cloudera®

WHITE PAPER

Reference Guide for Deploying and Configuring Apache Kafka

Version: 102

cloudera®

Table of Content

cloudera®

1. Company Overviews

This paper was co-written by Cloudera and Confluent

cloudera®

Cloudera is revolutionizing enterprise data management by offering the first unified platform for big data, an enterprise data hub built on Apache Hadoop. Cloudera offers enterprises one place to store, access, process, secure, and analyze all their data, empowering them to extend the value of existing investments while enabling fundamental new ways to derive value from their data. Cloudera's open source big data platform is the most widely adopted in the world, and Cloudera is the most prolific contributor to the open source Hadoop ecosystem. As the leading educator of Hadoop professionals, Cloudera has trained over 27,000 individuals worldwide. Over 1,400 partners and a seasoned professional services team help deliver greater time to value. Finally, only Cloudera provides proactive and predictive support to run an enterprise data hub with confidence. Leading organizations in every industry plus top public sector organizations globally run Cloudera in production.

Confluent is founded by the team that built Kafka at LinkedIn. They went through the process of fully instrumenting everything that happens in a company and making it available as realtime Kafka feeds to all data systems like Hadoop, Search, Newsfeed and so on. At LinkedIn, this platform scaled to hundreds of billions of messages per day covering everything happening in the company. They open sourced Kafka from its very early days, and have led it to impressive industry-wide adoption across several thousand companies. Now Confluent is focused on building a realtime data platform solution to help other companies get easy access to data as realtime streams.

2. Introduction

Apache Kafka is a distributed publish-subscribe messaging system that is designed to be fast, scalable, and durable. This open source project - licensed under the Apache license - has gained popularity within the Hadoop ecosystem, across multiple industries. Its key strength is the ability to make high volume data available as a real-time stream for consumption in systems with very different requirements—from batch systems like Hadoop, to real-time systems that require low-latency access, to stream processing engines like Apache Spark Streaming that transform the data streams as they arrive. Kafka's flexibility makes it ideal for a wide variety of use cases, from replacing traditional message brokers, to collecting user activity data, aggregating logs, operational application metrics and device instrumentation.

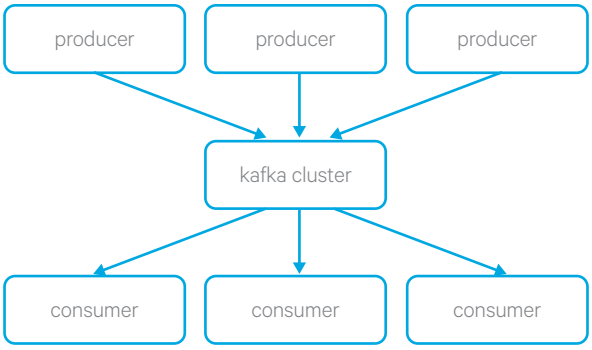
This reference paper provides an overview of the general best practices for deploying and running Kafka as a component of Cloudera's Enterprise Data Hub.

3. Apache Kafka Technology Overview

Kafka's strengths are:

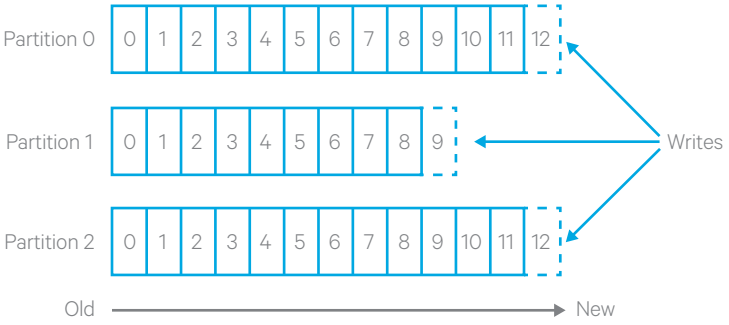
- **High-Throughput & Low Latency:** Even with very modest hardware, Kafka can support hundreds of thousands of messages per second, with latencies as low as a few milliseconds.
- **Scalability:** A Kafka cluster can be elastically and transparently expanded without downtime.
- **Durability & Reliability:** Messages are persisted on disk and replicated within the cluster to prevent data loss.
- **Fault-Tolerance:** Immune to machine failure in the Kafka cluster.
- **High Concurrency:** Ability to simultaneously handle a large number (thousands) of diverse clients, simultaneously writing to and reading from Kafka.

