Best practices for Confluent Kafka

NetApp Solutions

NetApp
October 05, 2022

Table of Contents

Best practices for Confluent Kafka

TR-4912: Best practice guidelines for Confluent Kafka tiered storage with NetApp

Karthikeyan Nagalingam, Joseph Kandatilparambil, NetApp
Rankesh Kumar, Confluent

Apache Kafka is a community-distributed event-streaming platform capable of handling trillions of events a day. Initially conceived as a messaging queue, Kafka is based on an abstraction of a distributed commit log. Since it was created and open-sourced by LinkedIn in 2011, Kafka has evolved from a messages queue to a full-fledged event-streaming platform. Confluent delivers the distribution of Apache Kafka with the Confluent Platform. The Confluent Platform supplements Kafka with additional community and commercial features designed to enhance the streaming experience of both operators and developers in production at a massive scale.

This document describes the best-practice guidelines for using Confluent Tiered Storage on a NetApp's Object storage offering by providing the following content:

- Confluent verification with NetApp Object storage – NetApp StorageGRID
- Tiered storage performance tests
- Best-practice guidelines for Confluent on NetApp storage systems

Why Confluent Tiered Storage?

Confluent has become the default real-time streaming platform for many applications, especially for big data, analytics, and streaming workloads. Tiered Storage enables users to separate compute from storage in the Confluent platform. It makes storing data more cost effective, enables you to store virtually infinite amounts of data and scale workloads up (or down) on-demand, and makes administrative tasks like data and tenant rebalancing easier. S3 compatible storage systems can take advantage of all these capabilities to democratize data with all events in one place, eliminating the need for complex data engineering. For more info on why you should use tiered storage for Kafka, check this article by Confluent.

Why NetApp StorageGRID for tiered storage?

StorageGRID is an industry-leading object storage platform by NetApp. StorageGRID is a software-defined, object-based storage solution that supports industry-standard object APIs, including the Amazon Simple Storage Service (S3) API. StorageGRID stores and manages unstructured data at scale to provide secure, durable object storage. Content is placed in the right location, at the right time, and on the right storage tier, optimizing workflows and reducing costs for globally distributed rich media.

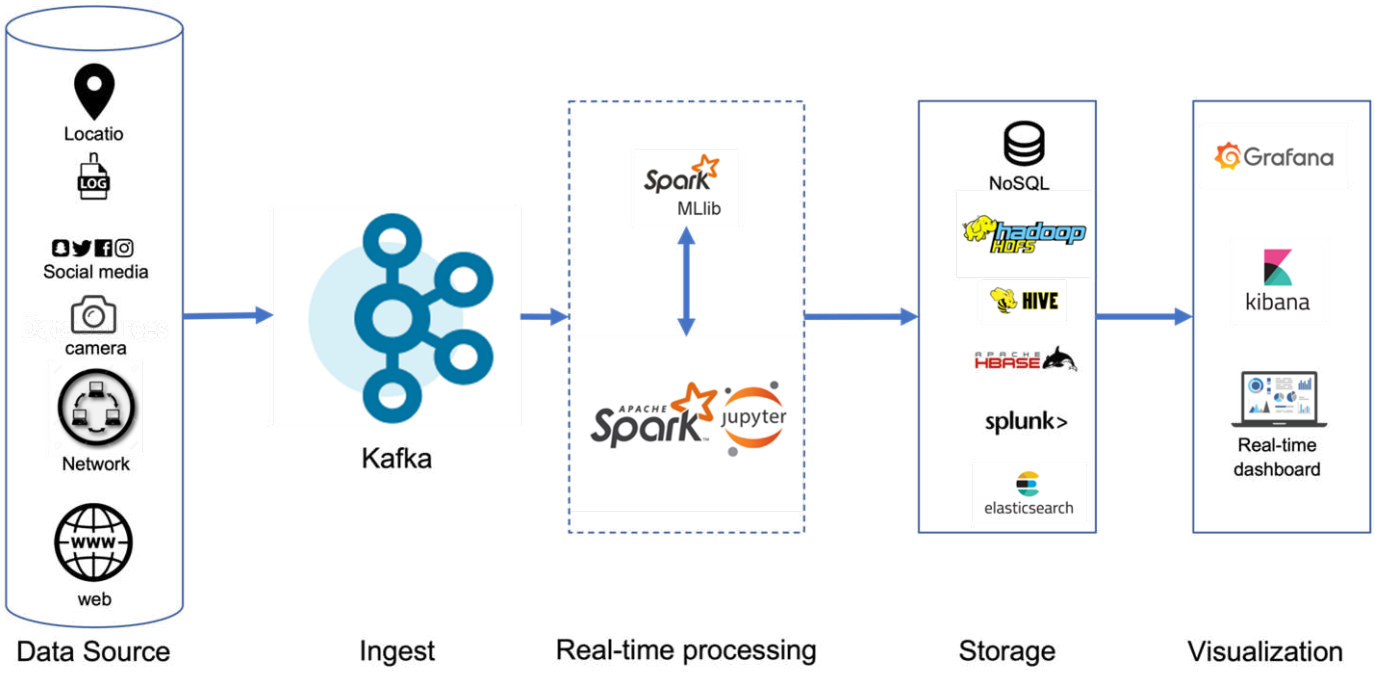
The greatest differentiator for StorageGRID is its Information Lifecycle Management (ILM) policy engine that enables policy-driven data lifecycle management. The policy engine can use metadata to manage how data is stored across its lifetime to initially optimize for performance and automatically optimize for cost and durability as data ages.

Enabling Confluent Tiered Storage

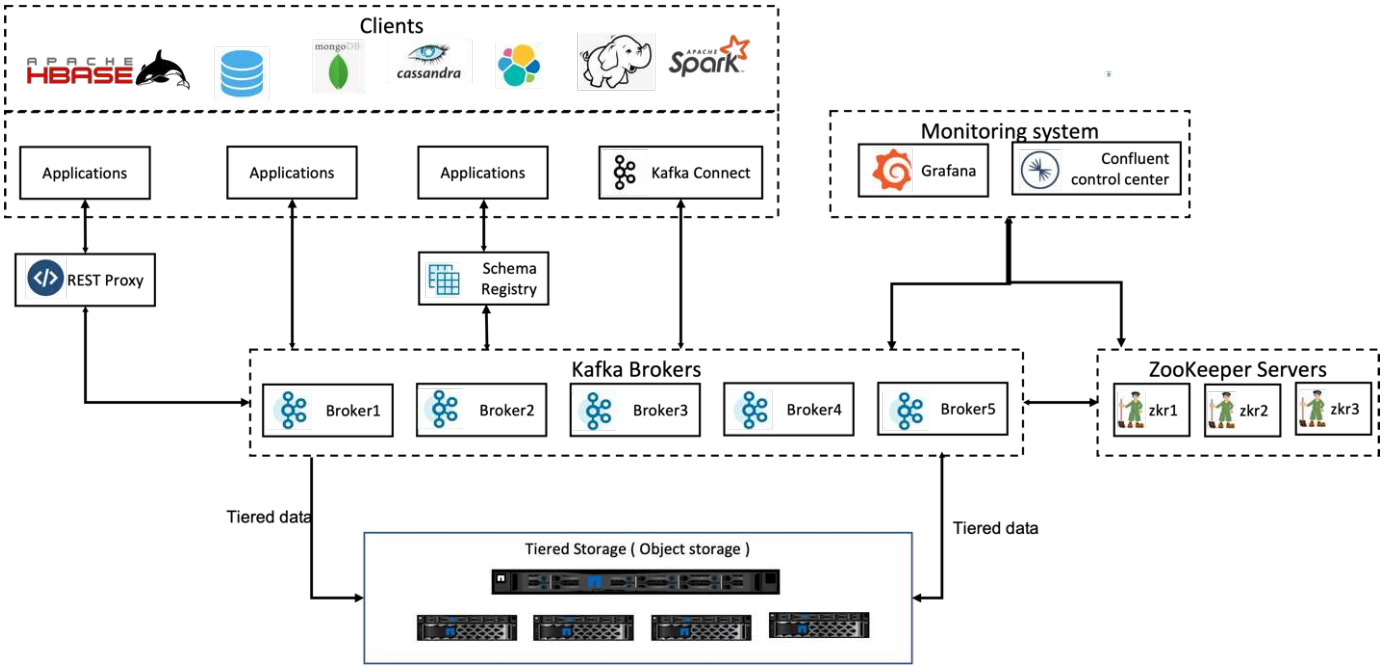
The basic idea of tiered storage is to separate the tasks of data storage from data processing. With this separation, it becomes much easier for the data storage tier and the data processing tier to scale independently.

A tiered storage solution for Confluent must contend with two factors. First, it must work around or avoid common object store consistency and availability properties, such as inconsistencies in LIST operations and occasional object unavailability. Secondly, it must correctly handle the interaction between tiered storage and Kafka's replication and fault tolerance model, including the possibility of zombie leaders continuing to tier offset ranges. NetApp Object storage provides both the consistent object availability and HA model make the tired storage available to tier offset ranges. NetApp object storage provides consistent object availability and an HA model to make the tired storage available to tier offset ranges.

With tiered storage, you can use high-performance platforms for low-latency reads and writes near the tail of your streaming data, and you can also use cheaper, scalable object stores like NetApp StorageGRID for high-throughput historical reads. We also have technical solution for Spark with netapp storage controller and details are here. The following figure shows how Kafka fits into a real-time analytics pipeline.



The following figure depicts how NetApp StorageGRID fits in as Confluent Kafka's object storage tier.



Next: Solution architecture details.

Solution architecture details

Previous: Introduction.

This section covers the hardware and software used for Confluent verification. This information is applicable to Confluent Platform deployment with NetApp storage. The following table covers the tested solution architecture and base components.

Solution components	Details
Confluent Kafka version 6.2	• Three zookeepers • Five broker servers • Five tools servers • One Grafana • One control center
Linux (ubuntu 18.04)	All servers
NetApp StorageGRID for tiered storage	• StorageGRID software • 1 x SG1000 (load balancer) • 4 x SGF6024 • 4 x 24 x 800 SSDs • S3 protocol • 4 x 100GbE (network connectivity between broker and StorageGRID instances)

Solution components	Details
15 Fujitsu PRIMERGY RX2540 servers	Each equipped with: * 2 CPUs, 16 physical cores total * Intel Xeon * 256GB physical memory * 100GbE dual port

Next: Technology overview.

