How to Simplify Kafka at Scale with Confluent Tiered Storage

This article originally appeared on Medium.com and is republished with permission from the author.

Confluent announced the general availability of Tiered Storage in the Confluent Platform 6.0 release, a new feature for managing and simplifying storage in large Kafka deployments. I have been able to test the solution with an on-prem object store, FlashBlade®.

Kafka provides a cornerstone functionality for any data pipeline: the ability to reliably pass data from one service or place to another. Kafka brings the advantages of microservice architectures to data engineering and data science projects. There are always more data sources to ingest and longer retention periods to increase pipeline reliability in the event of unplanned outages. Additionally, the use cases for Kafka as a permanent data store continue to grow.

But putting more data in a Kafka cluster results in operational challenges: more frequent node failures and longer rebalance times. Both impact the overall reliability of the cluster. Further, software upgrades become more challenging as rolling reboots need to be carefully staged so as not to introduce extra risk of data loss. And finally, storing more data in Kafka does not help if it cannot be predictably accessed with sufficient performance.

The rest of this post describes

- How Tiered Storage and FlashBlade work together
- How to set up and configure Tiered Storage
- Performance results of three realistic test scenarios

How Kafka Tiered Storage and FlashBlade Work Together

The Tiered Storage architecture augments Kafka brokers with a FlashBlade object store, storing data on FlashBlade instead of local storage on brokers. Brokers now contain significantly less state locally, making them more lightweight and rebalancing operations orders of magnitude faster. And finally, this architecture is transparent to the producers and consumers, which connect to brokers as always.

Tiered Storage simplifies the operation and scaling of a Kafka cluster enough so that it is easy to scale individual Kafka clusters to petabytes of data. And with FlashBlade as backend, Tiered Storage has the performance to make all Kafka data accessible for both streaming consumers and historical queries.

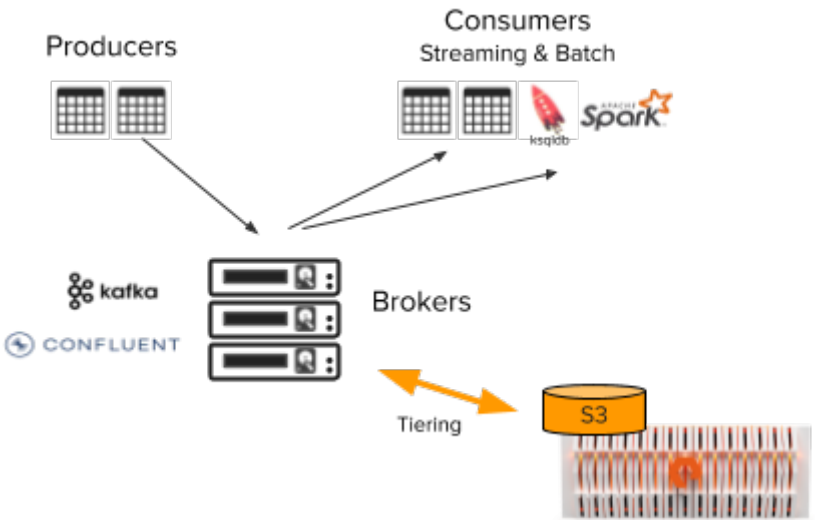


Figure 1: Kafka architecture with Tiered Storage and FlashBlade.

FlashBlade's design principles map well to disaggregated Kafka clusters; both FlashBlade and Kafka use scale-out architectures with linear and predictable scaling of performance and capacity. As an S3 backend for Tiered Storage, FlashBlade provides simple, performant, reliable, and scalable storage for on-premises data pipelines. Further, a FlashBlade system can be used for either or both filesystem (NFS) and object store (S3) use cases, both natively implemented on top of FlashBlade's flash-optimized, internal database.

How Tiered Storage Works

Overview of How Kafka Stores Data

Data in Kafka is organized into **topics**, which are a logical equivalent to tables. Each topic is subdivided into a configurable number of **partitions**, which allows parallelizing a topic across multiple Kafka **brokers**. Each broker is responsible for multiple partitions at a time. Partitions can have replicas on other brokers to create resiliency against failure, but only one broker is the **leader** for a given partition. **Brokers** ensure all messages are persistently stored.