Kafka provides a high-level abstraction called a ***Topic***. A Topic is a category or stream name to which messages are published. Users define a new Topic for each new category of messages. The clients that publish messages to a Topic are called ***Producers***. The clients that consume messages from a Topic are called Consumers. Producers and ***Consumers*** can simultaneously write to and read from multiple Topics. Each Kafka cluster consists of one or more servers called ***Brokers***. Message data is replicated and persisted on the Brokers.



Each Kafka Topic is partitioned, and messages are ordered within each ***partition***. Writes to each partition are sequential, and this is one of the key aspects of Kafka's performance. The messages in the partitions are each assigned a sequential id number called the ***offset*** that uniquely identifies each message within the partition.

Anatomy of a Topic



The partitions of a Topic are distributed over the Brokers of the Kafka cluster with each Broker handling data and requests for a share of the partitions. Each partition is ***replicated*** across a configurable number of Brokers for fault tolerance.

In Kafka, the offset or position of the last read message from a Topic's partition, are maintained by the corresponding Consumer. A Consumer will advance its offset linearly as it reads messages. However, the position is controlled by the Consumer and it can consume messages in any order it likes. For example, a Consumer can reset to an older offset to reprocess. To enable this flexible consumption of messages, the Kafka cluster retains all published messages—whether or not they have been consumed—for a configurable period of time.

Detailed documentation for Kafka is available here.

cloudera®

4. Common Use Cases for Kafka

Log Aggregation

Kafka can be used across an organization to collect logs from multiple services and make them available in standard format to multiple consumers, including Hadoop, Apache HBase, and Apache Solr.

Messaging

Message Brokers are used for a variety of reasons, such as to decouple data processing from data producers, to buffer unprocessed messages, etc. Kafka provides high-throughput, low latency, replication, and fault-tolerance - making it a good solution for large scale message processing applications.

Customer Activity Tracking

Kafka is often used to track the activity of customers on websites or mobile apps. User activity (pageviews, searches, clicks, or other actions users may take) is published by application servers to central Topics with one Topic per activity type. These Topics are available for subscription by downstream systems for monitoring usage in real-time, and for loading into Hadoop or offline data warehousing systems for offline processing and reporting.

Activity tracking is often very high volume as many messages are generated for each user pageview.

Operational Metrics

Kafka is often used for logging operational monitoring data. This involves aggregating statistics from distributed applications to produce centralized feeds of operational data, for alerting and reporting.

Stream Processing

Many users end up doing stage-wise processing of data, where data is consumed from topics of raw data and then aggregated, enriched, or otherwise transformed into new Kafka Topics for further consumption. For example, a processing flow for an article recommendation system might crawl the article content from RSS feeds and publish it to an “articles” Topic; further processing might help normalize or de-duplicate this content to a Topic of cleaned article content; and a final stage might attempt to match this content to users. This creates a graph of real-time data flow. Spark Streaming, Storm, and Samza are popular frameworks that are used in conjunction with Kafka to implement Stream Processing pipelines.

Event Sourcing

Event sourcing is a style of application design where 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.

5. Deploying a Kafka Cluster

Cloudera provides a Kafka Custom Service Descriptor (CSD) to enable easy deployment and administration of a Kafka cluster via Cloudera Manager. The CSD provides granular real-time view of the health of your Kafka Brokers, along with reporting and diagnostic tools. This CSD is available on Cloudera’s downloads page.