Technology overview

Previous: Solution architecture details.

This section describes the technology used in this solution.

NetApp StorageGRID

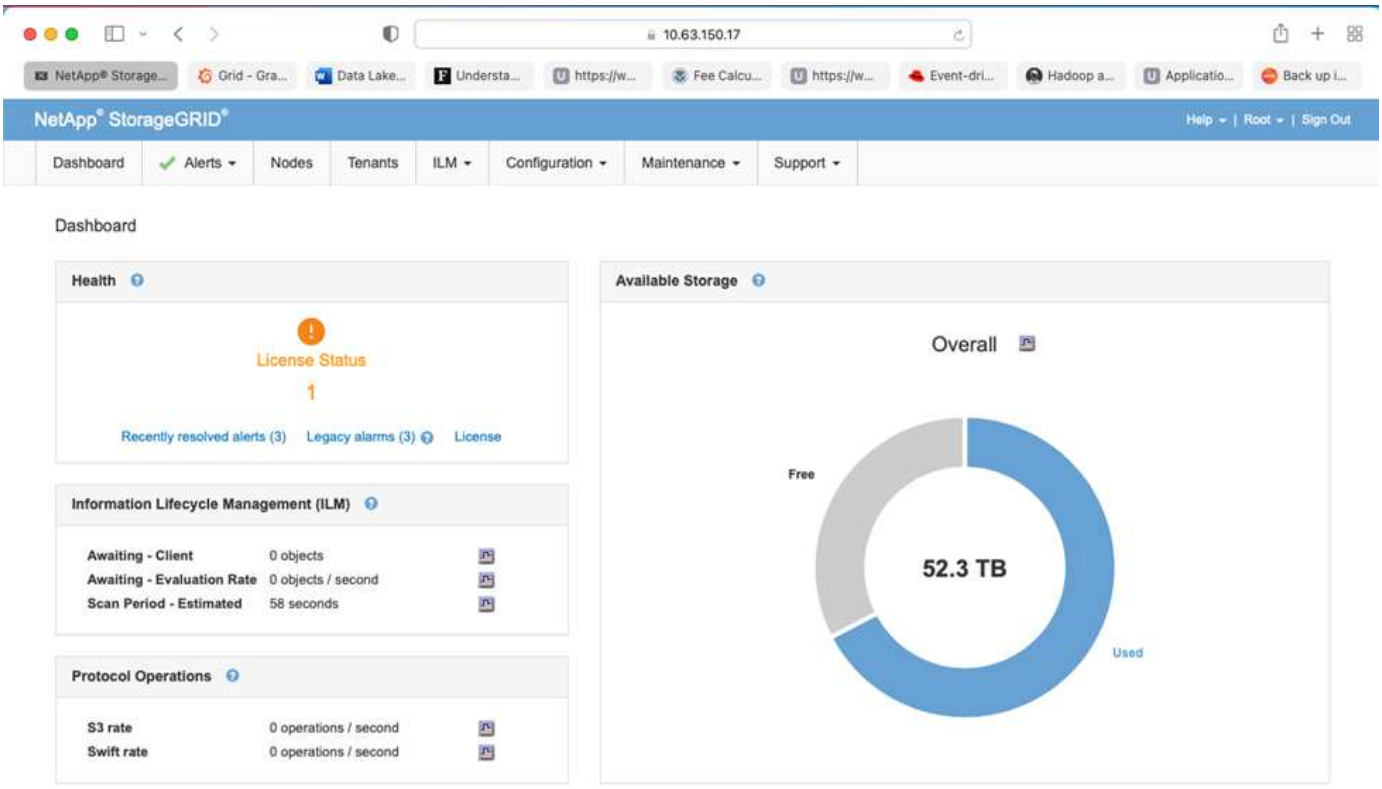
NetApp StorageGRID is a high-performance, cost-effective object storage platform. By using tiered storage, most of the data on Confluent Kafka, which is stored in local storage or the SAN storage of the broker, is offloaded to the remote object store. This configuration results in significant operational improvements by reducing the time and cost to rebalance, expand, or shrink clusters or replace a failed broker. Object storage plays an important role in managing data that resides on the object store tier, which is why picking the right object storage is important.

StorageGRID offers intelligent, policy-driven global data management using a distributed, node-based grid architecture. It simplifies the management of petabytes of unstructured data and billions of objects through its ubiquitous global object namespace combined with sophisticated data management features. Single-call object access extends across sites and simplifies high availability architectures while ensuring continual object access, regardless of site or infrastructure outages.

Multitenancy allows multiple unstructured cloud and enterprise data applications to be securely serviced within the same grid, increasing the ROI and use cases for NetApp StorageGRID. You can create multiple service levels with metadata-driven object lifecycle policies, optimizing durability, protection, performance, and locality across multiple geographies. Users can adjust data management policies and monitor and apply traffic limits to realign with the data landscape nondisruptively as their requirements change in ever-changing IT environments.

Simple management with Grid Manager

The StorageGRID Grid Manager is a browser-based graphical interface that allows you to configure, manage, and monitor your StorageGRID system across globally distributed locations in a single pane of glass.



You can perform the following tasks with the StorageGRID Grid Manager interface:

- Manage globally distributed, petabyte-scale repositories of objects such as images, video, and records.
- Monitor grid nodes and services to ensure object availability.
- Manage the placement of object data over time using information lifecycle management (ILM) rules. These rules govern what happens to an object's data after it is ingested, how it is protected from loss, where object data is stored, and for how long.
- Monitor transactions, performance, and operations within the system.

Information Lifecycle Management policies

StorageGRID has flexible data management policies that include keeping replica copies of your objects and using EC (erasure coding) schemes like 2+1 and 4+2 (among others) to store your objects, depending on specific performance and data protection requirements. As workloads and requirements change over time, it's common that ILM policies must change over time as well. Modifying ILM policies is a core feature, allowing StorageGRID customers to adapt to their ever-changing environment quickly and easily. Please check the ILM policy and ILM rules setup in StorageGRID.

Performance

StorageGRID scales performance by adding more storage nodes, which can be VMs, bare metal, or purpose-built appliances like the SG5712, SG5760, SG6060, or SGF6024. In our tests, we exceeded the Apache Kafka key performance requirements with a minimum-sized, three-node grid using the SGF6024 appliance. As customers scale their Kafka cluster with additional brokers, they can add more storage nodes to increase performance and capacity.

Load balancer and endpoint configuration

Admin nodes in StorageGRID provide the Grid Manager UI (user interface) and REST API endpoint to view,

configure, and manage your StorageGRID system, as well as audit logs to track system activity. To provide a highly available S3 endpoint for Confluent Kafka tiered storage, we implemented the StorageGRID load balancer, which runs as a service on admin nodes and gateway nodes. In addition, the load balancer also manages local traffic and talks to the GSLB (Global Server Load Balancing) to help with disaster recovery.

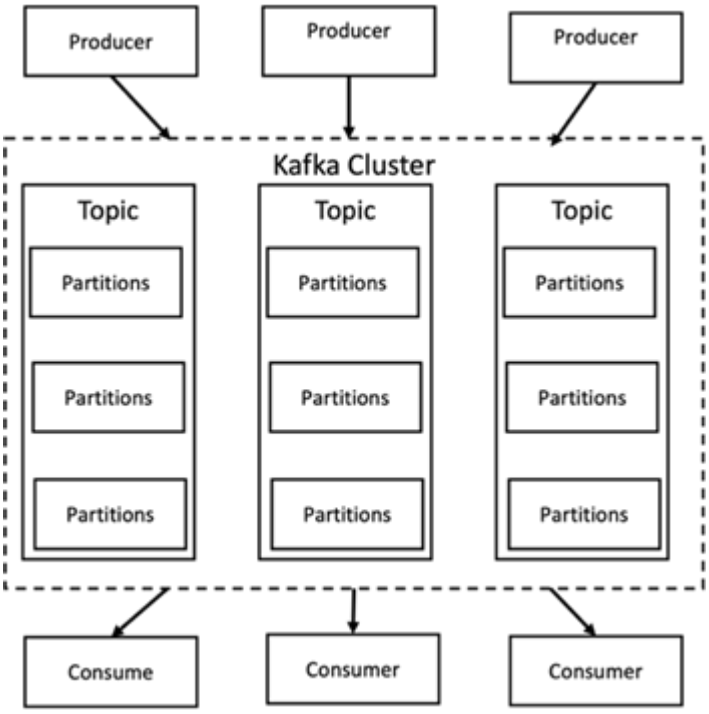
To further enhance endpoint configuration, StorageGRID provides traffic classification policies built into the admin node, lets you monitor your workload traffic, and applies various quality-of-service (QoS) limits to your workloads. Traffic classification policies are applied to endpoints on the StorageGRID Load Balancer service for gateway nodes and admin nodes. These policies can assist with traffic shaping and monitoring.

Traffic classification in StorageGRID

StorageGRID has built-in QoS functionality. Traffic classification policies can help monitor different types of S3 traffic coming from a client application. You can then create and apply policies to put limits on this traffic based on in/out bandwidth, the number of read/write concurrent requests, or the read/write request rate.

Apache Kafka

Apache Kafka is a framework implementation of a software bus using stream processing written in Java and Scala. It's aimed to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka can connect to an external system for data export and import through Kafka Connect and provides Kafka streams, a Java stream processing library. Kafka uses a binary, TCP-based protocol that is optimized for efficiency and relies on a "message set" abstraction that naturally groups messages together to reduce the overhead of the network roundtrip. This enables larger sequential disk operations, larger network packets, and contiguous memory blocks, thereby enabling Kafka to turn a bursty stream of random message writes into linear writes. The following figure depicts the basic data flow of Apache Kafka.



