# 5 - Data Acquisition

This chapter covers

- Data Acquisition Considerations
- Publish - Subscribe Messaging Frameworks
- Big Data Collection Systems
- Messaging Queues
- Custom Connectors

In Chapter-1, we proposed a big data stack and various analytics patterns. An important component of the analytics stack and the patterns is the data connectors which allow collecting data from various data sources into a distributed file system or a NoSQL database for batch analysis of data, or which connect the data sources to stream or in-memory processing frameworks for real-time analysis of data.

## 5.1 Data Acquisition Considerations

Before we look at specific tools and frameworks for data acquisition and data connectors, let us first look at the various considerations for data acquisition, which will drive the choice of the tools or frameworks.

### 5.1.1 Source Type

The type of the data source has to be kept into consideration while making a choice for a data connector. Data sources can either publish bulk data in batches or data in small batches (micro-batches) or streaming real-time data.

Some examples of batch data sources are:

- Files
- Logs
- Relational databases

Some examples of real-time data sources are:

- Machines generating sensor data
- Internet of Things (IoT) systems sending real-time data
- Social media feeds
- Stock market feeds

### 5.1.2 Velocity

The velocity of data refers to how fast the data is generated and how frequently it varies. For data with a high velocity (real-time or streaming data), communication mechanisms which have low overhead and low latency are required. Later in this Chapter, we describe how WebSocket and MQTT based connectors can be used for ingesting real-time and streaming data. For such applications, distributed publish-subscribe messaging frameworks such as Apache Kafka are also good choices as they support high throughput and low latency communication. With such frameworks the downstream data consumers can subscribe to the data feeds and receive data in near real-time.

### 5.1.3 Ingestion Mechanism

The data ingestion mechanism can either be a push or pull mechanism. The choice of the specific tool or framework for data ingestion will be driven by the data consumer. If the consumer has the capability (or requirement) to pull data, publish-subscribe messaging frameworks which allow the consumers to pull the data (such as Apache Kafka) or messaging queues can be used. The data producers push data to the a messaging framework or a queue from which the consumers can pull the data. An alternative design approach that is adopted in systems such as Apache Flume is the push approach, where the data sources first push

data to the framework and the framework then pushes the data to the data sinks. Figures 5.1 and 5.2 show the data flow in push-pull and publish-subscribe messaging.

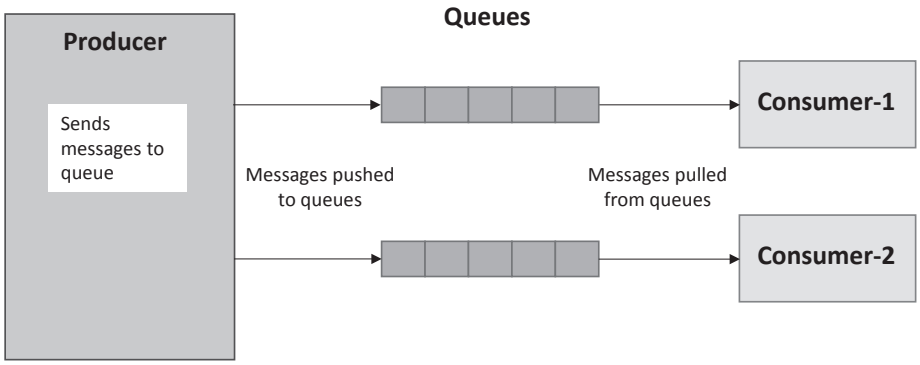


Figure 5.1: Push-Pull messaging

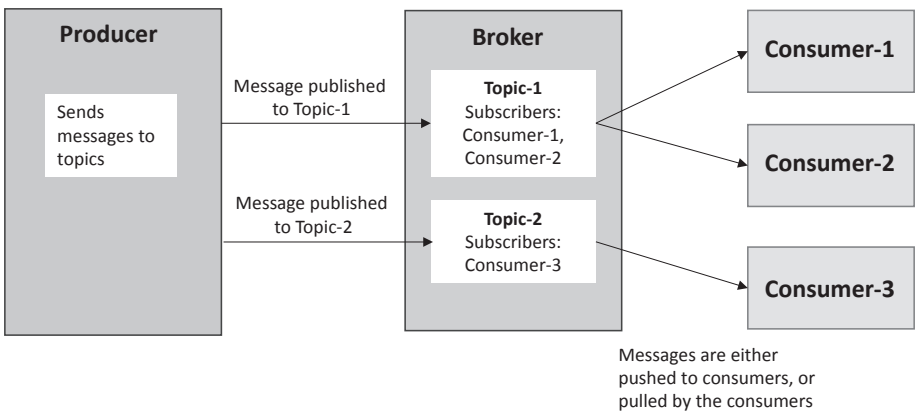


Figure 5.2: Publish - Subscribe Messaging

## 5.2 Publish - Subscribe Messaging Frameworks

Publish-Subscribe is a communication model that comprises publishers, brokers and consumers. Publishers are the sources of data. Publishers send data to topics which are managed by the broker. Publishers are not aware of the consumers. Consumers subscribe to the topics which are managed by the broker. When the broker receives data for a topic from a publisher, it sends the data to all the subscribed consumers. Alternatively, the consumers can pull data for specific topics from the broker.

In this section, we will describe two publish-subscribe messaging frameworks - Apache Kafka and Amazon Kinesis.

### 5.2.1 Apache Kafka

Apache Kafka is a high throughput distributed messaging system. Kafka can also be considered as a distributed, partitioned, replicated commit log service. Kafka can be used for applications such as stream processing, messaging, website activity tracking, metrics collection and monitoring, log aggregation, etc.

#### Architecture

Figure 5.3 shows the architecture and components of Kafka.

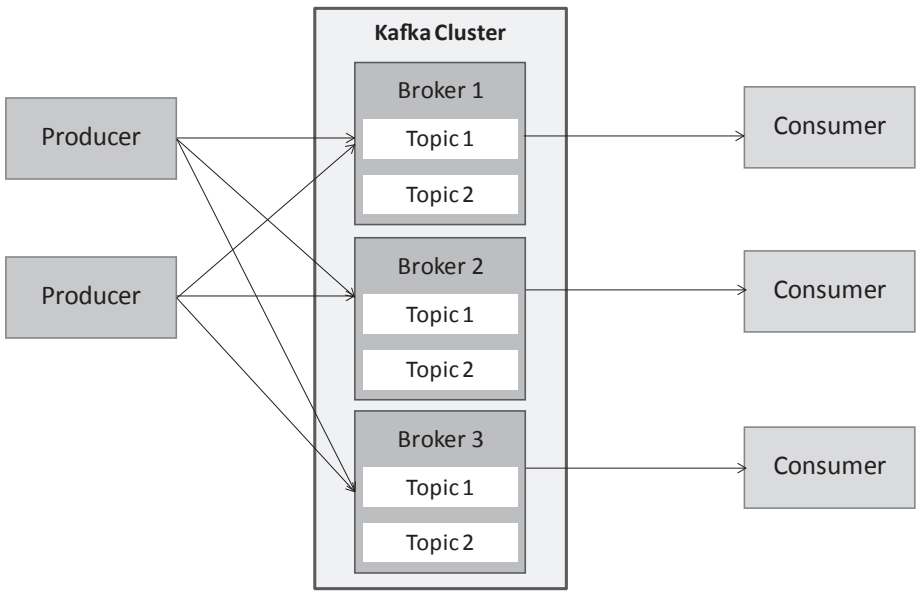


Figure 5.3: Kafka Architecture

A Kafka system includes the following components:

- **Topic:** A topic is a user-defined category to which messages are published.
- **Producer:** Producer is a component that publishes messages to one or more topics.
- **Consumer:** Consumer is a component that subscribes to one or more topics and processes the messages.
- **Broker:** Broker is a component that manages the topics and handles the persistence, partitioning, and replication of the data. A Kafka cluster can have multiple Kafka Brokers (or servers), with each Broker managing multiple topics.

