#### 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).

##### 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.

**Sinks**

File Roll Sink

Thrift Sink

Avro Sink

HDFS Sink

Logger Sink

|  |
| --- |
| Avro Source |
| Thrift Source |
| Exec Source |
| JMS Source |

**Flume**

Sink

Channel

Source

|  |
| --- |
| IRC Sink |
| HBase Sink |
| ElasticSearch Sink |
| Custom Sink |

Figure 5.6: Apache Flume architecture

**Sources**

Custom Source

HTTP Source

Syslog Source

Sequence Generator Source

NetCat Source

Twitter Source

Spooling Directory Source

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

**Flume Agent**

External Source

Data

Store

Sink

Channel

Source

**(a)**

**Flume Agent**

External Source

Data

Store

Sink

Channel

Sink

Channel

Source

Sink

Channel

**(b)**

External Source

**Flume Agent**

Source

**(c)**

Data

Store

External Source

**Flume Agent**

Source

**Flume Agent**

Source

Data

Store

External Source

**(d)**

Sink

Channel

Source

**Flume Agent**

Sink

Channel

Source

**Flume Agent**

Sink

Channel

Sink

Channel

Sink

Channel

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

##### 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 codegen  create-hive-table | Export an HDFS directory to a database table  Generate code to interact with database records 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

##### 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

Sqoop import command

Map tasks

created

**Hadoop**

Data imported

RDBMS

Map tasks

connect to RDBMS and import data

Sqoop

Map Task

Map Task

Map Task



|  |
| --- |
| **Data Store** |
| HDFS |
|  |
| Hive |
| HBase |
|  |

Figure 5.8: Importing data using Apache Sqoop

##### 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.

##### 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

##### 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

##### 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

##### 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”

##### 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

##### 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.

#### 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).

##### 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

wget [https://www.rabbitmq.com/rabbitmq-signing-key-public.asc](http://www.rabbitmq.com/rabbitmq-signing-key-public.asc) sudo apt-key add rabbitmq-signing-key-public.asc

sudo apt-get install rabbitmq-server sudo pip install pika

Box 5.14 shows an example of a producer that sends data to a RabbitMQ queue. This example uses the *pika* library which is a pure-Python implementation of AMQP. The producer sends synthetic data along with the timestamp to a RabbitMQ queue.

□ **Box 5.14: Example of a Producer that sends data to RabbitMQ**

import pika

from time import time import json

import pickle, re, os, urllib, urllib2 from datetime import datetime

from random import randrange import time

import datetime

connection = pika.BlockingConnection(pika.ConnectionParameters( host=‘localhost’))

channel = connection.channel()

channel.queue\_declare(queue=‘test’) while True:

data = str(randrange(0,60)) + ‘,’ +

str(randrange(0,100)) + ‘,’ + str(randrange(5000,12000)) + ‘,’ + str(randrange(50,350))

ts=time.time() timestamp =

datetime.datetime.fromtimestamp(ts).strftime(‘%Y-%m-%d %H:%M:%S’)

msg=‘timestamp’: timestamp, ‘data’: data print msg

channel.basic\_publish(exchange=‘’, routing\_key=‘test’, body=json.dumps(msg))

print data time.sleep(1)

Box 5.15 shows an example of a consumer that consumes data from RabbitMQ queue.

□ **Box 5.15: Example of a Consumer that consumes data from RabbitMQ**

import pika

connection = pika.BlockingConnection(pika.ConnectionParameters( host=‘localhost’))

channel = connection.channel() channel.queue\_declare(queue=‘hello’)

def callback(ch, method, properties, body): print "Received %r" % (body,)

channel.basic\_consume(callback, queue=‘hello’,

no\_ack=True)

channel.start\_consuming()

##### ZeroMQ

ZeroMQ is a high-performance messaging library which provides tools to build a messaging system. Unlike other message queuing systems, ZeroMQ can work without a message broker. ZeroMQ provides various messaging patterns such as Request-Reply, Publish-Subscribe, Push-Pull and Exclusive Pair.

The commands for setting up ZeroMQ are given in Box 5.16.

□ **Box 5.16: Setting up ZeroMQ**

sudo apt-get install libtool

autoconf automake uuid-dev build-essential

wget <http://download.zeromq.org/zeromq-4.0.4.tar.gz> tar zxvf zeromq-4.0.4.tar.gz && cd zeromq-4.0.4

./configure make

sudo make install

sudo apt-get install python-zmq

Box 5.17 shows an example of a producer that sends data to a ZeroMQ queue.

□ **Box 5.17: Example of Producer that sends data to ZeroMQ**

import zmq

from time import time import json

from random import randrange import time

import datetime

context = zmq.Context()

socket = context.socket(zmq.PUSH) socket.bind(‘tcp://127.0.0.1:5555’)

while True:

#Generate some synthetic data

data = str(randrange(0,60)) + ‘,’ +

str(randrange(0,100)) + ‘,’ + str(randrange(5000,12000)) +

‘,’ + str(randrange(50,350))

ts=time.time() timestamp =

datetime.datetime.fromtimestamp(ts).strftime(‘%Y-%m-%d %H:%M:%S’)

msg=‘timestamp’: timestamp, ‘data’: data print msg

data = zmq.Message(json.dumps(msg)) socket.send(data)

print data time.sleep(1)

Box 5.18 shows an example of a consumer that consumes data from ZeroMQ queue.

□ **Box 5.18: Example of a Consumer that consumes data from ZeroMQ**

import zmq

context = zmq.Context()

socket = context.socket(zmq.PULL) socket.connect(‘tcp://127.0.0.1:5555’)

while True:

data = socket.recv() print data

##### RestMQ

RESTMQ is a message queue which is based on a simple JSON-based protocol and uses HTTP as transport. The queue is organized as REST resources. RESTMQ can be used by any client which can make HTTP calls. The commands for setting up RestMQ are given in Box 5.19.

□ **Box 5.19: Setting up RESTMQ**

#Install RESTMQ

sudo apt-get install build-essential curl python-pip redis-server libffi-dev python-dev -y libssl-dev python-setuptools

git clone https://github.com/gleicon/restmq.git cd restmq

sudo pip install -r requirements.txt sudo python setup.py install

# Start RESTMQ

cd restmq/start\_scripts touch acl.conf

bash restmq\_server -acl=acl.conf -listen=0.0.0.0 &

Box 5.20 shows an example of a producer that sends data to a RESTMQ queue.

□ **Box 5.20: Example of a Producer that sends data to RESTMQ**

import requests import json import urllib2

from random import randrange import time

import datetime

while True:

#Generate some synthetic data

data = str(randrange(0,60)) + ‘,’ +

str(randrange(0,100)) + ‘,’ + str(randrange(5000,12000)) + ‘,’ + str(randrange(50,350))

ts=time.time() timestamp =

datetime.datetime.fromtimestamp(ts).strftime(‘%Y-%m-%d %H:%M:%S’) msg=‘timestamp’: timestamp, ‘data’: data

data = urllib.urlencode(‘queue’:‘test’, ‘value’:json.dumps(msg)) r = urllib2.Request(‘http://localhost:8888/’, data)

f = urllib2.urlopen(r) data = f.read() f.close()

Box 5.21 shows an example of a consumer that consumes data from RESTMQ queue.

□ **Box 5.21: Example of a Consumer that consumes data from RESTMQ**

import json

from twisted.web import client from twisted.python import log

from twisted.internet import reactor

class CometClient(object): def write(self, content):

try:

data = json.loads(content) except Exception, e:

log.err("cannot decode json: %s" % str(e)) log.err("json is: %s" % content)

else:

log.msg("got data: %s" % repr(data))

def close(self): pass

if name == " main ": log.startLogging(sys.stdout)

client.downloadPage("http://localhost:8888/c/test", CometClient()) reactor.run()

□ #Post data to RESTMQ

curl -X POST -d "value=data" http://localhost:8888/q/test

#Get data from RESTMQ

curl http://localhost:8888/c/test

##### Amazon SQS

Amazon SQS offers a highly scalable and reliable hosted queue for storing messages as they travel between distinct components of applications. SQS only guarantees that the messages will arrive, not that they will arrive in the same order in which they were put in the queue. Though, at first look, Amazon SQS may seem to be similar to Amazon Kinesis, however, both are intended for very different types of applications. While Kinesis is meant for real-time applications that involve high data ingress and egress rates, SQS is simply a queue system that stores and releases messages in a scalable manner.

SQS can be used in distributed applications in which various application components need to exchange messages. Let us look at some examples of using SQS. Box 5.22 shows the Python code for creating an SQS queue. In this example, a connection to SQS service is first established by calling *boto.sqs.connect*\_*to*\_*region*. The AWS region, access key and secret key are passed to this function. After connecting to SQS service, *conn.create*\_*queue* is called to create a new queue with queue name as an input parameter. The function *conn.get*\_*all*\_*queues* is used to retrieve all SQS queues.

□ **Box 5.22: Python program for creating an SQS queue**

import boto.sqs

ACCESS\_KEY="<enter access key>" SECRET\_KEY="<enter secret key>" REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect\_to\_region( REGION,

aws\_access\_key\_id=ACCESS\_KEY, aws\_secret\_access\_key=SECRET\_KEY)

queue\_name = ‘mytestqueue’

print "Creating queue with name: " + queue\_name q = conn.create\_queue(queue\_name)

print "Created queue with name: " + queue\_name print " \n Getting all queues"

rs = conn.get\_all\_queues()

for item in rs: print item

Box 5.23 shows the Python code for writing to an SQS queue. After connecting to an SQS queue, the *queue.write* method is called with the message as an input parameter.

□ **Box 5.23: Python program for writing to an SQS queue**

import boto.sqs

from boto.sqs.message import Message import time

ACCESS\_KEY="<enter access key>" SECRET\_KEY="<enter secret key>"

REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect\_to\_region( REGION,

aws\_access\_key\_id=ACCESS\_KEY, aws\_secret\_access\_key=SECRET\_KEY)

queue\_name = ‘mytestqueue’

print "Connecting to queue: " + queue\_name q = conn.get\_all\_queues(prefix=queue\_name)

msg\_datetime = time.asctime(time.localtime(time.time()))

msg = "Test message generated on: " + msg\_datetime print "Writing to queue: " + msg

m = Message() m.set\_body(msg) status = q[0].write(m)

print "Message written to queue" count = q[0].count()

print "Total messages in queue: " + str(count)

Box 5.24 shows the Python code for reading from an SQS queue. After connecting to an SQS queue, the *queue.read* method is called to read a message from a queue.

□ **Box 5.24: Python program for reading from an SQS queue**

import boto.sqs

from boto.sqs.message import Message

ACCESS\_KEY="<enter access key>"

SECRET\_KEY="<enter secret key>" REGION="us-east-1"

print "Connecting to SQS"

conn = boto.sqs.connect\_to\_region( REGION,

aws\_access\_key\_id=ACCESS\_KEY, aws\_secret\_access\_key=SECRET\_KEY)

queue\_name = ‘mytestqueue’

print "Connecting to queue: " + queue\_name q = conn.get\_all\_queues(prefix=queue\_name)

count = q[0].count()

print "Total messages in queue: " + str(count) print "Reading message from queue"

for i in range(count): m = q[0].read()

print "Message %d: %s" % (i+1,str(m.get\_body())) q[0].delete\_message(m)

print "Read %d messages from queue" % (count)

#### Custom Connectors

Custom connectors and web services for acquiring data from data producers can be developed to meet the application requirements.

##### REST-based Connectors

Figure 5.9 shows the architecture of a REST-based custom connector. The connector exposes a REST web service. Data producers can publish data to the connector using HTTP POST requests which contain the data payload. The request data received by the connector is stored to the sink (such as local filesystem, distributed filesystem or cloud storage). The data sinks in the connector provide the functionality for processing the HTTP request and storing the data to the sink. The benefit of using a REST-based connector is that any client that can make HTTP requests can send data to the connector. Requests are stateless in nature, and each request carries all the information that is required to process the request. The HTTP headers add to the request overhead making this method unsuitable for high-throughput and real-time applications.

**Implementing a REST-based Custom Connector**

Let us look at an example of implementing a custom REST-based connector as shown in Figure 5.9. Box 5.25 shows the Python implementation of the REST-based connector. In this example, we use the Flask Python web framework to implement the web service. This connector publishes a single end point (such as [‘http://public-ip/api/data’](http://public-ip/api/data) ), to which the

POST

Sink

Custom Connector

Data Sinks

REST

WebService

Sink

Producer

Request (JSON data)

Store data

Producer

Figure 5.9: REST-based custom connector

client applications can send an HTTP POST request along with the data payload. Box 5.26 shows an example of a client which sends some synthetic sensor data to the web service. The web service receives the data from the POST request payload and then publishes the data to an Amazon SQS queue and also writes the data to an Amazon DynamoDB table.

