Introduction to Flume

Flume is an open-source distributed system designed for efficient and reliable collection, aggregation, and movement of large volumes of log data from various sources to centralized storage systems. It is a part of the Apache Hadoop project and is commonly used in big data processing pipelines.

To use Flume run a Flume agent, which is a long-lived Java process that runs sources and sinks, connected by channels. A source in Flume produces events and delivers them to the channel, which stores the events until they are forwarded to the sink. A combination of Source-Channel-Sink forms the basic Flume building block.

Here's a brief introduction to the key concepts and components of Apache Flume:

Event: The basic unit of data in Flume is called an event. It represents the data that is being transported from the source to the destination.

Source: A source in Flume is responsible for ingesting data into the Flume pipeline. Sources collect events from external systems such as log files, directories, or network sources.

Channel: A channel acts as a transport buffer that stores events while they are being passed from the source to the sink. It decouples the source and sink, enabling better fault tolerance and reliability.

Sink: A sink is responsible for taking the events from the channel and delivering them to a destination, which can be a storage system like Hadoop HDFS, HBase, or another endpoint.

Agent: A Flume agent is an independent Java process that runs on a machine and is responsible for collecting, aggregating, and moving data. An agent consists of a source, one or more channels, and one or more sinks.

Interceptor: Interceptors are optional components that can be added to sources, channels, or sinks to perform transformations on events as they pass through the Flume pipeline.

Flume Configuration: Flume is highly configurable through a properties file where you define sources, channels, and sinks along with their configurations.

Event-driven Architecture: Flume operates on an event-driven architecture, where events trigger the flow of data through the pipeline. This allows for flexible and scalable data collection.

Reliability and Fault Tolerance: Flume is designed to be reliable and fault-tolerant. For example, if a sink fails to deliver an event to its destination, the event is retried or redirected to an alternative path.

Flume Agents in a Distributed Environment: Multiple Flume agents can be deployed in a distributed environment to handle large-scale data ingestion requirements. This makes Flume suitable for processing big data in Hadoop clusters.

Installing Flume

Download a stable release of the Flume binary distribution, and unpack the tarball in a suitable location:

% tar xzf apache-flume-x.y.z-bin.tar.gz

It's useful to put the Flume binary on your path:

% export FLUME_HOME=~/sw/apache-flume-x.y.z-bin

% export PATH=$PATH:$FLUME_HOME/bin

A Flume agent can then be started with the flume-ng command

An Example

To show how Flume works, let's start with a setup that:

1. Watches a local directory for new text files

2. Sends each line of each file to the console as files are added

Add the files to the directory, but it's easy to imagine a process like a web server creating new files that we want to continuously ingest with Flume.

In this example, the Flume agent runs a single source-channel-sink, configured using a Java properties file. The configuration controls the types of sources, sinks, and channels that are

used, as well as how they are connected together. For this example, we'll use the configuration in Example 14-1.

Example 14-1. Flume configuration using a spooling directory source and a logger sink

```
agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /home/cloudera/spooldir
agent1.sinks.sink1.type = logger
agent1.channels.channel1.type = file
```

Property names form a hierarchy with the agent name at the top level. In this example,

- we have a single agent, called agent1.
- The names for the different components in an agent are defined at the next level, so for example agent1.sources lists the names of the sources that should be run in agent1 (here it is a single source, source1). Similarly, agent1 has a sink (sink1) and a channel (channel1).

The properties for each component are defined at the next level of the hierarchy. The configuration properties that are available for a component depend on the type of the component.

- In this case, agent1.sources.source1.type is set to spooldir, which is a spooling directory source that monitors a spooling directory for new files.
- The spooling directory source defines a spoolDir property, so for source1 the full key is agent1.sources.source1.spoolDir.
- The source's channels are set with agent1.sources.source1.channels.

The sink is a logger sink for logging events to the console. It too must be connected to the channel (with the agent1.sinks.sink1.channel property). The channel is a file channel, which

means that events in the channel are persisted to disk for durability. The system is illustrated in Figure 14-1.

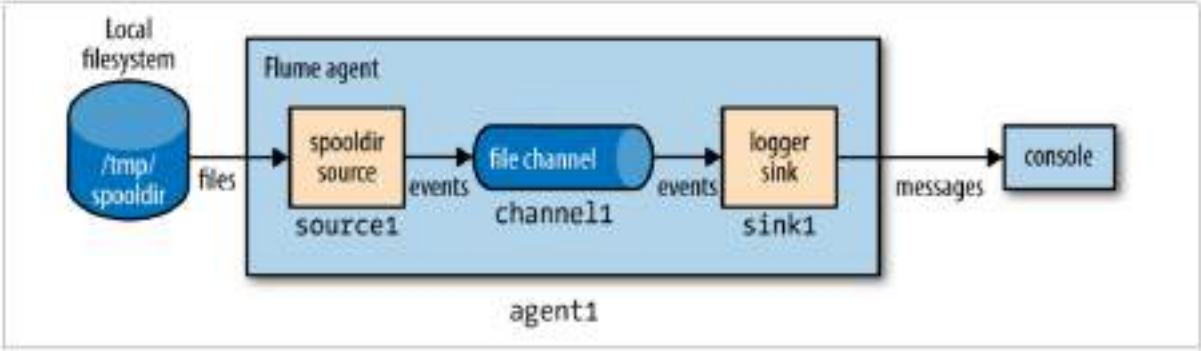


Figure 14-1. Flume agent with a spooling directory source and a logger sink connected by a file channel

Before running the example, we need to create the spooling directory on the local filesystem:

% mkdir spooldir

Then we can start the Flume agent using the flume-ng command:

**% flume-ng agent **

**--conf-file spool.properties **

**--name agent1 **

**--conf $FLUME_HOME/conf **

-Dflume.root.logger=INFO,console

The Flume properties file from Example 14-1 is specified with the --conf-file flag. The agent name must also be passed in with --name. The --conf flag tells Flume where to find its general configuration, such as environment settings.

In a new terminal, create a file in the spooling directory. The spooling directory source expects files to be immutable.