Refer to the great blog post here for more detailed discussion about Kafka storage internals.

Tiered Storage In-Depth

Digging deeper into how brokers store message data as files/objects, incoming records in each partition are written into a **segment** data structure. As this is logically an append-only log, segments are periodically closed and new segments created. The frequency of rolling over and resultant segment sizes are controlled by *segment.bytes* or less-commonly *segment.ms*.

With Tiered Storage, once segments are completed, they are then copied to the Tiered Storage backend and deleted from local storage. Only one copy of the partition is uploaded to Tiered Storage; replicas are not copied. The result is that the Tiered Storage backend, FlashBlade, becomes responsible for providing enterprise-grade resilience against failures. Because FlashBlade uses parity encoding instead of replicas for durability, Tiered Storage requires less raw capacity.

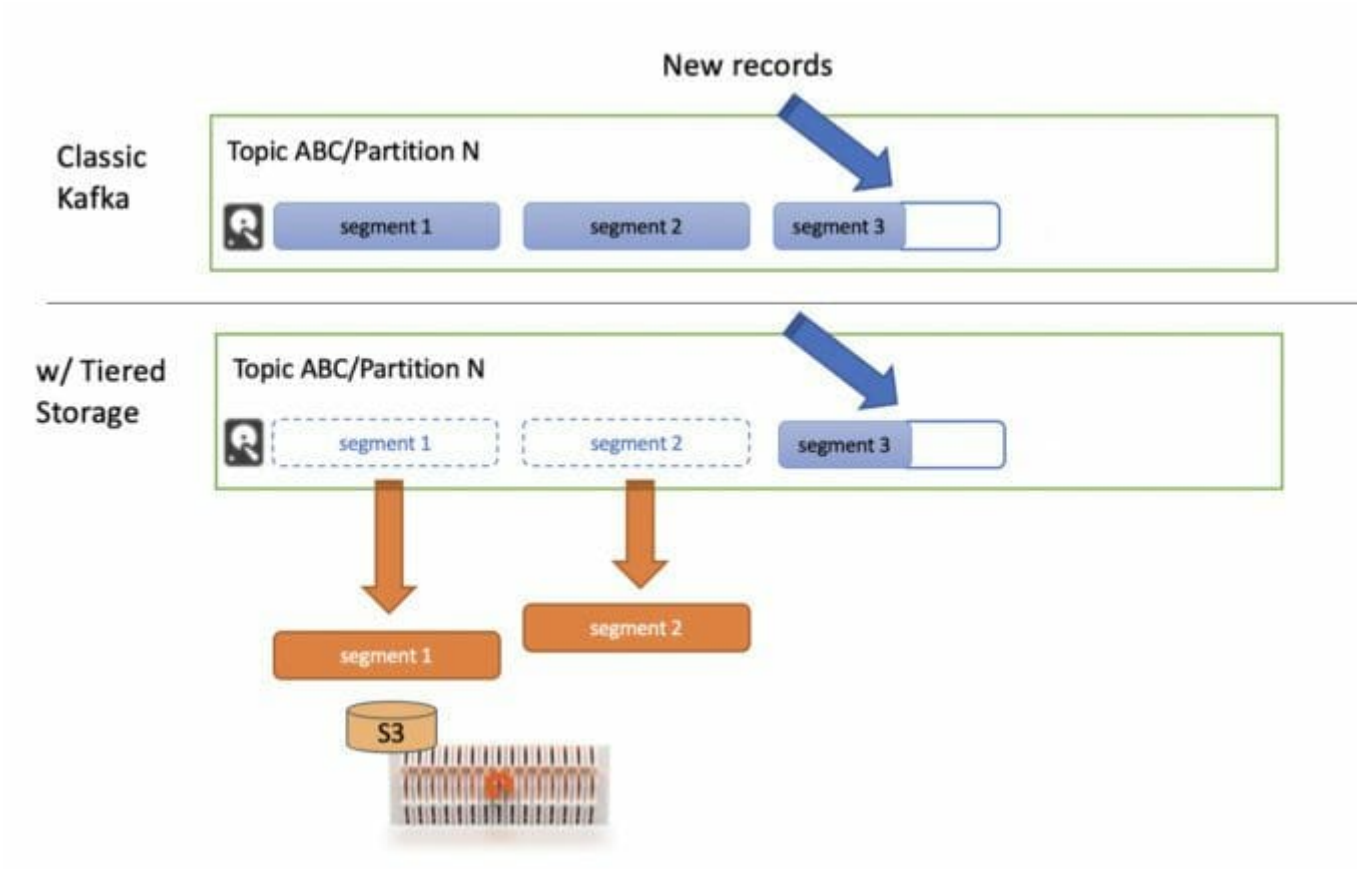


Figure 2: Illustration of how Kafka log segments are offloaded to Tiered Storage.

In the Tiered Storage implementation, the **local hotset** delays the deletion of segments for a specified time period. But because FlashBlade is a high-performance object store, this additional layer of caching (beyond DRAM) can be disabled to reduce local capacity required on brokers.

Instead of segments stored locally, brokers now keep metadata pointers to the data on the FlashBlade object store. When the broker hosting a partition fails, the new leader can simply read segment data directly from the object store. The net effect is that *any* broker can serve requests for a tiered partition with the same low-latency FlashBlade performance.

FlashBlade achieves high resiliency of its internal servers in a very similar manner. The FlashBlade low-level database shards all data and metadata, with a logical *authority* process responsible for each partition. The *authority* itself is a stateless, containerized version of a storage controller and can be run on any of the internal servers and still have access to all NVRAM and NAND flash in the FlashBlade. And similar to Tiered Storage, handling storage expansions or failures is very fast because only a constant amount of in-memory state needs to be rebuilt, as opposed to large amounts of data needing to be rebalanced.