The benefit of having such a custom connector is that the client and the server become independent of each other. The web service decouples the client from the server. The server can add or change the actions (such as publishing data to a queue or storing data in a database) without the client having to be aware of the changes. The client can use any tool or programming language from which it can make an HTTP POST request. (Note: Though we call this as a REST-connector, it is not fully REST compliant as we have only implemented the POST functionality. Other methods such as GET, PUT, DELETE may not be required if the connector only allows data to be ingested.)

□ **Box 5.25: Python implementation of a REST-based custom connector**

import boto.sqs

from boto.sqs.message import Message import boto.dynamodb2

from boto.dynamodb2.table import Table import cPickle as pickle

import time import datetime import json

from flask import Flask, jsonify, abort,

from flask import request, make\_response, url\_for app = Flask( name , static\_url\_path="")

ACCESS\_KEY = <Enter AWS Access Key> SECRET\_KEY = <Enter Secret Key> REGION="us-east-1"

queue\_name = ‘sensordata’ table\_name = ‘sensordata’

#Connect to AWS SQS

conn = boto.sqs.connect\_to\_region(REGION,aws\_access\_key\_id=ACCESS\_KEY, aws\_secret\_access\_key=SECRET\_KEY)

q = conn.get\_all\_queues(prefix=queue\_name) #Connect to AWS DynamoDB

conn\_dynamo = boto.dynamodb2.connect\_to\_region(REGION,

aws\_access\_key\_id=ACCESS\_KEY, aws\_secret\_access\_key=SECRET\_KEY)

table=Table(table\_name,connection=conn\_dynamo) #Publishes data to SQS

def publish\_to\_sqs(data):

m = Message() m.set\_body(data) status = q[0].write(m) return status

#Writes data to DynamoDB table def publish\_to\_dynamo(datadir):

item = table.put\_item(data=datadir)

@app.errorhandler(400) def bad\_request(error):

return make\_response(jsonify({‘error’: ‘Bad request’}), 400)

@app.errorhandler(404) def not\_found(error):

return make\_response(jsonify({‘error’: ‘Not found’}), 404)

@app.route(‘/api/data’, methods=[‘POST’]) def post\_data():

data = json.loads(request.data) publish\_to\_sqs(pickle.dumps(data)) publish\_to\_dynamo(data)

return jsonify({‘result’: ‘true’}), 201

if name == ‘ main ’: app.run(debug=True)

□ **Box 5.26: Python implementation of client program that publishes data to a custom connector**

from random import randint import time

import datetime import requests import json

def getData(): ts=time.time()

timestamp = datetime.datetime.fromtimestamp(ts).strftime(‘%Y-%m-%d %H:%M:%S’)

temp = str(randint(0,100)) humidity = str(randint(0,100)) co2 = str(randint(50,500)) light = str(randint(0,10000))

data = {"timestamp": timestamp, "temperature": temp, "humidity": humidity , "co2": co2, "light": light}

return data

def publish(datadir):

r = requests.post("http://localhost:5000/api/data", data = json.dumps(datadir),

headers={"Content-Type": "application/json"})

while True:

data = getData() print data publish(data) time.sleep(1)

##### WebSocket-based Connectors

Figure 5.10 shows the architecture of a WebSocket-based custom connector. The connector exposes a WebSocket web service. The Web Application Messaging Protocol (WAMP) which is a sub-protocol of WebSocket can be used for creating a WebSocket-based connector. WAMP provides publish-subscribe and remote procedure call (RPC) messaging patterns. Clients (or data producers) establish a TCP connection with the connector and send data frames. WebSocket connection is stateful in nature and allows full duplex communication over a single TCP connection. Data producers publish data to the WebSocket endpoints which are published by the connector. The subscribers subscribe to the WebSocket endpoints and receive data from the WebSocket web service.

Unlike request-response communication with REST, WebSockets allow full duplex communication and do not require a new connection to be setup for each message to be sent. WebSocket communication begins with a connection setup request sent by the client to the server. This request (called a WebSocket handshake) is sent over HTTP and the server interprets it as an upgrade request. If the server supports WebSocket protocol, the server responds to the WebSocket handshake response. After the connection is setup, the client and server can send data/messages to each other in full-duplex mode. There is no overhead for connection setup and termination requests for each message. WebSocket communication is

Subscriber

Data Frames

Producer

Custom Connector

WebSocket WebService

Producer

Subscriber

Store data

Figure 5.10: WebSocket-based custom connector

suitable for applications that have low latency or high throughput requirements.

##### MQTT-based Connectors

MQTT (MQ Telemetry Transport) is a lightweight publish-subscribe messaging protocol designed for constrained devices. MQTT is suitable for Internet of Things (IoT) applications that involve devices sending sensor data to a server or cloud-based analytics backends to be processed and analyzed. The entities involved in MQTT include:

* + - * Publisher: Publisher is the component which publishes data to the topics managed by the Broker.
      * Broker/Server: Broker manages the topics and forwards the data received on a topic to all the subscriber which are subscribed to the topic.
      * Subscriber: Subscriber is the component which subscribes to the topics and receives data published on the topics by the publishers.

**Implementing a MQTT-based Custom Connector**

Let us look at an example of implementing a custom MQTT-based connector. Boxes 5.27 and

5.28 show the Python implementation of the MQTT subscriber and publisher components. The subscriber component in this example runs on the server, which also has an MQTT Broker running. The publisher component runs on the devices which need to publish data to the server. The devices publish data to an MQTT topic (e.g. $iot/test). The subscriber which is subscribed to the topic receives the data and processes or forwards the data. The forwarding actions may include forwarding the data to a messaging queue, writing the data to a NoSQL database or storing the data to a distributed file system.

The benefit of using the MQTT-based custom connector is that it decouples the client and the server (in space, time and synchronization dimensions). By space decoupling, we mean that the client and server do not need to know about each other. Time decoupling means that the client and server do not need to be running simultaneously. Synchronization decoupling means that the communication between client and server can happen asynchronously. The client and server do not have to wait while the messages are being processed.

□ **Box 5.27: Python implementation of a MQTT-based custom connector (subscriber)**

import paho.mqtt.client as mqtt import json

import boto.sqs

from boto.sqs.message import Message

REGION="us-east-1" queue\_name = ‘sensordata’

ACCESS\_KEY = <Enter AWS Access Key> SECRET\_KEY = <Enter Secret Key>

conn = boto.sqs.connect\_to\_region(REGION,aws\_access\_key\_id=ACCESS\_KEY,

aws\_secret\_access\_key=SECRET\_KEY) q = conn.get\_all\_queues(prefix=queue\_name)

def publish\_to\_sqs(data): m = Message() m.set\_body(data) status = q[0].write(m) return status

def on\_connect(client, userdata, flags, rc): print("Connected with result code "+str(rc)) client.subscribe("$iot/test")

def on\_message(client, userdata, msg): data = json.loads(msg.payload) publish\_to\_sqs(data)

client = mqtt.Client() client.on\_connect = on\_connect client.on\_message = on\_message

client.connect("localhost", 1883, 60) client.loop\_forever()

□ **Box 5.28: Python implementation of a MQTT-based custom connector (publisher)**

import paho.mqtt.client as mqtt import paho.mqtt.publish as publish import time

from random import randint import datetime

import requests

import json

def getData(): ts=time.time()

timestamp = datetime.datetime.fromtimestamp(ts).strftime(‘%Y-%m-%d %H:%M:%S’)

temp = str(randint(0,100)) humidity = str(randint(0,100)) co2 = str(randint(50,500)) light = str(randint(0,10000))

data = {"timestamp": timestamp,

"temperature": temp, "humidity": humidity , "co2": co2, "light": light}

return data

def publish\_to\_topic(data):

publish.single("$iot/test", payload=json.dumps(data), hostname="localhost")

while True:

data = getData() print data

publish\_to\_topic(data) time.sleep(1)

##### Amazon IoT

Amazon IoT is a service for collecting data from Internet of Things (IoT) devices (such as sensors and smart appliances) into the AWS cloud. The data collected can be sent to various AWS services, e.g. stored in Amazon DynamoDB database, stored in a file on S3, sent to an Amazon Kinesis data stream, sent to Amazon SNS as a push notification and inserted into a code for executing it with Amazon Lambda service.

Figure 5.11 shows the various components of the AWS IoT service.

* + - * **Device Gateway**: Device Gateway enables devices to communicate with AWS IoT using MQTT or HTTP protocols. Devices can publish or subscribe to topics.
      * **Device Registry**: Device registry (also called things registry) maintains the resources associated with each device including attributes, certificates and meta-data.
      * **Device Shadow**: Device shadow maintains the state of a device as a JSON document. Applications can retrieve or update the device state using the AWS IoT REST APIs. Device shadow persists the state of the device even when the device is offline. When a device becomes online, the state is synchronized with the device shadow.
      * **Rules Engine**: Rules engine allows you to define rules for processing messages received from devices. Using an SQL-like language, you can define rules to select data, process data and send the data to other AWS services such as DynamoDB, S3, Kinesis, SNS and Lambda.
      * **Security and Identity Service**: This service allows devices to securely exchange data with the AWS IoT service. For devices communicating via MQTT, certificate-based authentication is used. Certificates have policies associated with them which authorize devices to access specific resources.

Figure 5.11: Amazon IoT components

**AWS IoT**

Messages

Messages

Messages

Security & Identity Service

Device Registry

Device Shadows

Device Gateway

Rules Engine

AWS Services

* S3
* DynamoDB
* Lambda
* Kinesis
* SNS

Applications

IoT Device

Let us look at some examples of using AWS IoT service. The first step is to create a thing from the AWS IoT dashboard as shown in Figure 5.12. Thing represents a device in the AWS IoT service. When a thing is created, an entry is created in the device registry for the device and a device shadow is also created. At this step, you can also add the optional attributes to describe the device capabilities.

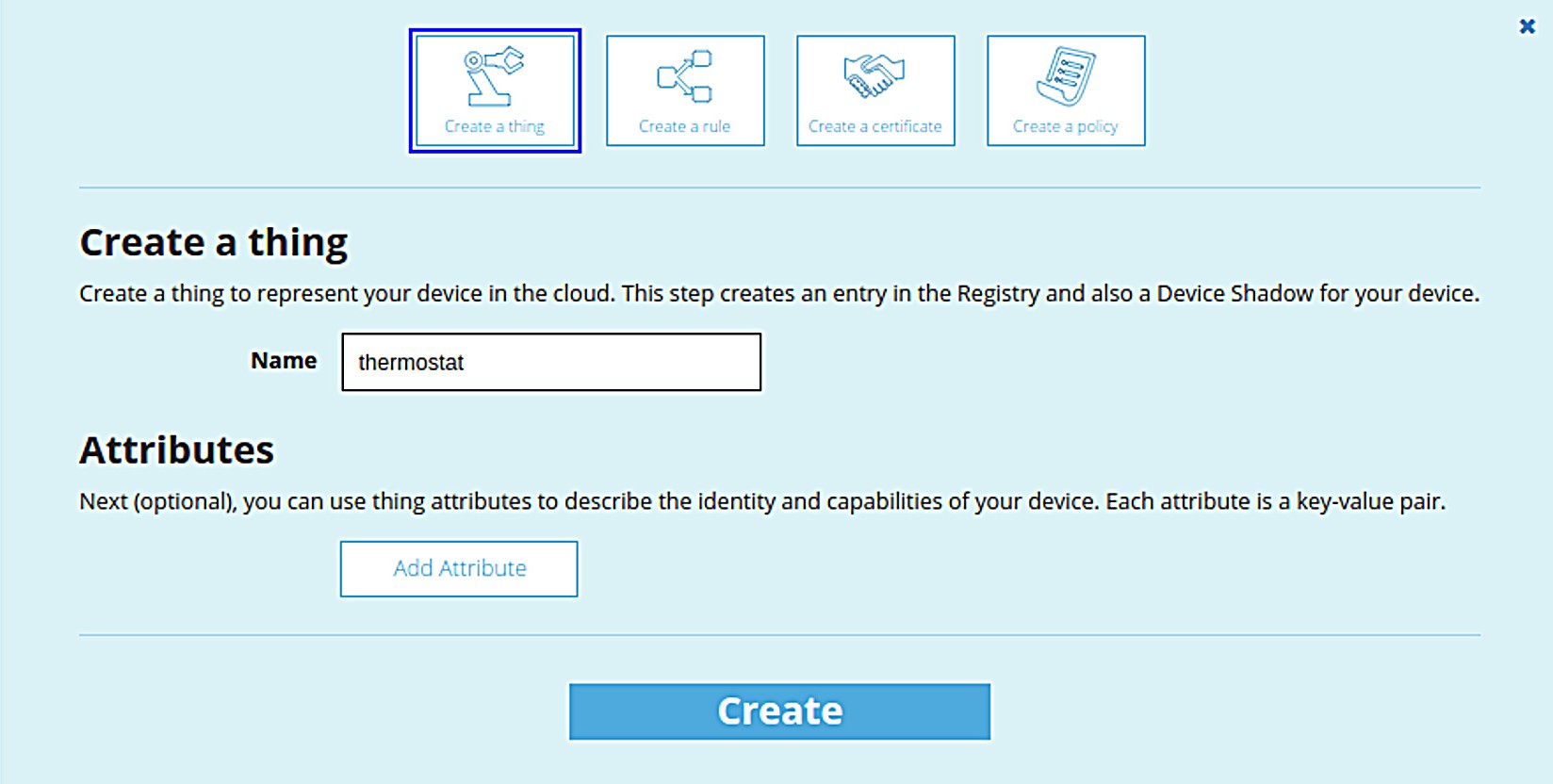


Figure 5.12: Creating a Thing from Amazon IoT dashboard

In the next step, we create a certificate which is used by the device for connecting to AWS IoT. The certificate is attached to a thing. Three files are created in this step - a certificate file, a public key file, and a private key file. Next, we create a policy and attach the policy to the certificate to assign permissions.