[cloudera@quickstart ~]$ vim input

Hello Flume

Back in the agent's terminal, we see that Flume has detected and processed the file:

Preparing to move file /spooldir/input to spooldir/input.COMPLETED

Event: { headers:{} body: 48 65 6C 6C 6F 20 46 6C 75 6D 65 Hello Flume }

- The spooling directory source ingests the file by splitting it into lines and creating a Flume event for each line.
- Events have optional headers and a binary body, which is the UTF-8 representation of the line of text.
- The body is logged by the logger sink in both hexadecimal and string form.
- The file we placed in the spooling directory was only one line long, so only one event was logged in this case.
- We also see that the file was renamed to input.COMPLETED by the source, which indicates that Flume has completed processing it and won't process it again.

Transactions and Reliability:

- Flume uses separate transactions to guarantee delivery from the source to the channel and from the channel to the sink.
- In the example in the previous section, the spooling directory source creates an event for each line in the file.
- The source will only mark the file as completed once the transactions encapsulating the delivery of the events to the channel have been successfully committed.
- Similarly, a transaction is used for the delivery of the events from the channel to the sink.
- If for some unlikely reason the events could not be logged, the transaction would be rolled back and the events would remain in the channel for later redelivery.
- The channel we are using is a file channel, which has the property of being durable: once an event has been written to the channel, it will not be lost, even if the agent restarts.
- The overall effect is that every event produced by the source will reach the sink.
- The major caveat here is that every event will reach the sink at least once—that is, duplicates are possible.

- Duplicates can be produced in sources or sinks: for example, after an agent restart, the spooling directory source will redeliver events for an uncompleted file, even if some or all of them had been committed to the channel before the restart.
- After a restart, the logger sink will re-log any event that was logged but not committed.

Batching:

For efficiency, Flume tries to process events in batches for each transaction, where possible, rather than one by one. Batching helps file channel performance in particular, since every transaction results in a local disk write and fsync call.

The batch size used is determined by the component in question, and is configurable in many cases. For example, the spooling directory source will read files in batches of 100 lines. Similarly, the Avro sink will try to read 100 events from the channel before sending them over RPC.

The HDFS Sink:

Note: The configuration in Example 14-2 updates the previous example to use an HDFS sink.

The only two settings that are required are the sink's type (hdfs) and hdfs.path, which specifies the directory where files will be placed. We've also specify a meaningful file prefix and suffix, and instructed Flume to write events to the files in text format.

Example 14-2. Flume configuration using a spooling directory source and an HDFS Sink

```
agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1
agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /home/cloudera/spooldir

agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
```

agent1.sinks.sink1.hdfs.fileSuffix = **.log**

agent1.sinks.sink1.hdfs.inUsePrefix = **_**

agent1.sinks.sink1.hdfs.fileType = **DataStream**

agent1.channels.channel1.type = file

Restart the agent to use the *hdfs.properties* configuration, and create a new file in the spooling directory:

[cloudera@quickstart ~]$ vim input

Hello

Again

- Events will now be delivered to the HDFS sink and written to a file.
- Files in the process of being written to have a *.tmp* in-use suffix added to their name to indicate that they are not yet complete.
- In this example, we have also set hdfs.inUsePrefix to be _(underscore; by default it is empty), which causes files in the process of being written to have that prefix added to their names.
- This is useful since MapReduce will ignore files that have a _ prefix. So, a typical temporary filename would be *_events.1399295780136.log.tmp*; the number is a timestamp generated by the HDFS sink.
- A file is kept open by the HDFS sink until it has either been open for a given time, has reached a given size, or has had a given number of events written to it.
- If any of these criteria are met, the file is closed and its in-use prefix and suffix are removed.
- New events are written to a new file.
- After 30 seconds, we can be sure that the file has been rolled and we can take a look at its contents:

  % **hadoop fs -cat /tmp/flume/events.1399295780136.log**

  Hello

  Again

Partitioning and Interceptors:

Large datasets are often organized into partitions, so that processing can be restricted to particular partitions if only a subset of the data is being queried. For Flume event data, it's very common to partition by time. A process can be run periodically that transforms completed partitions.

It's easy to change the example to store data in partitions by setting hdfs.path to include subdirectories that use time format escape sequences:

```
agent1.sinks.sink1.hdfs.path = /tmp/flume/year=%Y/month=%m/day=%d
```

Here we have chosen to have day-sized partitions, but other levels of granularity are possible, as are other directory layout schemes.

The partition that a Flume event is written to is determined by the timestamp header on the event. Events don't have this header by default, but it can be added using a Flume interceptor. Interceptors are components that can modify or drop events in the flow; they are attached to sources, and are run on events before the events have been placed in a channel. The following extra configuration lines add a timestamp interceptor to source1, which adds a timestamp header to every event produced by the source:

```
agent1.sources.source1.interceptors = interceptor1
agent1.sources.source1.interceptors.interceptor1.type = timestamp
```

Using the timestamp interceptor ensures that the timestamps closely reflect the times at which the events were created. For some applications, using a timestamp for when the event was written to HDFS might be sufficient—although, be aware that when there are multiple tiers of Flume agents there can be a significant difference between creation time and write time, especially in the event of agent downtime. For these cases, the HDFS sink has a setting, hdfs.useLocal TimeStamp, that will use a timestamp generated by the Flume agent running the HDFS sink.

File Formats

It's normally a good idea to use a binary format for storing your data in, since the resulting files are smaller than they would be if you used text. For the HDFS sink, the file format used is controlled using hdfs.fileType and a combination of a few other properties.

If unspecified, hdfs.fileType defaults to SequenceFile, which will write events to a sequence file with LongWritable keys that contain the event timestamp and BytesWritable values that contain the event body. It's possible to use Text Writable values in the sequence file instead of BytesWritable by setting hdfs.writeFormat to Text.

