



Confluent Platform Tiered Object Storage Throughput Benchmark

September 2023

Confluent Platform Tiered Object Storage

Throughput Benchmark

The growth of data and how to manage and monetize it is the defining characteristic of the modern enterprise.

The defining architecture for these enterprises follows the lead of the hyper-scalers where storage and compute are disaggregated, best-of-breed and designed for commodity hardware. This enables computing to become stateless, elastic, and independently scalable from storage.

The standard for this architecture is modern object storage. This is a function of modern object storage's RESTful APIs (S3), scalability, resiliency and throughput performance.

This document describes the observations and results of testing MinIO object storage as a backend for the tiered storage feature of Confluent Platform 7.1.0 on servers equipped with third generation Intel Xeon Scalable processors. Testing was performed by Confluent, Intel and MinIO. The scope of these tests was to observe the read, write and delete performance of MinIO object storage under heavy workloads originating from the Kafka broker related to tiered storage.

Confluent fully supports MinIO object storage as a Tiered Storage layer. Tiered Storage makes storing massive volumes of data in Kafka manageable by reducing operational burden and cost. The concept is to disaggregate data storage from data processing, allowing each to scale independently. With Confluent Tiered Storage, you can send warm data to cost-effective MinIO object storage, and scale Kafka Brokers and storage separately for greater efficiency. This combination enables individual Kafka clusters to grow to petabytes of data when deployed on the optimal hardware.

MinIO is a cloud-native object storage suite designed for high-performance workloads such as AI/ML, advanced analytics and databases. MinIO is software defined and open-sourced under the AGPL v3 license. The object storage suite consists of a server, and optional components such as a client, a management console, a Kubernetes Operator and Operator Console and a software development kit (SDK).

[Confluent Platform](#) is a full-scale data streaming platform that enables you to easily access, store, and manage data as continuous, real-time streams. 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 the majority of those leverage Confluent.

Confluent Tiered Storage using MinIO object storage improves Kafka scalability, elasticity and performance. With tiered storage, the warm tier handles elasticity and throughput for long-term storage, while the hot tier relies on costly local short-term storage. Previously, replication and re-replication affected all brokers, slowing down performance. Now, with compute and storage decoupled, brokers now only replicate hot data and MinIO's erasure coding protects warm data. Software-defined MinIO enables infinite storage in Confluent Platform.

Our results running a throughput test between 5 Kafka Brokers and an 8 node MinIO cluster with 64 NVMe drives across a 40 Gbps network can be summarized as follows:

Kafka Brokers	Drives per Broker	Kafka Topics	Producers / Tasks	Consumers / tasks	Partitions per topic	Replicas/ Minimum	Data Ingested	Kafka Ingestion	MinIO Data Rate	Notes
5	1	8	8/6	8/6	100	3/1	20TB	1.22 GB/s	7 GiB/s	Broker limited by the single drive
5	5	8	8/6	8/6	100	3/1	20TB	13.68 GB/s	12 GiB/s	Broker I/O at 100%, 40Gbps network saturated
6	8	8	8/6	8/6	100	3/1	20TB	12.41 GiB/s	N/A	Adding Brokers doesn't improve throughput as network is saturated.