The alternative for building a large Kafka cluster is the classic architecture, which stores all data on broker-local storage with replicas. This means that to store 1PB of data requires 3PB of raw local storage. The challenges here are 1) large numbers of brokers are required in order to physically house all the necessary drives, 2) it quickly becomes expensive to add more servers, especially if needing the performance of SSDs, and 3) more nodes means slower software upgrades, more node failures, and longer rebalancing times.

Understanding Tiered Storage IO Workloads

Storage

To understand Tiered Storage better, I will next break down the storage IO in a typical Kafka streaming workload.

A background streaming workload results in only writes to both broker-local storage and Tiered Storage. The streaming consumers are able to consistently read from memory only, requiring no storage IO. Further, the Tier Archiver is also able to read from memory and avoid reading from the broker local storage. The reason for choosing a smaller *segment.size* configuration of 100MB instead of 1GB is to ensure that the segment data is likely to still be in memory when written to Tiered Storage.

As an example, consider a streaming ingress of 1GB/s. The brokers write an aggregate of 3GB/s of data to local storage as a result of the standard replication-factor of 3. The Tier Archiver component within Kafka writes completed log

segments to FlashBlade at an average of 1GB/s, though in practice writes occur in short bursts of 3GB/s. These bursts are due to the fact that load is distributed evenly across partitions, resulting in a segment being completed in all partitions around the same time.

Network

Each Kafka message needs to traverse on the following network hops: 1) ingress from producer, 2) two additional copies for replicas, 3) egress to consumer, and 4) tiered storage traffic. The result is a 5x multiplier.

More generally:

```
networking_multiplier = replication_factor + 1 +
num_consumer_groups
```

Where the “1” term above corresponds to the copy made to Tiered Storage and the ingress traffic from producers is included as part of the “replication_factor” variable. This calculation is specific to a single topic so a cluster-wide total is simply a weighted average over all topics.

As an example again, if we assume a 1GB/s input stream with replication-factor=3 and one following consumer group, the total networking bandwidth required under normal conditions will be 5GB/s. In this example case, Tiered Storage increases networking bandwidth demands by 25%.

