Sqoop User Guide (v1.2.0-cdh3u0)

Table of Contents

1. Introduction

Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle into the Hadoop Distributed File System (HDFS), transform the data in Hadoop MapReduce, and then export the data back into an RDBMS.

Sqoop automates most of this process, relying on the database to describe the schema for the data to be imported. Sqoop uses MapReduce to import and export the data, which provides parallel operation as well as fault tolerance.

This document describes how to get started using Sqoop to move data between databases and Hadoop and provides reference information for the operation of the Sqoop command-line tool suite. This document is intended for:

- System and application programmers
- System administrators
- Database administrators
- Data analysts
- Data engineers

2. Supported Releases

This documentation applies to Sqoop v1.2.0-cdh3u0.

3. Sqoop Releases

Sqoop is an open source software product of Cloudera, Inc.

Software development for Sqoop occurs at http://github.com/cloudera/sqoop. At that site you can obtain:

- New releases of Sqoop as well as its most recent source code
- An issue tracker
- A wiki that contains Sqoop documentation

Sqoop is compatible with Apache Hadoop 0.21 and Cloudera's Distribution of Hadoop version 3.

4. Prerequisites

The following prerequisite knowledge is required for this product:

- Basic computer technology and terminology
- Familiarity with command-line interfaces such as bash
- Relational database management systems
- Basic familiarity with the purpose and operation of Hadoop

Before you can use Sqoop, a release of Hadoop must be installed and configured. We recommend that you download Cloudera's Distribution for Hadoop (CDH3) from the Cloudera Software Archive at http://archive.cloudera.com for straightforward installation of Hadoop on Linux systems.

This document assumes you are using a Linux or Linux-like environment. If you are using Windows, you may be able to use cygwin to accomplish most of the following tasks. If you are using Mac OS X, you should see few (if any) compatibility errors. Sqoop is predominantly operated and tested on Linux.

5. Basic Usage

With Sqoop, you can *import* data from a relational database system into HDFS. The input to the import process is a database table. Sqoop will read the table row-by-row into HDFS. The output of this import process is a set of files containing a copy of the imported table. The import process is performed in parallel. For this reason, the output will be in multiple files. These files may be delimited text files (for example, with commas or tabs separating each field), or binary SequenceFiles containing serialized record data.

A by-product of the import process is a generated Java class which can encapsulate one row of the imported table.

This class is used during the import process by Sqoop itself. The Java source code for this class is also provided to you, for use in subsequent MapReduce processing of the data. This class can serialize and deserialize data to and from the SequenceFile format. It can also parse the delimited-text form of a record. These abilities allow you to quickly develop MapReduce applications that use the HDFS-stored records in your processing pipeline. You are also free to parse the delimiteds record data yourself, using any other tools you prefer.

After manipulating the imported records (for example, with MapReduce or Hive) you may have a result data set which you can then *export* back to the relational database. Sqoop's export process will read a set of delimited text files from HDFS in parallel, parse them into records, and insert them as new rows in a target database table, for consumption by external applications or users.

Sqoop includes some other commands which allow you to inspect the database you are working with. For example, you can list the available database schemas (with the sqoop-list-databases tool) and tables within a schema (with the sqoop-list-tables tool). Sqoop also includes a primitive SQL execution shell (the sqoop-eval tool).

Most aspects of the import, code generation, and export processes can be customized. You can control the specific row range or columns imported. You can specify particular delimiters and escape characters for the file-based representation of the data, as well as the file format used. You can also control the class or package names used in generated code. Subsequent sections of this document explain how to specify these and other arguments to Sqoop.

6. Sqoop Tools

Sqoop is a collection of related tools. To use Sqoop, you specify the tool you want to use and the arguments that control the tool.

If Sqoop is compiled from its own source, you can run Sqoop without a formal installation process by running the bin/sqoop program. Users of a packaged deployment of Sqoop (such as an RPM shipped with Cloudera's Distribution for Hadoop) will see this program installed as /usr/bin/sqoop. The remainder of this documentation will refer to this program as sqoop. For example:

```
$ sqoop tool-name [tool-arguments]
```

Note

The following examples that begin with a $ character indicate that the commands must be entered at a terminal prompt (such as bash). The $ character represents the prompt itself; you should not start these commands by typing a $. You can also enter commands inline in the text of a paragraph; for example, sqoop help. These examples do not show a $ prefix, but you should enter them the same way. Don't confuse the $ shell prompt in the examples with the $ that precedes an environment variable name. For example, the string literal $HADOOP_HOME includes a "$".