Kafka stores key-value messages that come from an arbitrary number of processes called producers. The data can be partitioned into different partitions within different topics. Within a partition, messages are strictly ordered by their offsets (the position of a message within a partition) and indexed and stored together with a timestamp. Other processes called consumers can read messages from partitions. For stream processing, Kafka offers the Streams API that allows writing Java applications that consume data from Kafka and write results back to Kafka. Apache Kafka also works with external stream processing systems such as Apache

Apex, Apache Flink, Apache Spark, Apache Storm, and Apache NiFi.

Kafka runs on a cluster of one or more servers (called brokers), and the partitions of all topics are distributed across the cluster nodes. Additionally, partitions are replicated to multiple brokers. This architecture allows Kafka to deliver massive streams of messages in a fault-tolerant fashion and has allowed it to replace some of the conventional messaging systems like Java Message Service (JMS), Advanced Message Queuing Protocol (AMQP), and so on. Since the 0.11.0.0 release, Kafka offers transactional writes, which provide exactly once stream processing using the Streams API.

Kafka supports two types of topics: regular and compacted. Regular topics can be configured with a retention time or a space bound. If there are records that are older than the specified retention time or if the space bound is exceeded for a partition, Kafka is allowed to delete old data to free storage space. By default, topics are configured with a retention time of 7 days, but it's also possible to store data indefinitely. For compacted topics, records don't expire based on time or space bounds. Instead, Kafka treats later messages as updates to older message with the same key and guarantees never to delete the latest message per key. Users can delete messages entirely by writing a so-called tombstone message with the null value for a specific key.

There are five major APIs in Kafka:

- **Producer API.** Permits an application to publish streams of records.
- **Consumer API.** Permits an application to subscribe to topics and processes streams of records.
- **Connector API.** Executes the reusable producer and consumer APIs that can link the topics to the existing applications.
- **Streams API.** This API converts the input streams to output and produces the result.
- **Admin API.** Used to manage Kafka topics, brokers and other Kafka objects.

The consumer and producer APIs build on top of the Kafka messaging protocol and offer a reference implementation for Kafka consumer and producer clients in Java. The underlying messaging protocol is a binary protocol that developers can use to write their own consumer or producer clients in any programming language. This unlocks Kafka from the Java Virtual Machine (JVM) ecosystem. A list of available non-Java clients is maintained in the Apache Kafka wiki.

Apache Kafka use cases

Apache Kafka is most popular for messaging, website activity tracking, metrics, log aggregation, stream processing, event sourcing, and commit logging.

- Kafka has improved throughput, built-in partitioning, replication, and fault-tolerance, which makes it a good solution for large-scale message-processing applications.
- Kafka can rebuild a user's activities (page views, searches) in a tracking pipeline as a set of real-time publish-subscribe feeds.
- Kafka is often used for operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data.
- Many people use Kafka as a replacement for a log aggregation solution. Log aggregation typically collects physical log files off of servers and puts them in a central place (for example, a file server or HDFS) for processing. Kafka abstracts files details and provides a cleaner abstraction of log or event data as a stream of messages. This allows for lower-latency processing and easier support for multiple data sources and distributed data consumption.
- Many users of Kafka process data in processing pipelines consisting of multiple stages, in which raw input data is consumed from Kafka topics and then aggregated, enriched, or otherwise transformed into new topics for further consumption or follow-up processing. For example, a processing pipeline for

recommending news articles might crawl article content from RSS feeds and publish it to an "articles" topic. Further processing might normalize or deduplicate this content and publish the cleansed article content to a new topic, and a final processing stage might attempt to recommend this content to users. Such processing pipelines create graphs of real-time data flows based on the individual topics.

- Event souring is a style of application design for which state changes are logged as a time-ordered sequence of records. Kafka's support for very large stored log data makes it an excellent backend for an application built in this style.
- Kafka can serve as a kind of external commit-log for a distributed system. The log helps replicate data between nodes and acts as a re-syncing mechanism for failed nodes to restore their data. The log compaction feature in Kafka helps support this use case.

Confluent

Confluent Platform is an enterprise-ready platform that completes Kafka with advanced capabilities designed to help accelerate application development and connectivity, enable transformations through stream processing, simplify enterprise operations at scale, and meet stringent architectural requirements. Built by the original creators of Apache Kafka, Confluent expands the benefits of Kafka with enterprise-grade features while removing the burden of Kafka management or monitoring. Today, over 80% of the Fortune 100 are powered by data streaming technology – and most of those use Confluent.

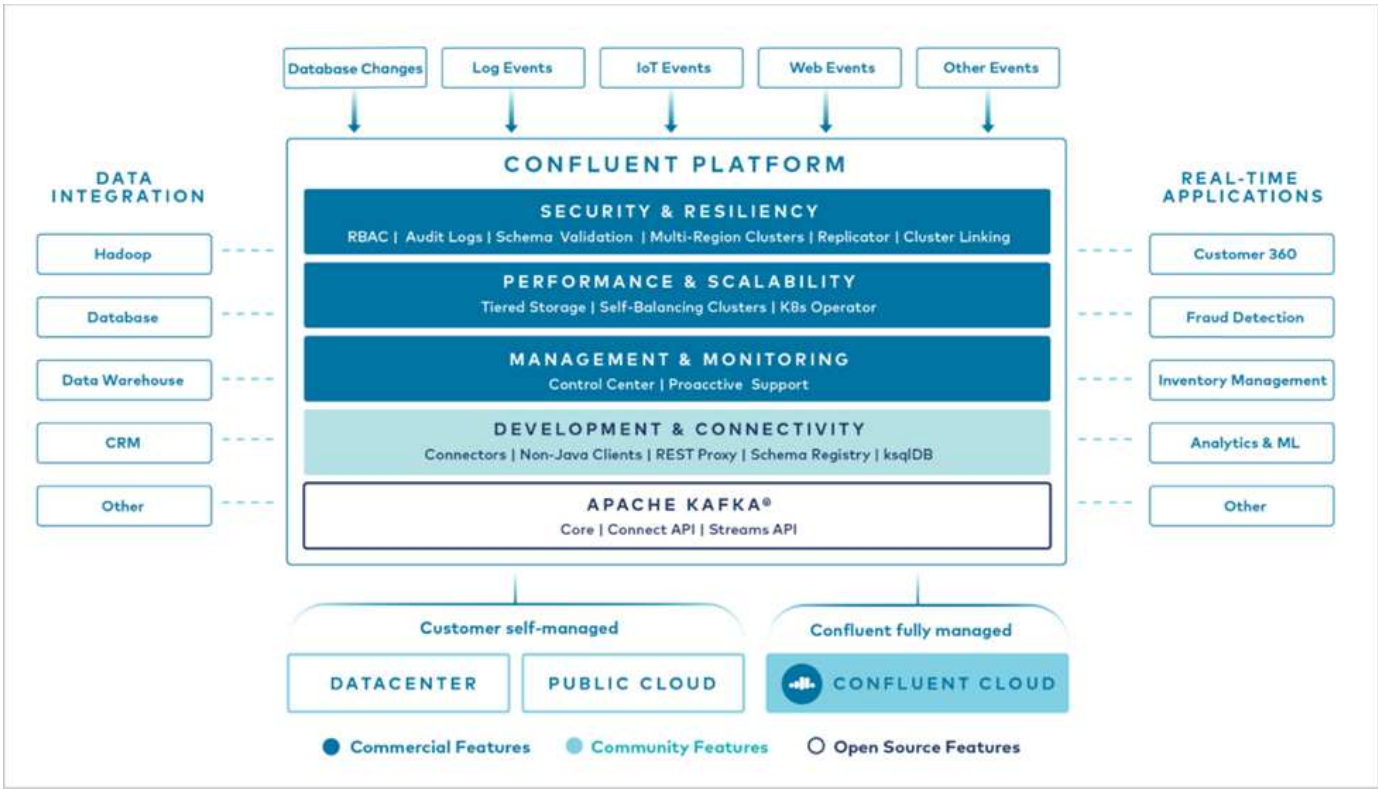
Why Confluent?

By integrating historical and real-time data into a single, central source of truth, Confluent makes it easy to build an entirely new category of modern, event-driven applications, gain a universal data pipeline, and unlock powerful new use cases with full scalability, performance, and reliability.

What is Confluent used for?

Confluent Platform lets you focus on how to derive business value from your data rather than worrying about the underlying mechanics, such as how data is being transported or integrated between disparate systems. Specifically, Confluent Platform simplifies connecting data sources to Kafka, building streaming applications, as well as securing, monitoring, and managing your Kafka infrastructure. Today, Confluent Platform is used for a wide array of use cases across numerous industries, from financial services, omnichannel retail, and autonomous cars, to fraud detection, microservices, and IoT.

The following figure shows Confluent Kafka Platform components.



Overview of Confluent's event streaming technology

At the core of Confluent Platform is Apache Kafka, the most popular open-source distributed streaming platform. The key capabilities of Kafka are as follows:

- Publish and subscribe to streams of records.
- Store streams of records in a fault tolerant way.
- Process streams of records.

Out of the box, Confluent Platform also includes Schema Registry, REST Proxy, a total of 100+ prebuilt Kafka connectors, and ksqlDB.

Overview of Confluent platform's enterprise features

- **Confluent Control Center.** A GUI-based system for managing and monitoring Kafka. It allows you to easily manage Kafka Connect and to create, edit, and manage connections to other systems.
- **Confluent for Kubernetes.** Confluent for Kubernetes is a Kubernetes operator. Kubernetes operators extend the orchestration capabilities of Kubernetes by providing the unique features and requirements for a specific platform application. For Confluent Platform, this includes greatly simplifying the deployment process of Kafka on Kubernetes and automating typical infrastructure lifecycle tasks.
- **Confluent connectors to Kafka.** Connectors use the Kafka Connect API to connect Kafka to other systems such as databases, key-value stores, search indexes, and file systems. Confluent Hub has downloadable connectors for the most popular data sources and sinks, including fully tested and supported versions of these connectors with Confluent Platform. More details can be found here.
- **Self- balancing clusters.** Provides automated load balancing, failure detection and self-healing. It provides support for adding or decommissioning brokers as needed, with no manual tuning.
- **Confluent cluster linking.** Directly connects clusters together and mirrors topics from one cluster to another over a link bridge. Cluster linking simplifies setup of multi-datacenter, multi-cluster, and hybrid

cloud deployments.