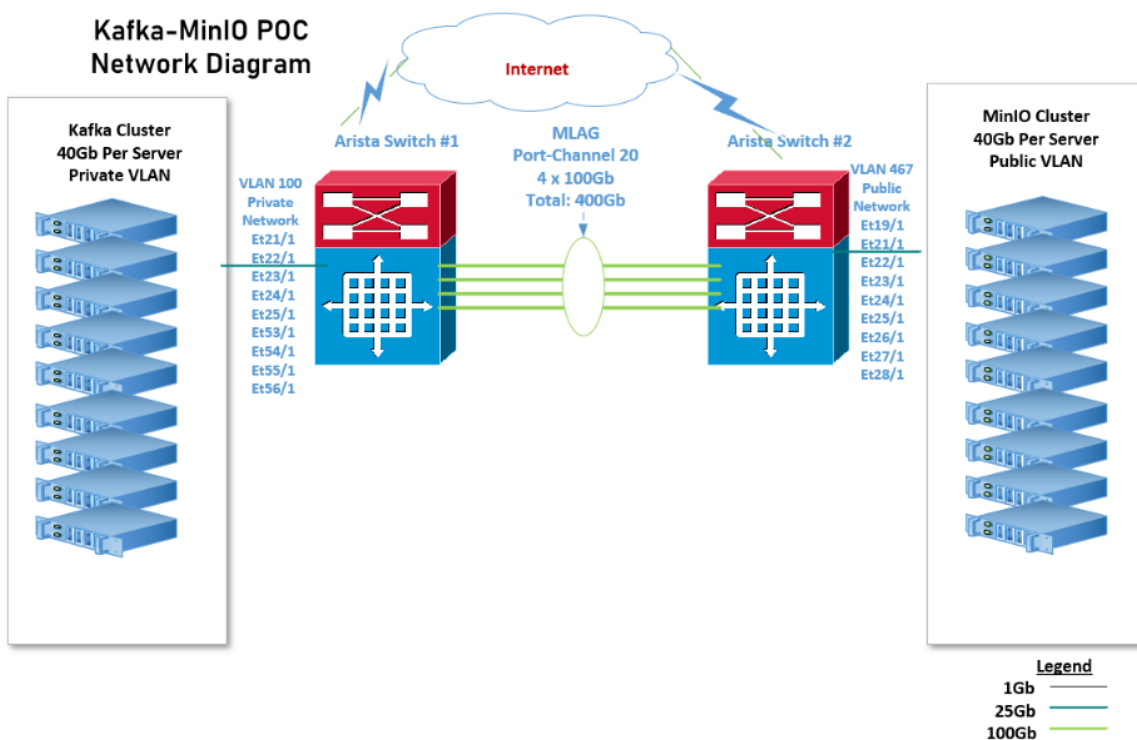
1. Benchmark Environment

1.1 Hardware

Testing was conducted using Intel hardware. MinIO Server nodes were configured with 40GbE networking and NVMe drives.

Instance	# Nodes	Server CPU type	CPU	MEM	Storage	Network
MinIO Server	8	Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz	1	512 GB	6 x 7.68TB NVMe	40 Gbps
Kafka Broker	6	Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz	1	384 GB	8 x 4TB NVMe	40 Gbps
Kafka Management	8	Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz	1	512 GB	1 x 800GB SSD	40 Gbps

Kafka Management nodes consist of 3 machines for Zookeeper and 5 machines for test tools. One Kafka Broker machine was dedicated to Grafana and Prometheus.



1.2 Software

Property	Value
Server OS	RHEL 9.1
MinIO Version	2023-06-09T07-32-12Z
Java	JDK 11
Grafana	Latest version: Download from Grafana
Prometheus server	Latest version: Download from Prometheus server
Prometheus JMX exporter	Download from here

1.3 Tier Fetch Benchmark

We performed the steps listed in Section 3. Tier Fetch Benchmark section of the Confluent Tier Object Store Compatibility Checker (TOCC) procedure. The TOCC framework is used to assess the compatibility of an object store with tiered storage.

The scope of the performance tests conducted was to observe the read, write and delete performance of MinIO object storage under the following heavy workloads originating from the Kafka Broker related to tiered storage:

1. **Background write workload** to archive stream data from Broker's local disk (or page cache) to MinIO object storage.
2. **Streaming read(fetch) workload** to serve historical fetch requests from consumers.
3. **Background deletion workload** that deletes stream data in MinIO object storage when data retention has expired.

An important part of the performance of the object store APIs for serving reads is its ability to serve range fetch read requests under heavy load. This benchmark is useful to measure the performance of the object store when serving range fetch requests from segments generated by the benchmark. In this benchmark, the client reading/writing from/to the object store is not the Kafka broker, but rather a custom client developed



using some of the core libraries directly used by Confluent internally to serve the tier fetch requests.

For each `record_size_bytes` chosen from the list: `[500, 50000, 500000, 1000000, 2000000]`, the benchmark performs 60 iterations with each iteration consisting of all of the following set of steps below. It then measures the avg/min/max time taken to complete the entire range fetch request across all 60 iterations.