Sqoop ships with a help tool. To display a list of all available tools, type the following command:

```
$ 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
  list-databases     List available databases on a server
  list-tables        List available tables in a database
  version            Display version information

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

You can display help for a specific tool by entering: sqoop help (tool-name); for example, sqoop help import.

You can also add the --help argument to any command: sqoop import --help.

6.1. Using Command Aliases

In addition to typing the sqoop (toolname) syntax, you can use alias scripts that specify the sqoop-(toolname) syntax. For example, the scripts sqoop-import, sqoop-export, etc. each select a specific tool.

6.2. Controlling the Hadoop Installation

You invoke Sqoop through the program launch capability provided by Hadoop. The sqoop command-line program is a wrapper which runs the bin/hadoop script shipped with Hadoop. If you have multiple installations of Hadoop present on your machine, you can select the Hadoop installation by setting the $HADOOP_HOME environment variable.

For example:

```
$ HADOOP_HOME=/path/to/some/hadoop sqoop import --arguments...
```

or:

```
$ export HADOOP_HOME=/some/path/to/hadoop
$ sqoop import --arguments...
```

If $HADOOP_HOME is not set, Sqoop will use the default installation location for Cloudera's Distribution for Hadoop, /usr/lib/hadoop.

The active Hadoop configuration is loaded from $HADOOP_HOME/conf/, unless the $HADOOP_CONF_DIR environment variable is set.

6.3. Using Generic and Specific Arguments

To control the operation of each Sqoop tool, you use generic and specific arguments.

For example:

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

Common arguments:
   --connect <jdbc-uri>      Specify JDBC connect string
   --connect-manager <jdbc-uri>      Specify connection manager class to use
   --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

[...]

Generic Hadoop command-line arguments:
```

```
(must preceed any tool-specific arguments)
Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|jobtracker:port>    specify a job tracker
-files <comma separated list of files>    specify comma separated files to be copied to the map reduce cluster
-libjars <comma separated list of jars>    specify comma separated jar files to include in the classpath.
-archives <comma separated list of archives>    specify comma separated archives to be unarchived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]
```

You must supply the generic arguments -conf, -D, and so on after the tool name but **before** any tool-specific arguments (such as --connect). Note that generic Hadoop arguments are preceeded by a single dash character (-), whereas tool-specific arguments start with two dashes (--), unless they are single character arguments such as -P.

The -conf, -D, -fs and -jt arguments control the configuration and Hadoop server settings. The -files, -libjars, and -archives arguments are not typically used with Sqoop, but they are included as part of Hadoop's internal argument-parsing system.

6.4. Using Options Files to Pass Arguments

When using Sqoop, the command line options that do not change from invocation to invocation can be put in an options file for convenience. An options file is a text file where each line identifies an option in the order that it appears otherwise on the command line. Option files allow specifying a single option on multiple lines by using the back-slash character at the end of intermediate lines. Also supported are comments within option files that begin with the hash character. Comments must be specified on a new line and may not be mixed with option text. All comments and empty lines are ignored when option files are expanded. Unless options appear as quoted strings, any leading or trailing spaces are ignored. Quoted strings if used must not extend beyond the line on which they are specified.

Option files can be specified anywhere in the command line as long as the options within them follow the otherwise prescribed rules of options ordering. For instance, regardless of where the options are loaded from, they must follow the ordering such that generic options appear first, tool specific options next, finally followed by options that are intended to be passed to child programs.

To specify an options file, simply create an options file in a convenient location and pass it to the command line via --options-file argument.

Whenever an options file is specified, it is expanded on the command line before the tool is invoked. You can specify

more than one option files within the same invocation if needed.