#### Partitions

Kafka topics are subdivided into multiple partitions as shown in Figure 5.4. Each partition is an ordered and immutable sequence of messages. Topics are stored on the disk in the form of partitioned logs (commit logs). The benefit of using partitions is that the log can scale to massive sizes, which will not fit onto the disk of a single server. Partitions also allow multiple consumers to consume messages in parallel.

Partitions are distributed among the brokers, where each broker is typically a separate physical server. Partitions are also replicated across multiple brokers for fault tolerance

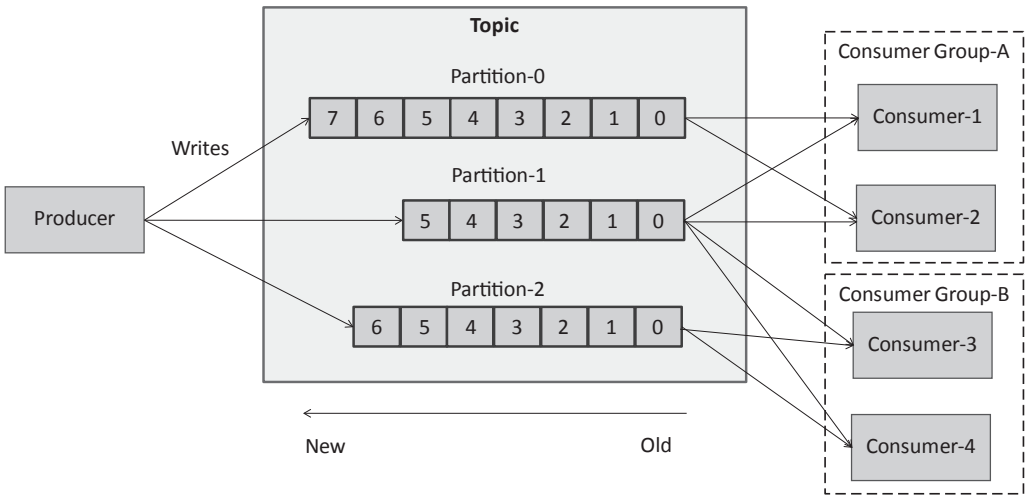


Figure 5.4: Kafka topic partitions

purposes. For each partition, one of the servers acts as the 'leader' for the partition and handles all the writes and reads for the partition. Other servers which hold the partition replicas are called the 'followers'. Partitions and replicas can be reassigned between the brokers as more brokers become available.

#### Publishing Messages

Producers publish messages to topics. A producer decides which message should be published to which partition of a topic. Producers can either publish messages to different partitions of a topic in a round-robin fashion (for load balancing purposes) or publish messages with specific keys to specific partitions.

#### Consuming Messages

Consumers consume (and later process) messages from topics. Each message in a partition is assigned a sequence ID called the offset. Offsets are used by the consumers to track which messages have been consumed. Kafka Brokers are not responsible for keeping a track of the messages consumed by the consumers. The published messages are retained on the disk for a configurable duration of time. Since the topics are retained on the disk, the consumers can replay the messages using the offset. The consumers increment the offset as they consume the messages in a sequence.

Consumers can be grouped together into consumer groups. Each message which is published to a topic by a producer is delivered to one consumer within a consumer group. The consumers within a consumer group can either be separate processes or separate instances. Having multiple consumers within a consumer group makes the system scalable and fault tolerant. The partitions in a topic are assigned to consumers such that each partition is consumed by exactly one consumer in a consumer group. The individual consumer processes can process different partitions in parallel. Since only one consumer process consumes from a given partition, messages are delivered in order. Kafka provides ordering of messages within a partition, but not between partitions.

### Log Storage & Compaction

Kafka is structured to store messages in the form of append-only logs. For each partition of each topic, a log file is maintained which contains an ordered and immutable sequence of messages. Messages are made available to consumers only after they have been committed to the log. This ensure that all the messages which are consumed by the consumers also persist in the logs so that they can be consumed by other consumers if need be. Unlike, queuing systems which delete the messages after they have been consumed, Kafka retains the messages in the log files, which are retained for a certain amount of time (that can be configured). Kafka provides two options for log cleanup policy - *delete* or *compact*. Log for a topic partition is stored as a directory of segment files. The maximum size to which a segment file can grow before a new segment is rolled over in the log can be configured. If the log cleanup policy is set to *delete*, the log segments are deleted when they reach the size or time limits set. If the log cleanup policy is set to *compact*, then log compaction is used to clean out obsolete records. While for temporal data the *delete* retention policy works well, the *compact* policy is useful when you have keyed and mutable data. For example, if each message which is published to a topic is uniquely identified by a primary key (like a row in a database table), and the message values are going to change over time, then it is preferable to use log compaction which removes old and obsolete values for a key. Log compaction ensures that at least the last known value for each message key for each topic partition is retained.

### Using Kafka

In this section, we will describe some examples of Kafka producers and consumers. We will the use the Kafka python library called *kafka-python* for the examples. The *kafka-python* library can be installed using *pip* as follows:

```
sudo pip install kafka-python
```

Kafka uses Zookeeper for state coordination. A Kafka topic can be created as follows:

```
bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic test
```

If Zookeeper is running on a separate machine than Kafka, then change the Zookeeper hostname from localhost to the correct hostname.

To make sure that the topic has been created, you can use the following command to list all topics:

```
bin/kafka-topics.sh --list --zookeeper localhost:2181
```

Box 5.1 shows an example of a Kafka producer that sends messages synchronously.

**Box 5.1: Kafka Producer for sending messages synchronously**

```
import time
from datetime import datetime
```

```
from kafka.client import KafkaClient
from kafka.producer import SimpleProducer

client = KafkaClient("localhost:6667")
producer = SimpleProducer(client)

while True:
   ts=time.time()
   timestamp = datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
   data = "This is a test string generated at:  " + str(timestamp)

   producer.send_messages('test', data)

   time.sleep(1)
```

Box 5.2 shows an example of a Kafka producer that sends messages asynchronously.

**■ Box 5.2: Kafka Producer for sending messages asynchronously**

```
import time
from datetime import datetime
from kafka.client import KafkaClient
from kafka.producer import SimpleProducer

client = KafkaClient("localhost:6667")
producer = SimpleProducer(client, async=True)

while True:
   ts=time.time()
   timestamp = datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
   data = "Test message sent asynchronously at:  " + str(timestamp)

   producer.send_messages('test', data)

   time.sleep(1)
```

Box 5.3 shows an example of a Kafka producer that sends messages in batch. The producer collects the messages and sends the messages after 20 messages are collected or after every 60 seconds.

**■ Box 5.3: Kafka Producer for sending messages in batch**

```
import time
from datetime import datetime
from kafka.client import KafkaClient
from kafka.producer import SimpleProducer

client = KafkaClient("localhost:6667")

#The following producer will collect messages in batch
#and send them to Kafka after 20 messages are
```

```
# collected or every 60 seconds

producer = SimpleProducer(client,batch_send=True,
        batch_send_every_n=20,
        batch_send_every_t=60)

while True:
   ts=time.time()
   timestamp = datetime.fromtimestamp(ts).strftime('%Y-%m-%d %H:%M:%S')
   data = "This is a test string generated at:  " + str(timestamp)

   producer.send_messages('test', data)

   time.sleep(1)
```

Box 5.4 shows an example of a Kafka producer that sends keyed messages. The default partitioner class for partitioning messages is the *HashedPartitioner* that partitions based on the hash of the key. Another option is to use the *RoundRobinPartitioner* that sends messages to different partitions in a round-robin fashion.