For optimal performance, it is highly recommended that production Kafka Brokers be deployed on dedicated machines, separate from the machines on which the rest of your Hadoop cluster runs. Kafka relies on dedicated disk access and large pagecache for peak performance, and sharing nodes with Hadoop processes may interfere with its ability to fully leverage the pagecache.

Kafka is meant to run on industry standard hardware. The machine specs for a Kafka Broker machine will be similar to that of your Hadoop TaskTracker or DataNodes. Though there is no minimum specification that is required, we would suggest machines that have ***at least***:

- Processor with four 2Ghz cores
- Six 7200 RPM SATA drives (JBOD or RAID10).
- 32GB of RAM
- 1Gb Ethernet

The size of your Kafka cluster will depend upon the hardware specifications of the cluster machines, and usage factors like the number of simultaneous Producers and Consumers, data replication parameter, data retention period, etc. For example, you need sufficient memory to buffer data for active readers and writers. Assuming readers and writers are fairly evenly distributed across the Brokers in your cluster, you can do a back-of-the-envelope estimate of memory needs by assuming you want to be able to buffer for at least 30 seconds and compute your memory need as *write_throughput*30*. The disk throughput is important, and can often be a performance bottleneck, and hence more disks are often better. For a more thorough guide to cluster sizing, please refer to Appendix A.

Kafka has a dependency on Apache Zookeeper. We would recommend a dedicated Zookeeper cluster with three or five nodes. Remember that a larger Zookeeper cluster will have slower writes than a smaller cluster, since data has to be propagated to a larger quorum of machines. However, a three-node cluster can only tolerate one machine failure for uninterrupted service, whereas a five-node cluster can tolerate two machine failures. There is no simple formula for the correct size of a Zookeeper cluster, and we ask the user to refer to Zookeeper documentation for more information. Note that Kafka can share a Zookeeper cluster with other applications. However, unless you completely understand the usage patterns of the other applications, we recommend a dedicated cluster for Kafka. Deploying and monitoring Zookeeper clusters is very easy with Cloudera Manager, and hence adding an additional cluster has very low overhead.

6. Configuring a Kafka Cluster

The following section assumes the user is using Kafka binaries based on Apache version 0.8.2.

6.1 Replication, Partitions, and Leaders

As described in Section 3, the data written to Kafka is replicated for fault tolerance and durability. Kafka allows users to set a separate replication factor for each Topic. The replication factor controls how many Brokers will replicate each message that is written. If you have a replication factor of three then up to two Brokers can fail before you will lose access to your data. We recommend using a replication factor of at least two, so that you can transparently bounce machines without interrupting data consumption. However, if you have stronger durability requirements, use a replication factor of three or above.

Topics in Kafka are partitioned, and each Topic has a configurable partition count. The partition count controls how many logs the topic will be sharded into. Producers assign data to partitions in round-robin fashion, to spread the data belonging to a Topic among its partitions. Each partition is of course replicated, but one replica of each partition is selected as a ***Leader Partition***, and all reads and writes go to this lead partition.

Here are some factors to consider while picking an appropriate partition count for a Topic. A partition can only be read by a single Consumer (however, a Consumer can read many partitions). Thus, if your partition count is less than your Consumer count, many Consumers will not receive data for that Topic. Hence, we recommend a partition count that is higher than the maximum number of simultaneous Consumers of the data, so that each Consumer receives data. Similarly, we recommend a partition count that is higher than the number of Brokers, so that the Leader Partitions are evenly distributed among Brokers,

thus distributing the read/write load (Kafka performs random and even distribution of partitions across Brokers). As a reference, many of our customers have Topics with hundreds of partitions each. However, note that Kafka will need to allocate memory for message buffer per partition. If there are a large number of partitions, make sure Kafka starts with sufficient heap space (number of partitions times *replica.fetch.max.bytes*).

6.2 Producers and Consumers

Producers push data to Kafka Brokers, and Consumers pull data from Kafka Brokers. To learn how to write data to and read from Kafka, please refer to the Kafka documentation and examples that ship with the Kafka code base.