For example, the following Sqoop invocation for import can be specified alternatively as shown below:

```
$ sqoop import --connect jdbc:mysql://localhost/db --username foo --table TEST

$ sqoop --options-file /users/homer/work/import.txt --table TEST
```

where the options file /users/homer/work/import.txt contains the following:

```
import
--connect
jdbc:mysql://localhost/db
--username
foo
```

The options file can have empty lines and comments for readability purposes. So the above example would work exactly the same if the options file /users/homer/work/import.txt contained the following:

```
#
# Options file for Sqoop import
#

# Specifies the tool being invoked
import

# Connect parameter and value
--connect
jdbc:mysql://localhost/db

# Username parameter and value
--username
foo

#
# Remaining options should be specified in the command line.
#
```

6.5. Using Tools

The following sections will describe each tool’s operation. The tools are listed in the most likely order you will find them useful.

7. sqoop-import

7.1. Purpose

The import tool imports an individual table from an RDBMS to HDFS. Each row from a table is represented as a separate record in HDFS. Records can be stored as text files (one record per line), or in binary representation in SequenceFiles.

7.2. Syntax

```
$ sqoop import (generic-args) (import-args)
$ sqoop-import (generic-args) (import-args)
```

While the Hadoop generic arguments must precede any import arguments, you can type the import arguments in any order with respect to one another.

Note

In this document, arguments are grouped into collections organized by function. Some collections are present in several tools (for example, the "common" arguments). An extended description of their functionality is given only on the first presentation in this document.

Table 1. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

7.2.1. Connecting to a Database Server

Sqoop is designed to import tables from a database into HDFS. To do so, you must specify a *connect string* that describes how to connect to the database. The *connect string* is similar to a URL, and is communicated to Sqoop with the --connect argument. This describes the server and database to connect to; it may also specify the port. For example:

```
$ sqoop import --connect jdbc:mysql://database.example.com/employees
```

This string will connect to a MySQL database named employees on the host database.example.com. It's important that you **do not** use the URL localhost if you intend to use Sqoop with a distributed Hadoop cluster. The connect string you supply will be used on TaskTracker nodes throughout your MapReduce cluster; if you specify the literal name localhost, each node will connect to a different database (or more likely, no database at all). Instead, you should use the full hostname or IP address of the database host that can be seen by all your remote nodes.

You might need to authenticate against the database before you can access it. You can use the --username and --password or -P parameters to supply a username and a password to the database. For example:

```
$ sqoop import --connect jdbc:mysql://database.example.com/employees \
    --username aaron --password 12345
```

Warning

The --password parameter is insecure, as other users may be able to read your password from the command-line arguments via the output of programs such as ps. The **-P** argument will read a password from a console prompt, and is the preferred method of entering credentials. Credentials may still be transferred between nodes of the MapReduce cluster using insecure means.

Sqoop automatically supports several databases, including MySQL. Connect strings beginning with jdbc:mysql:// are handled automatically in Sqoop. (A full list of databases with built-in support is provided in the "Supported Databases" section. For some, you may need to install the JDBC driver yourself.)

You can use Sqoop with any other JDBC-compliant database. First, download the appropriate JDBC driver for the type of database you want to import, and install the .jar file in the $SQOOP_HOME/lib directory on your client machine. (This will be /usr/lib/sqoop/lib if you installed from an RPM or Debian package.) Each driver .jar file also has a specific driver class which defines the entry-point to the driver. For example, MySQL's Connector/J library has a driver class of com.mysql.jdbc.Driver. Refer to your database vendor-specific documentation to determine the main driver class. This class must be provided as an argument to Sqoop with --driver.

For example, to connect to a SQLServer database, first download the driver from microsoft.com and install it in your Sqoop lib path.