**■ Box 5.4: Kafka Producer for sending keyed messages**

```
import time
import datetime
from kafka import KafkaClient, KeyedProducer,
from kafka import HashedPartitioner, RoundRobinPartitioner

kafka = KafkaClient("localhost:6667")

#Default partitioner is HashedPartitioner
producer = KeyedProducer(kafka)
producer.send("test", "key1", "Test message with key1")
producer.send("test", "key2", "Test message with key2")

#Using RoundRobinPartitioner
producer = KeyedProducer(kafka, partitioner=RoundRobinPartitioner)
producer.send("test", "key3", "Test message with key3")
producer.send("test", "key4", "Test message with key4")
```

Box 5.5 shows an example of a Kafka consumer that consumes messages from a topic.

**■ Box 5.5: Kafka Consumer**

```
from kafka.client import KafkaClient
from kafka.consumer import SimpleConsumer

client = KafkaClient("localhost:6667")
consumer = SimpleConsumer(client, "test-group", "test")

for message in consumer:
   #Print message object
```

```
   print message

   #Print only message value
   print message.message.value
```

Box 5.6 shows an alternative implementation of a Kafka consumer.

**■ Box 5.6: Kafka Consumer - alternative implementation**

```
from kafka.client import KafkaClient
from kafka.consumer import KafkaConsumer

client = KafkaClient("localhost:6667")
consumer = KafkaConsumer("test", metadata_broker_list=['localhost:6667'])

while True:
   data = consumer.next().value
   print data
```

### 5.2.2 Amazon Kinesis

Amazon Kinesis is a fully managed commercial service for ingesting real-time streaming data. Kinesis scales automatically to handle high volume streaming data coming from a large number of sources. The streaming data collected by Kinesis can be processed by applications running on Amazon EC2 instances or any other compute instance that can connect to Kinesis.

Kinesis allows rapid and continuous data intake and support data blobs of size upto 50Kb. The data producers write data records to Kinesis streams. A data record comprises a sequence number, a partition key, and the data blob. Data records in a Kinesis stream are distributed in shards. Each shard provides a fixed unit of capacity, and a stream can have multiple shards. A single shard of throughput allows capturing 1MB per second of data, at up to 1,000 PUT transactions per second and allows applications to read data at up to 2 MB per second.

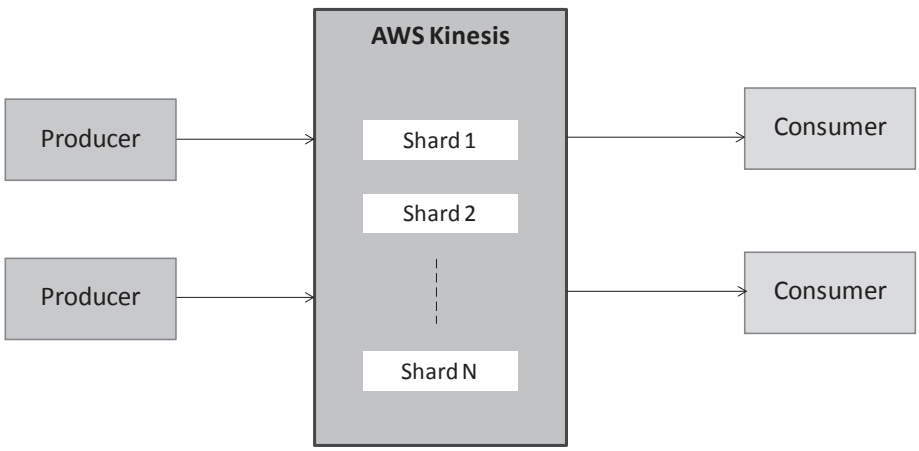


Figure 5.5: Amazon Kinesis architecture

Box 5.7 shows a Python program for writing to a Kinesis stream. In this example, a connection to the Kinesis service is first established and then a new Kinesis stream is either created (if not existing) or described. The data is written to the Kinesis stream using the *kinesis.put_record* function.

**■ Box 5.7: Python program for writing to a Kinesis stream**

```
from random import randrange
import time
import datetime
import boto
import json
from boto.kinesis.exceptions import ResourceNotFoundException

ACCESS_KEY = "<enter>"
SECRET_KEY = "<enter>"

kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

#Send some synthetic data to AWS Kinesis
while True:
 ts=time.time()

 data = str(ts) + `,' + str(randrange(0,60)) + `,' +
   str(randrange(0,100)) + `,' + str(randrange(5000,12000)) +
    `,' + str(randrange(0,100))

 print data
 response = kinesis.put_record(`forestfire', data, data)
 print response
 time.sleep(1)
```

Box 5.8 shows a Python program for reading from a Kinesis stream. In this example, a shard iterator is obtained using the *kinesis.get_shard_iterator* function. The shard iterator specifies the position in the shard from which you want to start reading data records sequentially. The data is read using the *kinesis.get_records* function which returns one or more data records from a shard.

**■ Box 5.8: Python program for reading from a Kinesis stream**

```
from random import randrange
import time
import datetime
import boto
import json
from boto.kinesis.exceptions import ResourceNotFoundException

ACCESS_KEY = "<enter>"
SECRET_KEY = "<enter>"
```

```
kinesis = boto.connect_kinesis(aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

response = kinesis.describe_stream('forestfire')

if response['StreamDescription']['StreamStatus'] == 'ACTIVE':
 shard_id = response['StreamDescription']['Shards'][0]['ShardId']

response = kinesis.get_shard_iterator('forestfire', shard_id,
'TRIM_HORIZON')
shard_iterator = response['ShardIterator']

response = kinesis.get_records(shard_iterator)
shard_iterator = response['NextShardIterator']

for record in response['Records']:
 print record
```

## 5.3 Big Data Collection Systems

Data collection systems allow collecting, aggregating and moving data from various sources (such as server logs, databases, social media, streaming sensor data from Internet of Things devices and other sources) into a centralized data store (such as a distributed file system or a NoSQL database).

### 5.3.1 Apache Flume

Apache Flume is a distributed, reliable, and available system for collecting, aggregating, and moving large amounts of data from different data sources into a centralized data store.

#### Flume Architecture

Flume's architecture is based on data flows and includes the following components:

- **Source**: Source is the component which receives or polls for data from external sources. A Flume data flow starts from a source. For example, Flume source can receive data from a social media network (using streaming APIs).
- **Channel**: After the data is received by a Flume source, the data is transmitted to a channel. Each channel in a data flow is connected to one sink to which the data is drained. A data flow can comprise of multiple channels, where a source writes the data to multiple channels.
- **Sink**: Sink is the component which drains data from a channel to a data store (such as a distributed file system or to another agent). Each sink in a data flow is connected to a channel. Sinks either deliver data to its final destination or are chained to other agents.
- **Agent**: A Flume agent is a collection of sources, channels and sinks. Agent is a process that hosts the sources, channels and sinks from which the data moves from an external source to its final destination.
- **Event**: An event is a unit of data flow having a payload and an optional set of attributes. Flume sources consume events generated by external sources.

Flume uses a data flow model which includes sources, channels and sinks, encapsulated into agents. Figure 5.7 shows some examples of Flume data flows. The simplest data flow

has one source, one channel and one sink. Sources can multiplex data to multiple channels for either load balancing purposes, or, for parallel processing. More complex data flows can be created by chaining multiple agents where the sink of one agent delivers data to a source of another agent.