Your applications will write data to Brokers using instances of the KafkaProducer class, which implements the Producer API. Producers send requests asynchronously and in parallel, but always return a Future response object that returns the offset, as well as any error that may have occurred when the request is complete. An important configuration to consider while setting up a Producer is the value of the parameter "***acks***", which is the number of acknowledgments the Producer requires the Leader Partition to have received (from the other replicas) before marking a request complete. If acks is set to 0, the Producer will not wait to acknowledge receipt of data by the Brokers, and hence the Producer can not know if data delivery failed and thus there can be data loss. However, setting acks to 0 is likely to give high throughput. For settings that achieve guaranteed data delivery, please refer to Section 6.3.

Kafka messages consist of a fixed-size header and variable length opaque byte array payload. By treating the data as a byte array, Kafka enables you to use a custom serialization format or existing popular serialization formats like Apache Avro, Protobuf, etc. Though there is no maximum message size enforced by Kafka, we recommend writing messages that are no more than 1MB in size. Most customers see optimal throughput with messages ranging from 1-10 KB in size.

Applications can read data from Kafka by using Kafka's Consumer APIs. Kafka has two levels of Consumer APIs. The low-level, "simple" API maintains a connection to a single Broker. This API is completely stateless, with the offset being passed in on every request, allowing the user to maintain this metadata however they choose.

The high-level API hides the details of Brokers from the user and allows consuming off the cluster of machines without concern for the underlying topology. It also maintains the state of what has been consumed. The high-level API also provides the ability to subscribe to topics that match a filter expression (i.e. either a whitelist or a blacklist regular expression).

To parallelize the consumption of data from a Topic, multiple Consumers can form a group and jointly consume a single Topic. Each Consumer in the same group should be given the same *group_id*. This *group_id* is provided in the configuration of the Consumer, and is your way to tell the Consumer which group it belongs to. The Consumers in a group divide up the partitions as fairly as possible, and each partition is consumed by exactly one Consumer in a Consumer group.

6.3 Settings for Guaranteed Message Delivery

First, we need to understand the concept on an "In-Sync Replica" (commonly referred to as ISR). For a Topic partition, an ISR is a follower replica that is caught-up with the Leader Partition, and is situated on a Broker that is alive. Thus, if a leader replica is replaced by an ISR, there will be no loss of data. However, if a non-ISR replica is made a Leader Partition, some data loss is possible since it may not have the latest messages. Thus, to ensure message delivery without data loss, the following settings are important:

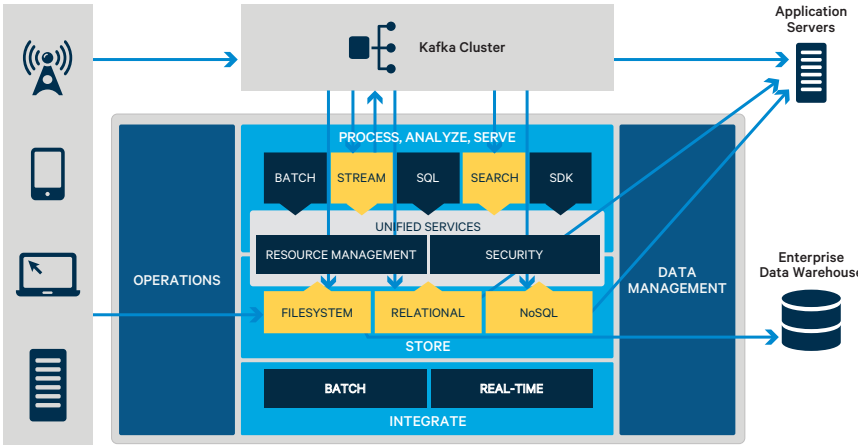
- While configuring a Producer, set "***acks=-1***". This ensures that a message is considered to be successfully delivered only after ALL the ISRs have acknowledged writing the message.