Box 5.29 shows a Python example for publishing messages to AWS IoT. This example uses the Paho Python MQTT client. For connecting to AWS IoT a Root Certificate Authority (CA) certificate, a client certificate and private key file is required. This example simulates a thermostat device sending the current state (temperature) to AWS IoT, which stores the state in the device shadow. To report the state over MQTT, a message is published on the topic

$aws/things/thingName/shadow/update.

□ **Box 5.29: Python code for publishing messages to AWS IoT**

import paho.mqtt.client as mqtt import ssl

import paho.mqtt.publish as publish

connection={

"host": "A26VGTA50P1HNL.iot.us-east-1.amazonaws.com", "port": 8883,

"clientId": "thermostat", "thingName": "thermostat", "caCert": "root-CA.crt",

"clientCert": "9795072c41-certificate.pem.crt", "privateKey": "9795072c41-private.pem.key"

}

tlsdict= {‘ca\_certs’:connection[‘caCert’], ‘certfile’:connection[‘clientCert’], ‘keyfile’:connection[‘privateKey’], ‘tls\_version’:ssl.PROTOCOL\_SSLv23, ‘ciphers’:None}

state="{ *\*"state*\*": {*\*"reported*\*": { *\*"temperature*\*": *\*"70*\*" } } }" publish.single("$aws/things/thermostat/shadow/update", payload=str(state),

qos=1, retain=False, hostname=connection[‘host’],

port=8883, client\_id=connection[‘clientId’], keepalive=60, will=None, auth=None, tls=tlsdict, protocol=mqtt.MQTTv311)

The current state for a device can be seen from the AWS IoT dashboard as shown in Figure 5.13. Box 5.30 shows a Python example for subscribing to the state updates for a device. To receive updates from the device shadow over MQTT, the device/application can subscribe to topic the $aws/things/thingName/shadow/update/accepted.

Applications can also use the AWS IoT REST API to query for the last reported state for a device or update the device state. For example, a mobile application that controls the temperature setting for a smart thermostat can be built. The thermostat reports its current state (temperature) to AWS IoT, and the state is stored in the device shadow. The mobile application can update the desired state in the device shadow instead of directly communicating with the thermostat. The desired state is synchronized with the device, the next time it is connected to the AWS IoT service. The device state can also be updated from the AWS IoT dashboard as shown in Figure 5.14.

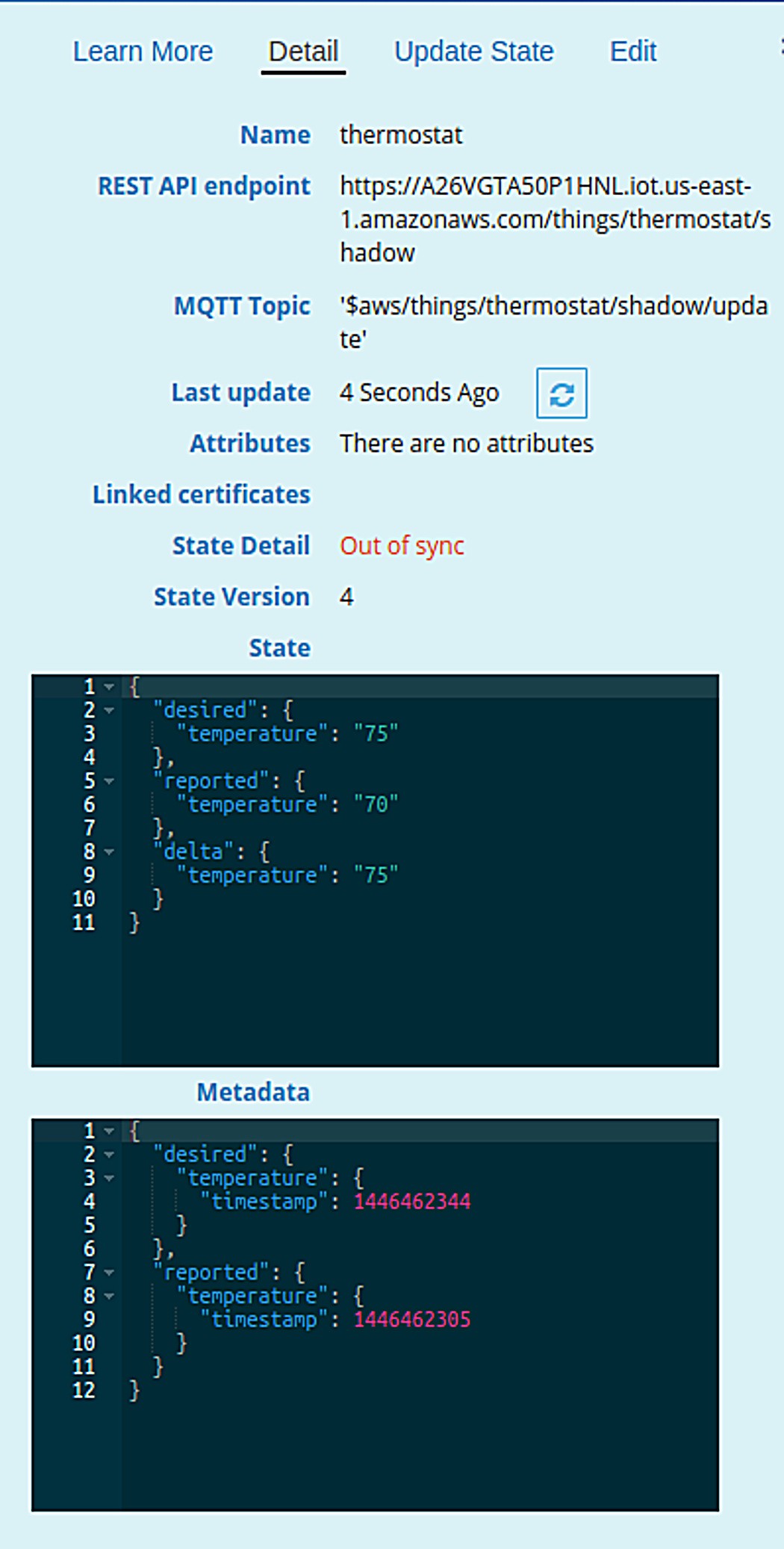


Figure 5.13: Viewing the state of a thing from Amazon IoT dashboard

□ **Box 5.30: Python code for subscribing to a topic in AWS IoT**

import paho.mqtt.client as mqtt import ssl

connection={

"host": "A26VGTA50P1HNL.iot.us-east-1.amazonaws.com", "port": 8883,

"clientId": "thermostat", "thingName": "thermostat", "caCert": "root-CA.crt",

"clientCert": "9795072c41-certificate.pem.crt", "privateKey": "9795072c41-private.pem.key"

}

def on\_connect(client, userdata, flags, rc): print("Connected with result code "+str(rc)) client.subscribe("$aws/things/rpi/shadow/update/accepted")

def on\_message(client, userdata, msg): print(msg.topic+" "+str(msg.payload))

client = mqtt.Client() client.on\_connect = on\_connect client.on\_message = on\_message

client.tls\_set(ca\_certs=connection[‘caCert’], certfile=connection[‘clientCert’], keyfile=connection[‘privateKey’], cert\_reqs=ssl.CERT\_REQUIRED, tls\_version=ssl.PROTOCOL\_SSLv23, ciphers=None)

client.connect(connection[‘host’], connection[‘port’]) client.loop\_forever()

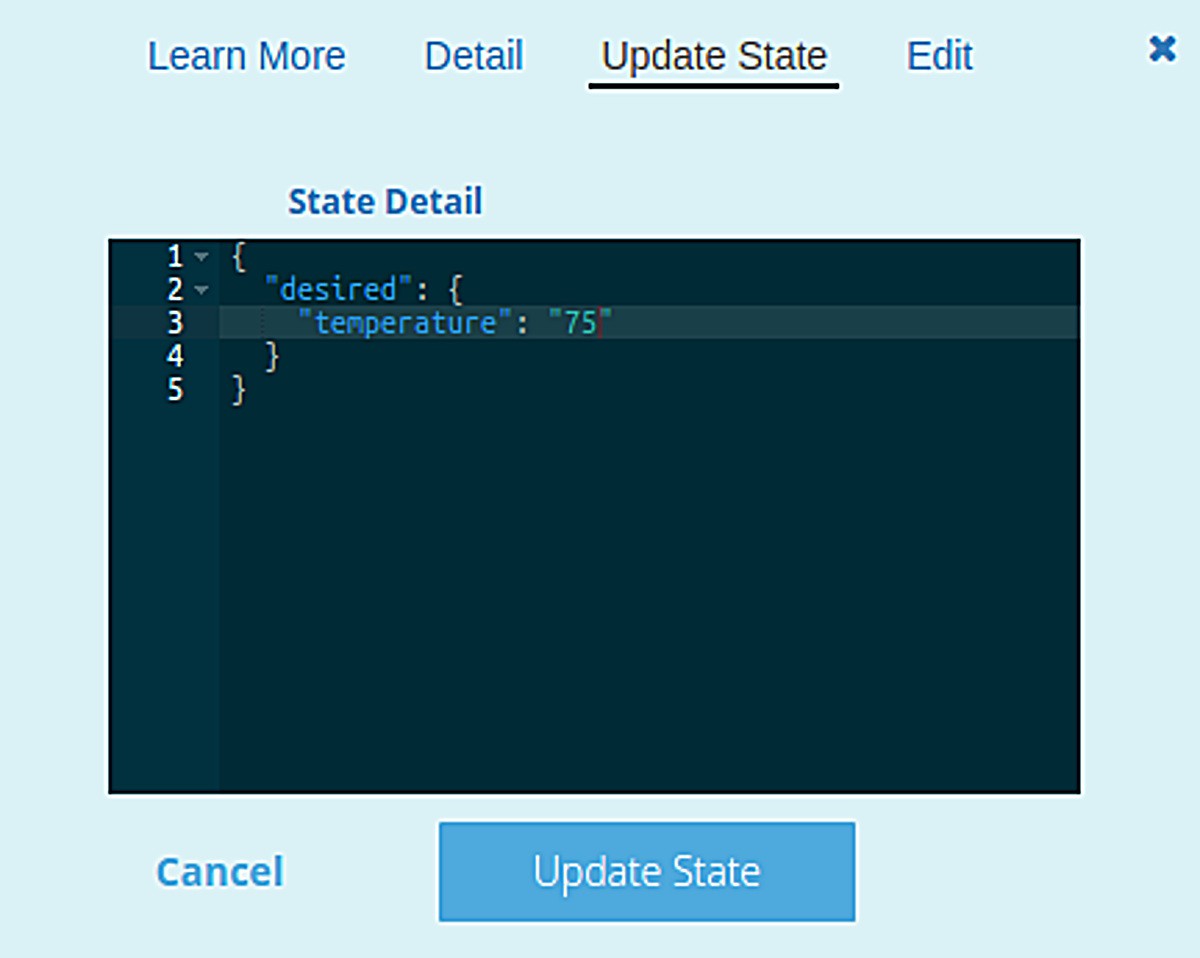


Figure 5.14: Updating the state of a thing from Amazon IoT dashboard

Let us now look at a more advanced example where we use the rule engine to send data collected from a device to different AWS services. For this example, we will create a new thing called ‘forest’ which represents a device deployed in a forest for reporting data

collected from various sensors (temperature, humidity, light, *CO*2). Data collected from multiple such devices deployed in a forest can be analyzed to detect forest fires. While you can use the AWS IoT Starter Kits to build physical devices with real sensors connected to them, for simplicity we will use a program which generates and sends synthetic data to AWS IoT. Box 5.31 shows a Python program for sending synthetic sensor data to AWS IoT.

