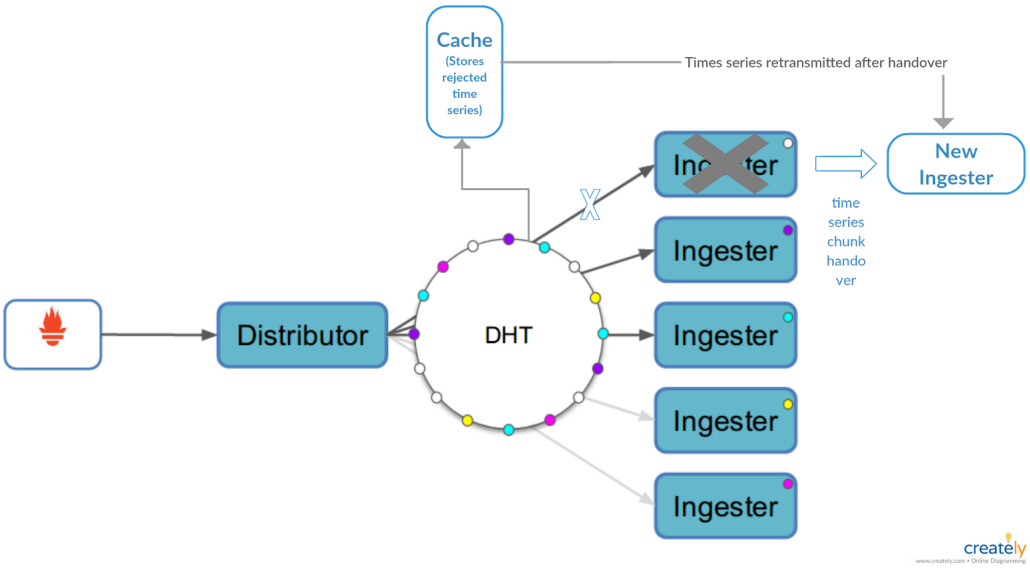
3. **Send samples to joining ingester during handover.**

   Currently, an ingester can in one of the following 4 states “ACTIVE”, “LEAVING”, “PENDING” and “JOINING”. Here, the idea is to add one more state i.e. "PREPARING_TO_LEAVE", apart from the existing ones and if an ingester is marked as “PREPARING_TO_LEAVE”, it can still receive time series.

   So, instead of directly changing the state of the leaving ingester from “ACTIVE” to "LEAVING" and stop accepting any data, we mark it "PREPARING_TO_LEAVE" and start transferring data chunk to the new ingester, whose state is marked as “JOINING”. And, during this time we still receive data. And once the transfer is complete, now we change the state of leaving one from "PREPARING_TO_LEAVE" to "LEAVING" and change the ownership of all the tokens belonging to the leaving ingester to the newly joined one (i.e where all the time series data is transferred for the leaving ingester) & from now on all the time series are transferred to this new ingester.

4. **Adding a Cache microservice to temporarily store the rejected time series.**

   As stated earlier that if the ingester state is “leaving” or “joining” it does not accept any data during. So, to solve this problem, instead of spilling data randomly in some other ingester we can store it temporarily in “Cache” (as demonstrated in the diagram below). So, once the handover is completed, all the rejected time series for that time period is retransmitted to the new token owner i.e. new ingester. In case of query, to find the relevant time series chunks, we have to look in the “Cache” microservice too, apart from ingester and long-term data storage.



A small variation to this idea is, instead of having a completely different “Cache” microservice, we can store the rejected data in a queue in the “Distributor”.

Section IV: What benefits does your proposed work have for Cortex and the overall CNCF community?

As currently there is no mechanism to avoid spilling of data during ingester handover. Hence, to avoid frequent occurrence of this scenario the current implementation only allows users to scale up their ingester pools by 1 ingester per 12 hour period, which can create a lot of problems when load changes dramatically. But, as this project’s AIM is to solve this problem. Hence, after successfully completion of this project, there will be a smooth handover of ingesters. Hence, there won’t be any limitations on the addition, removal and handover of ingesters.