Objects Created

Using s5cmd to list the objects on the Tiered Storage backend gives insight into how Kafka stores log segment as individual objects.

```
$ s5cmd --endpoint-url https://$FB_IP ls s3://kafka/*
2020/10/02 05:58:54 8 0/-0GFCOcFSDGD-
p37slsw/84/00000000000090550151_0_v0.epoch-state
2020/10/02 05:58:54 179288 0/-0GFCOcFSDGD-
p37slsw/84/00000000000090550151_0_v0.offset-index
2020/10/02 05:58:54 10 0/-0GFCOcFSDGD-
p37slsw/84/00000000000090550151_0_v0.producer-state
2020/10/02 05:58:54 104857556 0/-0GFCOcFSDGD-
p37slsw/84/00000000000090550151_0_v0.segment
2020/10/02 05:58:54 219744 0/-0GFCOcFSDGD-
p37slsw/84/00000000000090550151_0_v0.timestamp-index
```

For each segment offloaded to Tiered Storage, we see five distinct objects; the

object sizes are bolded in each line. The ".segment" object is the primary record data, with size closely matching the "segment.size" parameter (100MB). There are also smaller metadata objects: epoch-state, offset-index, producer-state, and timestamp-index. This further motivates why an object store needs to perform well with both large and small object sizes.

Deploying Tiered Storage with FlashBlade

Confluent Platform with Tiered Storage can be run in any deployment model: bare metal, VMs, containers, or Kubernetes, as long as you have the necessary license.

Capacity planning for Tiered Storage is straightforward as only one copy of the data is retained on Tiered Storage. As an example to calculate capacity needed, a 100MB/s input topic (measured post-compression) with one week of retention requires 60TB of usable capacity on a FlashBlade. In contrast, using the classic architecture and replication factor of 3 will require 180TB of raw storage. Then as Kafka usage grows to multiple petabytes, the FlashBlade backend can be non-disruptively scaled as needed.

A small amount of local storage is still required on the brokers to store partial log segments. Specifically, the maximum capacity needed is a function of the total number of partitions and segment size. The recommended segment size is 100MB, therefore, a relatively large cluster with 10k partitions would then need at most 1TB of local storage *spread across all brokers*.

Configuring Tiered Storage with FlashBlade

Please consult the Confluent recommendations for Tiered Storage for general guidance; this section will focus on using FlashBlade as a Tiered Storage backend.

I have shared an example yaml definition of a Kubernetes Statefulset to run Kafka with Tiered Storage from an internal use case at Pure. In this example, I use a ConfigMap to configure parameters as environment variables. While this example is specific to Kubernetes, these configuration options are available in any other deployment environment via either the *server.properties* file or via environment variables.

```
Enable Tiered Storage:

CONFLUENT_TIER_FEATURE=true

CONFLUENT_TIER_ENABLE=true
```

```
CONFLUENT_TIER_BACKEND=S3

CONFLUENT_TIER_S3_BUCKET=<BUCKET_NAME>

CONFLUENT_TIER_S3_REGION=<REGION>
```

Note that the region field needs to be set to any *valid* value but is not used when connecting to the FlashBlade.

The endpoint override configuration enables Kafka to connect to a FlashBlade data VIP for Tiered Storage endpoint:

```
CONFLUENT_TIER_S3_AWS_ENDPOINT_OVERRIDE=${ENDPOINT}
```

As an example, if the data VIP for the FlashBlade is 10.62.64.200, then the configuration would look like this:

```
CONFLUENT_TIER_S3_AWS_ENDPOINT_OVERRIDE=https://10.62.64.200
```

Next, you need to create S3 access keys on the FlashBlade and then pass those keys through environment variables:

```
AWS_ACCESS_KEY_ID=<YOUR_ACCESS_KEY_ID>

AWS_SECRET_ACCESS_KEY=<YOUR_SECRET_ACCESS_KEY>
```