Then run Sqoop. For example:

```
$ sqoop import --driver com.microsoft.jdbc.sqlserver.SQLServerDriver \
```

```
--connect <connect-string> ...
```

Table 2. Import control arguments:

Argument	Description
--append	Append data to an existing dataset in HDFS
--as-sequencefile	Imports data to SequenceFiles
--as-textfile	Imports data as plain text (default)
--columns <col,col,col...>	Columns to import from table
--direct	Use direct import fast path
--direct-split-size <n>	Split the input stream every *n* bytes when importing in direct mode
--inline-lob-limit <n>	Set the maximum size for an inline LOB
-m,--num-mappers <n>	Use *n* map tasks to import in parallel
-e,--query <statement>	Import the results of statement.
--split-by <column-name>	Column of the table used to split work units
--table <table-name>	Table to read
--target-dir <dir>	HDFS destination dir
--warehouse-dir <dir>	HDFS parent for table destination
--where <where clause>	WHERE clause to use during import
-z,--compress	Enable compression
--null-string <null-string>	The string to be written for a null value for string columns
--null-non-string <null-string>	The string to be written for a null value for non-string columns

The --null-string and --null-non-string arguments are optional.\ If not specified, then the string "null" will be used.

7.2.2. Selecting the Data to Import

Sqoop typically imports data in a table-centric fashion. Use the --table argument to select the table to import. For example, --table employees. This argument can also identify a VIEW or other table-like entity in a database.

By default, all columns within a table are selected for import. Imported data is written to HDFS in its "natural order;" that is, a table containing columns A, B, and C result in an import of data such as:

```
A1,B1,C1
A2,B2,C2
...
```

You can select a subset of columns and control their ordering by using the `--columns` argument. This should include a comma-delimited list of columns to import. For example: `--columns "name,employee_id,jobtitle"`.

You can control which rows are imported by adding a SQL `WHERE` clause to the import statement. By default, Sqoop generates statements of the form `SELECT <column list> FROM <table name>`. You can append a `WHERE` clause to this with the `--where` argument. For example: `--where "id > 400"`. Only rows where the `id` column has a value greater than 400 will be imported.

7.2.3. Free-form Query Imports

Sqoop can also import the result set of an arbitrary SQL query. Instead of using the `--table`, `--columns` and `--where` arguments, you can specify a SQL statement with the `--query` argument.

When importing a free-form query, you must specify a destination directory with `--target-dir`.

If you want to import the results of a query in parallel, then each map task will need to execute a copy of the query, with results partitioned by bounding conditions inferred by Sqoop. Your query must include the token `$CONDITIONS` which each Sqoop process will replace with a unique condition expression. You must also select a splitting column with `--split-by`.

For example:

```
$ sqoop import \
  --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id) WHERE $CONDITIONS' \
  --split-by a.id --target-dir /user/foo/joinresults
```

Alternately, the query can be executed once and imported serially, by specifying a single map task with `-m 1`:

```
$ sqoop import \
  --query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id) WHERE $CONDITIONS' \
  -m 1 --target-dir /user/foo/joinresults
```

Note

The facility of using free-form query in the current version of Sqoop is limited to simple queries where there are no ambiguous projections and no `OR` conditions in the `WHERE` clause. Use of complex queries such as queries that have sub-queries or joins leading to ambiguous projections can lead to unexpected results.

7.2.4. Controlling Parallelism

Sqoop imports data in parallel from most database sources. You can specify the number of map tasks (parallel processes) to use to perform the import by using the `-m` or `--num-mappers` argument. Each of these arguments takes an integer value which corresponds to the degree of parallelism to employ. By default, four tasks are used. Some databases may see improved performance by increasing this value to 8 or 16. Do not increase the degree of parallelism greater than that available within your MapReduce cluster; tasks will run serially and will likely increase the amount of time required to perform the import. Likewise, do not increase the degree of parallism higher than that which your database can reasonably support. Connecting 100 concurrent clients to your database may increase the load on the database server to a point where performance suffers as a result.

When performing parallel imports, Sqoop needs a criterion by which it can split the workload. Sqoop uses a *splitting column* to split the workload. By default, Sqoop will identify the primary key column (if present) in a table and use it as the splitting column. The low and high values for the splitting column are retrieved from the database, and the map tasks operate on evenly-sized components of the total range. For example, if you had a table with a primary key column of `id` whose minimum value was 0 and maximum value was 1000, and Sqoop was directed to use 4 tasks, Sqoop would run four processes which each execute SQL statements of the form `SELECT * FROM sometable WHERE id >= lo AND id < hi`, with `(lo, hi)` set to (0, 250), (250, 500), (500, 750), and (750, 1001) in the different tasks.