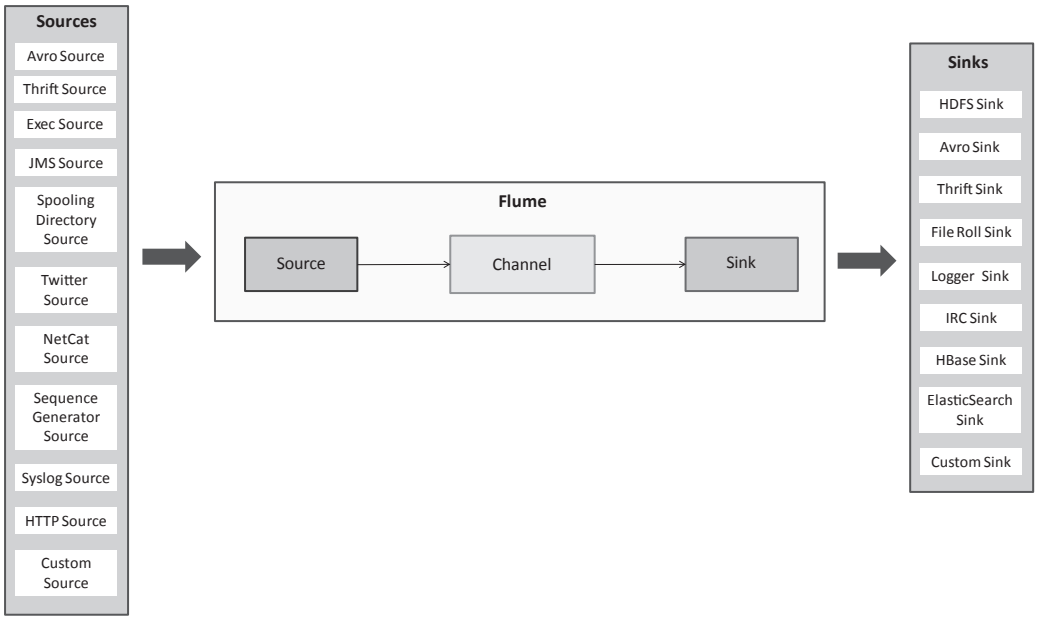


Figure 5.6: Apache Flume architecture

Flume agents are defined in the configuration files. Box 5.9 shows a generic definition of a Flume agent. In the configuration file, first the sources, channels and sinks for the agent are listed and then each source, channel and sink is defined. Finally the bindings between the sources, channels and sinks are defined.

**■ Box 5.9: Generic definition of a Flume agent**

```
<agent name>.sources = <source-1> <source-2> ...  <source-N>
<agent name>.channels = <channel-1> <channel-2> ...  <channel-N>
<agent name>.sinks = <sink-1> <sink-2> ...  <sink-N>

# Define sources
<agent name>.sources.<source-1>.type = <source type>
:
<agent name>.sources.<source-N>.type = <source type>

# Define sinks
<agent name>.sinks.<sink-1>.type = <sink type>
:
<agent name>.sinks.<sink-1>.type = <sink type>


# Define channels
```

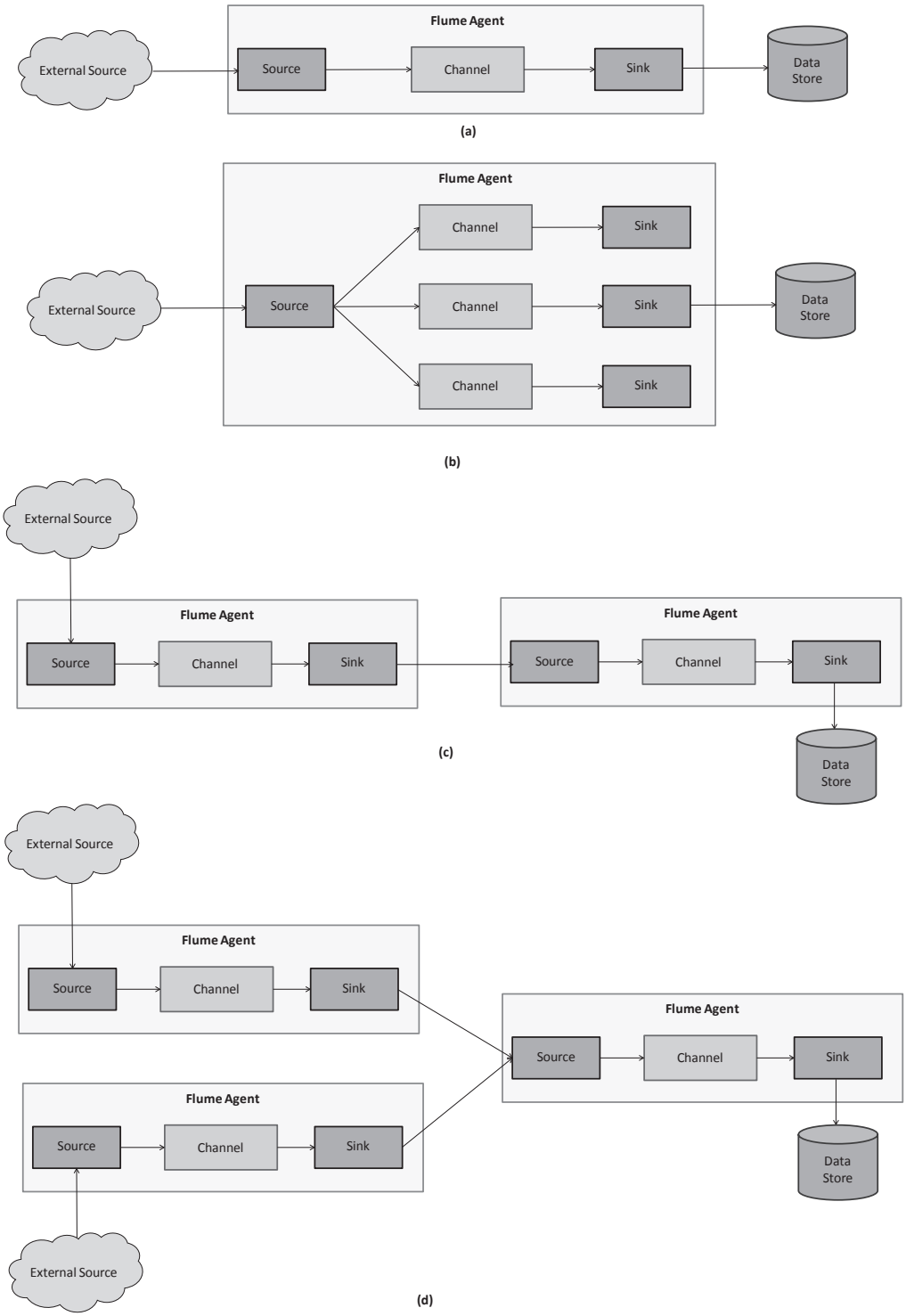


Figure 5.7: Flume data flow examples

```
myagent.channels.<channel-1>.type = <channel type>
:
myagent.channels.<channel-N>.type = <channel type>

# Bind the sources and sinks to the channels
myagent.sources.<source-1>.channels = <channel-1>
myagent.sinks.<sink-1>.channel = <channel-1
:
myagent.sources.<source-N>.channels = <channel-1> ...  <channel-N>
myagent.sinks.<sink-N>.channel = <channel-N
```

```
■ #Format of command to run a Flume agent
#sudo flume-ng agent -c <conf file path> -f <conf file> -n <agent name>

#Example
sudo flume-ng agent -c /etc/flume/conf -f
     /etc/flume/conf/flume.conf -n myagent
```

#### Flume Sources

Flume comes with multiple built-in sources that allow collecting and aggregating data from a wide range of external systems. Flume also provides the flexibility to add custom sources.