Here are the steps carried out by the benchmark per iteration:

1. Create a segment and populate it with records each having the specified `record_size_bytes`, until the segment reaches 100MB in size.
2. Upload the generated segment to the object store.
3. Fetch a range of 10MB from the segment in chunks. Measure the time taken to complete the entire range fetch request.
4. Print the benchmark output to `stderr`.

1.4 MinIO Configuration

The MinIO binary was downloaded onto each server node, and configured as follows:

```
# Remote volumes to be used for MinIO server.
MINIO_VOLUMES=http://data{1...8}/mnt/drive{1...8}/minio
# Use if you want to run MinIO on a custom port.
MINIO_OPTS="--console-address :9199"
# Root user for the server.
MINIO_ROOT_USER=minio
MINIO_STORAGE_CLASS_STANDARD=EC:4
# Root secret for the server.
MINIO_ROOT_PASSWORD=minio123
MINIO_PROMETHEUS_AUTH_TYPE="public"
MINIO_PROMETHEUS_URL=http://data1:9090
MINIO_PROMETHEUS_AUTH_TYPE="public"
```



2. Understanding Hardware Performance

2.1 Network Performance

In virtually all cases with MinIO, the network is the bottleneck. MinIO takes full advantage of the available underlying server hardware. In this test, a single 40 Gbps network connected Kafka Brokers, Kafka Management Tools and MinIO Servers.

Therefore, the maximum throughput that can be expected from each of these nodes would be 5 Gbytes/sec.

There are 8 MinIO nodes, making the theoretical maximum GET throughput 40 GB/sec and PUT throughput 20 GB/sec. Our results fall within those parameters.

3. Running the Confluent TOCC Performance Tests

We deployed and started the Confluent Platform using instructions provided [here](#). Particularly, the components required to be started were Kafka Brokers, ZooKeeper and Control Center.

We enable tiered storage on the test cluster using instructions provided [here](#).

We deployed the Brokers such that they exported JMX metrics at a specific port. TOCC JAR collects JMX metrics from this port for use during the tiered storage correctness test. It is common practice to configure the JMX settings for the brokers during startup using the `KAFKA_JMX_OPTS` environment variable. As an example, the following JMX settings can be supplied (please replace `<jmx_port>` and `<host_ip>` with real values):

```
export KAFKA_JMX_OPTS=' -Dcom.sun.management.jmxremote
-Djava.rmi.server.hostname=<host_ip>
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.local.only=false
-Dcom.sun.management.jmxremote.rmi.port=<jmx_port>
-Dcom.sun.management.jmxremote.port=<jmx_port> '
```



We deployed [Trogdor](#) to parallelize tests and workloads to run on Trogdor agents deployed across multiple machines (Kafka Management node type). For parallel produce-consume and retention workloads, the test script leverages Trogdor agent's in-built [ProduceBenchSpec](#) and [ConsumeBenchSpec](#) to generate the load. For parallel tier fetch benchmark and correctness tests, the script leverages the TOCC JAR described above using the agent's in-built [ExternalCommandSpec](#).

To start the Trogdor coordinator as a background process on the coordinator's node:

```
#!/bin/bash -xe
export KAFKA_HEAP_OPTS='-Xmx8G -Xms8G'
/home/bduser/confluent/confluent-7.1.0/bin/trogdor coordinator -c
/home/bduser/confluent/tier-storage-config/tools-conf/trogdor.conf -n node0
&>
/tmp/trogdor-coordinator.log &
```

To start each Trogdor agent as a background process on each agent's node:

```
#!/bin/bash -xe
export KAFKA_HEAP_OPTS='-Xmx8G -Xms8G'
/home/bduser/confluent/confluent-7.1.0/bin/trogdor coordinator -c
/home/bduser/confluent/tier-storage-config/tools-conf/trogdor.conf -n node0
&>
/tmp/trogdor-coordinator.log &
```

To run the tier fetch benchmark:

```
#3. Tier fetch benchmark test (Writes test output to /tmp directory)

./tier-object-store-parallel-test.sh
--test.jar.path=/data/tools/tiered-storage/bin/kafka-tier-compatibility-checker-7.1.0-0-ce-all.jar
--confluent.tools.root=/data/tools/confluent/confluent-7.1.0/bin
--test.config.path=/data/tools/tiered-storage/conf/3-config-tier-fetch-benchmark-test.properties --test.type=test-tiered-storage-fetch >&
/tmp/3-config-tier-fetch-benchmark-test.log &
```