□ **Box 5.31: Python program for sending synthetic sensor data to AWS IoT**

from random import randrange import time

import datetime

import paho.mqtt.client as mqtt import ssl

import paho.mqtt.publish as publish

connection={

"host": "A26VGTA50P1HNL.iot.us-east-1.amazonaws.com", "port": 8883,

"clientId": "forest", "thingName": "forest", "caCert": "root-CA.crt",

"clientCert": "9795072c41-certificate.pem.crt", "privateKey": "9795072c41-private.pem.key"

}

tlsdict= {‘ca\_certs’:connection[‘caCert’], ‘certfile’:connection[‘clientCert’], ‘keyfile’:connection[‘privateKey’], ‘tls\_version’:ssl.PROTOCOL\_SSLv23, ‘ciphers’:None}

#Send some synthetic data to AWS IoT while True:

ts=time.time()

data = "{ *\*"state*\*": { *\*"location*\*": *\*"123*\*",

*\*"timestamp*\*": *\*""+str(ts)+"*\*",

*\*"temperature*\*": "+str(randrange(0,60))+",

*\*"humidity*\*": "+str(randrange(0,60))+",

*\*"light*\*": "+str(randrange(0,60))+", *\*"co2*\*": "+str(randrange(0,60))+"

}}"

print data

publish.single("$aws/things/forest/test", payload=str(data), qos=1, retain=False, hostname=connection[‘host’],

port=8883, client\_id=connection[‘clientId’], keepalive=60, will=None, auth=None, tls=tlsdict, protocol=mqtt.MQTTv311)

time.sleep(1)

Next, we create two different rules in AWS IoT for analyzing this data further. The first rule as shown in Figure 5.15 sends data to an Amazon Kinesis data stream. The second rule

as shown in Figure 5.16 writes the data to an Amazon DynamoDB table.

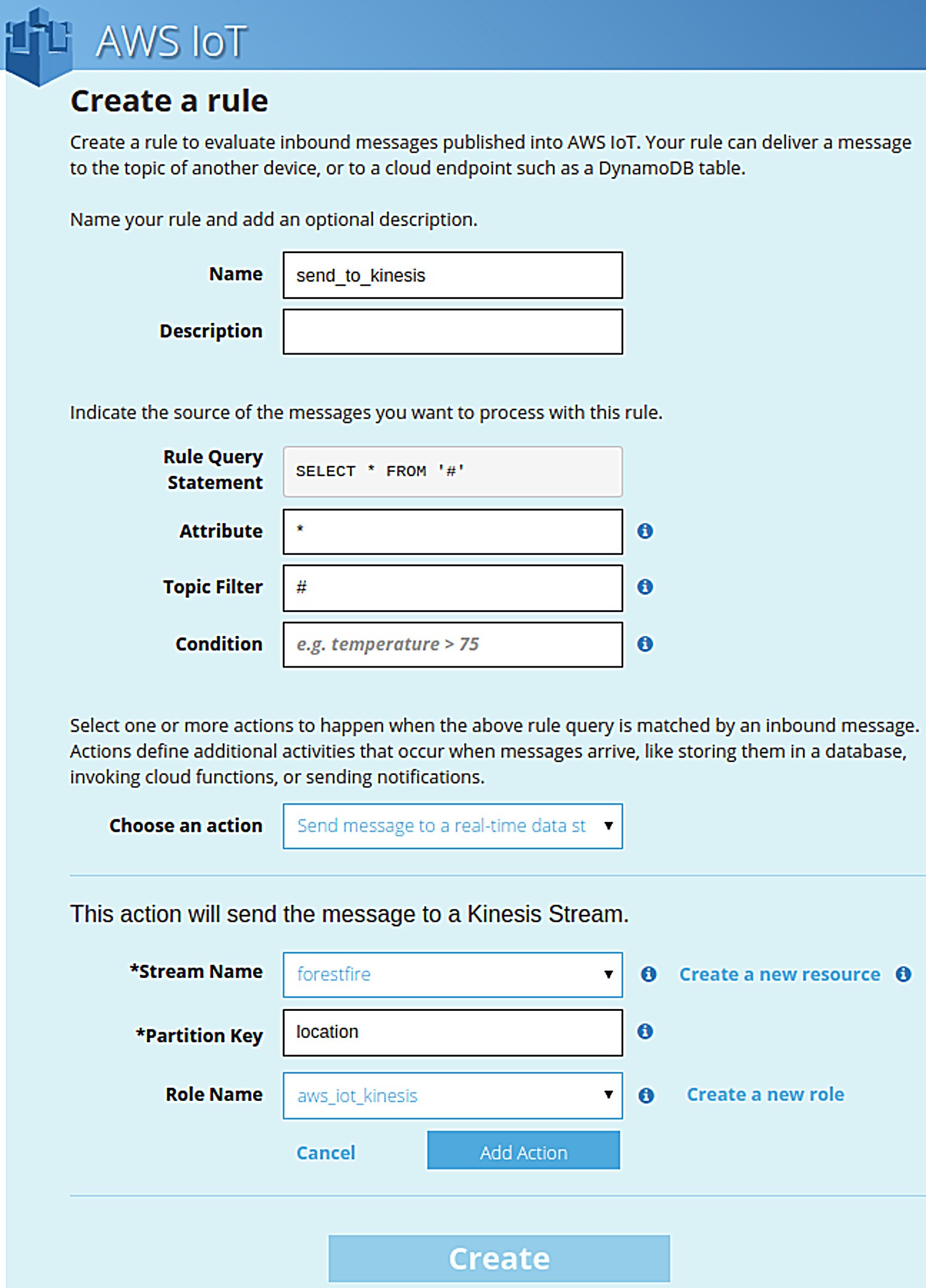


Figure 5.15: Creating a rule from Amazon IoT dashboard

With the rules defined, run the Python program in Box 5.31. The synthetic data generated by this program will be published to the topic $aws/things/forest/test in AWS IoT. The rules will send the data to Amazon Kinesis and Amazon DynamoDB. Figure 5.17 shows a screenshot of the Amazon DynamoDB table with the data published by the device. To read

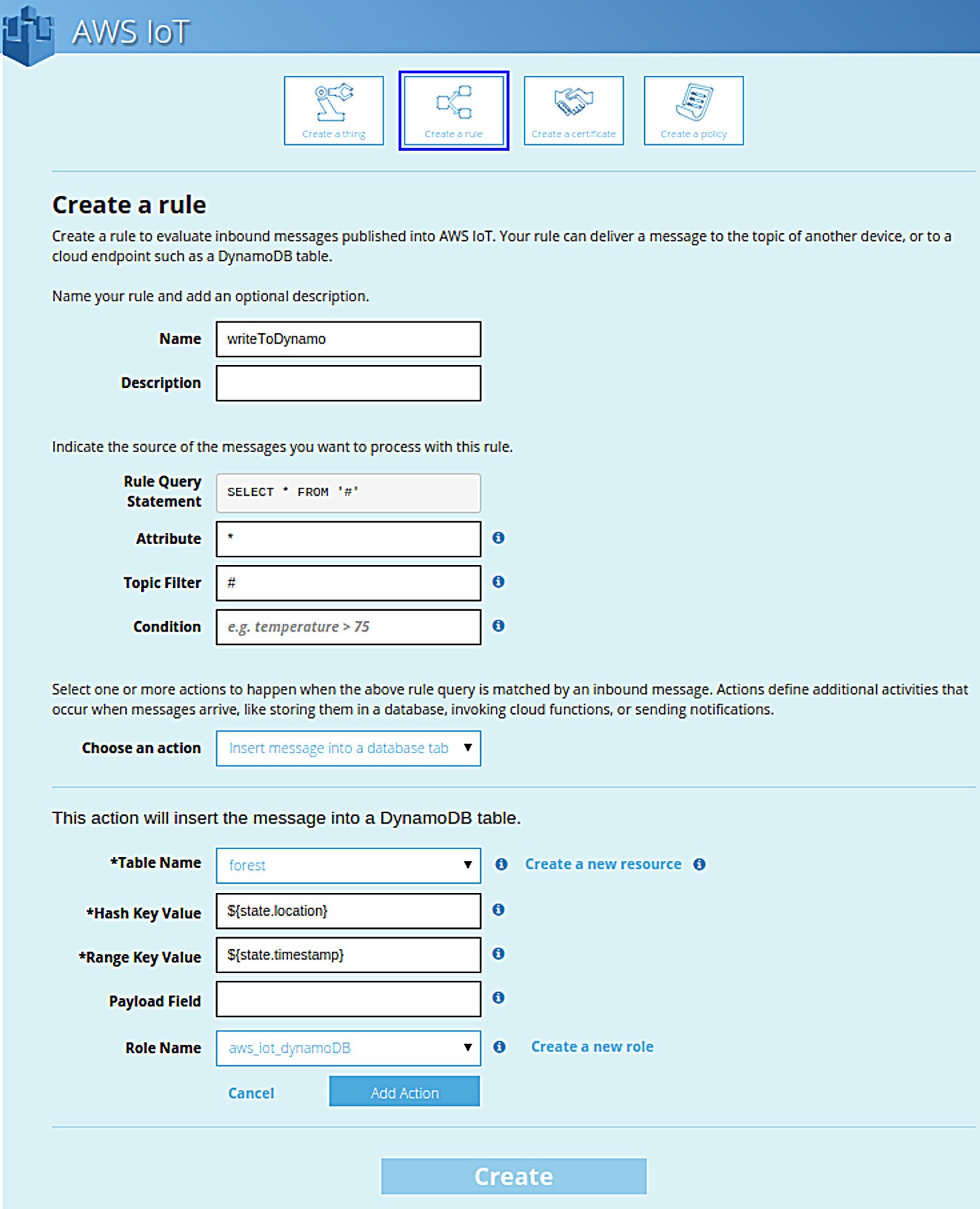


Figure 5.16: Creating a rule from Amazon IoT dashboard data from the Kinesis data stream, you can use the program shown in Box 5.8.

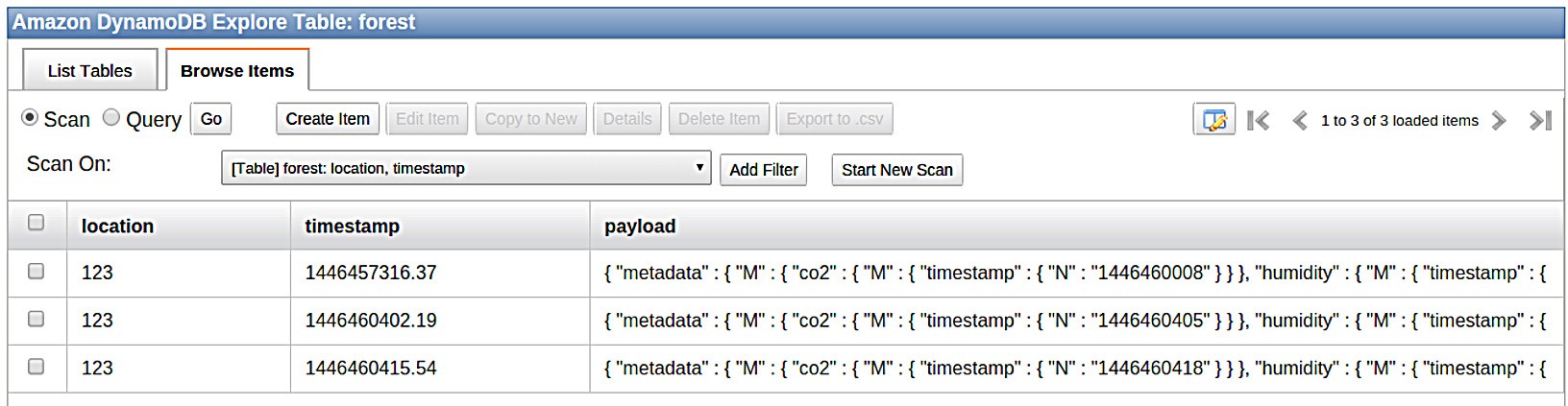


Figure 5.17: Viewing the data stored by an AWS IoT rule into a DynamoDB table

##### Azure IoT Hub

Azure IoT Hub is a fully managed service for bi-directional communication between IoT devices and the Azure cloud. Azure IoT Hub receives messages from IoT devices and sends them to various Azure services (such as Azure Stream Analytics) for further processing of messages. IoT Hub can store up to 7 days of data. Applications can use IoT Hub to send messages to the devices. Azure provides device libraries for connecting various devices to the IoT Hub. Supported protocols include HTTP 1.1 and AMQP 1.0. Support for MQTT can be added by running Azure IoT Protocol Gateway, an open source component, which can be run either locally or in the cloud. IoT Hub includes a device identity registry which is used to provision devices with their own security keys for securely connecting to the IoT Hub. Figure 5.18 shows the various components of Azure IoT Hub.