- **Avro Source**: Apache Avro is a data serialization system that provides a compact and fast binary data format. Avro uses an Interface Definition Language (IDL) to define the structure of data in the form of schemas. Avro is defined with JSON, and the schema is always stored with the data, which allows the programs reading the data to interpret the data. Avro can also be used with Remote Procedure Calls (RPC) where the client and server exchange the schemas in the handshake process. Avro provides serialization functionality similar to other systems such as Thrift and Protocol Buffers. The Flume Avro source receives events from external Avro client streams. An Avro source can be setup using the following properties in the Flume configuration file for the agent:

  ```
  ■ myagent.sources = source1
  myagent.sources.source1.type = avro
  myagent.sources.source1.bind = 0.0.0.0
  myagent.sources.source1.port = 4141
  ```

  The *bind* and *port* properties specify the hostname of the external Avro client and the Avro port.

- **Thrift Source**: Apache Thrift is a serialization framework similar to Avro. Thrift provides a software stack and a code generation engine to build services that transparently and efficiently work with multiple programming languages. Like Avro, Thrift also provides a stack for Remote Procedure Calls (RPC). The Flume Thrift source receives events from external Thrift client streams. A Thrift source can be setup using the following properties in the Flume configuration file for the agent:

  ```
  ■ myagent.sources = source1
  myagent.sources.source1.type = thrift
  ```

```
myagent.sources.source1.bind = 0.0.0.0
myagent.sources.source1.port = 4141
```

- **Exec Source**: Exec source can be used to ingest data from the standard output. When an agent with an Exec source is started, it runs the Unix command (specified in the Exec source definition) and continues to receive data from the standard output as long as the process runs. The typical use case for the Exec source is the *tail* command which emits few lines of any text file given to it as an input and writes them to standard output. The *tail* when used with the -F options outputs the appended data as the file grows. The box below shows an example of setting up an Exec source with the *tail* command.

```
■ myagent.sources = source1
myagent.sources.source1.type = exec
myagent.sources.source1.command = tail -F /var/log/eventlog.log
```

- **JMS Source**: Java Message Service (JMS) is a messaging service that can be used by Java applications to create, send, receive, and read messages. The JMS source receives messages from a JMS queue or topic. The box below shows an example of setting up a JMS source:

```
■ myagent.sources = s1
myagent.sources.s1.type = jms
myagent.sources.s1.initialContextFactory =
      org.apache.activemq.jndi.ActiveMQInitialContextFactory
myagent.sources.s1.connectionFactory = GenericConnectionFactory
myagent.sources.s1.providerURL = tcp://mqserver:61616
myagent.sources.s1.destinationName = DATA
myagent.sources.s1.destinationType = QUEUE
```

  To connect with a JMS destination an initial context factory name, a connection factory and a provider URL are required. The destination type can either be a queue or a topic.
- **Spooling Directory Source**: Spooling Directory source is useful for ingesting log files. A spool directory is setup on the disk from where the Spooling Directory source ingests the files. To use the source for ingesting logs, the log generation system is setup such that when the log files are rolled over they are moved to the spool directory. The Spooling Directory source parses the files and creates events. The parsing logic can be configured for the source. The default logic is to parse each line as an event. Though an alternative approach to Spooling Directory source is to use Exec source with *tail* command, however, it is not as reliable. The box below shows an example of setting up a Spooling Directory source:

```
■ myagent.sources = source1
myagent.sources.source1.type = spooldir
myagent.sources.source1.spoolDir = /var/log/apache/flumeSpool
myagent.sources.source1.fileHeader = true
```

- **Twitter Source**: The Flume Twitter source connects to the Twitter streaming API and receives tweets in real-time. The Twitter source converts the tweet objects to Avro

format before sending them to the downstream channel. The box below shows an example of setting up a Twitter source:

```
■ myagent.sources = source1
myagent.sources.source1.type =
      org.apache.flume.source.twitter.TwitterSource
myagent.sources.source1.consumerKey = CONSUMER_KEY
myagent.sources.source1.consumerSecret = CONSUMER_SECRET
myagent.sources.source1.accessToken = ACCESS_TOKEN
myagent.sources.source1.accessTokenSecret = ACCESS_TOKEN_SECRET
myagent.sources.source1.maxBatchSize = 10
myagent.sources.source1.maxBatchDurationMillis = 200
```

Before setting up the Twitter source, you will need to create a Twitter application from the Twitter developer account and obtain the consumer and access tokens and secrets for the application.

- **NetCat Source**: NetCat is a simple Unix utility which reads and writes data across network connections, using TCP or UDP protocol. The NetCat source listens to a specific port to which the data is written by a NetCat client and turns each line of text received into a Flume event. The box below shows an example of setting up a NetCat source:

```
■ myagent.sources = source1
myagent.sources.source1.type = netcat
myagent.sources.source1.bind = 0.0.0.0
myagent.sources.source1.port = 6666
```

- **Sequence Generator Source**: Sequence Generator source generates events with a sequence of numbers starting from 0 and incremented by 1. This source is mainly used for testing purposes. The box below shows an example of setting up a Sequence Generator source:

```
■ myagent.sources = source1
myagent.sources.source1.type = seq
```

- **Syslog Source**: Syslog source is used for ingesting syslog data. The box below shows an example of setting up a Syslog TCP source.

```
■ myagent.sources = source1
myagent.sources.source1.type = syslogtcp
myagent.sources.source1.host = localhost
myagent.sources.source1.port = 5140
```

- **HTTP Source**: HTTP source receives HTTP events (POST or GET requests) and converts them into Flume events. While the source can receive events in the form of HTTP POST and GET requests, GET command is used for experimentation only. To convert the HTTP requests into events, a pluggable handler is used. The default handler is JSONHandler, which expects an array of JSON objects. The box below shows an example of setting up a HTTP source:

```
■ myagent.sources = source1
myagent.sources.source1.type = http
myagent.sources.source1.bind = localhost
myagent.sources.source1.port = 81
myagent.sources.source1.handler =
      org.apache.flume.source.http.JSONHandler
```

The *bind* and *port* properties specify the hostname and port on which the source should listen to.

- **Custom Source**: Flume allows customs sources to be integrated into the system. Custom sources are implemented in Java. The Java class files of the custom source along with the dependencies are included in the classpath of the Flume agent and also specified in the agent configuration file shown below:

```
■ myagent.sources = source1
myagent.sources.source1.type = org.example.MySource
```

#### Flume Sinks

Flume comes with multiple built-in sinks. Each sink in a Flume agent connects to a channel and drains the data from the channel to a data store.