The configuration is a little different for Avro files. The hdfs.fileType property is set to DataStream, just like for plain text. Additionally, serializer must be set to avro_event. To enable compression, set the serializer.compressionCodec property. Here is an example of an HDFS sink configured to write Snappy-compressed Avro files:

agent1.sinks.sink1.type = hdfs

agent1.sinks.sink1.hdfs.path = /tmp/flume

agent1.sinks.sink1.hdfs.filePrefix = events

agent1.sinks.sink1.hdfs.fileSuffix = .avro

agent1.sinks.sink1.hdfs.fileType = DataStream

agent1.sinks.sink1.serializer = avro_event

agent1.sinks.sink1.serializer.compressionCodec = snappy

An event is represented as an Avro record with two fields: headers, an Avro map with string values, and body, an Avro bytes field.

If you want to use a custom Avro schema, there are a couple of options. If you have Avro in-memory objects that you want to send to Flume, then the Log4jAppender is appropriate.

It allows you to log an Avro Generic, Specific, or Reflect object using a log4j Logger and send it to an Avro source running in a Flume agent. In this case, the serializer property for the HDFS sink should be set to org.apache.flume.sink.hdfs.AvroEventSerializer$Builder, and the Avro schema set in the header (see the class documentation).

Alternatively, if the events are not originally derived from Avro objects, you can write a custom serializer to convert a Flume event into an Avro object with a custom schema.

The helper class AbstractAvroEventSerializer in the org.apache.flume.serialization package is a good starting point.

Fan Out

Fan out is the term for delivering events from one source to multiple channels, so they reach multiple sinks. For example, the configuration in Example 14-3 delivers events to both an HDFS sink (sink1a via channel1a) and a logger sink (sink1b via channel1b).

Example 14-3. Flume configuration using a spooling directory source, fanning out to an HDFS sink and a logger sink

```
agent1.sources = source1
agent1.sinks = sink1a sink1b
agent1.channels = channel1a channel1b

agent1.sources.source1.channels = channel1a channel1b
agent1.sinks.sink1a.channel = channel1a
agent1.sinks.sink1b.channel = channel1b

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /home/cloudera/spooldir
agent1.sinks.sink1a.type = hdfs
agent1.sinks.sink1a.hdfs.path = /tmp/flume
agent1.sinks.sink1a.hdfs.filePrefix = events
agent1.sinks.sink1a.hdfs.fileSuffix = .log

agent1.sinks.sink1a.hdfs.fileType = DataStream
agent1.sinks.sink1b.type = logger
agent1.channels.channel1a.type = file
agent1.channels.channel1b.type = memory
```

- The key change here is that the source is configured to deliver to multiple channels by setting agent1.sources.source1.channels to a space-separated list of channel names, channel1a and channel1b.
- This time, the channel feeding the logger sink (channel1b) is a memory channel, since we are logging events for debugging purposes and don't mind losing events on agent restart.

- Also, each channel is configured to feed one sink, just like in the previous examples. The flow is illustrated in Figure 14-2.

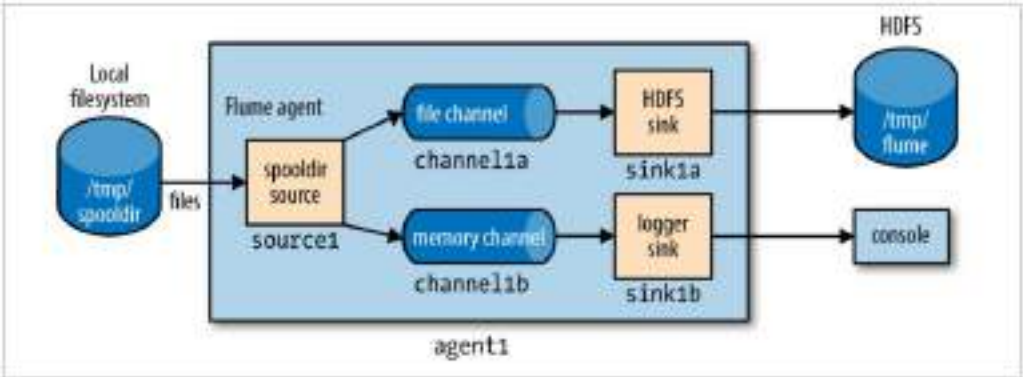


Figure 14-2. Flume agent with a spooling directory source and fanning out to an HDFS sink and a logger sink

Delivery Guarantees

- Flume uses a separate transaction to deliver each batch of events from the spooling directory source to each channel.
- In this example, one transaction will be used to deliver to the channel feeding the HDFS sink, and then another transaction will be used to deliver the same batch of events to the channel for the logger sink.
- If either of these transactions fails (if a channel is full, for example), then the events will not be removed from the source, and will be retried later.
- In this case, since we don't mind if some events are not delivered to the logger sink, we can designate its channel as an optional channel, so that if the transaction associated with it fails, this will not cause events to be left in the source and tried again later.
- To do this, we set the selector.optional property on the source, passing it a space-separated list of channels:

```
agent1.sources.source1.selector.optional = channel1b
```

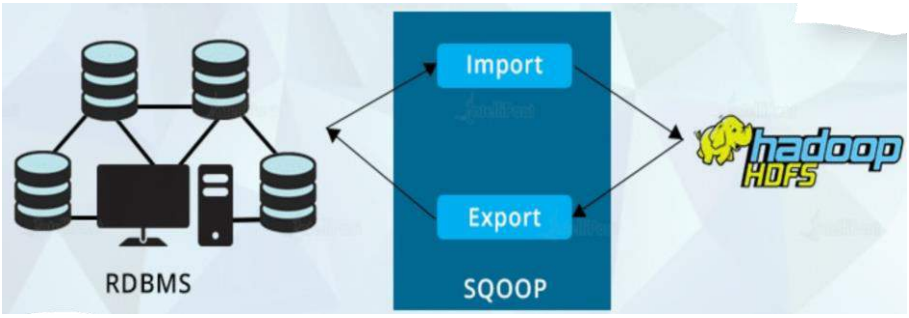
Replicating and Multiplexing Selectors

In normal fan-out flow, events are replicated to all channels—but sometimes more selective behavior might be desirable, so that some events are sent to one channel and others to another. This can be achieved by setting a multiplexing selector on the source, and defining routing rules that map particular event header values to channels.