IoT Device

Figure 5.18: Azure IoT components

Messages

(HTTP or AMQP)

Azure IoT

Messages

Messages

(MQTT)

Messages

(AMQP)

Azure IoT Protocol Gateway

Azure Services

* Stream Analytics
* Event Hub

Authentication & Authorization

Device Identity Registry

IoT Hub

IoT Device

In the previous section, we described the example of IoT devices deployed in a forest for reporting data collected from various sensors (temperature, humidity, light, *CO*2) for detecting forest fires. Let us repeat the same example using Azure IoT Hub. The first step is to create an IoT Hub that will receive data from devices. Log into the Azure Preview Portal and create a new IoT Hub as shown in Figure 5.19. Once the IoT Hub has been created, open the IoT hub tile in the Preview Portal, and note down the IoT Hub Hostname. Next, select the Key icon in the IoT Hub and click on the *iothubowner* shared access policy as shown in Figure 5.20. Note down the connection string and primary key.

Now that the IoT Hub is operational, let us register a device with the Hub. To create a

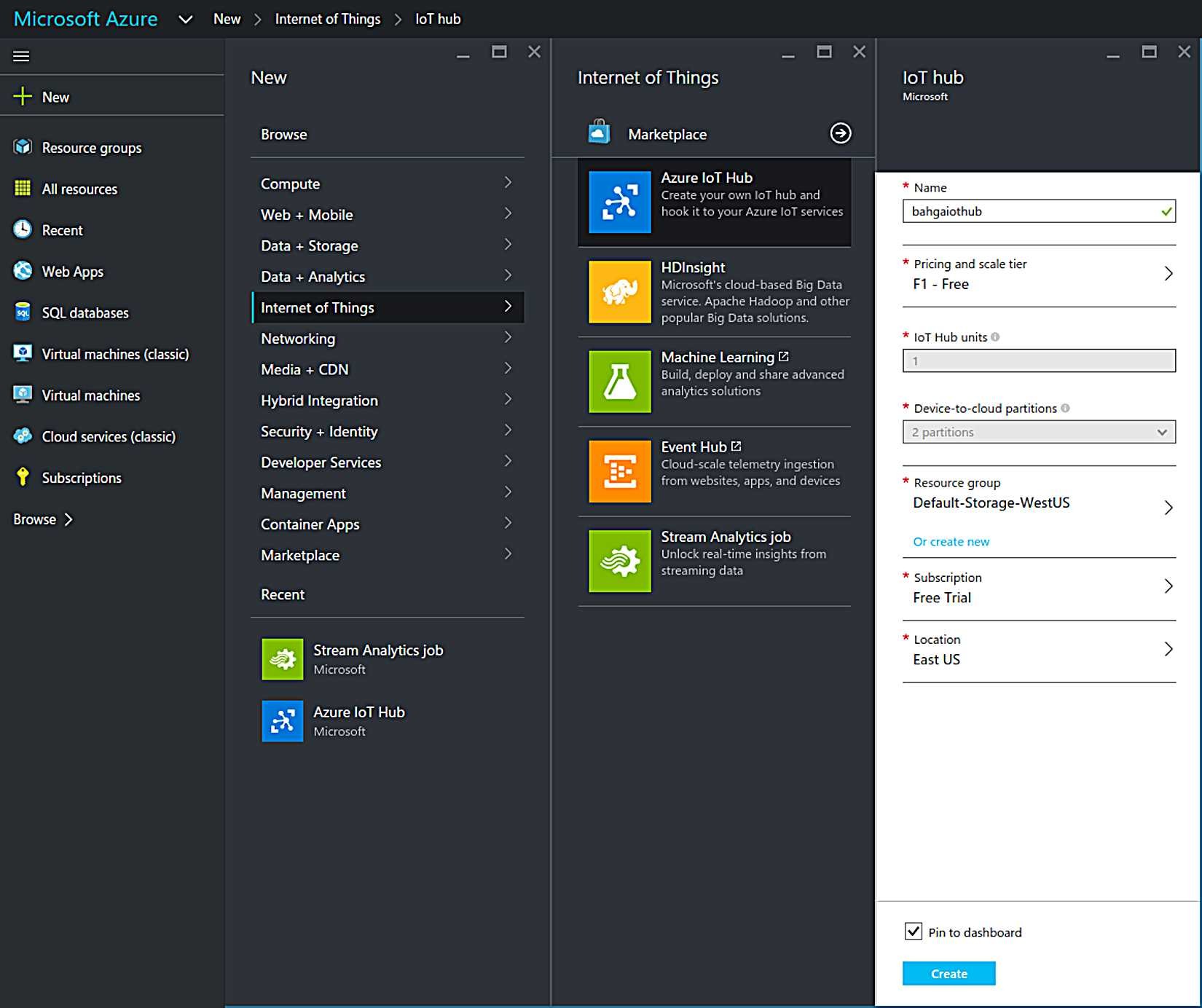


Figure 5.19: Creating an Azure IoT hub

new device identity, you can either use a standalone tool called Device Explorer (which runs on Windows) or a NodeJS tool called *iothub-explorer*. NodeJS can be installed as follows:

□ #Installing NodeJS

curl -sL https://deb.nodesource.com/setup\_4.x | sudo -E bash - sudo apt-get install -y nodejs

Next, install the *iothub-explorer* tool and then generate a unique identity and connection string as follows:

□ #Create a new device identity in the IoT Hub npm install -g iothub-explorer

node iothub-explorer "<enter iothubowner connection string>" create mydevice -connection-string

Note down the device connection string generated by *iothub-explorer*. As of writing this book, Python support libraries for IoT Hub have not been released. Therefore, we will provide an example using NodeJS. Box 5.32 shows a simple example of sending data to IoT Hub using NodeJS. In this example, use the device connection string generated by the

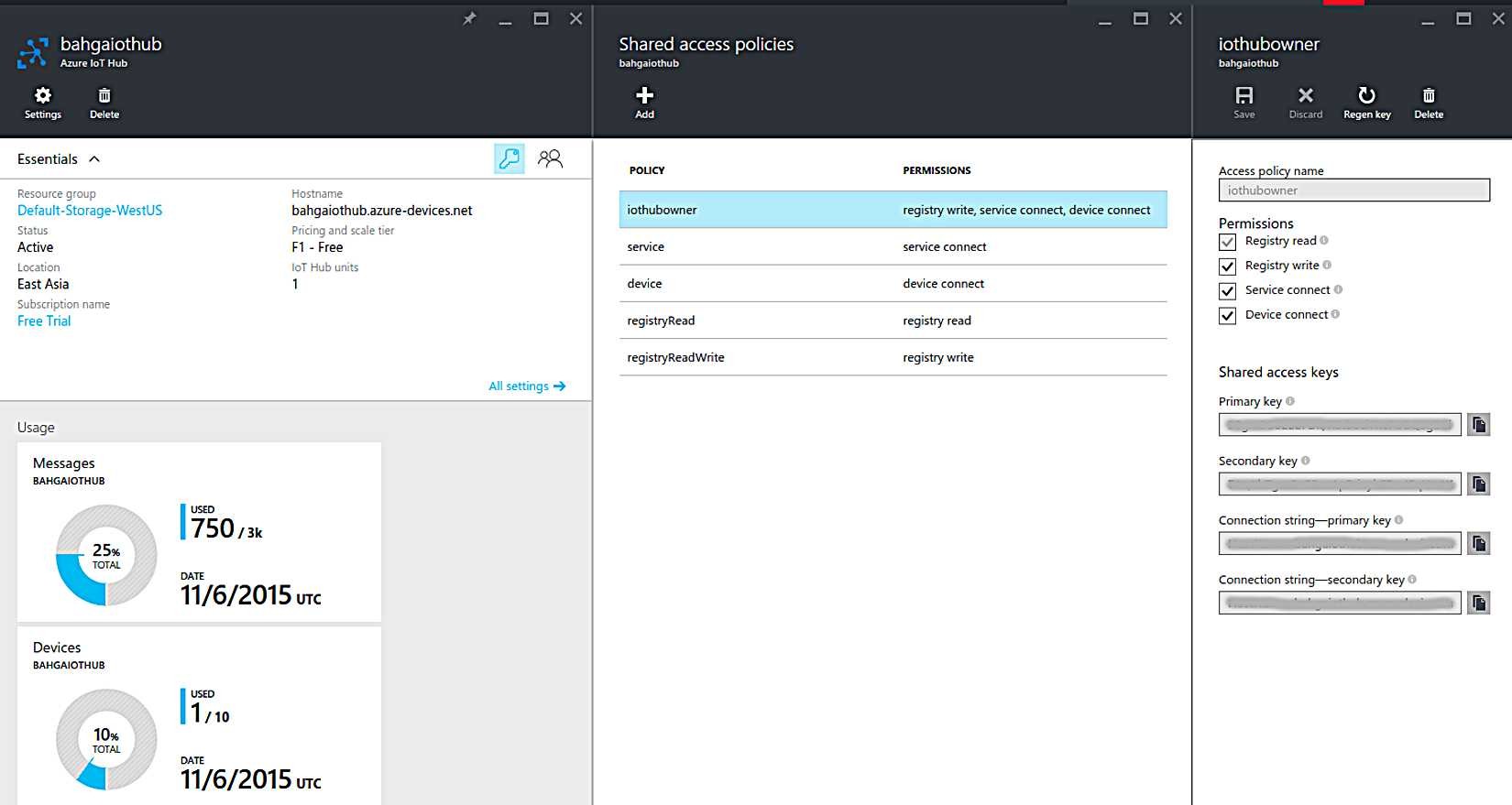


Figure 5.20: Viewing details of an Azure IoT hub

*iothub-explorer* tool. This program generates some random synthetic data and sends it to the IoT Hub every second.

□ **Box 5.32: NodeJS code for sending data to Azure IoT Hub**

var device = require(‘azure-iot-device’); var connectionString = ‘<enter>’;

var client = new device.Client(connectionString, new device.Https());

// Send some synthetic data to IoT Hub every second setInterval(function(){

var temperature = Math.random() \* 100 ; var humidity = Math.random() \* 100 ; var light = Math.random() \* 10000 ; var co2 = Math.random() \* 300 ;

var data = "{¨deviceid¨:¨" + "mydevice" + ",¨¨temperature¨:" + String(temperature) + ", ¨humidity¨:" + String(humidity) +

", ¨light¨:" + String(light) + ", ¨co2¨:" + String(co2) + " }";

var message = new device.Message(data); console.log("Sending message: " + message.getData()); client.sendEvent(message);

}, 1000);

To run the program use the commands show in box below:

□ #Running NodeJS program shown in Box 5.32 npm install node .

You will be able to see the count of messages received and the devices connected in the IoT Hub dashboard as seen in Figure 5.20. Now that we can send messages to IoT Hub, let us define some rules for further processing of messages. In the case of AWS IoT, we used the Rule Engine to define the rules using an SQL-like language. Azure provides a real-time event processing engine called Azure Stream Analytics. Azure Stream Analytics allows defining real-time analytic computations on streaming data using an SQL-like language (called Stream Analytics query language). Let us create a Stream Analytics job as shown in Figure 5.21 from the Azure dashboard. A Stream Analytics job includes an input source of streaming data, a query expressed in SQL-like language and an output sink to which the results are sent. Figures 5.22, 5.23 and 5.24 show the settings for the input, query and output of the Stream Analytics job. The input source, in this case, is the IoT Hub we created previously and the output sink is an Azure Event Hub. Event Hub is a managed service for reliably collecting and processing massive amounts of data with low latency. Event Hub provides similar functionality as Amazon Kinesis.