- Set the Topic level configuration ***min.insync.replicas***, which specifies the number of replicas that must acknowledge a write, for the write to be considered successful. If this minimum cannot be met, and acks=-1, then the Producer will raise an exception.
- Set the Broker configuration param ***unclean.leader.election.enable*** to false. This essentially means you are picking durability over availability since Kafka would avoid electing a leader, and instead make the partition unavailable, if no ISR is available to become the next leader safely.

A typical scenario would be to create a Topic with a replication factor of 3, *set min.insync.replicas* to 2, and produce with *request.required.acks* set to -1. However, you can increase these numbers is you have stronger durability requirements.

7. Integrating Kafka with Other Components of Hadoop

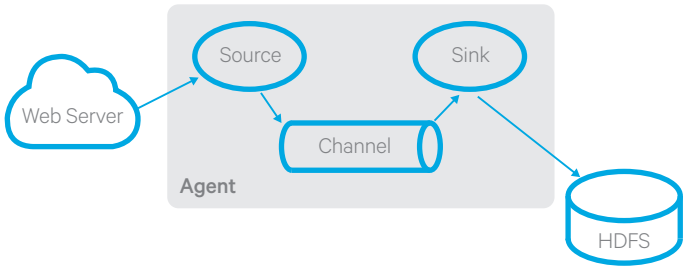
This section describes the seamless integration between Kafka and other data storage, processing and serving layers of CDH.



7.1 Using Apache Flume to Move Data from Kafka to HDFS, HBase, or Solr

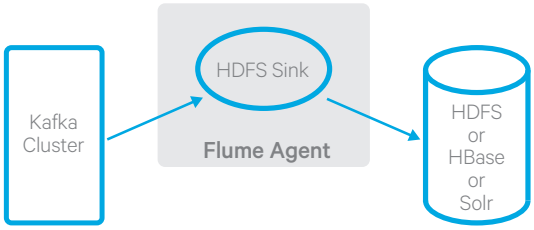
Apache Flume is a very popular service for large scale ingestion of data into HDFS, HBase, or Solr.

Flume has a ***Source, Channel, Sink*** architecture. A Flume Source will read data from the data's original source, the Flume Channel will buffer the data, and the Flume Sink will write the data out to HDFS/HBase/Solr. A Flume ***Agent*** is a (JVM) process that hosts *Sources*, *Sinks*, and *Channels*.



Flume comes with an out-of-the-box library of *Sources* the are optimized for continuous data ingestion from many common sources of data (syslog, http, log4j, etc). Thus, users can perform data ingestion via configuration, without having to write a single line of code.

Flume can use Kafka as a Channel, i.e. Flume *Sources* write to Kafka, and Flume *Sinks* can read data from Kafka. To persist a Kafka Topic to HDFS, setup Flume *Agents* running Flume's HDFS *Sinks*, and configure these *Sinks* to read from Kafka (to write data to HBase or Solr, you will use the corresponding Flume *Sinks*).



Please review the Flume documentation for additional details on how to configure Flume Agents. The following table describes parameters used to configure Kafka as a Flume Channel; required properties are listed in bold.

Table 1: Flume's Kafka Channel Properties

Property Name	Default Value	Description
type		Must be set to org.apache.flume.channel.kafka.KafkaChannel.
brokerList		The brokers the Kafka channel uses to discover topic partitions, formatted as a comma-separated list of hostname:port entries. You do not need to specify the entire list of brokers, but Cloudera recommends that you specify at least two for high availability.
zookeeperConnect		The URIs of the ZooKeeper cluster used by Kafka. This can will be comma-separated list of nodes in the ZooKeeper quorum (for example: zk01.example.com:2181,zk02.example.com:2181, zk03.example.com:2181).
topic	flume-channel	The Kafka topic the channel will use.
groupID	flume	The unique identifier of the Kafka consumer group the channel uses to register with Kafka.
parseAsFlumeEvent	true	Set to true if a Flume source is writing to the channel and will expect AvroDatums with the FlumeEvent schema parseAsFlumeEvent true (org.apache.flume.source.avro. AvroFlumeEvent) in the channel. Set to false if other producers are writing to the topic that the channel is using
readSmallestOffset	false	If true will read all data in the topic. If false will only read data written after the channel has started. Only used when parseAsFlumeEvent is false.
consumer.timeout.ms	100	kafka.consumer.timeout.ms (polling interval when writing to the sink)
Other properties supported by the Kafka producer		Used to configure the Kafka producer. You can use any producer properties supported by Kafka. Prepend the producer property name with the prefix kafka. (for example, kafka. compression.codec).