In my Kubernetes environment, I use a secret to manage the access keys. Creation of the secret is a one-time operation:

```
kubectl create secret generic my-s3-keys --from-
literal=AWS_ACCESS_KEY_ID="$ACCESS" --from-
literal=AWS_SECRET_ACCESS_KEY="$SECRET"
```

The access keys are then available to the Confluent Kafka pod through environment variables:

```
env:

- name: AWS_ACCESS_KEY_ID

 valueFrom:

   secretKeyRef:

     name: my-s3-keys

     key: access-key
```

```
- name: AWS_SECRET_ACCESS_KEY

 valueFrom:

  secretKeyRef:

     name: my-s3-keys

     key: secret-key
```

This secret can be used by other applications, e.g., Apache Spark or PrestoSql, using the same FlashBlade for other purposes.

Since multiple distinct Kafka clusters can share the same FlashBlade as a Tiered Storage backend, I create one bucket per Kafka cluster. Bucket creation can be easily automated using either the FlashBlade REST API or a tool like Ansible.

For example, with Ansible, install the FlashBlade collection:

```
ansible-galaxy collection install purestorage.flashblade
```

And then an Ansible task to create the bucket requires the management IP and Token for access to the FlashBlade and the bucket name. The bucket is then created under the given account.

```
collections:

- purestorage.flashblade

tasks:

- name: Create Bucket

 purefb_bucket:

   name: “{{ BUCKET }}”

  account: “{{ ACCOUNT }}”

   fb_url: “{{ FB_MGMTVIP }}”

   api_token: “{{ FB_TOKEN }}”

   state: present
```

As recommended by Confluent, set the parameter *segment.size* to 100MB. This

setting controls the size of data blocks that are written to Tiered Storage.

```
KAFKA_LOG_SEGMENT_BYTES=104857600
```

The advantage of the smaller segment size (the default is 1GB) is that the data is most likely to still be cached in broker DRAM by the time it needs to be written to Tiered Storage, avoiding reads from the local drives.

Even though the FlashBlade also compresses all data if possible, I configure compression at the producer if possible or otherwise the broker. Earlier compression reduces load on local storage and the network.

The tier-archiver and tier-fetcher are responsible for writing to and reading from tiered storage respectively. For high performance, it's important to increase the threads available:

```
CONFLUENT_TIER_ARCHIVER_NUM_THREADS=8
```

```
CONFLUENT_TIER_FETCHER_NUM_THREADS=16
```

The "hotset" configuration controlled by *tier.local.hotset.ms* or *tier.local.hotset.bytes* keeps a segment on local storage for a specified amount of time, even after the data has been offloaded to Tiered Storage. As discussed previously, this can be disabled with FlashBlade as backend in order to reduce local storage requirements:

```
CONFLUENT_TIER_LOCAL_HOTSET_MS=0
```

The default best practices involve using three replicas for each topic (*–replication-factor 3*), such that each incoming message is written on three brokers. By default, we recommend following this same practice when using Tiered Storage.

Keep in mind that currently, the Tiered Storage feature supports the default "cleanup.policy" of "delete" but does not support compacted topics.

Experimental Results

To test the performance benefits of Tiered Storage and FlashBlade, I used the testbed diagrammed below with six brokers, eight load generators, and a FlashBlade with 160TB usable capacity. Each broker has local NVME drives and all components are connected over 100Gb Ethernet.

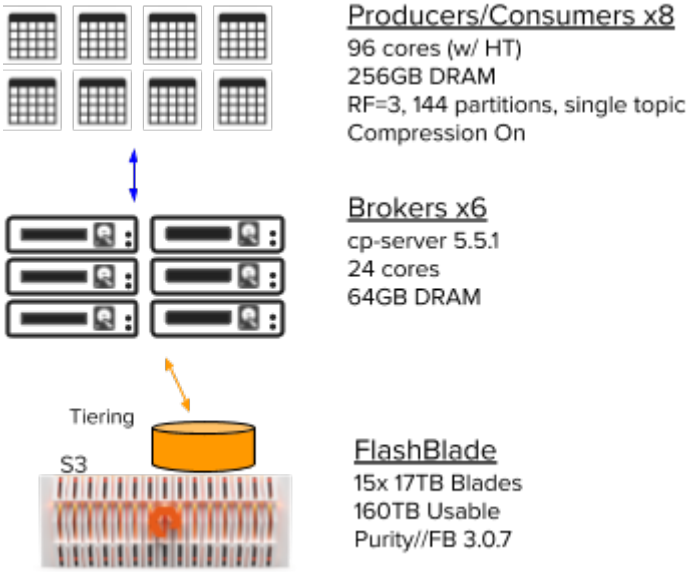


Figure 3: Experimental testbed configuration.