- **Confluent auto data balancer.** Monitors your cluster for the number of brokers, the size of partitions, number of partitions, and the number of leaders within the cluster. It allows you to shift data to create an even workload across your cluster, while throttling rebalance traffic to minimize the effect on production workloads while rebalancing.
- **Confluent replicator.** Makes it easier than ever to maintain multiple Kafka clusters in multiple data centers.
- **Tiered storage.** Provides options for storing large volumes of Kafka data using your favorite cloud provider, thereby reducing operational burden and cost. With tiered storage, you can keep data on cost-effective object storage and scale brokers only when you need more compute resources.
- **Confluent JMS client.** Confluent Platform includes a JMS-compatible client for Kafka. This Kafka client implements the JMS 1.1 standard API, using Kafka brokers as the backend. This is useful if you have legacy applications using JMS and you would like to replace the existing JMS message broker with Kafka.
- **Confluent MQTT proxy.** Provides a way to publish data directly to Kafka from MQTT devices and gateways without the need for a MQTT broker in the middle.
- **Confluent security plugins.** Confluent security plugins are used to add security capabilities to various Confluent Platform tools and products. Currently, there is a plugin available for the Confluent REST proxy that helps to authenticate the incoming requests and propagate the authenticated principal to requests to Kafka. This enables Confluent REST proxy clients to utilize the multitenant security features of the Kafka broker.

Next: Confluent verification.

Confluent verification

Previous: Technology overview.

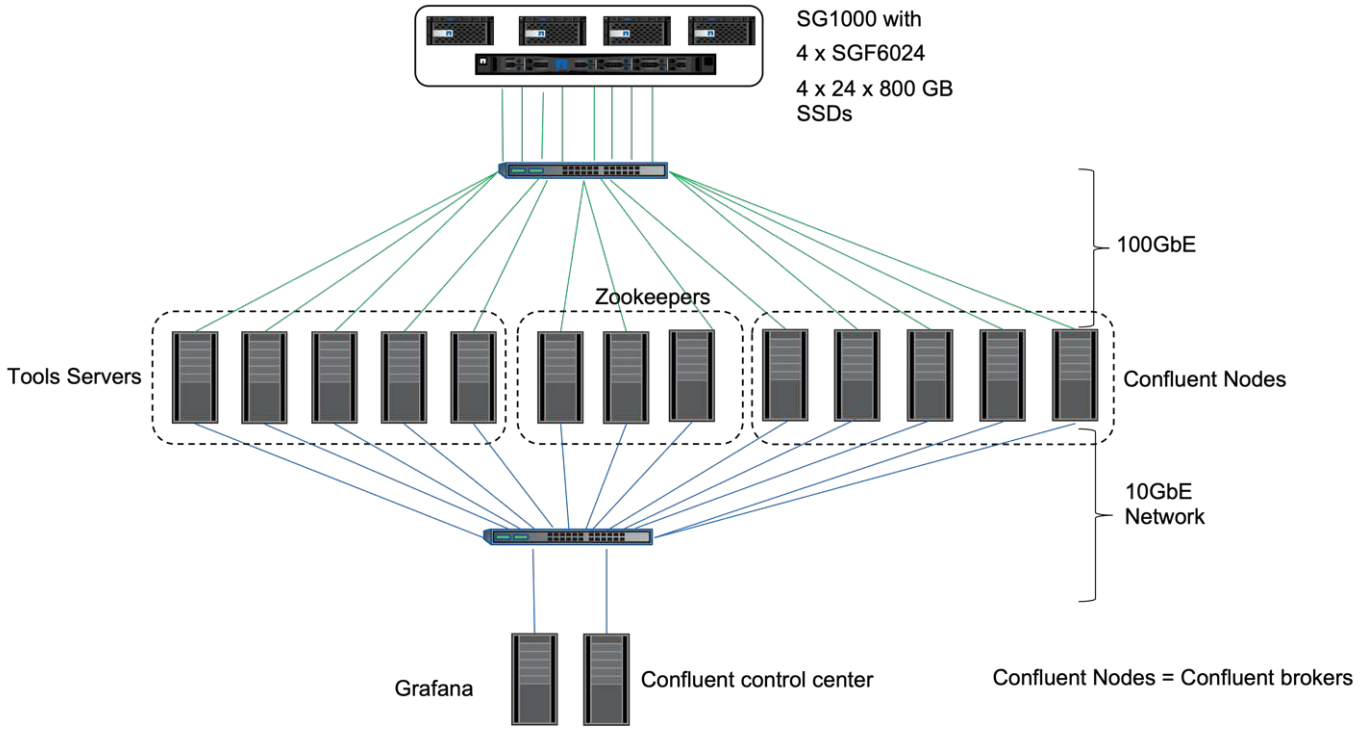
We performed verification with Confluent Platform 6.2 Tiered Storage in NetApp StorageGRID. The NetApp and Confluent teams worked on this verification together and ran the test cases required for verification.

Confluent Platform setup

We used the following setup for verification.

For verification, we used three zookeepers, five brokers, five test-script executing servers, named tools servers with 256GB RAM, and 16 CPUs. For NetApp storage, we used StorageGRID with an SG1000 load balancer with four SGF6024s. The storage and brokers were connected via 100GbE connections.

The following figure shows the network topology of configuration used for Confluent verification.



The tools servers act as application clients that send requests to Confluent nodes.

Confluent tiered storage configuration

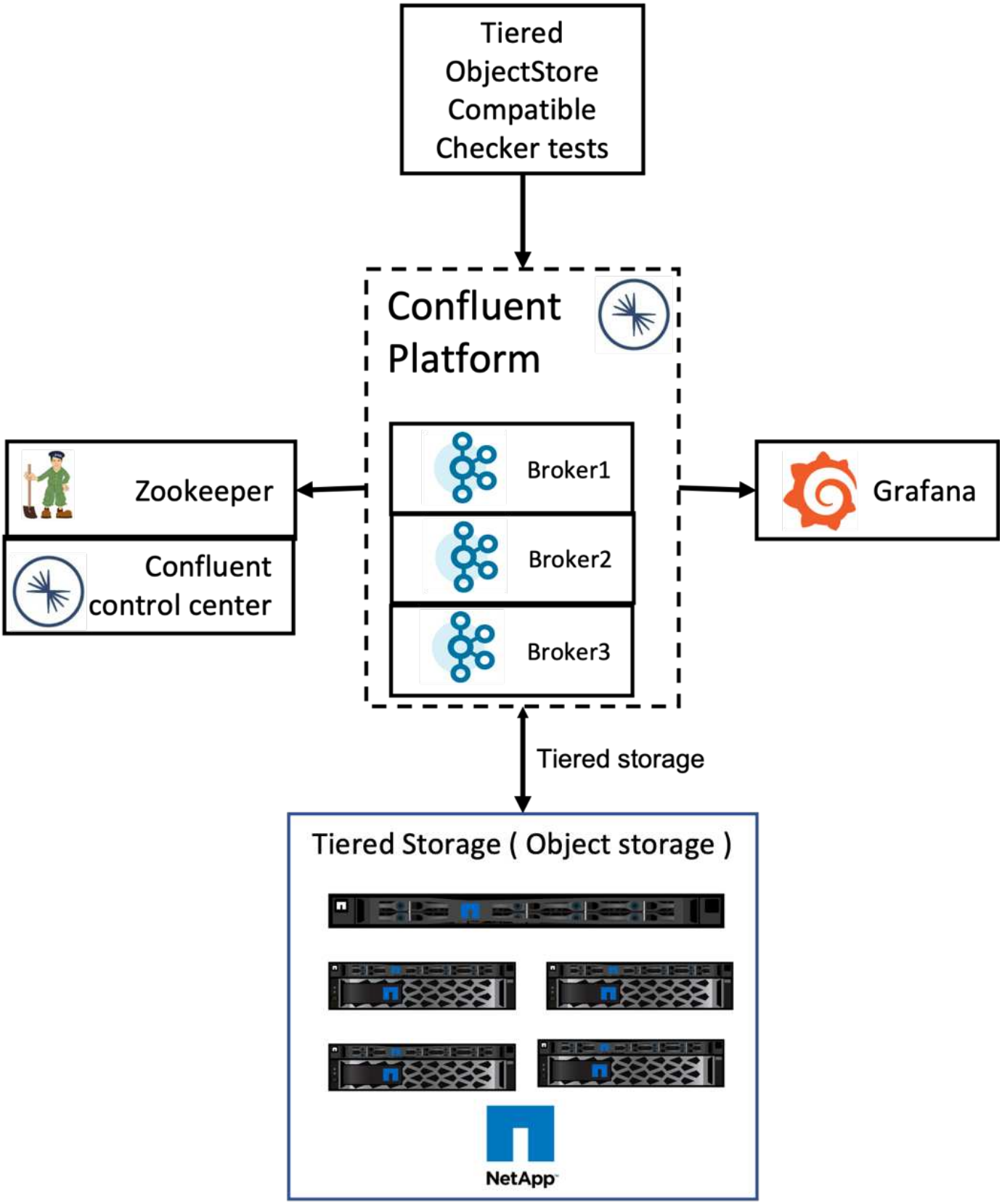
The tiered storage configuration requires the following parameters in Kafka:

```
Confluent.tier.archiver.num.threads=16
confluent.tier.fetcher.num.threads=32
confluent.tier.enable=true
confluent.tier.feature=true
confluent.tier.backend=S3
confluent.tier.s3.bucket=kafkasgdbucket1-2
confluent.tier.s3.region=us-west-2
confluent.tier.s3.cred.file.path=/data/kafka/.ssh/credentials
confluent.tier.s3.aws.endpoint.override=http://kafkasgd.rtpppe.netapp.com:
10444/
confluent.tier.s3.force.path.style.access=true
```

For verification, we used StorageGRID with the HTTP protocol, but HTTPS also works. The access key and secret key are stored in the file name provided in the `confluent.tier.s3.cred.file.path` parameter.

NetApp object storage - StorageGRID

We configured single-site configuration in StorageGRID for verfication.



Verification tests

We completed the following five test cases for the verification. These tests are executed on the Trogdor framework. The first two were functionality tests and the remaining three were performance tests.