Introduction to Sqoop:

Sqoop is a data ingestion tool that is designed to transfer data between RDBMS systems (such as Oracle, MySQL, SQL Server, Postgres, Teradata, etc) and Hadoop HDFS.

Sqoop stands for — “SQL to Hadoop & Hadoop to SQL”. It is developed by Cloudera.



Apache Sqoop is an open-source tool that allows users to extract data from a structured data store into Hadoop for further processing. This processing can be done with MapReduce programs or other higher-level tools such as Hive. When the final results of an analytic pipeline are available, Sqoop can export these results back to the data store for consumption by other clients.

Getting Sqoop:

Running Sqoop with no arguments does not do much of interest:

```
% sqoop
Try sqoop help for usage.
```

Sqoop is organized as a set of tools or commands. If you don’t select a tool, Sqoop does not know what to do. help is the name of one such tool; it can print out the list of available tools, like this:

```
% sqoop help
usage: sqoop COMMAND [ARGS]

Available commands:
  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
  export             Export an HDFS directory to a database table
  help               List available commands
  import             Import a table from a database to HDFS
  import-all-tables  Import tables from a database to HDFS
  job                Work with saved jobs
  list-databases     List available databases on a server
  list-tables        List available tables in a database
  merge              Merge results of incremental imports
  metastore          Run a standalone Sqoop metastore
  version            Display version information
```

See 'sqoop help COMMAND' for information on a specific command.

As it explains, the help tool can also provide specific usage instructions on a particular tool when you provide that tool's name as an argument:

```
% sqoop help import
usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]

Common arguments:
   --connect <jdbc-uri>     Specify JDBC connect string
   --driver <class-name>    Manually specify JDBC driver class to use
   --hadoop-home <dir>      Override $HADOOP_HOME
   --help                   Print usage instructions
-P                          Read password from console
   --password <password>    Set authentication password
   --username <username>    Set authentication username
   --verbose                Print more information while working
...
```

Sqoop Connectors:

- Sqoop has an extension framework that makes it possible to import data from—and export data to—any external storage system that has bulk data transfer capabilities.
- A Sqoop connector is a modular component that uses this framework to enable Sqoop imports and exports.
- Sqoop ships with connectors for working with a range of popular databases, including MySQL, PostgreSQL, Oracle, SQL Server, DB2, and Netezza.
- There is also a generic JDBC connector for connecting to any database that supports Java's JDBC protocol. Sqoop provides optimized MySQL, PostgreSQL, Oracle, and Netezza connectors that use database-specific APIs to perform bulk transfers more efficiently.
- As well as the built-in Sqoop connectors, various third-party connectors are available for data stores, ranging from enterprise data warehouses (such as Teradata) to NoSQL stores (such as Couchbase).
- These connectors must be downloaded separately and can be added to an existing Sqoop installation by following the instructions that come with the connector.

A Sample Import:

let's log in to mysql and create a database (Example 15-1).

Example 15-1. Creating a new MySQL database schema

```
% mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 235
Server version: 5.6.21 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

mysql> CREATE DATABASE hadoopguide;
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON hadoopguide.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)
```

- we created a new database schema called hadoopguide.
- We then allowed any local user to view and modify the contents of the hadoopguide schema

```
mysql> CREATE TABLE emp (id int, name varchar (40), age int, dept varchar (20));
mysql> Insert into emp values (100, 'ravi',25, 'HR');
mysql> Insert into emp values (101, 'raju',25, 'IT');
mysql> Insert into emp values (102, 'rajesh',27, 'IT');
mysql> Insert into emp values (103, 'raj',27, 'IT');
mysql> Insert into emp values (104, 'rajiv',29, 'HR');
mysql> Insert into emp values (105, 'ram',26, 'HR');
mysql> select * from emp;
+-----+--------+------+------+
| id  | name   | age  | dept |
+-----+--------+------+------+
| 100 | Ravi   | 25   | HR   |
| 101 | Raju   | 25   | IT   |
| 102 | Rajesh | 27   | IT   |
| 103 | Raj    | 27   | IT   |
| 104 | Rajiv  | 29   | HR   |
| 105 | Ram    | 26   | HR   |
+-----+--------+------+------+
6 rows in set (0.00 sec)
```

- we created a new table called emp and then inserted data into the table emp.

Now let's use Sqoop to import this table into HDFS:

```
[cloudera@quickstart ~]$ sqoop import \
> --connect jdbc:mysql://localhost/hadoopguide \
> --username root \
> --password cloudera \
> --table emp \
> --m 1
```

Sqoop's import tool will run a MapReduce job that connects to the MySQL database and reads the table. By default, this will use four map tasks in parallel to speed up the import process. Each task will write its imported results to a different file, but all in a common directory. Because we knew that we had only three rows to import in this example, we specified that Sqoop should use a single map task (-m 1) so we get a single file in HDFS.

We can inspect this file's contents like so:

% **hadoop fs -cat emp/part-m-00000**
100, Ravi, 25, HR
101, Raju, 25, IT
102, Rajesh, 27, IT
103, Raj, 27, IT
104, Rajiv, 29, HR
105, Ram, 26, HR

By default, Sqoop will generate comma-delimited text files for our imported data. Delimiters can be specified explicitly, as well as field enclosing and escape characters, to allow the presence of delimiters in the field contents.

Text and Binary File Formats:

- Sqoop is capable of importing into a few different file formats.
- Text files (the default) offer a human-readable representation of data, platform independence, and the simplest structure.
- However, they cannot hold binary fields (such as database columns of type VARBINARY), and distinguishing between null values and String-based fields containing the value "null" can be problematic.
- To handle these conditions, Sqoop also supports SequenceFiles, Avro datafiles, and Parquet files.
- These binary formats provide the most precise representation possible of the imported data.
- They also allow data to be compressed while retaining MapReduce's ability to process different sections of the same file in parallel.
- However, current versions of Sqoop cannot load Avro datafiles or SequenceFiles into Hive.
- Another disadvantage of SequenceFiles is that they are Java specific, whereas Avro and Parquet files can be processed by a wide range of languages.