I also compared with clusters set up in AWS with spec matching as closely as possible, except for the unavoidable shared networking within AWS.

Historical Query Test

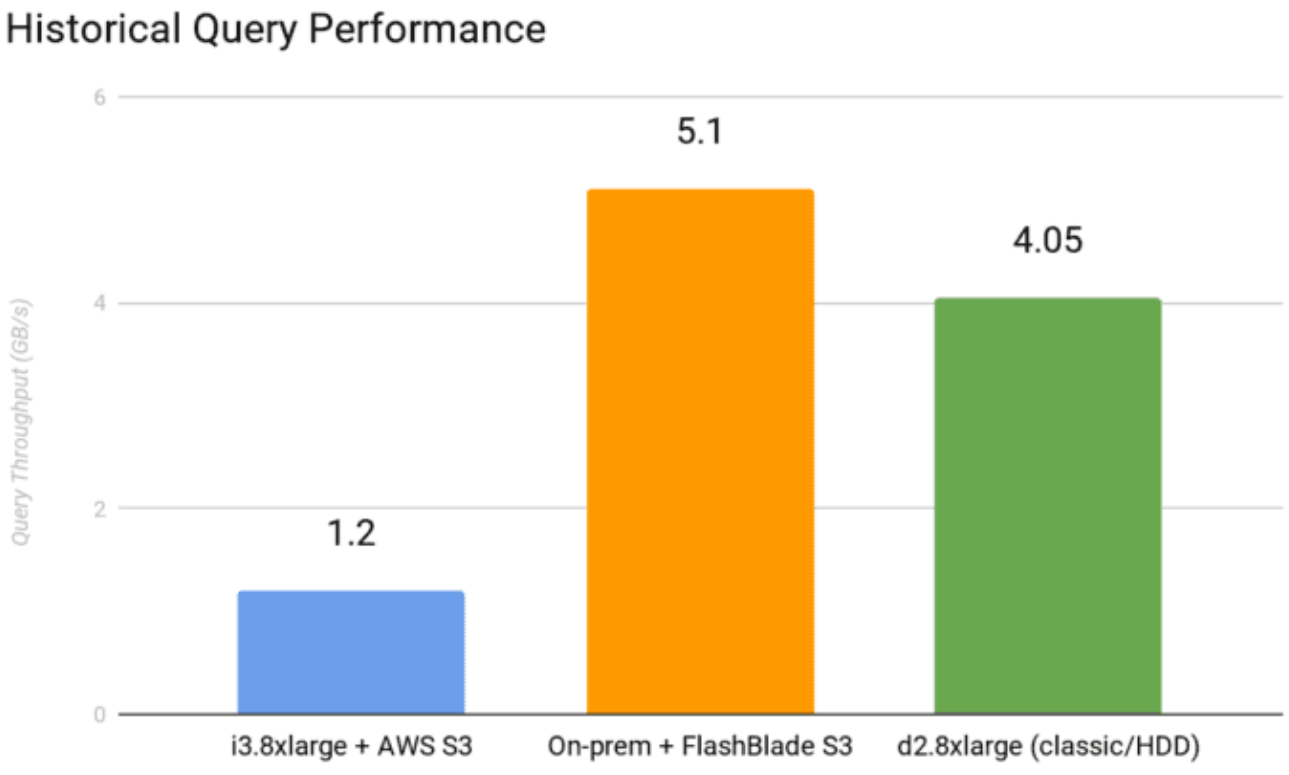
Since FlashBlade is an all-flash object store, the first experiment tests the performance to read back from Tiered Storage for a historical query. For a realistic scenario, I created a steady-state streaming workload of 500MB/s ingress and egress (1GB/s total traffic). I then added a historical query from a new consumer group and measured the read throughput for this consumer group.