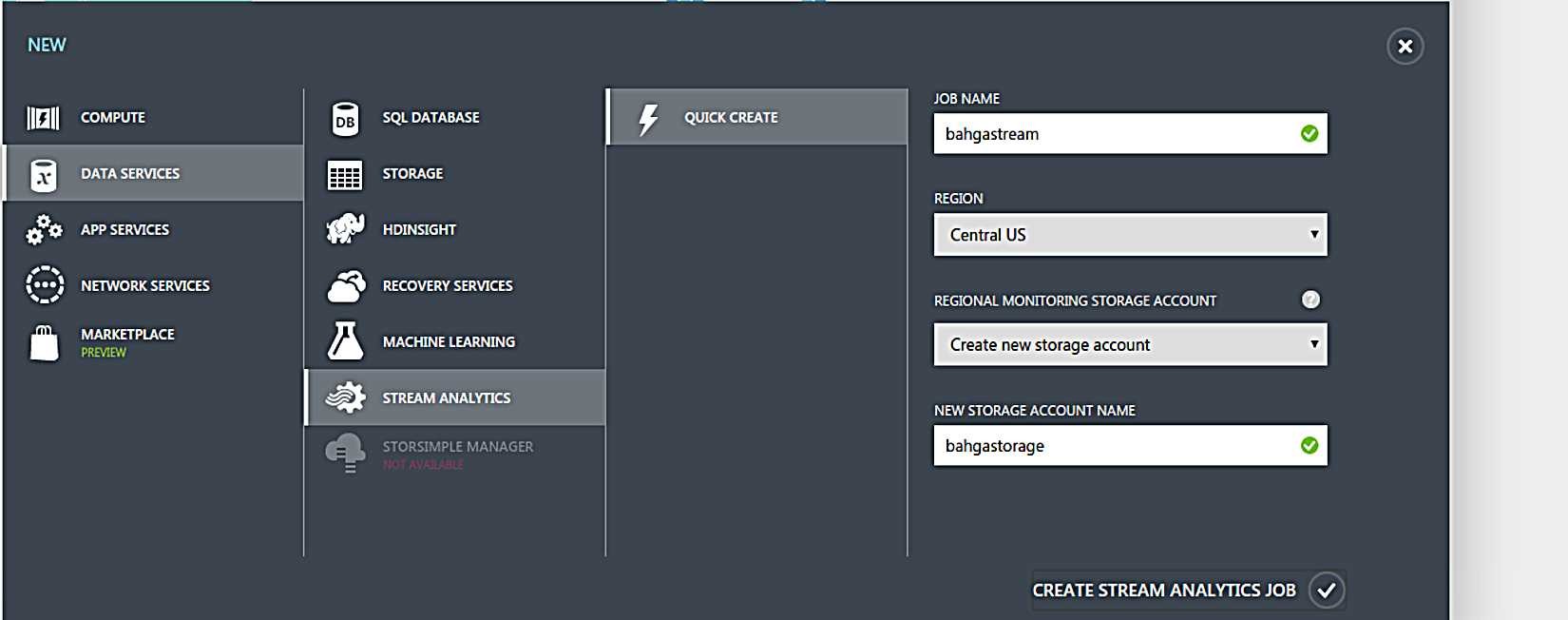


Figure 5.21: Creating an Azure stream analytics job

Figure 5.25 shows how to create a new Event Hub from the Azure dashboard. Once the Event Hub is created, go to the configure tab and add a new shared access policy (with Name

= "read-write" and Permissions = Send, Listen). Copy the Primary Key for the read-write policy. This policy name and the primary key are used while creating the output sink for the Stream Analytics job as shown in Figure 5.24.

Next, run the program show in Box 5.32 and monitor the Stream Analytics job and Events Hub from the Azure dashboard. You will be able to see messages being processed by the Stream Analytics job and the output being posted to the Events Hub as seen in Figures 5.26 and 5.27.

□ #Running the Javascript code in Box 5.32 npm install node .