Object store correctness test

This test determines whether all basic operations (for example, get/put/delete) on the object store API work well according to the needs of tiered storage. It is a basic test that every object store service should expect to pass ahead of the following tests. It is an assertive test that either passes or fails.

Tiering functionality correctness test

This test determines if end-to-end tiered storage functionality works well with an assertive test that either passes or fails. The test creates a test topic that by default is configured with tiering enabled and highly a reduced hotset size. It produces an event stream to the newly created test topic, it waits for the brokers to archive the segments to the object store, and it then consumes the event stream and validates that the consumed stream matches the produced stream. The number of messages produced to the event stream is configurable, which lets the user generate a sufficiently large workload according to the needs of testing. The reduced hotset size ensures that the consumer fetches outside the active segment are served only from the object store; this helps test the correctness of the object store for reads. We have performed this test with and without an object-store fault injection. We simulated node failure by stopping the service manager service in one of the nodes in StorageGRID and validating that the end-to-end functionality works with object storage.

Tier fetch benchmark

This test validated the read performance of the tiered object storage and checked the range fetch read requests under heavy load from segments generated by the benchmark. In this benchmark, Confluent developed custom clients to serve the tier fetch requests.

Produce-consume workload benchmark

This test indirectly generated write workload on the object store through the archival of segments. The read workload (segments read) was generated from object storage when consumer groups fetched the segments. This workload was generated by the test script. This test checked the performance of read and write on the object storage in parallel threads. We tested with and without object store fault injection as we did for the tiering functionality correctness test.

Retention workload benchmark

This test checked the deletion performance of an object store under a heavy topic-retention workload. The retention workload was generated using a test script that produces many messages in parallel to a test topic. The test topic was configuring with an aggressive size-based and time-based retention setting that caused the event stream to be continuously purged from the object store. The segments were then archived. This led to a large number of deletions in the object storage by the broker and collection of the performance of the object-store delete operations.

Next: Performance tests with scalability.

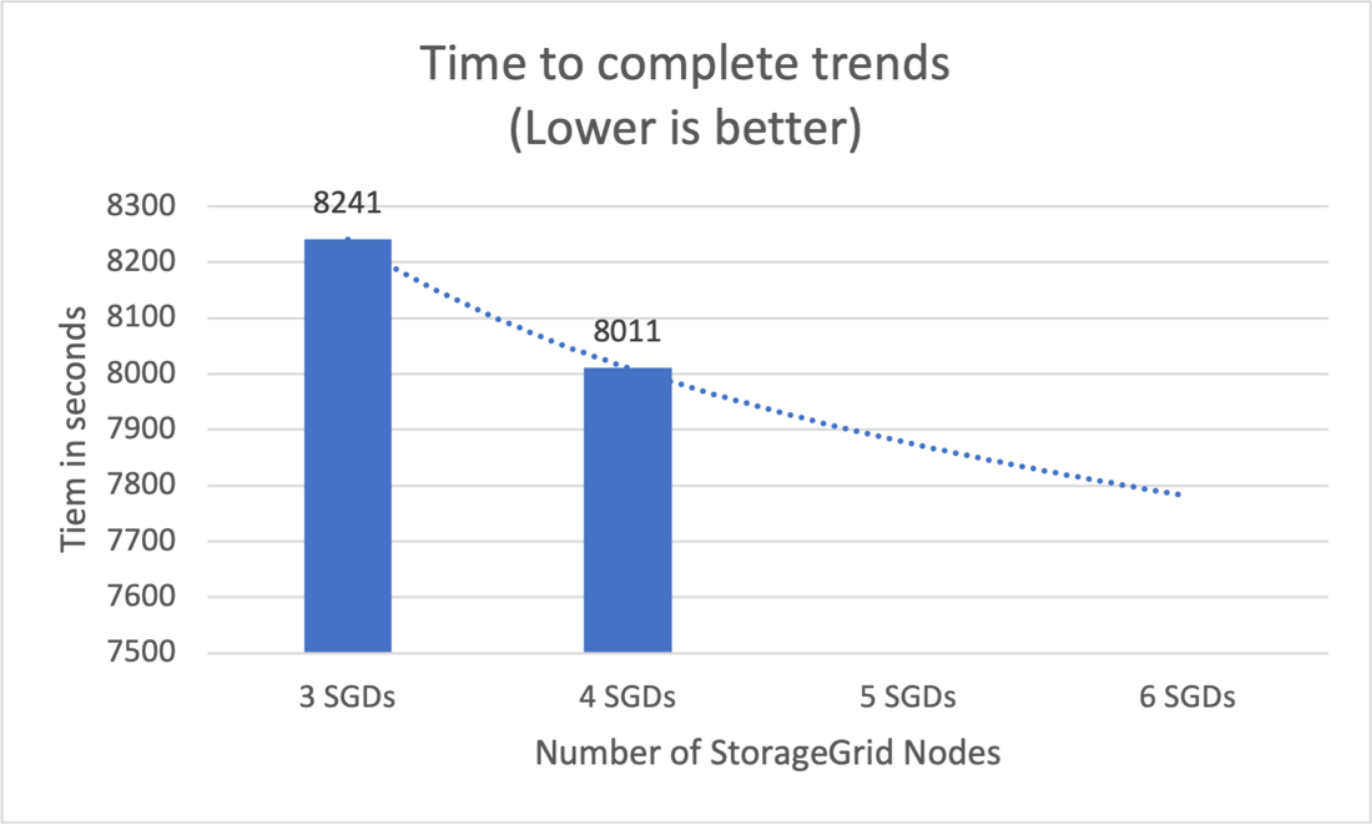
Performance tests with scalability

Previous: Confluent verification.

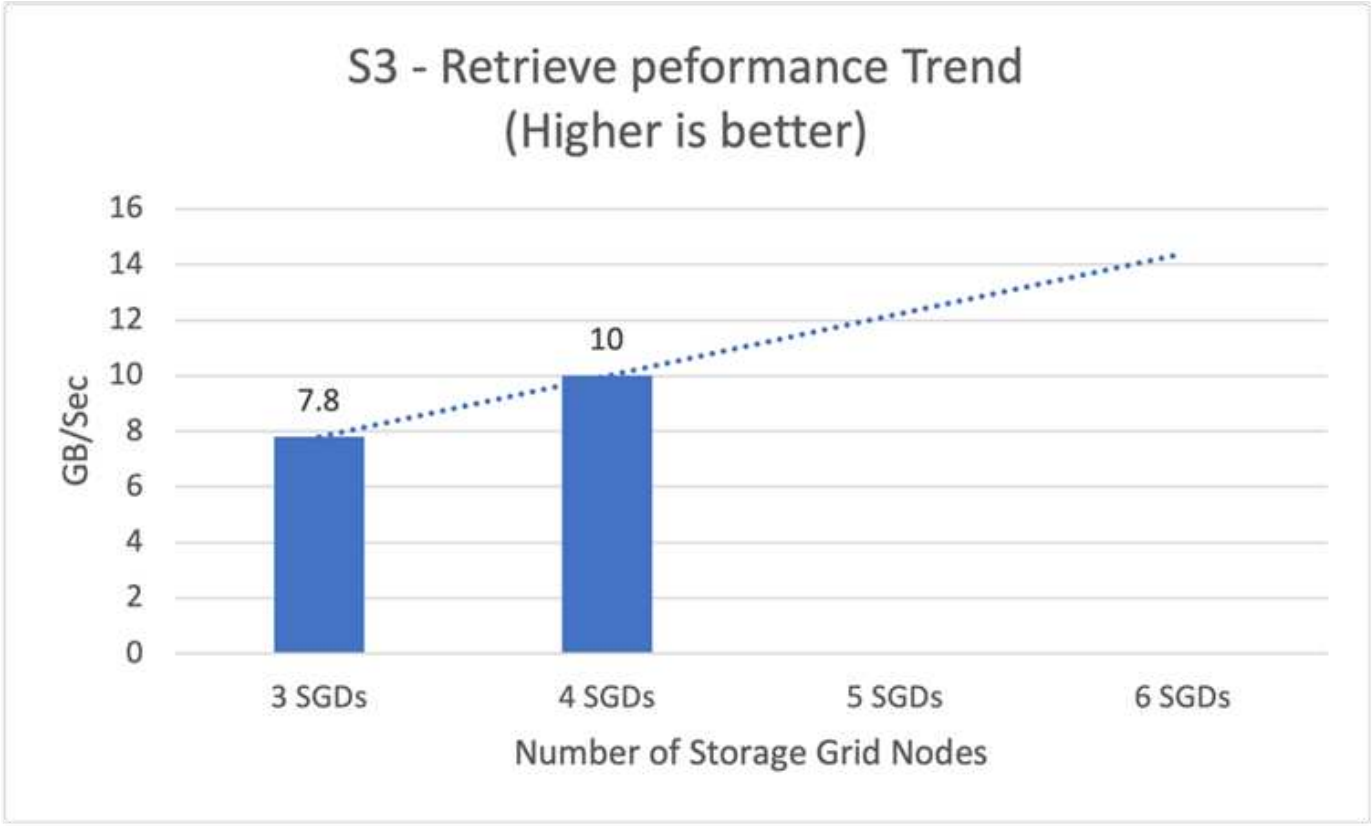
We performed the tiered storage testing with three to four nodes for producer and consumer workloads with the NetApp StorageGRID setup. According to our tests, the time to completion and the performance results were directly proportional to the number of StorageGRID nodes. The StorageGRID setup required a minimum of three nodes.

- The time to complete the produce and consumer operation decreased linearly when the number of storage

nodes increased.



- The performance for the s3 retrieve operation increased linearly based on number of StorageGRID nodes. StorageGRID supports up to 200 StorgeGRID nodes.



Next: Confluent s3 connector.

Confluent s3 connector

Previous: Performance tests with scalability.

The Amazon S3 Sink connector exports data from Apache Kafka topics to S3 objects in either the Avro, JSON, or Bytes formats. The Amazon S3 sink connector periodically polls data from Kafka and in turn uploads it to S3. A partitioner is used to split the data of every Kafka partition into chunks. Each chunk of data is represented as an S3 object. The key name encodes the topic, the Kafka partition, and the start offset of this data chunk.