If the actual values for the primary key are not uniformly distributed across its range, then this can result in unbalanced tasks. You should explicitly choose a different column with the `--split-by` argument. For example, `--split-by employee_id`. Sqoop cannot currently split on multi-column indices. If your table has no index column, or has a multi-column key, then you must also manually choose a splitting column.

7.2.5. Controlling the Import Process

By default, the import process will use JDBC which provides a reasonable cross-vendor import channel. Some databases can perform imports in a more high-performance fashion by using database-specific data movement tools. For example, MySQL provides the `mysqldump` tool which can export data from MySQL to other systems very quickly. By

supplying the --direct argument, you are specifying that Sqoop should attempt the direct import channel. This channel may be higher performance than using JDBC. Currently, direct mode does not support imports of large object columns.

When importing from PostgreSQL in conjunction with direct mode, you can split the import into separate files after individual files reach a certain size. This size limit is controlled with the --direct-split-size argument.

By default, Sqoop will import a table named foo to a directory named foo inside your home directory in HDFS. For example, if your username is someuser, then the import tool will write to /user/someuser/foo/(files). You can adjust the parent directory of the import with the --warehouse-dir argument. For example:

```
$ sqoop import --connnect <connect-str> --table foo --warehouse-dir /shared \
    ...
```

This command would write to a set of files in the /shared/foo/ directory.

You can also explicitly choose the target directory, like so:

```
$ sqoop import --connnect <connect-str> --table foo --target-dir /dest \
    ...
```

This will import the files into the /dest directory. --target-dir is incompatible with --warehouse-dir.

When using direct mode, you can specify additional arguments which should be passed to the underlying tool. If the argument -- is given on the command-line, then subsequent arguments are sent directly to the underlying tool. For example, the following adjusts the character set used by mysqldump:

```
$ sqoop import --connect jdbc:mysql://server.foo.com/db --table bar \
    --direct -- --default-character-set=latin1
```

By default, imports go to a new target location. If the destination directory already exists in HDFS, Sqoop will refuse to import and overwrite that directory's contents. If you use the --append argument, Sqoop will import data to a temporary directory and then rename the files into the normal target directory in a manner that does not conflict with existing filenames in that directory.

> **Note**
> When using the direct mode of import, certain database client utilities are expected to be present in the shell path of the task process. For MySQL the utilities mysqldump and mysqlimport are required, whereas for PostgreSQL the utility psql is required.

7.2.6. Incremental Imports

Sqoop provides an incremental import mode which can be used to retrieve only rows newer than some previously-imported set of rows.

The following arguments control incremental imports:

Table 3. Incremental import arguments:

Argument	Description
--check-column (col)	Specifies the column to be examined when determining which rows to import.
--incremental (mode)	Specifies how Sqoop determines which rows are new. Legal values for mode include append and lastmodified.
--last-value (value)	Specifies the maximum value of the check column from the previous import.

Sqoop supports two types of incremental imports: append and lastmodified. You can use the --incremental argument to specify the type of incremental import to perform.

You should specify append mode when importing a table where new rows are continually being added with increasing row id values. You specify the column containing the row's id with --check-column. Sqoop imports rows where the check column has a value greater than the one specified with --last-value.

An alternate table update strategy supported by Sqoop is called lastmodified mode. You should use this when rows of the source table may be updated, and each such update will set the value of a last-modified column to the current timestamp. Rows where the check column holds a timestamp more recent than the timestamp specified with --last-value are imported.

At the end of an incremental import, the value which should be specified as --last-value for a subsequent import is printed to the screen. When running a subsequent import, you should specify --last-value in this way to ensure you

import only the new or updated data. This is handled automatically by creating an incremental import as a saved job, which is the preferred mechanism for performing a recurring incremental import. See the section on saved jobs later in this document for more information.

7.2.7. File Formats

You can import data in one of two file formats: delimited text or SequenceFiles.

Delimited text is the default import format. You can also specify it explicitly by using the `--as-textfile` argument. This argument will write string-based representations of each record to the output files, with delimiter characters between individual columns and rows. These delimiters may be commas, tabs, or other characters. (The delimiters can be selected; see "Output line formatting arguments.") The following is the results of an example text-based import:

```
1,here is a message,2010-05-01
2,happy new year!,2010-01-01
3,another message,2009-11-12
```

Delimited text is appropriate for most non-binary data types. It also readily supports further manipulation by other tools, such as Hive.