cloudera®

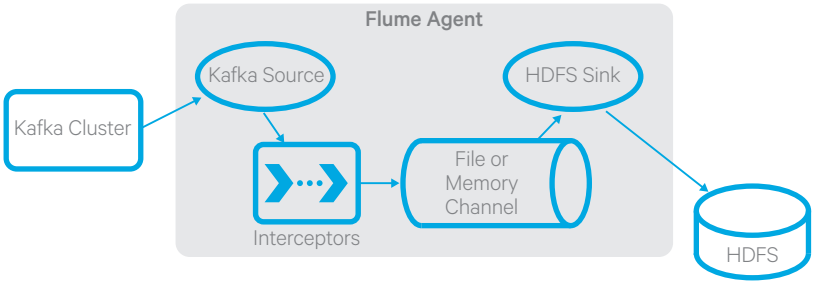
7.2 Using Flume to Write Data to Kafka

The Flume Kafka integration also enables Flume *Sources* to write directly to Kafka (by setting up Kafka as a Flume Channel). This makes it easy to write data to Kafka from data sources and APIs such as log files, http, log4j, syslog, thrift, etc. Flume sources can often be hooked up simply via configuration, without the need to write any code. Thus, the Flume Kafka integration enables the user to write data into Kafka via configuration, without the need for writing Producer code.

7.3 Simple In-Flight Data Processing with Flume Interceptors

Flume provides a key component called the ***Interceptor***, which is part of the Flume extensibility model, to do simple processing of incoming events (data masking, filtering, etc), as they pass through the system. Thus, *Interceptors* provide a lightweight, low latency method to process data flowing through Kafka, before writing the data out to its final destination.

To use Flume Interceptors, use Flume's Kafka *Source* to consume from Kafka, and configure Flume *Agents* as illustrated in this diagram:



Please review the Flume documentation for additional details on how to configure Flume.

7.4 Kafka and Spark Streaming for Complex Real-Time Stream Processing

Spark Streaming is an extension of the Apache Spark platform. It enables high-throughput, scalable processing of continuous streams of data. Combining Kafka with Spark Streaming enables the creation of real-time complex event processing architectures, which can process your data in seconds (or even hundreds of milliseconds). This enables the user to get deep insights from data in near real-time.

In Spark Streaming, the abstraction that represents a continuous stream of data is called a ***DStream***. Spark Streaming can be configured to consume Topics from Kafka, and create corresponding Kafka *DStreams*. Each *DStreams* batches incoming messages into an abstraction called the ***RDD***, which is just an immutable collection of the incoming messages. Each RDD is a micro-batch of the incoming messages, and the micro-batching window is configurable.

Details on initializing Kafka DStreams are provided here, along with a sample application. While configuring a Kafka DStream, you can specify the number of parallel consumer threads. However, the Consumers of a DStream will run on the same Spark Driver node. Thus, to do parallel consumption of a Kafka Topic from multiple machines, instantiate multiple Kafka DStreams, and union their corresponding RDDs before processing.

The processed data can be sent downstream to a data serving platform like HBase or a RDBMS. Often, the processed data is published back into Kafka from Spark Streaming (to Topics that are different from the source Topics), from which multiple downstream applications can consume the processed data.

cloudera®

8. Security

At the moment, Kafka does not provide its own authentication mechanism. However, it integrates perfectly with secured Hadoop and can even use a secured ZooKeeper cluster. The Kafka community is currently working on adding authentication, topic level authorization, and data encryption functionality.