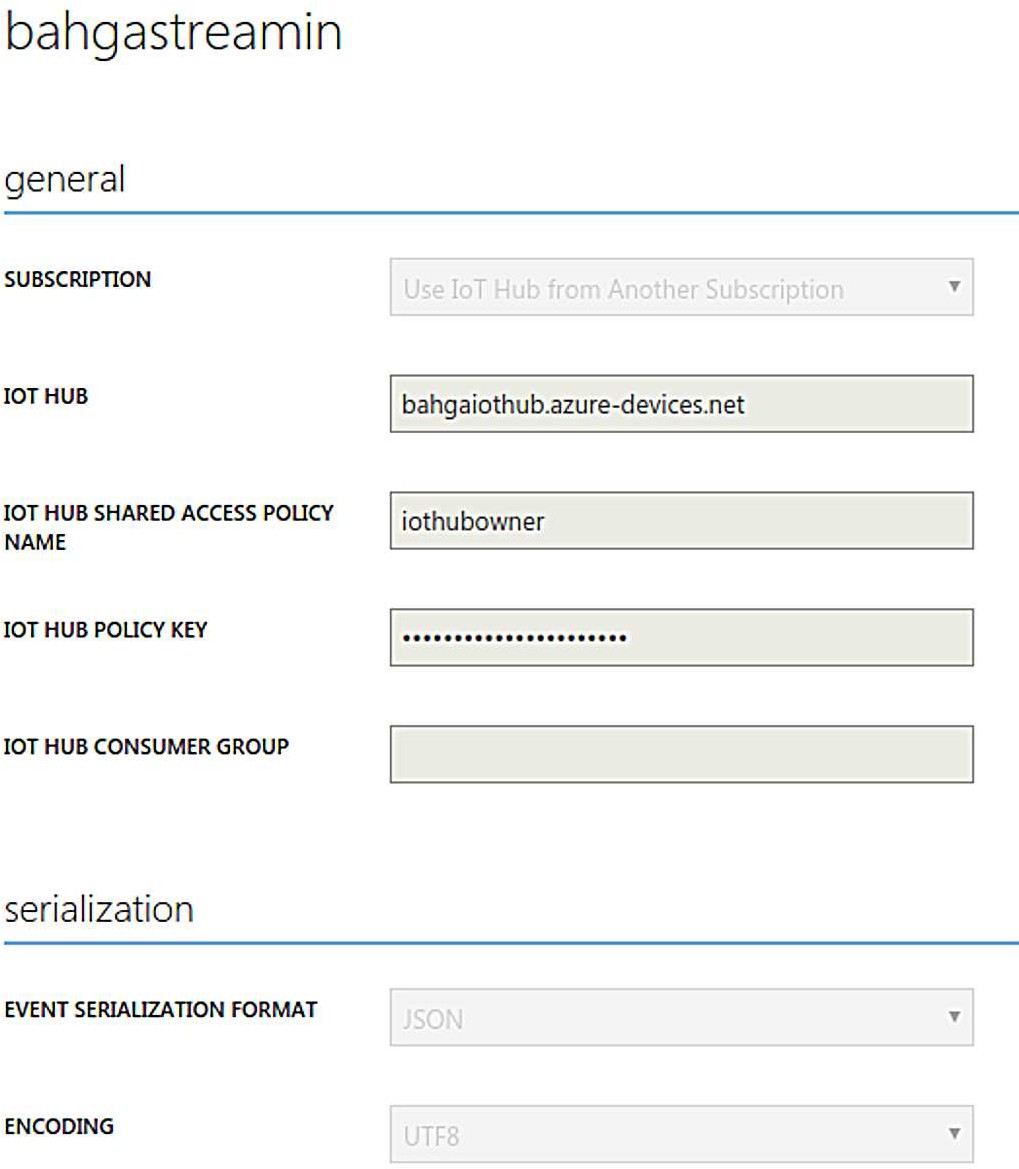


Figure 5.22: Input settings for stream analytics job

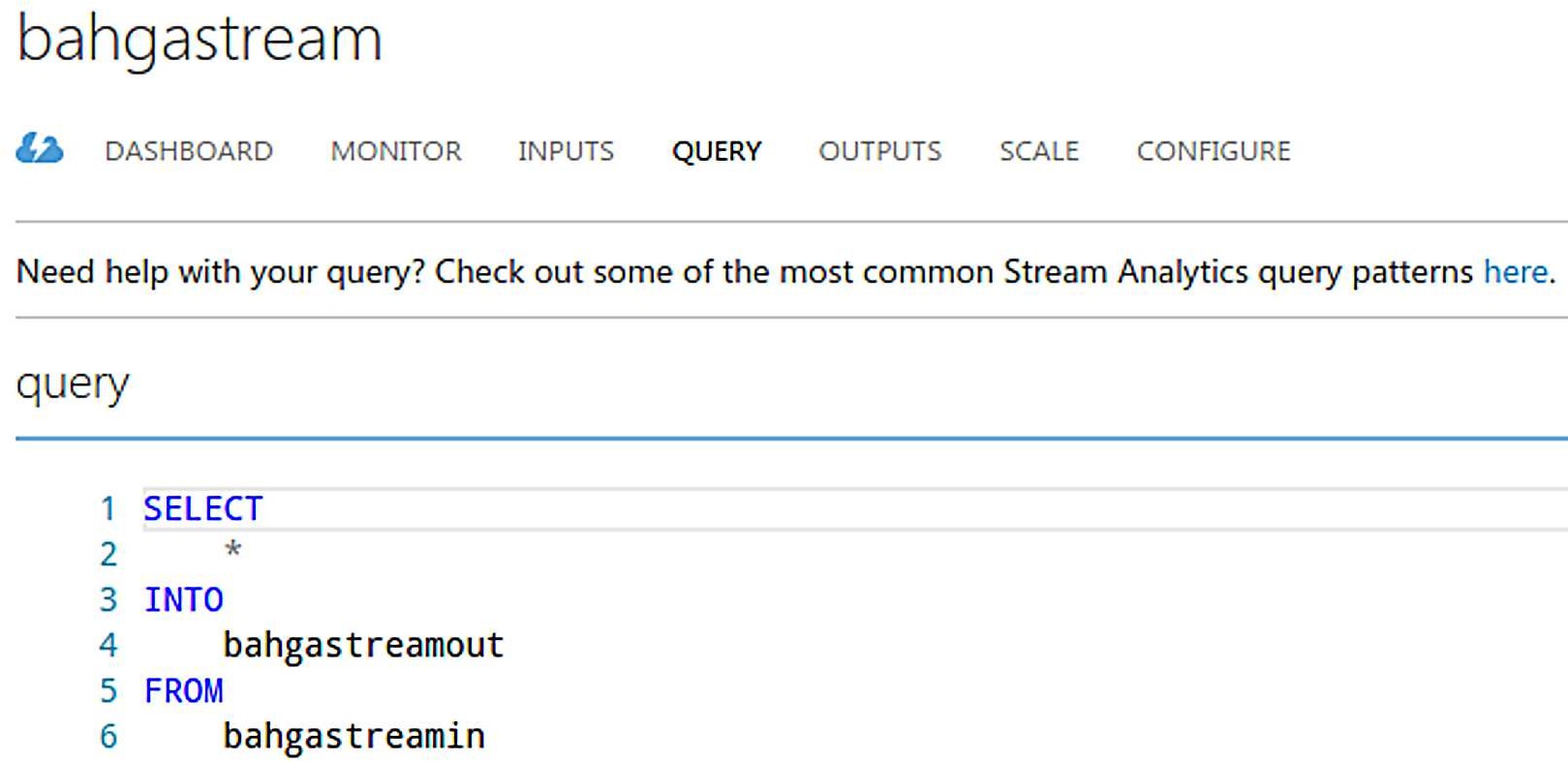


Figure 5.23: Query settings for stream analytics job

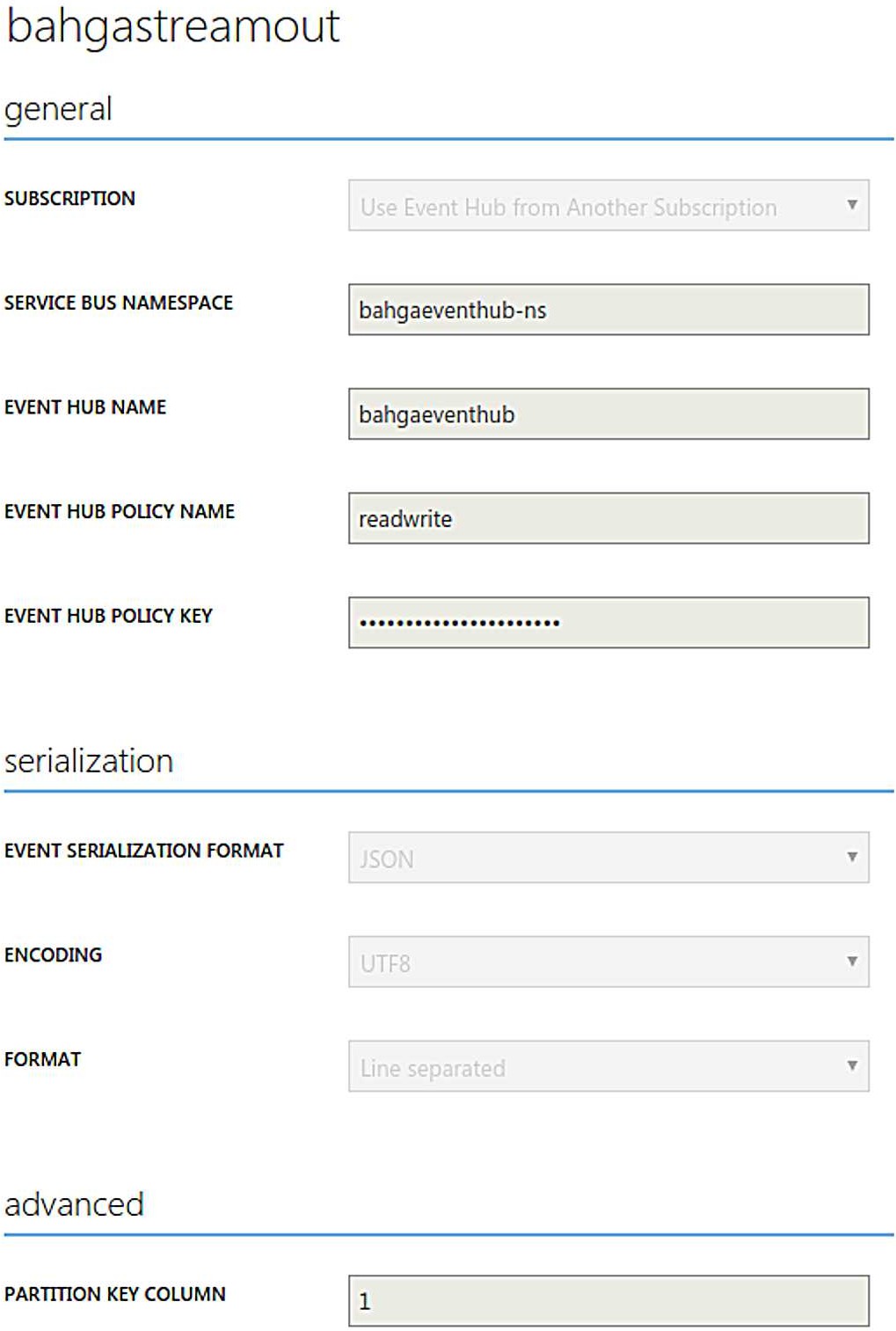


Figure 5.24: Output settings for stream analytics job

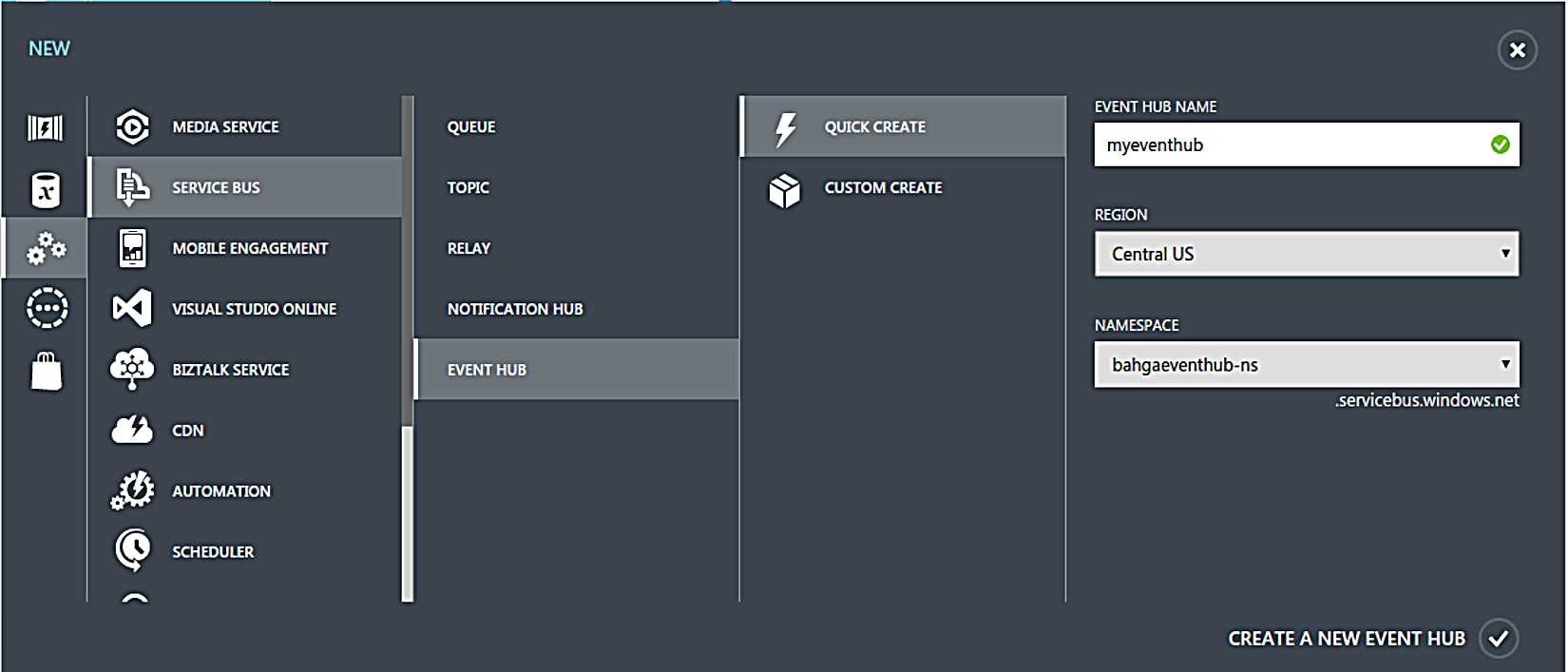


Figure 5.25: Creating an Event Hub from Azure dashboard

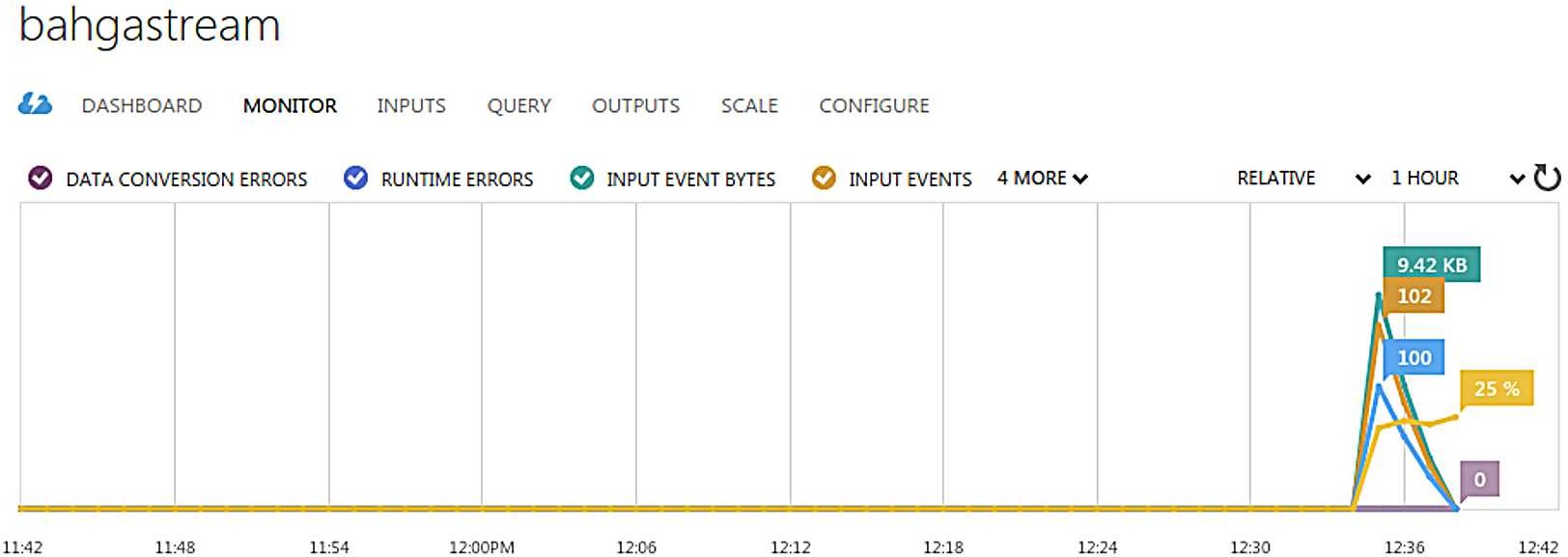


Figure 5.26: Azure Events Hub output

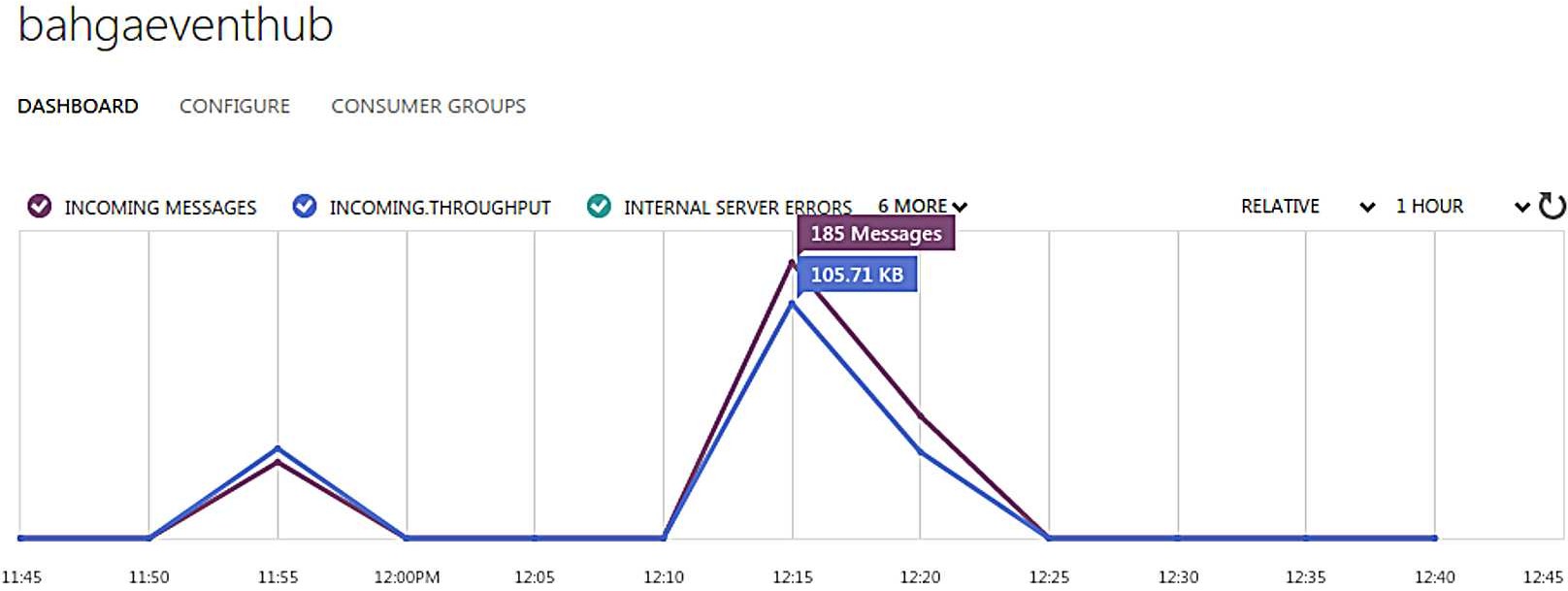


Figure 5.27: Azure Events Hub output

#### Summary

In this chapter, we described various data connectors which allow collecting data from raw data sources for ingesting 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. We described the publish-subscribe and push-pull messaging models. Publish-Subscribe is a communication model that involves publishers, brokers and consumers. Publishers send the data to the topics which are managed by the broker. When the broker receives data for a topic from the publisher, it sends the data to all the subscribed consumers. We described the Apache Kafka and Amazon Kinesis publish-subscribe messaging frameworks. Next, we described a source-sink data collection framework called 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. Next, we described Apache Sqoop, which is a tool that allows importing data from relational database management systems (RDBMS) into the HDFS, Hive or HBase tables. We described various messaging queues such as RabbitMQ, ZeroMQ, RestMQ and Amazon SQS. Examples of building REST-based and MQTT-based custom connectors were provided. Finally, we described IoT services from Amazon and Azure which allow collecting data from Internet of Things (IoT) devices into the cloud, where the data can be processed further

BIG DATA STORAGE :

In the previous chapter we described tools and frameworks for the acquisition of data from various types of sources and ingesting the data into a big data stack. The options for data storage within a big data stack include a distributed filesystem or a NoSQL database. In this chapter, we will describe the Hadoop Distributed File System (HDFS) for big data storage. Once the data is moved from the data source to HDFS, we can use specialized frameworks for batch analysis or interactive querying for analyzing the data.