For this test, I used the Confluent Platform built-in load generators *kafka-producer-perf-test* and *kafka-consumer-perf-test*. I then compared against a matching Tiered Storage config using i3.8xlarge instances and a classic Kafka architecture (no Tiered Storage) using d2.8xlarge HDD-based instances.

The Grafana dashboard below shows the shape of this workload, a background streaming workload with an overlay of high read throughput for the historical query.



The results show that FlashBlade is 4x faster as a Tiered Storage backend than AWS S3 as measured by the read throughput of the historical query.



An important caveat to the above results is that the classic architecture with spinning disks did not keep up with the 500MB/s ingress, dropping to as low as

80MB/s ingress during the historical query.

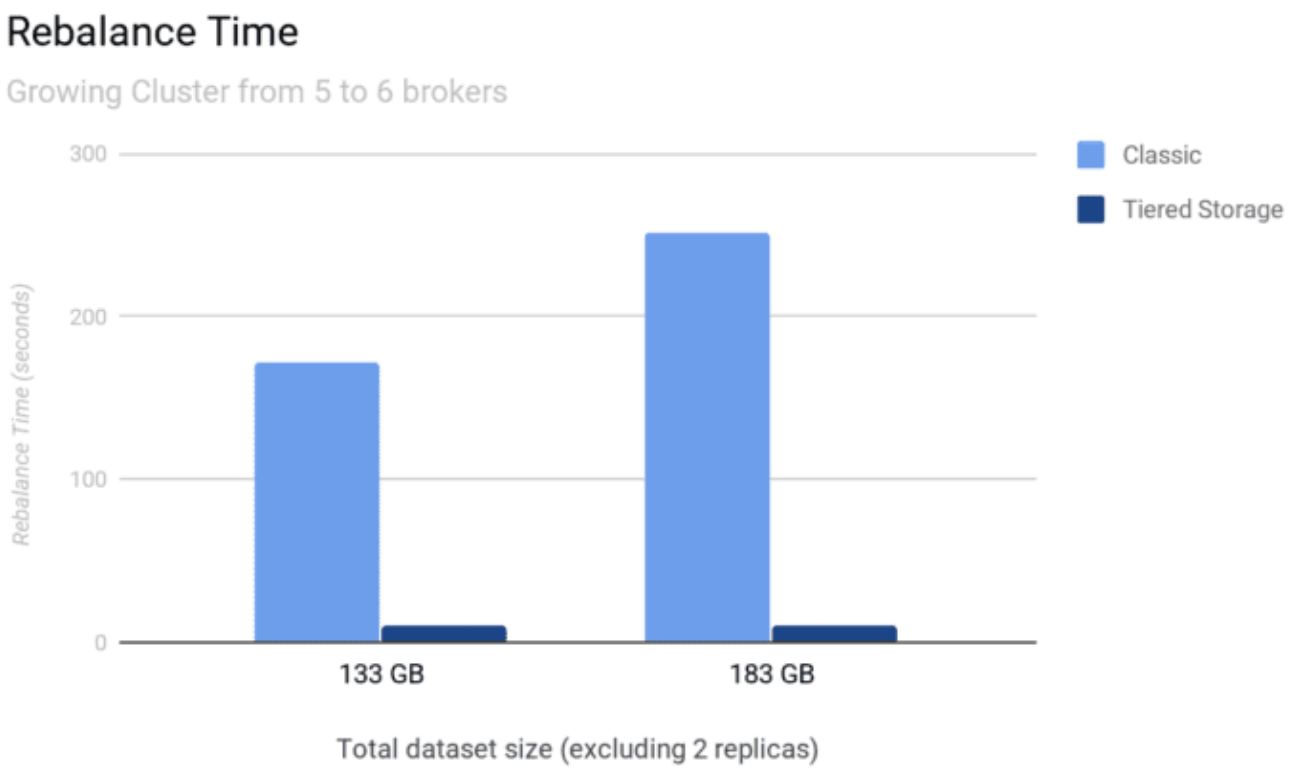
Rebalance Test

The next test demonstrates the operational simplicity gained by switching to Tiered Storage by measuring time taken to scale by adding a new broker. Many cluster operations, like scaling or handling node failures, require rebalancing data across brokers. In classic configurations, the amount of data needing to be rebalanced grows as the cluster grows, but in Tiered Storage, the rebalancing is bounded to a small, constant amount.

Rebalancing Kafka clusters is even easier with the newly released Self-Balancing Clusters feature, which automates rebalancing based on cluster topology changes or uneven load.

The experimental results validate the new architecture; rebalances in Tiered Storage take **seconds** and are constant as the cluster grows, whereas rebalances with classic architecture take minutes and grow linearly as the cluster grows. Extrapolating the results below means that a classic 150TB Kafka cluster would take 57 hours to rebalance, whereas Tiered Storage would take less than one minute!

These results use an effectively infinite throttle (10GB/s) on rebalance traffic, meaning that the rebalance times can be significantly worse in realistic scenarios where a throttle slows down rebalances to protect streaming traffic.



Due to the architecture of Tiered Storage, it is possible to provide an upper bound on rebalance time as:

```
max_rebalance_time = (segment_size * num_partitions) /
rebalance_throttle
```