In this setup, we show you how to read and write topics in object storage from Kafka directly using the Kafka s3 sink connector. For this test, we used a stand-alone Confluent cluster, but this setup is applicable to a distributed cluster.

1. Download Confluent Kafka from the Confluent website.
2. Unpack the package to a folder on your server.
3. Export two variables.

```
Export CONFLUENT_HOME=/data/confluent/confluent-6.2.0
export PATH=$PATH:/data/confluent/confluent-6.2.0/bin
```

4. For a stand-alone Confluent Kafka setup, the cluster creates a temporary root folder in `/tmp`. It also creates Zookeeper, Kafka, a schema registry, connect, a ksql-server, and control-center folders and copies their respective configuration files from `$CONFLUENT_HOME`. See the following example:

```
root@stlrx2540m1-108:~# ls -ltr /tmp/confluent.406980/
total 28
drwxr-xr-x 4 root root 4096 Oct 29 19:01 zookeeper
drwxr-xr-x 4 root root 4096 Oct 29 19:37 kafka
drwxr-xr-x 4 root root 4096 Oct 29 19:40 schema-registry
drwxr-xr-x 4 root root 4096 Oct 29 19:45 kafka-rest
drwxr-xr-x 4 root root 4096 Oct 29 19:47 connect
drwxr-xr-x 4 root root 4096 Oct 29 19:48 ksql-server
drwxr-xr-x 4 root root 4096 Oct 29 19:53 control-center
root@stlrx2540m1-108:~#
```

5. Configure Zookeeper. You don’t need to change anything if you use the default parameters.

```
root@stlrx2540m1-108:~# cat
/tmp/confluent.406980/zookeeper/zookeeper.properties  | grep -iv ^#
dataDir=/tmp/confluent.406980/zookeeper/data
clientPort=2181
maxClientCnxns=0
admin.enableServer=false
tickTime=2000
initLimit=5
syncLimit=2
server.179=controlcenter:2888:3888
root@stlrx2540m1-108:~#
```

In the above configuration, we updated the `server. xxx` property. By default, you need three Zookeepers for the Kafka leader selection.

6. We created a myid file in `/tmp/confluent.406980/zookeeper/data` with a unique ID:

```
root@stlrx2540m1-108:~# cat /tmp/confluent.406980/zookeeper/data/myid
179
root@stlrx2540m1-108:~#
```

We used the last number of IP addresses for the myid file. We used default values for the Kafka, connect, control-center, Kafka, Kafka-rest, ksql-server, and schema-registry configurations.

7. Start the Kafka services.

```
root@stlrx2540m1-108:/data/confluent/confluent-6.2.0/bin# confluent
local services  start
The local commands are intended for a single-node development
environment only,
NOT for production usage.

Using CONFLUENT_CURRENT: /tmp/confluent.406980
ZooKeeper is [UP]
Kafka is [UP]
Schema Registry is [UP]
Kafka REST is [UP]
Connect is [UP]
ksqlDB Server is [UP]
Control Center is [UP]
root@stlrx2540m1-108:/data/confluent/confluent-6.2.0/bin#
```

There is a log folder for each configuration, which helps troubleshoot issues. In some instances, services take more time to start. Make sure all services are up and running.

8. Install Kafka connect using `confluent-hub`.

```
root@stlrx2540m1-108:/data/confluent/confluent-6.2.0/bin# ./confluent-
hub install confluentinc/kafka-connect-s3:latest
The component can be installed in any of the following Confluent
Platform installations:
  1. /data/confluent/confluent-6.2.0 (based on $CONFLUENT_HOME)
  2. /data/confluent/confluent-6.2.0 (where this tool is installed)
Choose one of these to continue the installation (1-2): 1
Do you want to install this into /data/confluent/confluent-
6.2.0/share/confluent-hub-components? (yN) y

Component's license:
Confluent Community License
http://www.confluent.io/confluent-community-license
I agree to the software license agreement (yN) y
Downloading component Kafka Connect S3 10.0.3, provided by Confluent,
Inc. from Confluent Hub and installing into /data/confluent/confluent-
6.2.0/share/confluent-hub-components
Do you want to uninstall existing version 10.0.3? (yN) y
Detected Worker's configs:
  1. Standard: /data/confluent/confluent-6.2.0/etc/kafka/connect-
distributed.properties
  2. Standard: /data/confluent/confluent-6.2.0/etc/kafka/connect-
standalone.properties
  3. Standard: /data/confluent/confluent-6.2.0/etc/schema-
registry/connect-avro-distributed.properties
  4. Standard: /data/confluent/confluent-6.2.0/etc/schema-
registry/connect-avro-standalone.properties
  5. Based on CONFLUENT_CURRENT:
/tmp/confluent.406980/connect/connect.properties
  6. Used by Connect process with PID 15904:
/tmp/confluent.406980/connect/connect.properties
Do you want to update all detected configs? (yN) y
Adding installation directory to plugin path in the following files:
  /data/confluent/confluent-6.2.0/etc/kafka/connect-
distributed.properties
  /data/confluent/confluent-6.2.0/etc/kafka/connect-
standalone.properties
  /data/confluent/confluent-6.2.0/etc/schema-registry/connect-avro-
distributed.properties
  /data/confluent/confluent-6.2.0/etc/schema-registry/connect-avro-
standalone.properties
  /tmp/confluent.406980/connect/connect.properties
  /tmp/confluent.406980/connect/connect.properties
```

```
Completed
root@stlrx2540m1-108:/data/confluent/confluent-6.2.0/bin#
```

You can also install a specific version by using `confluent-hub install confluentinc/kafka-connect-s3:10.0.3`.

9. By default, `confluentinc-kafka-connect-s3` is installed in `/data/confluent/confluent-6.2.0/share/confluent-hub-components/confluentinc-kafka-connect-s3`.
10. Update the plug-in path with the new `confluentinc-kafka-connect-s3`.

```
root@stlrx2540m1-108:~# cat /data/confluent/confluent-
6.2.0/etc/kafka/connect-distributed.properties | grep plugin.path
#
plugin.path=/usr/local/share/java,/usr/local/share/kafka/plugins,/opt/co
nnectors,
plugin.path=/usr/share/java,/data/zookeeper/confluent/confluent-
6.2.0/share/confluent-hub-components,/data/confluent/confluent-
6.2.0/share/confluent-hub-components,/data/confluent/confluent-
6.2.0/share/confluent-hub-components/confluentinc-kafka-connect-s3
root@stlrx2540m1-108:~#
```

11. Stop the Confluent services and restart them.

```
confluent local services  stop
confluent local services  start
root@stlrx2540m1-108:/data/confluent/confluent-6.2.0/bin# confluent
local services  status
The local commands are intended for a single-node development
environment only,
NOT for production usage.

Using CONFLUENT_CURRENT: /tmp/confluent.406980
Connect is [UP]
Control Center is [UP]
Kafka is [UP]
Kafka REST is [UP]
ksqlDB Server is [UP]
Schema Registry is [UP]
ZooKeeper is [UP]
root@stlrx2540m1-108:/data/confluent/confluent-6.2.0/bin#
```

12. Configure the access ID and secret key in the `/root/.aws/credentials` file.

```
root@stlrx2540m1-108:~# cat /root/.aws/credentials
[default]
aws_access_key_id = xxxxxxxxxxxx
aws_secret_access_key = xxxxxxxxxxxxxxxxxxxxxxxxx
root@stlrx2540m1-108:~#
```

13. Verify that the bucket is reachable.

```
root@stlrx2540m4-01:~# aws s3 –endpoint-url
http://kafkasgd.rtpppe.netapp.com:10444 ls kafkasgdbucket1-2
2021-10-29 21:04:18       1388 1
2021-10-29 21:04:20       1388 2
2021-10-29 21:04:22       1388 3
root@stlrx2540m4-01:~#
```

14. Configure the s3-sink properties file for s3 and bucket configuration.

```
root@stlrx2540m1-108:~# cat /data/confluent/confluent-
6.2.0/share/confluent-hub-components/confluentinc-kafka-connect-
s3/etc/quickstart-s3.properties | grep -v ^#
name=s3-sink
connector.class=io.confluent.connect.s3.S3SinkConnector
tasks.max=1
topics=s3_testtopic
s3.region=us-west-2
s3.bucket.name=kafkasgdbucket1-2
store.url=http://kafkasgd.rtpppe.netapp.com:10444/
s3.part.size=5242880
flush.size=3
storage.class=io.confluent.connect.s3.storage.S3Storage
format.class=io.confluent.connect.s3.format.avro.AvroFormat
partitioner.class=io.confluent.connect.storage.partitioner.DefaultPartit
ioner
schema.compatibility=NONE
root@stlrx2540m1-108:~#
```

15. Import a few records to the s3 bucket.

```
kafka-avro-console-producer --broker-list localhost:9092 --topic
s3_topic \
--property
value.schema='{"type":"record","name":"myrecord","fields":[{"name":"f1",
"type":"string"}]}'
{"f1": "value1"}
{"f1": "value2"}
{"f1": "value3"}
{"f1": "value4"}
{"f1": "value5"}
{"f1": "value6"}
{"f1": "value7"}
{"f1": "value8"}
{"f1": "value9"}
```

16. Load the s3-sink connector.

```
root@stlrx2540m1-108:~# confluent local services connect connector load
s3-sink  --config /data/confluent/confluent-6.2.0/share/confluent-hub-
components/confluentinc-kafka-connect-s3/etc/quickstart-s3.properties
The local commands are intended for a single-node development
environment only,
NOT for production usage.
https://docs.confluent.io/current/cli/index.html
{
  "name": "s3-sink",
  "config": {
    "connector.class": "io.confluent.connect.s3.S3SinkConnector",
    "flush.size": "3",
    "format.class": "io.confluent.connect.s3.format.avro.AvroFormat",
    "partitioner.class":
"io.confluent.connect.storage.partitioner.DefaultPartitioner",
    "s3.bucket.name": "kafkasgdbucket1-2",
    "s3.part.size": "5242880",
    "s3.region": "us-west-2",
    "schema.compatibility": "NONE",
    "storage.class": "io.confluent.connect.s3.storage.S3Storage",
    "store.url": "http://kafkasgd.rtpppe.netapp.com:10444/",
    "tasks.max": "1",
    "topics": "s3_testtopic",
    "name": "s3-sink"
  },
  "tasks": [],
  "type": "sink"
}
root@stlrx2540m1-108:~#
```