Generated Code:

- In addition to writing the contents of the database table to HDFS, Sqoop also provides you with a generated Java source file (widgets.java) written to the current local directory.
- Sqoop can use generated code to handle the deserialization of table-specific data from the database source before writing it to HDFS.
- The generated class (emp) is capable of holding a single record retrieved from the imported table.
- It can manipulate such a record in MapReduce or store it in a SequenceFile in HDFS.
- It is likely that you don't want to name your generated class emp, since each instance of the class refers to only a single record.
- We can use a different Sqoop tool to generate source code without performing an import; this generated code will still examine the database table to determine the appropriate data types for each field:

```
% sqoop codegen  \
> --connect jdbc:mysql://localhost/hadoopguide \
> --table emp  \
> --class-name Emp
```

- The codegen tool simply generates code; it does not perform the full import.
- We specified that we'd like it to generate a class named Emp; this will be written to Emp.java.
- We also could have specified --class-name and other code-generation arguments during the import process we performed earlier.
- This tool can be used to regenerate code if you accidentally remove the source file, or generate code with different settings than were used during the import.
- If you're working with records imported to SequenceFiles, it is inevitable that you'll need to use the generated classes (to deserialize data from the SequenceFile storage).

Imports: A Deeper Look:

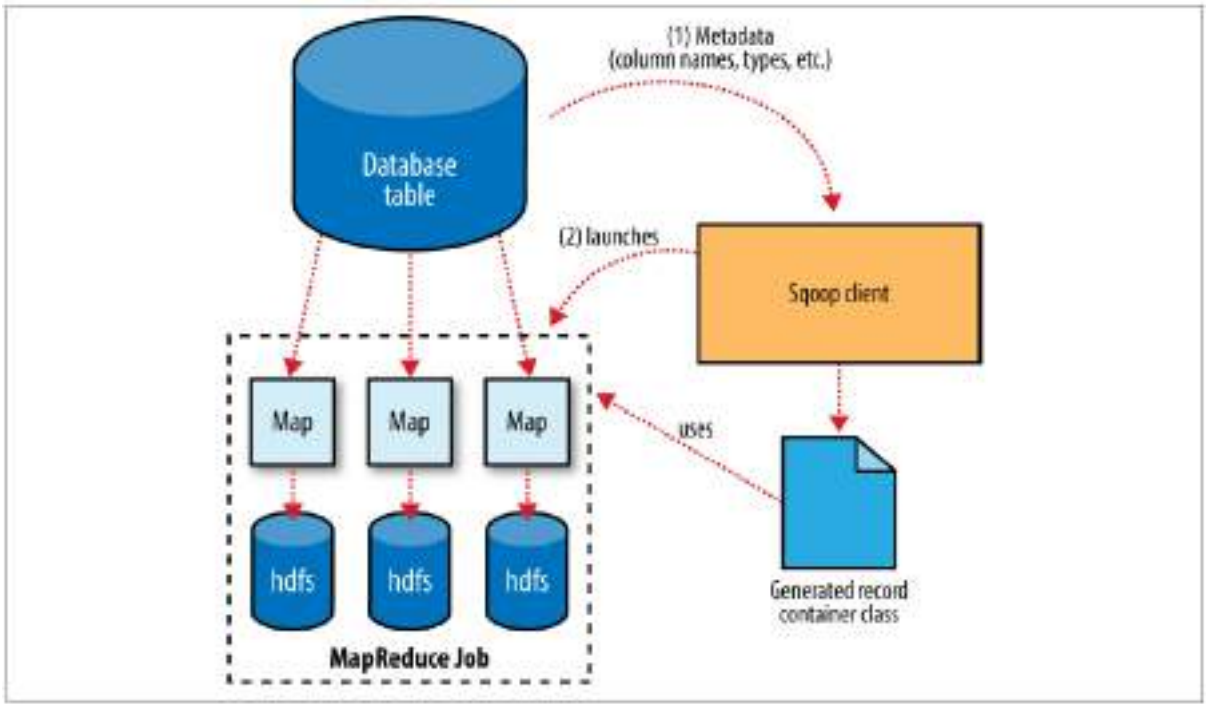


Figure 15-1. Sqoop's import process

- At a high level, Figure 15-1 demonstrates how Sqoop interacts with both the database and Hadoop.
- Like Hadoop itself, Sqoop is written in Java.
- Java provides an API called Java Database Connectivity, or JDBC, that allows applications to access data stored in an RDBMS.
- Most database vendors provide a JDBC driver that implements the JDBC API and contains the necessary code to connect to their database servers.
- Before the import can start, Sqoop uses JDBC to examine the table it is to import.
- It retrieves a list of all the columns and their SQL data types.
- These SQL types (VARCHAR, INTEGER, etc.) can then be mapped to Java data types (String, Integer, etc.), which will hold the field values in MapReduce applications.
- Sqoop's code generator will use this information to create a table-specific class to hold a record extracted from the table.

mysql> CREATE TABLE widgets(id INT, name VARCHAR(64), dept VARCHAR(20));

- The Widget class from earlier, for example, contains the following methods that retrieve each column from an extracted record:

```
public Integer get_id();
public String get_name();
public String get_dept();
```

- More critical to the import system's operation, though, are the serialization methods that form the DBWritable interface, which allow the Widget class to interact with JDBC:

```
public void readFields(ResultSet __dbResults) throws SQLException;
public void write(PreparedStatement __dbStmt) throws SQLException;
```

- JDBC's ResultSet interface provides a cursor that retrieves records from a query; the readFields() method here will populate the fields of the Widget object with the columns from one row of the ResultSet's data.
- The write() method shown here allows Sqoop to insert new Widget rows into a table, a process called *exporting*.
- The MapReduce job launched by Sqoop uses an InputFormat that can read sections of a table from a database via JDBC.
- The DataDrivenDBInputFormat provided with Hadoop partitions a query's results over several map tasks.