As, Cortex is infinitely scalable and is perfect for operational system monitoring in distributed microservices based environment. Hence, it is the future of operational system monitoring. And, as resolving this critical problem is very important to Cortex. Hence, equally important to the operational system monitoring toolkit world. And, hence the project is really crucial to CNCF community.

Section V: Schedule Of Deliverables

Days	Task to be done
May 6 - May 26	1. Community bonding period 2. Getting used to with the Cortex code base.
May 27 - June 7	Hands on with current Cortex code base.
June 8 - June 15	Complete prototyping of possible solution-1 i.e. Transfer ownership of tokens one at a time.
June 16 - June 24	Complete prototyping of possible solution-2 i.e. redistribute all the tokens during the handover.
June 24 - June 28 Phase 1 Evaluation	
June 28 - July 5	Complete prototyping of possible solution-3 i.e. Send samples to joining ingester during handover.
July 6- July 13	Complete prototyping of possible solution-4 i.e Adding a Cache microservice to temporarily store the rejected time series.
July 14 - July 20	Compare all of the above prototype and analyse which solution is best for smooth handover, based on the above mentioned parameters.
July 22 - July 26 Phase 2 Evaluation	
July 21 - Aug 15	Fully implement the selected prototype.
Aug 15 - Aug 19	Documentation

Final Submission

Section VI: About Me

a. Personal Information & Contact Details

Name: Saurav Malani
Email ID: sauravmalani1@gmail.com
Resume: https://researchweb.iiit.ac.in/~saurav.malani/Resume.pdf
Linkedin: https://www.linkedin.com/in/saurav-malani-686854114/
Github: https://github.com/saurav-malani
Slack/IRC: malani
Phone no.: +91 8239472824

Location:
Hyderabad, India
Time zone: UTC +5:30 (IST)

Education:
4th year, B.Tech and MS By Research in CSE, IIIT-Hyderabad

b. Why Me?

1. As the project is highly technical, so I till date I have spent a decent amount of time in understanding the project. I went through the original design document of Cortex project, Cortex community meeting notes, relevant youtube talks in the process of information gathering related to project. Apart from this 10's of hours brainstorming the possible solution after understanding the core architecture of Cortex. Proposed and discussed in great detail several potential solutions with the mentor: Bryan Boreham. So, I can assure you that I have understood the architecture of the project very well and the reason behind the problem. And, based on the above potential solutions, that I have proposed above. Hence, I am pretty confident that, if I will be given the opportunity to work on this project, then I will be able to successfully complete it.

2. Apart from the above experiences, I have solved a few bugs in Kubernetes Python-Client issue-70684, here I worked on updating OpenAPI SecurityDefinition in Kubernetes Python-Client to use Authorization header name, following HTTP standard. Also, currently I am also working on issue: 229 which requires solving "test directory listing" issue & then Creating and Uploading corresponding Kubernetes Python-Client package to Python Package Index. This still requires a little bit of work to be done. So, will be submitting the PR in sometime. Hence, I have a decent understanding of in's and out's of Kubernetes, which will be definitely helpful directly or indirectly.
3. Also, I have submitted a few PR in Linux Foundation, OpendayLight Organization
   https://git.opendaylight.org/gerrit/69452
   https://git.opendaylight.org/gerrit/#/c/69361/

4. I have decent experience of debugging in distributed system (microservice) environment using Jaeger. Check out the project where I used Jaeger here.

5. Also, to know more about my work experience, please check out my resume here.

c. Availability?

I will work 6 - 7 hours on weekdays and at least 10 hours on weekends (Saturday and Sunday). So I will be able to devote at least 50 hours per week for the project (6.5 * 5 + 10 * 2 = 52.5 hours).