Configuration:

```
confluent.tier.backend=S3
confluent.tier.enable=true
confluent.tier.feature=true
confluent.tier.s3.cred.file.path=/home/bduser/.ssh/credentials
confluent.tier.s3.bucket=confluent
confluent.tier.s3.region=us-east-1
#confluent.tier.s3.sse.algorithm=none
confluent.tier.metadata.replication.factor=1
confluent.tier.s3.aws.endpoint.override=http://poc.min.io:9000

bootstrap.server=10.105.193.208:9092,10.105.193.209:9092,10.105.193.210:9092
,10.
105.193.211:9092,10.105.193.212:9092
debug=true
jmx.port=7203

### DISTRIBUTED ###

trogdor.agent.nodes=node0,node1,node2,node3,node4
trogdor.coordinator.hostname.port=10.105.193.216:8889

#test.binary.task.max.heap.size=10G
test.binary.num.tasks=5
tiering.completed.timeout=1800
test.binary.task.timeout.sec=14400

#output directory
test.metrics.output.dir=/tmp/custom
```

To run a produce-consume workload:

```
#3. Produce-consume workload generator test
./tier-object-store-parallel-test.sh
--test.jar.path=/home/bduser/confluent/tier-storage-config/test-jar/kafka-ti
er-compatibility-checker-7.3.0-0-ce-all.jar
--confluent.tools.root=/home/bduser/confluent/confluent-7.1.0/bin
--test.config.path=/home/bduser/confluent/config/4-config-produce-consume-wo
rkload-generator-test.properties --test.type=test-tiered-storage-load >&
/tmp/4-config-produce-consume-workload-generator-test.log &
```



Configuration:

```
confluent.tier.backend=S3
confluent.tier.enable=true
confluent.tier.feature=true
confluent.tier.s3.cred.file.path=/home/bduser/.ssh/credentials
confluent.tier.s3.bucket=confluent
confluent.tier.s3.region=us-east-1
#confluent.tier.s3.sse.algorithm=none
confluent.tier.metadata.replication.factor=1
confluent.tier.s3.aws.endpoint.override=http://poc.min.io:9000

bootstrap.server=10.105.193.208:9092,10.105.193.209:9092,10.105.193.210:9092
,10.105.193.211:9092,10.105.193.212:9092
debug=true
jmx.port=7203

### DISTRIBUTED ###

trogdor.agent.nodes=node0,node1,node2,node3,node4
trogdor.coordinator.hostname.port=10.105.193.216:8889

#### WORKLOAD GENERATOR ###
num.records=20000000
num.producers=5
num.head.consumers=3
num.tail.consumers=1
num.partitions=100

#test.binary.task.max.heap.size=10G
test.binary.num.tasks=5
tiering.completed.timeout=1800
test.binary.task.timeout.sec=14400
```

To run the produce-consume workload generator with Object Store fault injection:

```
#3. Produce-consume workload generator test
./tier-object-store-parallel-test.sh
--test.jar.path=/data/tools/tiered-storage/bin/kafka-tier-compatibility-checker-7.1.0-0-ce-all.jar
--confluent.tools.root=/data/tools/confluent/confluent-7.1.0/bin
--test.config.path=/data/tools/tiered-storage/conf/3-config-produce-consume-workload-generator-test.properties --test.type=test-tiered-storage-load >&
/tmp/3-config-produce-consume-workload-generator-test.log &
```



Configuration:

```
confluent.tier.backend=S3
confluent.tier.s3.bucket=test-s3bucket
confluent.tier.s3.region=us-east-1
confluent.tier.s3.cred.file.path=/path/to/s3_credentials.properties
bootstrap.server=172.31.81.2:9092,172.31.17.2:9092,172.31.33.3:9092

num.records=200000000
num.producers=6
num.head.consumers=3
num.tail.consumers=1
num.partitions=100

trogdor.agent.nodes=node0,node1,node2
trogdor.coordinator.hostname.port=172.31.17.1:8889
```