17. Check the s3-sink status.

```
root@stlrx2540m1-108:~# confluent local services connect connector
status s3-sink
The local commands are intended for a single-node development
environment only,
NOT for production usage.
https://docs.confluent.io/current/cli/index.html
{
  "name": "s3-sink",
  "connector": {
    "state": "RUNNING",
    "worker_id": "10.63.150.185:8083"
  },
  "tasks": [
    {
      "id": 0,
      "state": "RUNNING",
      "worker_id": "10.63.150.185:8083"
    }
  ],
  "type": "sink"
}
root@stlrx2540m1-108:~#
```

18. Check the log to make sure that s3-sink is ready to accept topics.

```
root@stlrx2540m1-108:~# confluent local services connect log
```

19. Check the topics in Kafka.

```
kafka-topics --list --bootstrap-server localhost:9092
…
connect-configs
connect-offsets
connect-statuses
default_ksql_processing_log
s3_testtopic
s3_topic
s3_topic_new
root@stlrx2540m1-108:~#
```

20. Check the objects in the s3 bucket.

```
root@stlrx2540m1-108:~# aws s3 --endpoint-url
http://kafkasgd.rtpppe.netapp.com:10444 ls --recursive kafkasgdbucket1-
2/topics/
2021-10-29 21:24:00        213
topics/s3_testtopic/partition=0/s3_testtopic+0+0000000000.avro
2021-10-29 21:24:00        213
topics/s3_testtopic/partition=0/s3_testtopic+0+0000000003.avro
2021-10-29 21:24:00        213
topics/s3_testtopic/partition=0/s3_testtopic+0+0000000006.avro
2021-10-29 21:24:08        213
topics/s3_testtopic/partition=0/s3_testtopic+0+0000000009.avro
2021-10-29 21:24:08        213
topics/s3_testtopic/partition=0/s3_testtopic+0+0000000012.avro
2021-10-29 21:24:09        213
topics/s3_testtopic/partition=0/s3_testtopic+0+0000000015.avro
root@stlrx2540m1-108:~#
```

21. To verify the contents, copy each file from S3 to your local filesystem by running the following command:

```
root@stlrx2540m1-108:~# aws s3 --endpoint-url
http://kafkasgd.rtpppe.netapp.com:10444 cp s3://kafkasgdbucket1-
2/topics/s3_testtopic/partition=0/s3_testtopic+0+0000000000.avro
tes.avro
download: s3://kafkasgdbucket1-
2/topics/s3_testtopic/partition=0/s3_testtopic+0+0000000000.avro to
./tes.avro
root@stlrx2540m1-108:~#
```

22. To print the records, use avro-tools-1.11.0.1.jar (available in the Apache Archives).

```
root@stlrx2540m1-108:~# java -jar /usr/src/avro-tools-1.11.0.1.jar
tojson tes.avro
21/10/30 00:20:24 WARN util.NativeCodeLoader: Unable to load native-
hadoop library for your platform... using builtin-java classes where
applicable
{"f1":"value1"}
{"f1":"value2"}
{"f1":"value3"}
root@stlrx2540m1-108:~#
```

Next: Confluent self-rebalancing clusters.

Confluent Self-balancing Clusters

Previous: Kafka s3 connector.

If you have managed a Kafka cluster before, you are likely familiar with the challenges that come with manually reassigning partitions to different brokers to make sure that the workload is balanced across the cluster. For organizations with large Kafka deployments, reshuffling large amounts of data can be daunting, tedious, and risky, especially if mission-critical applications are built on top of the cluster. However, even for the smallest Kafka use cases, the process is time consuming and prone to human error.

In our lab, we tested the Confluent self-balancing clusters feature, which automates rebalancing based on cluster topology changes or uneven load. The Confluent rebalance test helps to measure the time to add a new broker when node failure or the scaling node requires rebalancing data across brokers. In classic Kafka configurations, the amount of data to be rebalanced grows as the cluster grows, but, in tiered storage, rebalancing is restricted to a small amount of data. Based on our validation, rebalancing in tiered storage takes seconds or minutes in a classic Kafka architecture and grows linearly as the cluster grows.

In self-balancing clusters, partition rebalances are fully automated to optimize Kafka's throughput, accelerate broker scaling, and reduce the operational burden of running a large cluster. At steady-state, self-balancing clusters monitor the skew of data across the brokers and continuously reassigns partitions to optimize cluster performance. When scaling the platform up or down, self-balancing clusters automatically recognize the presence of new brokers or the removal of old brokers and trigger a subsequent partition reassignment. This enables you to easily add and decommission brokers, making your Kafka clusters fundamentally more elastic. These benefits come without any need for manual intervention, complex math, or the risk of human error that partition reassignments typically entail. As a result, data rebalances are completed in far less time, and you are free to focus on higher-value event-streaming projects rather than needing to constantly supervise your clusters.

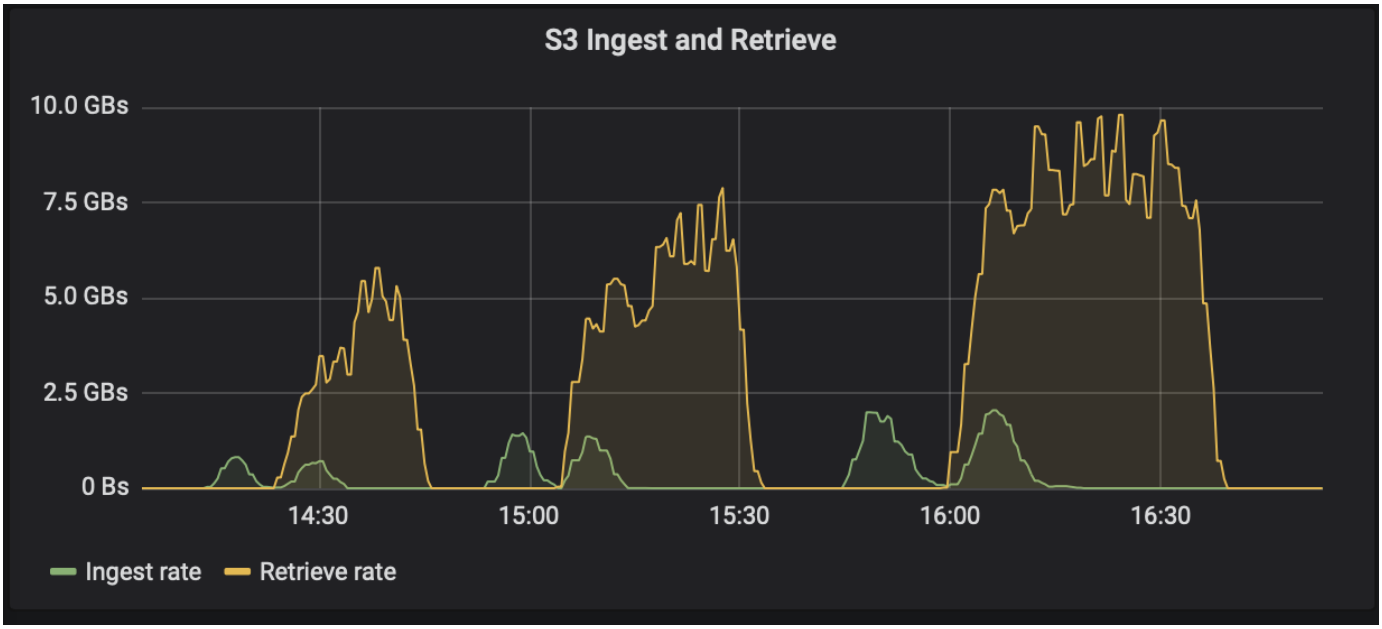
Next: Best practice guidelines.

Best practice guidelines

Previous: Confluent self-rebalancing clusters.

- Based on our validation, S3 object storage is best for Confluent to keep data.
- We can use high-throughput SAN (specifically FC) to keep the broker hot data or local disk, because, in the Confluent tiered storage configuration, the size of the data held in the brokers data directory is based on the segment size and retention time when the data is moved to object storage.
- Object stores provide better performance when segment.bytes is higher; we tested 512MB.
- In Kafka, the length of the key or value (in bytes) for each record produced to the topic is controlled by the `length.key.value` parameter. For StorageGRID, S3 object ingest and retrieve performance increased to higher values. For example, 512 bytes provided a 5.8GBps retrieve, 1024 bytes provided a 7.5GBps s3 retrieve, and 2048 bytes provided close to 10GBps.

The following figure presents the S3 object ingest and retrieve based on `length.key.value`.



- **Kafka tuning.** To improve the performance of tiered storage, you can increase TierFetcherNumThreads and TierArchiverNumThreads. As a general guideline, you want to increase TierFetcherNumThreads to match the number of physical CPU cores and increase TierArchiverNumThreads to half the number of CPU cores. For example, in server properties, if you have a machine with eight physical cores, set confluent.tier.fetcher.num.threads = 8 and confluent.tier.archiver.num.threads = 4.
- **Time interval for topic deletes.** When a topic is deleted, deletion of the log segment files in object storage does not immediately begin. Rather, there is a time interval with a default value of 3 hours before deletion of those files takes place. You can modify the configuration, confluent.tier.topic.delete.check.interval.ms, to change the value of this interval. If you delete a topic or cluster, you can also manually delete the objects in the respective bucket.
- **ACLs on tiered storage internal topics.** A recommended best practice for on-premises deployments is to enable an ACL authorizer on the internal topics used for tiered storage. Set ACL rules to limit access on this data to the broker user only. This secures the internal topics and prevents unauthorized access to tiered storage data and metadata.

```
kafka-acls --bootstrap-server localhost:9092 --command-config adminclient-
configs.conf \
--add --allow-principal User:<kafka> --operation All --topic "_confluent-
tier-state"
```

Replace the user `<kafka>` with the actual broker principal in your deployment.