9. Summary

Apache Kafka is an open source, industry standard messaging solution. It provides unmatched throughput, reliability, durability and flexibility. It complements and is well integrated with other components of the Hadoop ecosystem, enabling businesses to process large volumes of data in real-time and thus derive value from the data in real-time. Kafka provides an unparalleled platform to implement use cases that leverage big data with High Velocity.

Appendix A

Cluster Sizing

There are many variables that go into determining the correct hardware footprint for a Kafka cluster. The most accurate way to model your use case is to simulate the load you expect on your own hardware, and you can do this using the load generation tools that ship with Kafka.

However, if we want to size our cluster without simulation, a very simple rule could be to size the cluster based on the amount of disk-space required (which can be computed from the estimated rate at which you get data * the required data retention period).

A slightly more sophisticated estimation can be done based on network and disk throughput requirements.

Let's walk through the details of this estimation.

Let's say we want to plan for a use case with the following characteristics:

W - MB/sec of data that will be written

R - Replication factor

C - Number of Consumer groups (i.e. the number of readers for each write)

Mostly Kafka will be limited by the disk and network throughput so let's first describe the disk and network requirements for the cluster.

The volume of writing that will be done is *W* * *R* (i.e. each replica will write each message). Data will be read by replicas as part of the internal cluster replication and also by Consumers. Since all the replicas other than the master will read each write this means a read volume of (*R*-1) * *W* for replication. In addition each of the *C* Consumers will read each write, so there will be a read volume of *C***W*. This gives the following:

- Writes: *W* * *R*
- Reads: (*R* + *C* - 1) * *W*

However, note that reads may actually be cached, in which case no actual disk I/O will be done. We can model the effect of caching fairly easily. If the cluster has *M* MB of memory, then a write rate of *W* MB/sec will allow *M/(W*R)* seconds of writes to be cached. So a server with 32GB of memory taking writes at 50MB/sec will serve roughly the last 10 minutes of data from cache.

Readers may fall out of cache for a variety of reasons — a slow Consumer, or a failed server that recovers and needs to catch up. An easy way to model this is to assume a number of

lagging readers you will budget for. To model this, let's call the number of lagging readers *L*. A very pessimistic assumption would be that *L* = R + C -1, that is that all Consumers are lagging all the time. A more realistic assumption might be to assume no more than two Consumers are lagging at any given time.

Based on this, we can calculate our cluster-wide I/O requirements:

Disk Throughput (Read + Write): *W* * *R* + *L* * *W*

Network Read Throughput: (*R* + *C* -1) * *W*

Network Write Throughput: *W* * *R*

A single server will provide a given disk throughput as well as network throughput. For example if we had a 1 Gigabit ethernet card with full duplex, then that would give us 125MB/sec read and 125MB/sec write; likewise 6 7200 SATA drives might give us roughly 300 MB/sec read + write throughput. Once we know the total requirements we have, as well as what is provided by one machine, we can divide to get the total number of machines we need.

This will give us a machine count running at maximum capacity, assuming no overhead for network protocols, as well as perfect balance of data and load. Since there is protocol overhead as well as imbalance, we will want to have at least 2x this ideal capacity to ensure we have sufficient capacity.

About Cloudera

Cloudera delivers the modern platform for data management and analytics. The world's leading organizations trust Cloudera to help solve their most challenging business problems with Cloudera Enterprise, the fastest, easiest, and most secure data platform built on Apache Hadoop. Our customers can efficiently capture, store, process, and analyze vast amounts of data, empowering them to use advanced analytics to drive business decisions quickly, flexibly, and at lower cost than has been possible before. To ensure our customers are successful, we offer comprehensive support, training, and professional services. Learn more at cloudera.com.

cloudera.com

1-888-789-1488 or 1-650-362-0488
Cloudera, Inc. 1001 Page Mill Road, Palo Alto, CA 94304, USA