- **HDFS Sink**: The Hadoop Distributed File System (HDFS) Sink drains events from a channel to HDFS. The data is written to HDFS in the form of a configurable file type. HDFS sink supports SequenceFile, DataStream and CompressedStream file types. HDFS sink allows the files to be rolled either when the size of the file exceeds a certain limit, or after a specified interval, or after a certain number of events have been written to a file. The box below shows an example of setting up an HDFS sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = hdfs
myagent.sinks.sink1.hdfs.fileType = DataStream
myagent.sinks.sink1.hdfs.path = /flume/events
myagent.sinks.sink1.hdfs.filePrefix = eventlog
myagent.sinks.sink1.hdfs.fileSuffix = .log
myagent.sinks.sink1.hdfs.batchSize = 1000
```

- **Avro Sink**: An Avro sink retrieves events from a channel and drains the events to a downstream host. The box below shows an example of setting up an Avro sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = avro
myagent.sinks.sink1.hostname = 10.10.10.10
myagent.sinks.sink1.port = 4545
```

- **Thrift Sink**: A Thrift sink retrieves events from a channel and drains the events to a downstream host. The box below shows an example of setting up an Thrift sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = thrift
myagent.sinks.sink1.hostname = 10.10.10.10
myagent.sinks.sink1.port = 4545
```

- **File Roll Sink**: A File Roll sink drains the events to a file on the local filesystem. The box below shows an example of setting up an File Roll sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = file_roll
myagent.sinks.sink1.sink.directory = /var/log/flume
```

- **Logger Sink**: A Logger sink retrieves events from a channel and logs the events. The box below shows an example of setting up an Logger sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = logger
```

- **IRC Sink**: An IRC sink retrieves events from a channel and drains the events to an IRC host. The box below shows an example of setting up an IRC sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = irc
myagent.sinks.sink1.hostname = irc.example.com
myagent.sinks.sink1.nick = flume
myagent.sinks.sink1.chan = #flume
```

- **HBaseSink**: An HBase sink retrieves events from a channel and drains the events to an HBase table. The box below shows an example of setting up an HBase sink:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = hbase
myagent.sinks.sink1.table = mytable
myagent.sinks.sink1.columnFamily = myfam
myagent.sinks.sink1.serializer =
     org.apache.flume.sink.hbase.RegexHbaseEventSerializer
```

- **Custom Sink**: Flume allows customs sinks to be integrated into the system. Custom sinks are implemented in Java. The Java class files of the custom sink along with the dependencies are included in the classpath of the Flume agent and also specified in the agent configuration file shown below:

```
■ myagent.sinks = sink1
myagent.sinks.sink1.type = org.example.MySink
```

#### Flume Channels

Channels store the events while they are being moved from a source to sink.

- **Memory Channel**: Memory channel stores the events in the memory and provides

high throughput. However, in the event of an agent failure, the events can be lost. The box below shows an example of setting up a memory channel.

```
■ myagent.channels = channel1
myagent.channels.channel1.type = memory
myagent.channels.channel1.capacity = 10000
myagent.channels.channel1.transactionCapacity = 10000
myagent.channels.channel1.byteCapacityBufferPercentage = 20
myagent.channels.channel1.byteCapacity = 800000
```

- **File Channel**: File channel stores the events in files on the local filesystem. Events are stored in a checkpoint file in the data directory specified in the channel configuration. The a maximum file size for the checkpoint file can be specified. The box below shows an example of setting up a file channel.

```
■ myagent.channels = channel1
myagent.channels.channel1.type = file
myagent.channels.channel1.checkpointDir = /mnt/flume/checkpoint
myagent.channels.channel1.dataDirs = /mnt/flume/data
```

- **JDBC Channel**: JDBC channel stores the events in an embedded Derby database. This channel provides a durable storage for events, and the events can be recovered easily in case of agent failures. The box below shows an example of setting up a JDBC channel.

```
■ myagent.channels = channel1
myagent.channels.channel1.type = jdbc
```

- **Spillable Memory Channel**: Spillable Memory channel stores events in an in-memory queue and when the queue fills up, the events are spilled onto the disk. This channel provides high throughput and fault tolerance. The box below shows an example of setting up a Spillable Memory channel.

```
■ myagent.channels = channel1
myagent.channels.channel1.type = SPILLABLEMEMORY
myagent.channels.channel1.memoryCapacity = 10000
myagent.channels.channel1.overflowCapacity = 1000000
myagent.channels.channel1.byteCapacity = 800000
myagent.channels.channel1.checkpointDir = /mnt/flume/checkpoint
myagent.channels.channel1.dataDirs = /mnt/flume/data
```

  Maximum number of events stored in a memory queue are specified using the *memoryCapacity* property and the maximum size of the memory queue is specified using the *byteCapacity* property. The in-memory queue is considered full, and the events are spilled to the disk when either the *memoryCapacity* or *byteCapacity* limit is reached.
- **Custom Channel**: Flume allows customs channels to be integrated into the system. Custom channels are implemented in Java. The Java class files of the custom channel along with the dependencies are included in the classpath of the Flume agent and also

specified in the agent configuration file shown below:

```
■ myagent.channels = channel1
myagent.channels.channel1.type = org.example.MyChannel
```

### Channel Selectors

Flume agents can have a single source connected to multiple channels. In such cases, the channel selector defines policy about distributing the events among the channels connected to a single source.

- **Replicating Channel Selector**: The default channel selected is the replicating selector, which replicates events received from the source to all the connected channels. The box below shows an example of the configuration of an agent which has a single source connected to three channels and uses a replicating channel selector.

```
■ myagent.sources = source1
myagent.channels = channel1 channel2 channel3
myagent.source.source1.selector.type = replicating
myagent.source.source1.channels = channel1 channel2 channel3
myagent.source.source1.selector.optional = channel3
```

- **Multiplexing Channel Selector**: Multiplexing channel selector distributes events from a source to all the connected channels. The box below shows an example of the configuration of an agent which has a single source connected to three channels and uses a multiplexing channel selector.

```
■ myagent.sources = source1
myagent.channels = channel1 channel2 channel3
myagent.sources.source1.selector.type = multiplexing
myagent.sources.source1.selector.header = country
myagent.sources.source1.selector.mapping.IN = channel1
myagent.sources.source1.selector.mapping.US = channel2
myagent.sources.source1.selector.default = channel3
```

  The *header* property specifies the attribute name to check for distributing the events among the channels and the *mapping* properties specify the mappings between the attribute values and the channels. For example, in the above configuration the *header* property is set to the country attribute. All the events which have the country attribute value as IN are sent to channel1 while all the events with country attribute value as the US are sent to channel2. The default channel is set as channel3.

- **Custom Channel Selector**: Flume allows customs channel selectors to be integrated into the system. Custom channel selectors are implemented in Java. The Java class files of the custom channel selector along with the dependencies are included in the classpath of the Flume agent and also specified in the agent configuration file shown below:

```
■ myagent.sources = source1
myagent.channels = channel1
myagent.sources.source1.selector.type =
```

```
      org.example.MyChannelSelector
```

#### Sink Processors

Flume allows creating sink groups where a channel can be attached to a sink group to which the events are drained. A sink processor defines how the events are drained from a channel to a sink. Sink processors enable parallelism, priorities, and automatic failover.

- **Load balancing Sink Processor**: The load balancing sink processor allows load balancing of events drained from a channel between the sinks in the attached sink group. The load is distributed among the list of sinks specified using a round robin or random selection mechanism. The box below shows an example of an agent with a sink group and a load balancing sink processor.

```
■ myagent.sinkgroups = group1
myagent.sinkgroups.group1.sinks = sink1 sink2
myagent.sinkgroups.group1.processor.type = load_balance
myagent.sinkgroups.group1.processor.backoff = true
myagent.sinkgroups.group1.processor.selector = random
```

- **Failover Sink Processor**: With Failover Sink processor, priorities can be assigned to sinks between in a sink group. The attached channel then drains the events to the highest priority sink. When the highest priority sink fails, the events are drained to the sink with one lower priority, providing automatic failover. The box below shows an example of an agent with a sink group and a failover sink processor.