For example, the command `confluent-tier-state` sets ACLs on the internal topic for tiered storage. Currently, there is only a single internal topic related to tiered storage. The example creates an ACL that provides the principal Kafka permission for all operations on the internal topic.

Next: Sizing.

Sizing

Previous: Best practice guidelines.

Kafka sizing can be performed with four configuration modes: simple, granular, reverse, and partitions.

Simple

The simple mode is appropriate for the first-time Apache Kafka users or early state use cases. For this mode, you provide requirements such as throughput MBps, read fanout, retention, and the resource utilization percentage (60% is default). You also enter the environment, such as on-premises (bare-metal, VMware, Kubernetes, or OpenStack) or cloud. Based on this information, the sizing of a Kafka cluster provides the number of servers required for the broker, the zookeeper, Apache Kafka connect workers, the schema registry, a REST Proxy, ksqlDB, and the Confluent control center.

For tiered storage, consider the granular configuration mode for sizing a Kafka cluster. Granular mode is appropriate for experienced Apache Kafka users or well-defined use cases. This section describes sizing for producers, stream processors, and consumers.

Producers

To describe the producers for Apache Kafka (for example a native client, REST proxy, or Kafka connector), provide the following information:

- **Name.** Spark.
- **Producer type.** Application or service, proxy (REST, MQTT, other), and existing database (RDBMS, NOSQL, other). You can also select “I don’t know.”
- **Average throughput.** In events per second (1,000,000 for example).
- **Peak throughput.** In events per second (4,000,000 for example).
- **Average message size.** In bytes, uncompressed (max 1MB; 1000 for example).
- **Message format.** Options include Avro, JSON, protocol buffers, binary, text, “I don’t know,” and other.
- **Replication factor.** Options are 1, 2, 3 (Confluent recommendation), 4, 5, or 6.
- **Retention time.** One day (for example). How long do you want your data to be stored in Apache Kafka? Enter -1 with any unit for an infinite time. The calculator assumes a retention time of 10 years for infinite retention.
- Select the check box for "Enable Tiered Storage to Decrease Broker Count and Allow for Infinite Storage?"
- When tiered storage is enabled, the retention fields control the hot set of data that is stored locally on the broker. The archival retention fields control how long data is stored in archival object storage.
- **Archival Storage Retention.** One year (for example). How long do you want your data to be stored in archival storage? Enter -1 with any unit for an infinite duration. The calculator assumes a retention of 10 years for infinite retention.
- **Growth Multiplier.** 1 (for example). If the value of this parameter is based on current throughput, set it to 1. To size based on additional growth, set this parameter to a growth multiplier.
- **Number of producer instances.** 10 (for example). How many producer instances will be running? This input is required to incorporate the CPU load into the sizing calculation. A blank value indicates that CPU load is not incorporated into the calculation.

Based on this example input, sizing has the following effect on producers:

- Average throughput in uncompressed bytes: 1GBps. Peak throughput in uncompressed bytes: 4GBps. Average throughput in compressed bytes: 400MBps. Peak throughput in compressed bytes: 1.6GBps. This is based on a default 60% compression rate (you can change this value).
    - Total on-broker hotset storage required: 31,104TB, including replication, compressed. Total off-broker archival storage required: 378,432TB, compressed. Use https://fusion.netapp.com for StorageGRID sizing.

Stream Processors must describe their applications or services that consume data from Apache Kafka and produce back into Apache Kafka. In most cases these are built in ksqlDB or Kafka Streams.

- **Name.** Spark streamer.
- **Processing time.** How long does this processor take to process a single message?
    - 1 ms (simple, stateless transformation) [example], 10ms (stateful in-memory operation).
    - 100ms (stateful network or disk operation), 1000ms (3rd party REST call).
    - I have benchmarked this parameter and know exactly how long it takes.
- **Output Retention.** 1 day (example). A stream processor produces its output back to Apache Kafka. How long do you want this output data to be stored in Apache Kafka? Enter -1 with any unit for an infinite duration.
- Select the check box "Enable Tiered Storage to Decrease Broker Count and Allow for Infinite Storage?"
- **Archival Storage Retention.** 1 year (for example). How long do you want your data to be stored in archival storage? Enter -1 with any unit for an infinite duration. The calculator assumes a retention of 10 years for infinite retention.
- **Output Passthrough Percentage.** 100 (for example). A stream processor produces its output back to Apache Kafka. What percentage of inbound throughput will be outputted back into Apache Kafka? For example, if inbound throughput is 20MBps and this value is 10, the output throughput will be 2MBps.
- From which applications does this read from? Select “Spark,” the name used in producer type-based sizing.
  Based on the above input, you can expect the following effects of sizing on stream processer instances and topic partition estimates:
- This stream processor application requires the following number of instances. The incoming topics likely require this many partitions as well. Contact Confluent to confirm this parameter.
    - 1,000 for average throughput with no growth multiplier
    - 4,000 for peak throughput with no growth multiplier
    - 1,000 for average throughput with a growth multiplier
    - 4,000 for peak throughput with a growth multiplier

Consumers

Describe your applications or services that consume data from Apache Kafka and do not produce back into Apache Kafka; for example, a native client or Kafka Connector.

- **Name.** Spark consumer.
- **Processing time.** How long does this consumer take to process a single message?
    - 1ms (for example, a simple and stateless task like logging)
    - 10ms (fast writes to a datastore)

    - 100ms (slow writes to a datastore)
    - 1000ms (third party REST call)
    - Some other benchmarked process of known duration.
- **Consumer type.** Application, proxy, or sink to an existing datastore (RDBMS, NoSQL, other).
- From which applications does this read from? Connect this parameter with producer and stream sizing determined previously.

Based on the above input, you must determine the sizing for consumer instances and topic partition estimates. A consumer application requires the following number of instances.

- 2,000 for average throughput, no growth multiplier
- 8,000 for peak throughput, no growth multiplier
- 2,000 for average throughput, including growth multiplier
- 8,000 for peak throughput, including growth multiplier

The incoming topics likely need this number of partitions as well. Contact Confluent to confirm.

In addition to the requirements for producers, stream processors, and consumers, you must provide the following additional requirements:

- **Rebuild time.** For example, 4 hours. If an Apache Kafka broker host fails, its data is lost, and a new host is provisioned to replace the failed host, how fast must this new host rebuild itself? Leave this parameter blank if the value is unknown.
- **Resource utilization target (percentage).** For example, 60. How utilized do you want your hosts to be during average throughput? Confluent recommends 60% utilization unless you are using Confluent self-balancing clusters, in which case utilization can be higher.

Describe your environment

- **What environment will your cluster be running in?** Amazon Web Services, Microsoft Azure, Google cloud platform, bare-metal on premises, VMware on premises, OpenStack on premises, or Kubernates on premises?
- **Host details.** Number of cores: 48 (for example), network card type (10GbE, 40GbE, 16GbE, 1GbE, or another type).
- **Storage volumes.** Host: 12 (for example). How many hard drives or SSDs are supported per host? Confluent recommends 12 hard drives per host.
- **Storage capacity/volume (in GB).** 1000 (for example). How much storage can a single volume store in gigabytes? Confluent recommends 1TB disks.
- **Storage configuration.** How are storage volumes configured? Confluent recommends RAID10 to take advantage of all Confluent features. JBOD, SAN, RAID 1, RAID 0, RAID 5, and other types are also supported.
- **Single volume throughput (MBps).** 125 (for example). How fast can a single storage volume read or write in megabytes per second? Confluent recommends standard hard drives, which typically have 125MBps throughput.
- **Memory capacity (GB).** 64 (for example).

After you have determined your environmental variables, select Size my Cluster. Based on the example parameters indicated above, we determined the following sizing for Confluent Kafka:

- **Apache Kafka.** Broker count: 22. Your cluster is storage-bound. Consider enabling tiered storage to decrease your host count and allow for infinite storage.
- **Apache ZooKeeper.** Count: 5; Apache Kafka Connect Workers: Count: 2; Schema Registry: Count: 2; REST Proxy: Count: 2; ksqlDB: Count: 2; Confluent Control Center: Count: 1.

Use reverse mode for platform teams without a use case in mind. Use partitions mode to calculate how many partitions a single topic requires. See https://eventsizer.io for sizing based on the reverse and partitions modes.

Next: Conclusion.

Conclusion

Previous: Sizing.

This document provides best practice guidelines for using Confluent Tiered Storage with NetApp storage, including verification tests, tiered storage performance results, tuning, Confluent S3 connectors, and the self-balancing feature. Considering ILM policies, Confluent performance with multiple performance tests for verification, and industry-standard S3 APIs, NetApp StorageGRID object storage is an optimal choice for Confluent tiered storage.

Where to find additional information

To learn more about the information that is described in this document, review the following documents and/or websites:

- What is Apache Kafka

  https://www.confluent.io/what-is-apache-kafka/

- NetApp Product Documentation

  https://www.netapp.com/support-and-training/documentation/

- S3-sink parameter details

  https://docs.confluent.io/kafka-connect-s3-sink/current/configuration_options.html#s3-configuration-options

- Apache Kafka

  https://en.wikipedia.org/wiki/Apache_Kafka

- Infinite Storage in Confluent Platform

  https://www.confluent.io/blog/infinite-kafka-storage-in-confluent-platform/

- Confluent Tiered Storage - Best practices and sizing

  https://docs.confluent.io/platform/current/kafka/tiered-storage.html#best-practices-and-recommendations

- Amazon S3 sink connector for Confluent Platform

  https://docs.confluent.io/kafka-connect-s3-sink/current/overview.html

- Kafka sizing

https://eventsizer.io

- StorageGRID sizing

  https://fusion.netapp.com/

- Kafka use cases

  https://kafka.apache.org/uses

- Self-balancing Kafka clusters in confluent platform 6.0

  https://www.confluent.io/blog/self-balancing-kafka-clusters-in-confluent-platform-6-0/

  https://www.confluent.io/blog/confluent-platform-6-0-delivers-the-most-powerful-event-streaming-platform-to-date/

Version history

Version	Date	Document version history
Version 1.0	December 2021	Initial release.