Controlling the Import:

- Sqoop does not need to import an entire table at a time.
- For example, a subset of the table's columns can be specified for import.
- Users can also specify a WHERE clause to include in queries via the --where argument, which bounds the rows of the table to import.
- For example, if widgets 0 through 99,999 were imported last month, but this month our vendor catalog included 1,000 new types of widget, an import could be configured with the clause WHERE id >= 100000; this will start an import job to retrieve all the new rows added to the source database since the previous import run.
- For more control—to perform column transformations, for example—users can specify a --query argument.

**sqoop import **
**--connect jdbc:mysql://localhost/kmit **
**--username root **
**--password cloudera **
**--table emp **
**--target-dir /subset **
--where "id<103"
--m 1

Incremental Imports:

It's common to run imports on a periodic basis so that the data in HDFS is kept synchronized with the data stored in the database. To do this, there needs to be some way of identifying the new data. Sqoop will import rows that have a column value (for the column specified with --check-column) that is greater than some specified value (set via --last-value).

The value specified as --last-value can be a row ID that is strictly increasing, such as an AUTO_INCREMENT primary key in MySQL. This is suitable for the case where new rows are added to the database table, but existing rows are not updated. This mode is called append mode, and is activated via --incremental append. Another option is timebased incremental imports (specified by --incremental lastmodified), which is appropriate when existing rows may be updated, and there is a column (the check column) that records the last modified time of the update. At the end of an incremental import, Sqoop will print out the value to be specified as --last-value on the next import.

**sqoop import **
**--connect jdbc:mysql://localhost/kmit **
**--username root **
**--password cloudera **
**--table emp1 **
**--target-dir /subset **
**--incremental append **
**--check-column id **
**--last-value 102 **
--m 1

Direct-Mode Imports:

When performing an import in Sqoop, you have two options for specifying the data to import: direct mode and non-direct mode.

In direct mode imports, Sqoop uses a direct link between the Hadoop cluster and the database server. This direct link can result in faster data transfers by leveraging database-specific connectors and parallel processing. However, not all databases support direct mode, and it might not be suitable for all scenarios.

Here's an example of using direct mode in a Sqoop import command:

```
sqoop import \
  --connect jdbc:mysql://hostname:port/database \
  --username your_username \
  --password your_password \
  --table your_table \
  --direct
```

In this example:

--connect: Specifies the JDBC connection URL for the database.

--username: Specifies the database username.

--password: Specifies the database password.

--table: Specifies the table to import.

--direct: Enables direct mode.

Imported Data and Hive:

you have a MySQL table named "emp" with columns "id," "name," and "dept," and it contains some sample data.

mysql> select * from emp;

id	name	dept
100	raj	IT
101	raju	IT
102	rajiv	IT
103	rajesh	IT

Next create a database in hive using the following

hive> create database hi;

Now, we can transfer the data from Mysql to Hive using the following

```
sqoop import \
  --connect jdbc:mysql://localhost/hello \
  --username root \
  --password cloudera \
  --table emp \
  --m 1 \
  --hive-import \
  --create-hive-table \
  --hive-table employee
```

Using --hive-import and --create-hive-table, Sqoop will automatically create a Hive table and manage its location.

When specifying the --hive-table parameter, you should only provide the table name (employee), not the database and table. The database is already set using the USE hi; command in Hive.

Importing Large Objects

Most databases provide the capability to store large amounts of data in a single field. Depending on whether this data is textual or binary in nature, it is usually represented as a CLOB or BLOB column in the table. These "large objects" are often handled specially by the database itself. In particular, most tables are physically laid out on disk as in Figure 15-2.

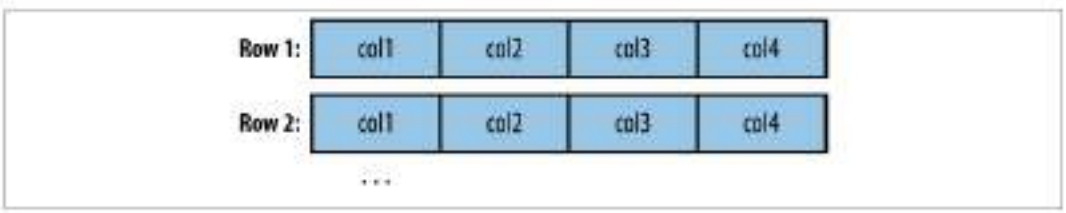


Figure 15-2. Database tables are typically physically represented as an array of rows, with all the columns in a row stored adjacent to one another

When scanning through rows to determine which rows match the criteria for a particular query, this typically involves reading all columns of each row from disk.

If large objects were stored "inline" in this fashion, they would adversely affect the performance of such scans. Therefore, large objects are often stored externally from their rows, as in Figure 15-3.

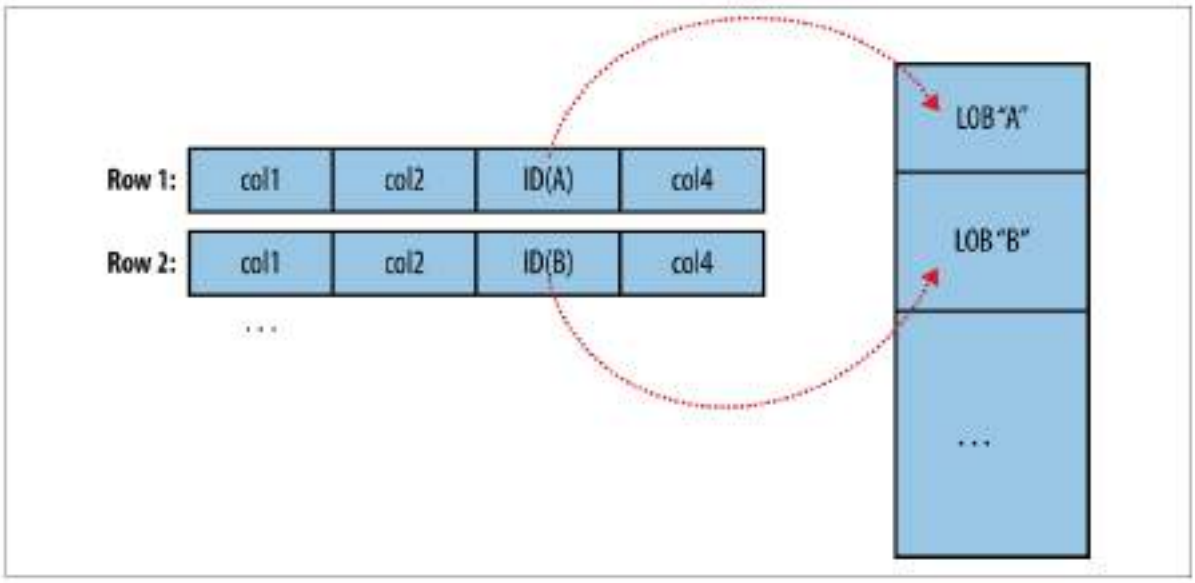


Figure 15-3. Large objects are usually held in a separate area of storage; the main row storage contains indirect references to the large objects

Accessing a large object often requires "opening" it through the reference contained in the row.