```
■ myagent.sinkgroups = group1
myagent.sinkgroups.group1.sinks = sink1 sink2
myagent.sinkgroups.group1.processor.type = failover
myagent.sinkgroups.group1.processor.priority.sink1 = 2
myagent.sinkgroups.group1.processor.priority.sink2 = 4
myagent.sinkgroups.group1.processor.maxpenalty = 10000
```

#### Flume Interceptors

Flume interceptors allow events to be modified, filtered or dropped as they flow from the source to a channel. Interceptors are connected to the source. Interceptors can also be chained to each other.

- **Timestamp Interceptor**: The Timestamp interceptor adds the current timestamp to the headers of the events processed. Timestamp interceptor can be configured as follows:

```
■ myagent.sources = source1
myagent.sources.source1.interceptors = i1
myagent.sources.source1.interceptors.i1.type = timestamp
```

- **Host Interceptor**: The Host interceptor adds the hostname of the Flume agent to the headers of the events processed. Host interceptor can be configured as follows:

```
■ myagent.sources = source1
myagent.sources.source1.interceptors = i1
```

```
myagent.sources.source1.interceptors.i1.type = host
myagent.sources.source1.interceptors.i1.hostHeader = hostname
myagent.sources.source1.interceptors.i1.useIP = false
```

- **Static Interceptor**: Static interceptor adds a static header to the events processed. The box below shows an example of adding a static header, country, with the value set to US.

```
■ myagent.sources = source1
myagent.sources.source1.interceptors = i1
myagent.sources.source1.interceptors.i1.type = static
myagent.sources.source1.interceptors.i1.key = country
myagent.sources.source1.interceptors.i1.value = US
```

- **UUID Interceptor**: The UUID adds a universally unique identifier to the headers of the events processed. UUID interceptor can be configured as follows:

```
■ myagent.sources = source1
myagent.sources.source1.interceptors = i1
myagent.sources.source1.interceptors.i1.type = uuid
myagent.sources.source1.interceptors.i1.headerName=id
```

- **Regex Filtering Interceptor**: Regex Filtering interceptor applies a regular expression to the event body and filters the matching events. The events matching the regular expression can either be included or excluded. Regex Filtering interceptor can be configured as follows:

```
■ myagent.sources = source1
myagent.sources.source1.interceptors = i1
myagent.sources.source1.interceptors.i1.type = regex_filter
myagent.sources.source1.interceptors.i1.regex = .*
myagent.sources.source1.interceptors.i1.excludeEvents = false
```

#### Flume Examples

Box 5.10 shows an example of setting up a Flume agent with NetCat Source & File Roll Sink.

**■ Box 5.10: Flume agent with NetCat Source & File Roll Sink**

```
myagent.sources = r1
myagent.channels = c1
myagent.sinks = k1

# Define source
myagent.sources.r1.type = netcat
myagent.sources.r1.bind = 0.0.0.0
myagent.sources.r1.port = 6666

#Define Sink
myagent.sinks.k1.type = file_roll
myagent.sinks.k1.sink.directory = /var/log/flume
```

```
#Define Channel
myagent.channels.c1.type = file
myagent.channels.c1.checkpointDir = /var/flume/checkpoint
myagent.channels.c1.dataDirs = /var/flume/data

# Bind the source and sink to the channel
myagent.sources.r1.channels = c1
myagent.sinks.k1.channel = c1
```

To test the agent, run the Flume agent and then open a new terminal and run the following command:

```
■ nc localhost 6666
```

Type some text. The same text will be sent to sink file.

```
■ sudo flume-ng agent -c /etc/flume/conf -f /etc/flume/conf/flume.conf -n
myagent
```

Box 5.11 shows an example of setting up a Flume agent with Twitter Source & HDFS Sink.

**■ Box 5.11: Flume agent with Twitter Source & HDFS Sink**

```
myagent.sources = r1
myagent.channels = c1
myagent.sinks = k1

# Define source
myagent.sources.r1.type = org.apache.flume.source.twitter.TwitterSource
myagent.sources.r1.consumerKey = <enter key here>
myagent.sources.r1.consumerSecret = <enter secret here>
myagent.sources.r1.accessToken = <enter token here>
myagent.sources.r1.accessTokenSecret = <enter token secret here>
myagent.sources.r1.maxBatchSize = 10
myagent.sources.r1.maxBatchDurationMillis = 200

#Define sink
myagent.sinks.k1.type = hdfs
myagent.sinks.k1.hdfs.fileType = DataStream
myagent.sinks.k1.hdfs.path = /flume/events
myagent.sinks.k1.hdfs.filePrefix = eventlog
myagent.sinks.k1.hdfs.fileSuffix = .log
myagent.sinks.k1.hdfs.batchSize = 1000

#Define Channel
myagent.channels.c1.type = file
myagent.channels.c1.checkpointDir = /var/flume/checkpoint
myagent.channels.c1.dataDirs = /var/flume/data

# Bind the source and sink to the channel
myagent.sources.r1.channels = c1
myagent.sinks.k1.channel = c1
```

Box 5.12 shows an example of setting up a Flume agent with HTTP Source & File Roll Sink.

**■ Box 5.12: Flume agent with HTTP Source & File Roll Sink**

```
myagent.sources = r1
myagent.channels = c1
myagent.sinks = k1

# Define source
myagent.sources.r1.type = http
myagent.sources.r1.bind = 0.0.0.0
myagent.sources.r1.port = 8000
myagent.sources.r1.handler = org.apache.flume.source.http.JSONHandler
myagent.sources.r1.handler.nickname = randomprops

#Define sink
myagent.sinks.k1.type = file_roll
myagent.sinks.k1.sink.directory = /var/log/flume

#Define Channel
myagent.channels.c1.type = file
myagent.channels.c1.checkpointDir = /var/flume/checkpoint
myagent.channels.c1.dataDirs = /var/flume/data

# Bind the source and sink to the channel
myagent.sources.r1.channels = c1
myagent.sinks.k1.channel = c1
```

### 5.3.2 Apache Sqoop

Apache Sqoop is a tool that allows importing data from relational database management systems (RDBMS) into the Hadoop Distributed File System (HDFS), Hive or HBase tables. Sqoop also allows exporting data from HDFS to RDBMS. Table 5.1 lists the various Sqoop commands.

| Tool | Function |
|---|---|
| import | Import a table from a database to HDFS |
| import-all-tables | Import tables from a database to HDFS |
| export | Export an HDFS directory to a database table |
| codegen | Generate code to interact with database records |
| create-hive-table | Import a table definition into Hive |
| eval | Evaluate a SQL statement and display the results |
| list-databases | List available databases on a server |
| list-tables | List available tables in a database |

Table 5.1: Sqoop Tools

### 5.3.3 Importing Data with Sqoop

Figure 5.8 shows the process of importing data from RDBMS using Sqoop. The import process begins with the user submitting a Sqoop import command. The format of an import command is shown below:

```
■ sqoop import --connect jdbc:mysql://<IP Address>/<Database Name>
--username <Username> --password <Password> --table <Table Name>
```

The import command includes a connection string which specifies the database type, database server hostname (or IP address) and database name. Sqoop can connect to any JDBC compliant database.

An example of an import command for importing data from a table named *Courses* from MySQL database named *Department* is shown below:

```
■ sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Courses
```

Sqoop import command launches multiple Map tasks (default is four tasks) which connect to the database and import the rows in the table in parallel to HDFS as delimited text files, binary Avro files or Hadoop SequenceFiles. The number of Map tasks used for importing data (and hence the parallelism) can be controlled using the —*m* option as shown in example below:

```
■ #Use 8 map tasks to import sqoop import --connect
 jdbc:mysql://localhost/Department
--username admin --password admin123 --table Courses --m 8
```

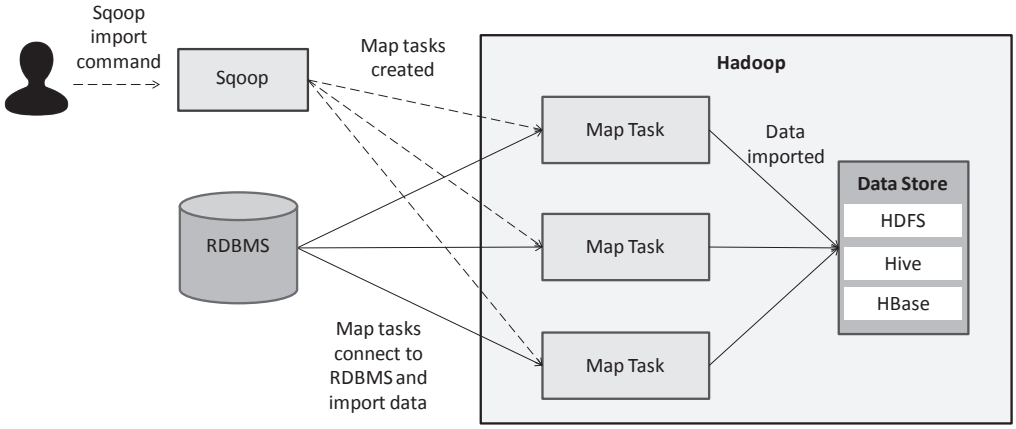


Figure 5.8: Importing data using Apache Sqoop

### 5.3.4 Selecting Data to Import

While in the previous example, we imported all the data from a table, Sqoop also allows importing selected data. With Sqoop import, it is possible to select a subset of columns (using the *columns* option) to import from a table as shown in the example below:

```
■ sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Courses
--columns "name,semester,year"
```

You can also use an SQL query with Sqoop import to select the data to import as shown in the example below:

```
■ sqoop import --connect jdbc:mysql://localhost/myDB --username admin
--password admin123 --query 'SELECT a.*, b.* FROM a JOIN b on
(a.id == b.id) WHERE $CONDITIONS'
--split-by a.id --target-dir /user/admin/joinresults
```

In the above example, Sqoop will import the results of the query in parallel. Since each Map task will execute the same query, certain conditions are required to split the data that each Map task imports. The $CONDITIONS token is replaced with the conditions by the Sqoop import tool at the run time. The *split-by* option specifies, on which column the data split should be performed to import data in parallel. When using an SQL query to specify what data to import, the *target-dir* option is required to provide the target location for the data to be imported.

### 5.3.5 Custom Connectors

While Sqoop ships with a generic JDBC connector, it may be preferable to use a vendor-specific JDBC connector as they can provide higher performance. Moreover, some databases provide data movement tools which can move data with higher performance. For example, MySQL provides *mysqldump* tool which can be used to export data from MySQL databases. Sqoop allows such database-specific tools to be used with the Sqoop import command using the *direct* option. The box below shows an example of importing data from MySQL with Sqoop using the *mysqldump* tool:

```
■ sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Courses --direct

#Passing additional arguments to database-specific tool
sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123
--table Courses --direct -- --default-character-set=latin1
```

### 5.3.6 Importing Data to Hive

Sqoop allows importing data into Hive using the *hive-import* option as shown in the example below. When this option is set, Sqoop will automatically create a Hive table and import data into the table.

```
■ sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Courses --hive-import
```

### 5.3.7 Importing Data to HBase

Sqoop allows importing data into HBase using the *hbase-table* option along with a target HBase table name, as shown in the example below. Sqoop also supports bulk loading of data into HBase using the *hbase-bulkload* option.

```
■ sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Courses
--hbase-table Courses
```

### 5.3.8 Incremental Imports

Incremental imports are useful when you have previously imported some rows from a table, and you want to import the newer rows. Sqoop provides an *incremental* option for incremental imports. When this option is used, a mode is also required, which can either be *append* or *lastmodified*. The *append* mode is used when a table is updated with new rows with increasing row ID values. The column to check for the row IDs is specified using the *check-column* option.

The *lastmodified* mode is used when the rows of a table are updated and the timestamp when a row was last modified is set in a last-modified column. The column to check for the last modified timestamp is specified using the *check-column* option. The *last-value* option is used in the *lastmodified* mode to specify the timestamp. When Sqoop import process completes it prints the *last-value*. In the next import, this *last-value* is specified, so that Sqoop can import only rows which have a last-modified timestamp greater than the *last-value*.

The box below shows examples of incremental imports:

```
■ sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Students
--check-column id --incremental append

#Import last modified rows
sqoop import --connect jdbc:mysql://localhost/Department
--username admin --password admin123 --table Students
--check-column last-modified --incremental
lastmodified --last-value "2015-04-03 15:08:45.66"
```

### 5.3.9 Importing All Tables

The Sqoop *import-all-tables* command can be used to import all tables from a database to HDFS, as shown the following example:

```
■ sqoop import-all-tables --connect jdbc:mysql://localhost/Department
```

### 5.3.10 Exporting Data with Sqoop

The Sqoop *export* command can be used to export files from HDFS to RDBMS, as shown in the following example:

```
■ sqoop export --connect jdbc:mysql://localhost/Department -table Courses
-export-dir /user/admin/courses
```

The target table must exist in the database. Sqoop translates the export command into a set of INSERT statements to append new rows to the table. The data from the input files is parsed and inserted into the target table. Instead of the default "insert" mode, you can also specify an "update" mode, in which Sqoop will use UPDATE statements to replace existing records in the target table. The *update-mode* option can be used to specify the "update" mode. With *update-mode* option, a mode needs to be specified which can either be *updateonly* or *allowinsert*. When *updateonly* mode is specified, the rows in the table are updated in the export process only if they exist. With *allowinsert* mode, the rows are updated if they exist in the table already or inserted if they do not exist.

## 5.4 Messaging Queues

Messaging queues are useful for push-pull messaging where the producers push data to the queues, and the consumers pull the data from the queues. The producers and consumers do not need to be aware of each other. Messaging queues allow decoupling of producers of data from the consumers. In this section, we will describe some message queuing systems based on protocols such as Advanced Message Queuing Protocol (AMQP) and ZeroMQ Message Transfer Protocol (ZMTP).

### 5.4.1 RabbitMQ

RabbitMQ implements the Advanced Message Queuing Protocol (AMQP), which is an open standard that defines the protocol for exchanges of messages between systems. AMQP clients can either be producers or consumers. The clients conforming with the standard can communicate with each other through brokers. Broker is a middleware application that receives messages from producers and routes them to consumers. The producers publish messages to the exchanges, which then distribute the messages to queues based on the defined routing rules (or bindings). AMQP brokers provide four types of exchanges: direct exchange (for point-to-point messaging), fanout exchange (for multicast messaging ), topic exchange (for publish-subscribe messaging) and header exchange (that uses header attributes for making routing decisions). Exchanges use bindings which are the rules to route messages to the queues. The consumers consume the messages from the queues. AMQP is an application level protocol that uses TCP for reliable delivery. A logical connection between a producer or consumer and a broker is called a Channel. For applications which need to establish multiple connections with a broker, it is undesirable to have multiple TCP connections. For such applications, multiple Channels can be setup over a single connection.

RabbitMQ is an AMQP Broker implemented in Erlang and is designed to be highly scalable and reliable. The commands for setting up RabbitMQ are given in Box 5.13.

**■ Box 5.13: Setting up RabbitMQ**

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' |
sudo tee /etc/apt/sources.list
```