SequenceFiles are a binary format that store individual records in custom record-specific data types. These data types are manifested as Java classes. Sqoop will automatically generate these data types for you. This format supports exact storage of all data in binary representations, and is appropriate for storing binary data (for example, `VARBINARY` columns), or data that will be principly manipulated by custom MapReduce programs (reading from SequenceFiles is higher-performance than reading from text files, as records do not need to be parsed).

By default, data is not compressed. You can compress your data by using the deflate (gzip) algorithm with the `-z` or `--compress` argument. This applies to both SequenceFiles or text files.

7.2.8. Large Objects

Sqoop handles large objects (`BLOB` and `CLOB` columns) in particular ways. If this data is truly large, then these columns should not be fully materialized in memory for manipulation, as most columns are. Instead, their data is handled in a streaming fashion. Large objects can be stored inline with the rest of the data, in which case they are fully materialized in memory on every access, or they can be stored in a secondary storage file linked to the primary data storage. By default, large objects less than 16 MB in size are stored inline with the rest of the data. At a larger size, they are stored

in files in the _lobs subdirectory of the import target directory. These files are stored in a separate format optimized for large record storage, which can accomodate records of up to 2^63 bytes each. The size at which lobs spill into separate files is controlled by the --inline-lob-limit argument, which takes a parameter specifying the largest lob size to keep inline, in bytes. If you set the inline LOB limit to 0, all large objects will be placed in external storage.

Table 4. Output line formatting arguments:

Argument	Description
--enclosed-by <char>	Sets a required field enclosing character
--escaped-by <char>	Sets the escape character
--fields-terminated-by <char>	Sets the field separator character
--lines-terminated-by <char>	Sets the end-of-line character
--mysql-delimiters	Uses MySQL's default delimiter set: fields: , lines: \n escaped-by: \ optionally-enclosed-by: '
--optionally-enclosed-by <char>	Sets a field enclosing character

When importing to delimited files, the choice of delimiter is important. Delimiters which appear inside string-based fields may cause ambiguous parsing of the imported data by subsequent analysis passes. For example, the string "Hello, pleased to meet you" should not be imported with the end-of-field delimiter set to a comma.

Delimiters may be specified as:

- a character (--fields-terminated-by X)
- an escape character (--fields-terminated-by \t). Supported escape characters are:
    - \b (backspace)
    - \n (newline)
    - \r (carriage return)
    - \t (tab)
    - \" (double-quote)
    - \\' (single-quote)
    - \\ (backslash)
    - \0 (NUL) - This will insert NUL characters between fields or lines, or will disable enclosing/escaping if used for one of the --enclosed-by, --optionally-enclosed-by, or --escaped-by arguments.

- The octal representation of a UTF-8 character's code point. This should be of the form `\0ooo`, where *ooo* is the octal value. For example, `--fields-terminated-by \001` would yield the `^A` character.
- The hexadecimal representation of a UTF-8 character's code point. This should be of the form `\0xhhh`, where *hhh* is the hex value. For example, `--fields-terminated-by \0x10` would yield the carriage return character.

The default delimiters are a comma (`,`) for fields, a newline (`\n`) for records, no quote character, and no escape character. Note that this can lead to ambiguous/unparsible records if you import database records containing commas or newlines in the field data. For unambiguous parsing, both must be enabled. For example, via `--mysql-delimiters`.

If unambiguous delimiters cannot be presented, then use *enclosing* and *escaping* characters. The combination of (optional) enclosing and escaping characters will allow unambiguous parsing of lines. For example, suppose one column of a dataset contained the following values:

```
Some string, with a comma.
Another "string with quotes"
```

The following arguments would provide delimiters which can be unambiguously parsed:

```
$ sqoop import --fields-terminated-by , --escaped-by \\ --enclosed-by '\"' ...
```

(Note that to prevent the shell from mangling the enclosing character, we have enclosed that argument itself in single-quotes.)