The difficulty of working with large objects in a database suggests that a system such as Hadoop, which is much better suited to storing and processing large, complex data objects, is an ideal repository for such information. Sqoop can extract large objects from tables and store them in HDFS for further processing.

As in a database, MapReduce typically materializes every record before passing it along to the mapper. If individual records are truly large, this can be very inefficient.

As shown earlier, records imported by Sqoop are laid out on disk in a fashion very similar to a database's internal structure: an array of records with all fields of a record concatenated together. When running a MapReduce program over imported records, each map task must fully materialize all fields of each record in its input split. If the contents of a large object field are relevant only for a small subset of the total number of records used as input to a MapReduce program, it would be inefficient to fully materialize all these records. Furthermore, depending on the size of the large object, full materialization in memory may be impossible.

To overcome these difficulties, Sqoop will store imported large objects in a separate file called a LobFile, if they are larger than a threshold size of 16 MB (configurable via the sqoop.inline.lob.length.max setting, in bytes). The LobFile format can store individual records of very large size (a 64-bit address space is used). Each record in a LobFile holds a single large object. The LobFile format allows clients to hold a reference to a record without accessing the record contents. When records are accessed, this is done through a java.io.InputStream (for binary objects) or java.io.Reader (for character-based objects).

When a record is imported, the "normal" fields will be materialized together in a text file, along with a reference to the LobFile where a CLOB or BLOB column is stored. For example, suppose our widgets table contained a BLOB field named schematic holding the actual schematic diagram for each widget.

An imported record might then look like:

2,gizmo,4.00,2009-11-30,4,null,externalLob(lf,lobfile0,100,5011714)

The externalLob(...) text is a reference to an externally stored large object, stored in LobFile format (lf) in a file named lobfile0, with the specified byte offset and length inside that file.

When working with this record, the Widget.get_schematic() method would return an object of type BlobRef referencing the schematic column, but not actually containing its contents. The BlobRef.getDataStream() method actually opens the LobFile and returns an InputStream, allowing you to access the schematic field's contents.

When running a MapReduce job processing many Widget records, you might need to access the schematic fields of only a handful of records. This system allows you to incur the I/O costs of accessing only the required large object entries—a big savings, as individual schematics may be several megabytes or more of data.

The BlobRef and ClobRef classes cache references to underlying LobFiles within a map task. If you do access the schematic fields of several sequentially ordered records, they will take advantage of the existing file pointer's alignment on the next record body.

Performing an Export:

In Sqoop, an import refers to the movement of data from a database system into HDFS. By contrast, an export uses HDFS as the source of data and a remote database as the destination.

Before exporting a table from HDFS to a database, we must prepare the database to receive the data by creating the target table.

Then we run the export command:

```
sqoop export \
  --connect jdbc:<db_type>://<db_host>:<db_port>/<database> \
  --username <db_username> \
  --password <db_password> \
  --table <db_table> \
  --export-dir /your/source/directory
```

Replace placeholders such as <db_type>, <db_host>, <db_port>, <database>, <db_username>, <db_password>, <db_table>, and /your/source/directory with your specific database and directory details.

Exports: A Deeper Look:

The Sqoop exports are similar in nature to how Sqoop performs imports. Before performing the export, Sqoop picks a strategy based on the database connect string. For most systems, Sqoop uses JDBC. Sqoop then generates a Java class based on the target table definition. This generated class has the ability to parse records from text files and insert values of the appropriate types into a table. A MapReduce job is then launched that reads the source datafiles from HDFS, parses the records using the generated class, and executes the chosen export strategy.

The JDBC-based export strategy builds up batch INSERT statements that will add multiple records to the target table. Inserting many records per statement performs much better than executing many single-row INSERT statements on most database systems. Separate threads are used to read from HDFS and communicate with the database, to ensure that I/O operations involving different systems are overlapped as much as possible.

