Building Fully Managed Service
for Beam Jobs with Flink on Kubernetes

Rishabh Kedia / Talat Uyarer
Streaming Team
Palo Alto Networks
Agenda

- What is Cortex Data Lake
  - Streaming Service Overall Design
- Design of Kubernetes Streaming Infrastructure
  - How we run Beam Flink Jobs on Kubernetes
- What kind of Challenges that we faced
  - Checkpoint Challenges
  - Scale Challenges
- Autoscaling to use resources efficiently
What is Cortex Data Lake
What is Cortex Data Lake

- PANW provides cloud-based, centralized log storage and aggregation for any kind of firewalls.
- One of our locations receives more than 20+ million records per second and can be scaled to receive more than 100 million records per second.
- We serve more than 10 different applications with 20 thousands streaming jobs in 10+ geographical regions.
Streaming Architecture at PANW

- Application
- Api Gateway
- Api Server
- Job Deployer
- Beam Job
- Dataflow Job
- Metadata Service
- Schema Registry

- Producer
- kafka
- Datastore

- Self Healing
- Cold Starter
- Kafka Partition Scaler

Application's Endpoint
“Thanks to Beam, we are not locked to any vendor, and we don’t need to change anything else if we make the switch.”

– Tahat Upener
Sr Principal Software Engineer

Streaming Architecture at PANW
Kubernetes
1. Create a BEAM pipeline and compile into a Uber JAR.
2. Create a Docker image containing the JAR and Flink Configuration files
3. Create a Yaml to Deploy Kubernetes make entry point is flink-run command
Deploying a Beam Job on Kubernetes

Source: https://jbcodeforce.github.io/flink-studies/architecture/
How About Deploying Thousand Jobs?

- Creating deployments via a client is not feasible and makes integration difficult.
- Flink introduced a Kubernetes Operator to handle all deployment needs via extending Kubernetes API Server with Custom Resource Definition.
How to integrate in current production?

- Complex Kubernetes Interactions
- Inconsistent Flink Job Management
- Deployment Errors and Inefficiencies
- Barriers to Innovation

In addition to above items, our existing production service used Dataflow API. So how we can integrate Operator without an API support?

```
$ kubectl apply -f flink-job1.yaml
$ kubectl scale flinkdeployment flink-job1 --replicas=16
```
Solution: FKO Library
Benefits

- Authentication and Authorization
- Standardized Job Management
- Abstracted Kubernetes Complexity
- Easy Upgrades
- Effortless Deployment
- Empowering Data teams
Challenges
Checkpointing is important to have healthy job. Let’s calculate possible cost for our scale

- 20 K+ jobs
- 10 Seconds targeted checkpointing time
- 200 write calls (20000 / 10) per seconds per job
- ~ 20 Operator per Job
- (20 x 200) ~ 4000 write calls per second ($2 per second)

[Source: https://cloud.google.com/storage/pricing]
What we did for Checkpointing

- Reduce Checkpoint size by removing PipelineOptions
- Beam assign default parallelism for all independent pipelines without checking Kafka Partition Count.
- Define Memory Threshold to prevent creating so many small files (state.storage.fs.memory-threshold)
- Enable Unaligned Checkpointing
Flink Task Assignment Issue

- Flink supports evenly distribution if you have one source.
- If you have multiple independent pipeline like us, Flink starts scheduling each partitions of source from zeroth pod/machine
- This makes first couple machines’ load heavier than tail nodes.
Autoscaling
Problem: Scaling

- Variable Traffic Pattern
- Performance Bottleneck
- Resource Underutilization (HPA is not enough)
Solution: Custom Autoscaler

- **Producer**
  - Add to **Queue**
  - Consumed by **Consumer 1**
  - Consumed by **Consumer 2**
  - Consumed by **Consumer 3**

- **FKO Library**
  - Get Flink Jobs

- **Flink Kube Cluster**

- **FKO Library**

  - **Flink REST API**
    - 1. Collect Metrics
    - 2. Analyze Metrics
    - 3. Action / Execute

  - 1. Get Topological graph
  - 2. Get Metrics
<table>
<thead>
<tr>
<th>Meter</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Throughput</td>
<td>How fast we process data on a given pipeline.</td>
</tr>
<tr>
<td>2. Backlog</td>
<td>The current lag of the Kafka topic</td>
</tr>
</tbody>
</table>
| 3. Backlog Growth   | Backlog is increasing/decreasing or constant \[
|                     | \( \text{Backlog Growth} = \frac{d\text{Backlog}}{dt} \)                  |
| 4. Input Rate       | Input Rate = Backlog Growth + Throughput                                   |
| 5. Backlog Time     | We should have some backlog for healthy processing. (~10 sec) \[
|                     | \( \text{Backlog Time} = \frac{\text{Backlog}}{\text{throughput}} \)      |
| 6. CPU Utilization  | What % of CPU is busy                                                      |

Note: https://www.infoq.com/presentations/google-cloud-dataflow/
Scale Up

Pre-condition: cpu > 75% and backlogTime > 10s

1. Increasing Backlog aka Backlog Growth > 0:
   \[ \text{Worker}_{\text{require}} = \text{Worker}_{\text{current}} \frac{\text{Input Rate}}{\text{Throughput}} \]

2. Consistent Backlog aka Backlog Growth = 0:
   \[ \text{Worker}_{\text{extra}} = \text{Worker}_{\text{current}} \frac{\text{Backlog Time}}{\text{Time to Reduce Backlog}} \]

To sum up:
\[ \text{Worker}_{\text{scaleup}} = \min(\text{Worker}_{\text{require}} + \text{Worker}_{\text{extra}}, \text{Worker}_{\text{max}}) \]

Scale Down

Pre-condition: cpu < 75% and backlogGrowth < 0 and backlogTime < 10s

1. So the only driving factor to calculate the required resources after a scale down is CPU.
\[ \text{Cpu Rate}_{\text{desired}} = \frac{\text{Worker}_{\text{current}}}{\text{Worker}_{\text{new}}} \frac{\text{Cpu Rate}_{\text{current}}}{\text{Cpu Rate}_{\text{current}}} \]

Note: https://www.infoq.com/presentations/google-cloud-dataflow/
Benefits

● Efficient Resource Utilization

● Customizable Scaling Parameters

● Lower Costs (reduced cost by 50%)
Scaling Lessons

- Active Resource Manager vs Standalone
  - Reducing provisioning time for scale actions
- Even though we use Standalone Mode, Internally Flink Submits jobs when we do scale up and scale down.
  - We hit Metascape Out of Memory issue.
  - Solution add your jar in Flink lib directory to prevent creating multiple classloader
- Beam did not expose backlog metrics
- Flink cache metrics when tasks are rescheduled
QUESTIONS?

linkedIn.com/in/talatuyarer/
linkedIn.com/in/rishabhkedia/