#### HDFS

HDFS is a distributed file system (DFS) that runs on large clusters and provides high-throughput access to data. HDFS is a highly fault-tolerant system and is designed to work with commodity hardware. HDFS stores each file as a sequence of blocks. The blocks of each file are replicated on multiple machines in a cluster to provide fault tolerance.

Let us look at the characteristics of HDFS:

* **Scalable Storage for Large Files**: HDFS has been designed to store large files (typically from gigabytes to terabytes in size). Large files are broken into chunks or blocks and each block is replicated across multiple machines in the cluster. HDFS has been designed to scale to clusters comprising of thousands of nodes.
* **Replication**: HDFS replicates data blocks to multiple machines in a cluster which makes the system reliable and fault-tolerant. The default block size used is 64MB and the default replication factor is 3.
* **Streaming Data Access**: HDFS has been designed for streaming data access patterns and provides high throughput streaming reads and writes. The HDFS design relaxes some of the POSIX requirements to enable streaming data access and make it suitable for batch operations thus trading off interactive access capability. This design choice has been made to meet the requirements of applications that involve write-once, read many times data access patterns. HDFS is not suited for applications that require low-latency access to data. Instead, HDFS provides high throughput data access.
* **File Appends**: HDFS was originally designed to have immutable files. Files once written to HDFS could not be modified by writing at arbitrary locations in the file or appending to the file. Recent versions of HDFS have introduced the append capability. The file append process is discussed later in the chapter.

##### HDFS Architecture

Figure 6.1 shows the architecture of HDFS. HDFS has two types of nodes: Namenode and Datanode.

**Namenode**

Namenode manages the filesystem namespace. All the filesystem meta-data is stored on the Namenode. While Namenode is responsible for executing operations such as opening and closing of files, no data actually flows through the Namenode. Namenode executes the read and write operations while the data is transferred directly to/from the Datanodes. HDFS splits files into blocks, and the blocks are stored on the Datanodes. For each block, multiple replicas are kept. Namenode persistently stores the filesystem meta-data and the mappings of the blocks to the datanodes, on the disk as two files: *fsimage* and *edits* files. The

*fsimage* contains a complete snapshot of the filesystem meta-data. The *edits* file stores the incremental updates to the meta-data.

When the Namenode starts, it loads the *fsimage* file into the memory and applies the *edits* file to bring the in-memory view of the filesystem up-to-date. Namenode then writes a new *fsimage* file to the disk.

Rack-1

Rack-N

DataNode

DataNode

DataNode

DataNode

NameNode

HDFS Client

Secondary NameNode

Figure 6.1: HDFS architecture

**Secondary Namenode**

The *edits* file keeps growing in size, over time, as the incremental updates are stored. The responsibility of applying the updates to the *fsimage* file is delegated to the Secondary Namenode, as the Namenode may not have enough resources available, as it is performing other operations. This process is called checkpointing. The checkpointing process is done either periodically (default 1 hour) or after a certain number of uncheckpointed transactions have been reached on the Namenode.

When the checkpointing process begins, the Secondary Namenode downloads the *fsimage* and *edits* files from the Namenode to the checkpoint directory on the Secondary Namenode. The Secondary Namenode then applies the *edits* on the *fsimage* file and creates a new *fsimage* file. The new *fsimage* is uploaded by the Secondary Namenode to the Namenode.

**Datanode**

While the Namenode stores the filesystem meta-data, the Datanodes store the data blocks and serve the read and write requests. Datanodes periodically send heartbeat messages and block reports to the Namenode. While the heartbeat messages tell the Namenode that a Datanode is alive, the block reports contain information on the blocks on a Datanode.

**Data Blocks & Replication**

Blocks are replicated on the Datanodes and by default three replicas are created. The placement of replicas on the Datanodes is determined by a rack-aware placement policy. This placement policy ensures reliability and availability of the blocks. For a replication factor of three, one replica is placed on a node on a local rack, the second replica is placed on a different node on a remote rack and the third replica is placed on a different node on the same

remote rack. This ensures that even if the rack becomes unavailable, at least one replica will remain available. Placement of replicas on different nodes in the same rack minimizes the network traffic between the racks.

**HDFS Read Path**

Figure 6.2 shows the HDFS read path. The read process begins with the client sending a request to the Namenode to obtain the locations of the data blocks for a file. The Namenode checks if the file exists and whether the client has sufficient permissions to read the file. The Namenode responds with the data block locations sorted by the distance to the client. This helps in minimizing the traffic between the nodes as the client can read the blocks from the nearest node. For example, if the client is on the same node as a data block, it can read the data block locally. The client reads the data blocks directly from the Datanodes in order, till all the blocks have been read. The Datanodes stream the data to the client. During the read process, if a replica becomes unavailable, the client can read another replica on a different Datanode.

1. Get block

locations

NameNode

Metadata

{file.txt Block A: 1,3,5

Block B: 2,4,5

..}

DataNode-N

2. Read

blocks

HDFS Client

DataNode-1

Figure 6.2: HDFS read path

**HDFS Write Path**

Figure 6.3 shows the HDFS write path. The write process begins with the client sending a request to the Namenode to create a new file in the filesystem namespace. The Namenode checks if the user has sufficient permissions to create the file and whether the file doesn’t already exist in the filesystem. The Namenode responds to the client with an output stream object. The client writes data to the output stream object which splits the data into packets and enqueues them into a data queue. The packets are consumed from the data queue in a separate thread, which requests the Namenode to allocate new blocks on the Datanodes to which the data should be written. Namenode responds with the locations of the blocks on the Datanodes. The client then establishes direct connections to the Datanodes on which the blocks are to be replicated forming a replication pipeline. The data packets consumed from the data queue are written to the first Datanode on the replication pipeline, which writes data to the second Datanode in the pipeline and so on. Once the packets are successfully written, each Datanode in the pipeline sends an acknowledgment. The client keeps a track of which all packets are acknowledged by the Datanodes. The process of writing data packets to the Datanodes proceeds till the block size is reached. Upon reaching the block size, the client again requests the Namenode to return a set of new blocks on the Datanodes. The client then streams the packets to the Datanodes. This process repeats till all the data packets are written and acknowledged. Finally, the client closes the output stream and sends a request to the Namenode to close the file.

1. Create file

NameNode

Metadata

{file.txt Blocks: [],

…}

HDFS Client

8. Complete

7. Ack

DataNode-1

6. Ack

5. Ack

1. Write block
2. Write block
3. Write block

Figure 6.3: HDFS write path

DataNode-N

DataNode-N

##### HDFS Usage Examples

**HDFS Command Line Tools**

□ #Copy file to HDFS #Format of command:

hdfs dfs -put <local source> <destination on HDFS>

#Example:

hdfs dfs -put file /user/hadoop/file

□ #Get file from HDFS #Format of command:

hdfs dfs -get <source on hdfs> <local destination>

#Example:

hdfs dfs -get /user/hadoop/file file

□ #List files on HDFS #Format of command: hdfs dfs -ls <args>

#Example:

hdfs dfs -ls /user/hadoop/

□ #Show contents of a file on HDFS #Format of command:

hdfs dfs -cat <HDFS Path>

#Example:

hdfs dfs -cat /user/hadoop/file

□ #Remove a file on HDFS #Format of command:

hdfs dfs -rm <HDFS Path>

#Example:

hdfs dfs -rm /user/hadoop/file

□ #Create a directory on HDFS #Format of command:

hdfs dfs -mkdir <paths>

#Example:

hdfs dfs -mkdir /user/hadoop/dir

**Accessing HDFS with Python**

In this section we provide Python examples of accessing HDFS using the Snakebite python package.

□ #Listing files on HDFS with Python from snakebite.client import Client

client = Client("localhost", 8020, use\_trash=False) list(client.ls(["/"]))

□ #Reading a file from HDFS with Python from snakebite.client import Client

client = Client("localhost", 8020, use\_trash=False) list(client.text(["/user/input.txt"]))

□ #Copying a file from HDFS with Python from snakebite.client import Client

client = Client("localhost", 8020, use\_trash=False) list(client.copyToLocal(["/user/input.txt"], ’/home/ubuntu/’))

**HDFS Web Interface**

HDFS provides a web interface from where you can browse the filesystem and also also download specific files as shown in Figures 6.4 and 6.5.

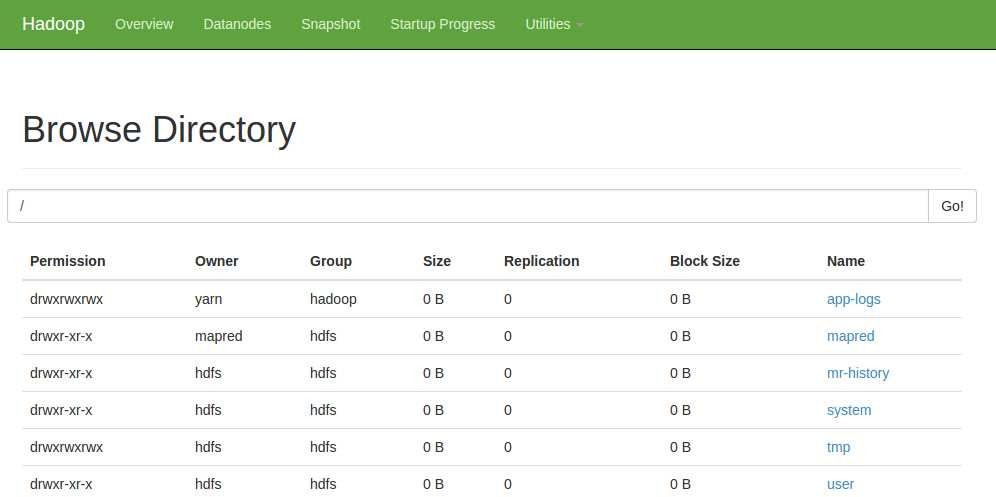


Figure 6.4: Browsing files on HDFS using web interface

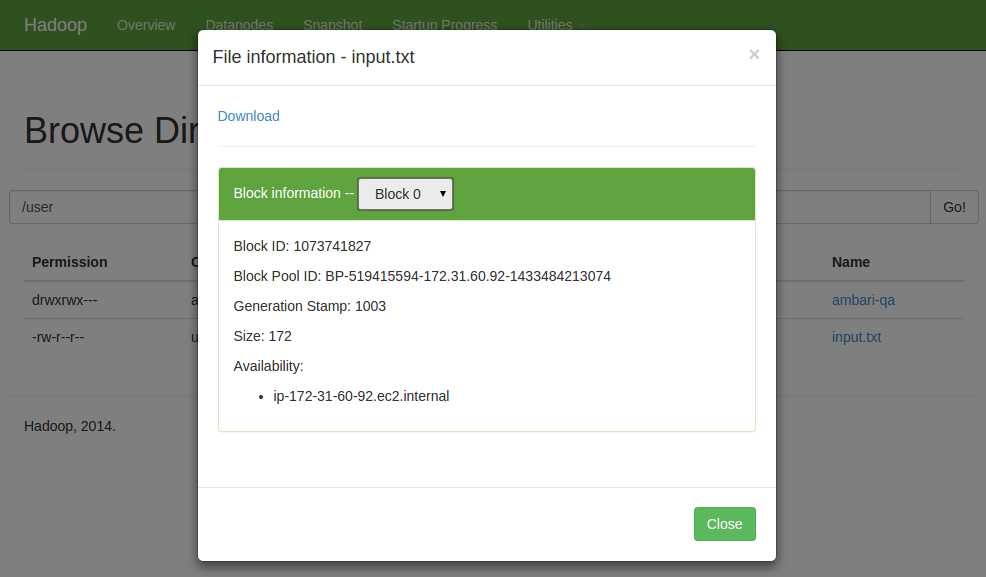


Figure 6.5: Download a file from HDFS using web interface

#### Summary

HDFS is a distributed file system that runs on large clusters and provides high-throughput access to data. HDFS provides scalable storage for large files which are broken into blocks. The blocks are replicated to make the system reliable and fault-tolerant. The HDFS Namenode stores the filesystem meta-data and is responsible for executing operations such as opening and closing of files. The Secondary Namenode helps in the checkpointing process by applying the updates in the *edits* file to the *fsimage* file which contains a complete snapshot of the filesystem meta-data. Datanodes store the data blocks which are replicated. The placement of replicas on the Datanodes is determined by a rack-aware placement policy. We described examples of accessing HDFS using the command line tools, a Python library for HDFS and the HDFS web interface.