Data that has not yet been tiered is limited to the set of open segments, i.e., segment_size times the total number of partitions across all topics. As an example, if a cluster has 1,000 partitions on each broker and 100MB segment_size, then the upper bound of data to be rebalanced is 100MB * 1,000 = 100GB. The amount of time taken to rebalance is then a function of the throttled rebalance rate, which defaults to 10MB/s, and for this example is close to three hours.

Data Pipeline: Kafka and Elasticsearch

Kafka frequently sits in front of another downstream data store for transformed or filtered messages. For log analytics, Elasticsearch is a powerful way of making streaming data searchable for unpredictable usage.

The standard way to run Kafka and Elasticsearch together would be two different silos of infrastructure, both compute and storage dedicated to either Kafka or Elasticsearch, plus additional servers for filebeats. With FlashBlade, both clusters can run on the same storage infrastructure.

A simple filebeats configuration reads from Kafka and pushes data into

Elasticsearch:

```
filebeat.inputs:

- type: kafka

 hosts:

 - confluentkafka-0:9092

 - confluentkafka-1:9092

 - confluentkafka-2:9092

 topics: ["flog"]

 group_id: "flogbeats"

setup.template.settings:

 index.number_of_shards: 36

 index.number_of_replicas: 0

 index.refresh_interval: 30s

output.elasticsearch:

 hosts: '${ELASTICSEARCH_HOST}:9200'

 worker: 2

 bulk_max_size: 4096
```

The Grafana dashboard below shows a simultaneous mix of **three workloads:** streaming Kafka, Elasticsearch indexing, and a historical Kafka query.



The above IO profile represents multidimensional performance: 1GB/s of S3 writes, 4GB/s of S3 reads, and 200-300MB/s of mixed NFS traffic from Elasticsearch. By being able to run both Kafka and Elasticsearch on the same infrastructure, Kubernetes, and FlashBlade, I also have the ability to scale flexibly in any dimension. For example, I can double Kafka retention for a period of time, then revert and instead double Elasticsearch retention, all without changes to the underlying infrastructure.

I have also posted a demo video walking through the benefits of Tiered Storage, including the tests described previously.

Summary

By disaggregating compute and storage, Confluent Tiered Storage and FlashBlade make scaling Kafka dramatically simpler. Performant S3 storage offloads storage from Kafka brokers, improving operational simplicity and rebalancing performance, as well as powering historical queries across all historical data. Confluent has pushed Kafka a huge leap forward with this cloud-native architecture; go and build more reliable data pipelines with *more* Kafka.