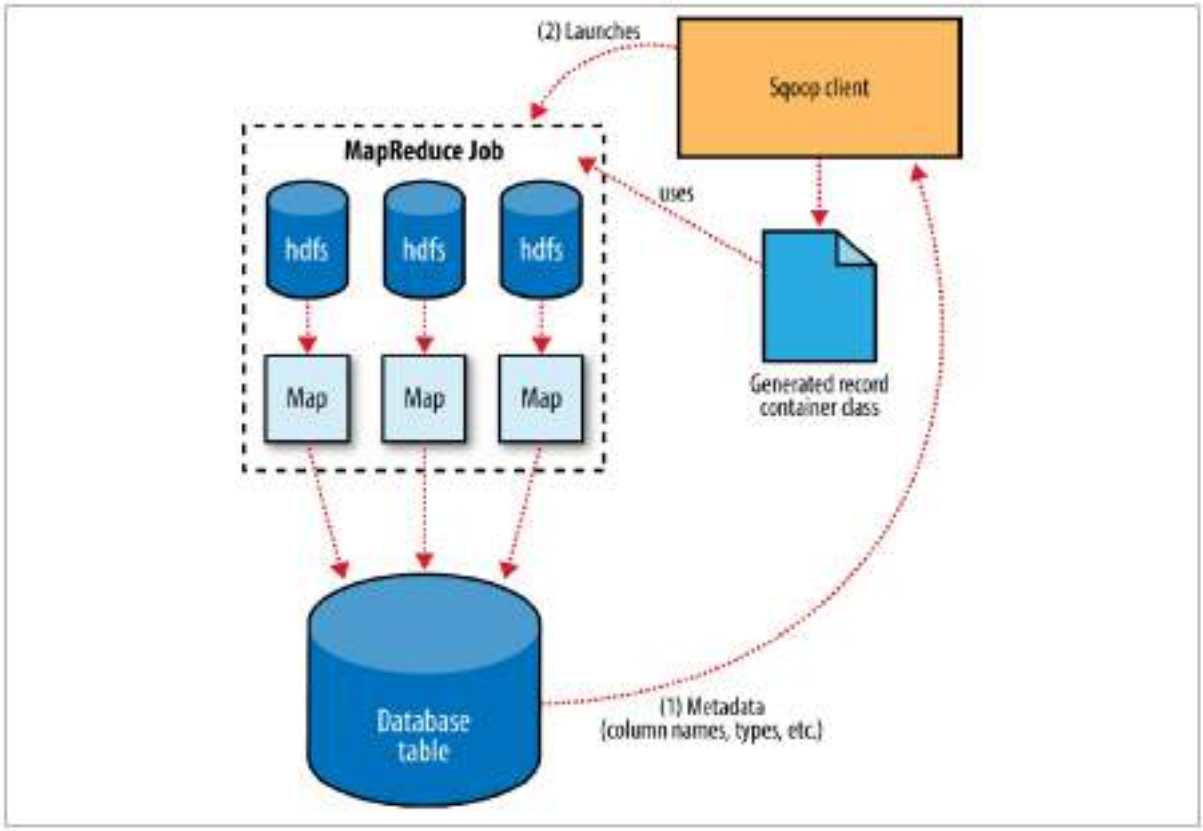


Figure 15-4. Exports are performed in parallel using MapReduce

For MySQL, Sqoop can employ a direct-mode strategy using mysqlimport. Each map task spawns a mysqlimport process that it communicates with via a named FIFO file on the local filesystem. Data is then streamed into mysqlimport via the FIFO channel, and from there into the database.

Whereas most MapReduce jobs reading from HDFS pick the degree of parallelism (number of map tasks) based on the number and size of the files to process, Sqoop's export system allows users explicit control over the number of tasks. The performance of the export can be affected by the number of parallel writers to the database, so Sqoop uses the CombineFileInputFormat class to group the input files into a smaller number of map tasks.

Exports and Transactionality:

Due to the parallel nature of the process, often an export is not an atomic operation. Sqoop will spawn multiple tasks to export slices of the data in parallel. These tasks can complete at different times, meaning that even though transactions are used inside tasks, results from one task may be visible before the results of another task. Moreover, databases often use fixed-size

buffers to store transactions. As a result, one transaction cannot necessarily contain the entire set of operations performed by a task. Sqoop commits results every few thousand rows, to ensure that it does not run out of memory. These intermediate results are visible while the export continues. Applications that will use the results of an export should not be started until the export process is complete, or they may see partial results.

To solve this problem, Sqoop can export to a temporary staging table and then, at the end of the job—if the export has succeeded—move the staged data into the destination table in a single transaction. You can specify a staging table with the –stagingtable option. The staging table must already exist and have the same schema as the destination. It must also be empty, unless the --clear-staging-table option is also supplied.

Exports and Sequence Files.

The example export reads source data from a Hive table, which is stored in HDFS as a delimited text file. Sqoop can also export delimited text files that were not Hive tables. For example, it can export text files that are the output of a MapReduce job. Sqoop can export records stored in SequenceFiles to an output table too, although some restrictions apply. A SequenceFile cannot contain arbitrary record types. Sqoop's export tool will read objects from SequenceFiles and send them directly to the Output Collector, which passes the objects to the database export OutputFormat.

To work with Sqoop, the record must be stored in the "value" portion of the SequenceFile's key-value pair format and must subclass the org.apache.sqoop.lib.SqoopRecord abstract class. If you use the codegen tool to generate a SqoopRecord implementation for a record based on your export target table, you can write a MapReduce program that populates instances of this class and writes them to SequenceFiles. sqoop-export can then export these SequenceFiles to the table. Another means by which data may be in SqoopRecord instances in SequenceFiles is if data is imported from a database table to HDFS and modified in some fashion, and then the results are stored in SequenceFiles holding records of the same data type.

In this case, Sqoop should reuse the existing class definition to read data from SequenceFiles, rather than generating a new (temporary) record container class to perform the export, as is done when converting text-based records to database rows. You can suppress code generation and instead use an existing record class and JAR by providing the --class-name and --jar-file arguments to Sqoop. Sqoop will use the specified class, loaded from the specified JAR, when exporting records.

In the following example, we reimport the widgets table as SequenceFiles, and then export it back to the database in a different table:

```
% sqoop import  \
--connect jdbc:mysql://localhost/hadoopguide \
--table widgets \
--m 1 \
--class-name WidgetHolder \
--as-sequencefile \
--target-dir widget_sequence_files
--bindir .
...

14/10/29 12:25:03 INFO mapreduce.ImportJobBase: Retrieved 3 records.

% mysql hadoopguide

mysql> CREATE TABLE widgets2(id INT, widget_name VARCHAR(100), price DOUBLE, designed DATE, version INT, notes VARCHAR(200));
Query OK, 0 rows affected (0.03 sec)

mysql> exit;

% sqoop export --connect jdbc:mysql://localhost/hadoopguide \
> --table widgets2 -m 1 --class-name WidgetHolder \
> --jar-file WidgetHolder.jar --export-dir widget_sequence_files
...

14/10/29 12:28:17 INFO mapreduce.ExportJobBase: Exported 3 records.
```

During the import, we specified the SequenceFile format and indicated that we wanted the JAR file to be placed in the current directory (with --bindir) so we can reuse it. Otherwise, it would be placed in a temporary directory. We then created a destination table for the export, which had a slightly different schema. Finally, we ran an export that used the existing generated code to read the records from the SequenceFile and write them to the database.