To run the retention workload generator test:

```
#3. Retention workload generator test
./tier-object-store-parallel-test.sh
--test.jar.path=/home/bduser/confluent/tier-storage-config/test-jar/kafka-ti
er-compatibility-checker-7.3.0-0-ce-all.jar
--confluent.tools.root=/home/bduser/confluent/confluent-7.1.0/bin
--test.config.path=/home/bduser/confluent/config/5-config-retention-workload
-generator-test.properties --test.type=test-tiered-storage-load >&
/tmp/5-config-retention-workload-generator-test.log &
```

Config:

```
confluent.tier.backend=S3
confluent.tier.enable=true
confluent.tier.feature=true
confluent.tier.s3.cred.file.path=/home/bduser/.ssh/credentials
confluent.tier.s3.bucket=confluent
confluent.tier.s3.region=us-east-1
#confluent.tier.s3.sse.algorithm=none
confluent.tier.metadata.replication.factor=1
confluent.tier.s3.aws.endpoint.override=http://poc.min.io:9000

bootstrap.server=10.105.193.208:9092,10.105.193.209:9092,10.105.193.210:9092
,10.105.193.211:9092,10.105.193.212:9092
debug=true
```



```
jmx.port=7203

### DISTRIBUTED ###

trogdor.agent.nodes=node0,node1,node2,node3,node4
trogdor.coordinator.hostname.port=10.105.193.216:8889

#### FOR FETCHER ###
num.records=100000000
num.producers=3
num.head.consumers=0
num.tail.consumers=0
# 100 MB
segment.bytes=104857600
# 200 MB (not more than 2 segments are retained)
retention.bytes=209715200

#test.binary.task.max.heap.size=10G
test.binary.num.tasks=5
tiering.completed.timeout=1800
test.binary.task.timeout.sec=1440
```

3.1 Results

The throughput between the Kafka hot tier and the MinIO object storage tier was measured under a variety of conditions.

Kafka Brokers	Drives per Broker	Kafka Topics	Producers / Tasks	Consumers / tasks	Partitions per topic	Replicas/ Minimum	Data Ingested	Kafka Ingestion	MinIO Data Rate	Notes
5	1	8	8/6	8/6	100	3/1	20TB	1.22 GB/s	7 GiB/s	Broker limited by the single drive
5	5	8	8/6	8/6	100	3/1	20TB	13.68 GB/s	12 GiB/s	Broker I/O at 100%, 40Gbps network saturated
6	8	8	8/6	8/6	100	3/1	20TB	12.41 GiB/s	N/A	Adding Brokers doesn't improve throughput as network is saturated.



3.2 Interpretation of Results

The first test run using five Kafka Brokers each equipped with 1 NVMe drive saw total throughput limited by I/O.

The second test run added 4 NVMe drives to each Kafka Broker to increase I/O throughput. During this test run, all Kafka Brokers were at 30%-35% CPU utilization and close to 100% I/O utilization. Network utilization was very close to 100% of the 40Gbps network links. At the same time, MinIO nodes experienced 10%-15% CPU utilization and 40% I/O utilization. MinIO performance was gated by network throughput.

To test this theory, we add another Kafka Broker and 3 more NVMe drives to each Kafka Broker, yielding 6 Kafka Brokers with 8 NVMe drives each. No changes were made to the MinIO configuration. Network utilization again approached 100%. Adding an additional Kafka Broker and adding drives to each Kafka Broker did not improve overall throughput – in fact Kafka throughput declined.

In this case the 40 Gbps network is, again, the bottleneck as MinIO gets close to hardware performance for both reads and writes.

4. Conclusion

Based on the results above, we found that MinIO is more than capable of providing high-performance error-free tiering of Kafka data. The 8 Intel Xeon nodes equipped with NVMe drives running MinIO more than met the throughput demands of the Kafka Brokers in each test. We found that MinIO scales up more efficiently than Kafka Brokers.

Finally, the importance of network bandwidth cannot be stressed enough. While MinIO performance will increase on a near linear basis with additional servers, bandwidth will often be the bottleneck and architects should build with those constraints in mind.