The result of the above arguments applied to the above dataset would be:

```
"Some string, with a comma.","1","2","3"...
"Another \"string with quotes\"","4","5","6"...
```

Here the imported strings are shown in the context of additional columns (`"1","2","3"`, etc.) to demonstrate the full effect of enclosing and escaping. The enclosing character is only strictly necessary when delimiter characters appear in the imported text. The enclosing character can therefore be specified as optional:

```
$ sqoop import --optionally-enclosed-by '\"' (the rest as above)...
```

Which would result in the following import:

```
"Some string, with a comma.",1,2,3...
"Another \"string with quotes\"",4,5,6...
```

> **Note**
> Even though Hive supports escaping characters, it does not handle escaping of new-line character. Also, it does not support the notion of enclosing characters that may include field delimiters in the enclosed string. It is therefore recommended that you choose unambiguous field and record-terminating delimiters without the help of escaping and enclosing characters when working with Hive; this is due to limitations of Hive's input parsing abilities.

The --mysql-delimiters argument is a shorthand argument which uses the default delimiters for the mysqldump program. If you use the mysqldump delimiters in conjunction with a direct-mode import (with --direct), very fast imports can be achieved.

While the choice of delimiters is most important for a text-mode import, it is still relevant if you import to SequenceFiles with --as-sequencefile. The generated class' toString() method will use the delimiters you specify, so subsequent formatting of the output data will rely on the delimiters you choose.

Table 5. Input parsing arguments:

Argument	Description
--input-enclosed-by <char>	Sets a required field encloser
--input-escaped-by <char>	Sets the input escape character
--input-fields-terminated-by <char>	Sets the input field separator
--input-lines-terminated-by <char>	Sets the input end-of-line character
--input-optionally-enclosed-by <char>	Sets a field enclosing character

When Sqoop imports data to HDFS, it generates a Java class which can reinterpret the text files that it creates when doing a delimited-format import. The delimiters are chosen with arguments such as --fields-terminated-by; this controls both how the data is written to disk, and how the generated parse() method reinterprets this data. The delimiters used by the parse() method can be chosen independently of the output arguments, by using --input-fields-terminated-by, and so on. This is useful, for example, to generate classes which can parse records created with one set of delimiters, and emit the records to a different set of files using a separate set of delimiters.

Table 6. Hive arguments:

Argument	Description

Argument	Description
--hive-home <dir>	Override $HIVE_HOME
--hive-import	Import tables into Hive (Uses Hive's default delimiters if none are set.)
--hive-overwrite	Overwrite existing data in the Hive table.
--hive-table <table-name>	Sets the table name to use when importing to Hive.

7.2.9. Importing Data Into Hive

Sqoop's import tool's main function is to upload your data into files in HDFS. If you have a Hive metastore associated with your HDFS cluster, Sqoop can also import the data into Hive by generating and executing a CREATE TABLE statement to define the data's layout in Hive. Importing data into Hive is as simple as adding the **--hive-import** option to your Sqoop command line.

If the Hive table already exists, you can specify the **--hive-overwrite** option to indicate that existing table in hive must be replaced. After your data is imported into HDFS or this step is omitted, Sqoop will generate a Hive script containing a CREATE TABLE operation defining your columns using Hive's types, and a LOAD DATA INPATH statement to move the data files into Hive's warehouse directory.

The script will be executed by calling the installed copy of hive on the machine where Sqoop is run. If you have multiple Hive installations, or hive is not in your $PATH, use the **--hive-home** option to identify the Hive installation directory. Sqoop will use $HIVE_HOME/bin/hive from here.

Note

This function is incompatible with --as-sequencefile.

Even though Hive supports escaping characters, it does not handle escaping of new-line character. Also, it does not support the notion of enclosing characters that may include field delimiters in the enclosed string. It is therefore recommended that you choose unambiguous field and record-terminating delimiters without the help of escaping and enclosing characters when working with Hive; this is due to limitations of Hive's input parsing abilities. If you do use --escaped-by, --enclosed-by, or --optionally-enclosed-by when importing data into Hive, Sqoop will print a warning message.

Sqoop will pass the field and record delimiters through to Hive. If you do not set any delimiters and do use --hive-import, the field delimiter will be set to ^A and the record delimiter will be set to \n to be consistent with Hive's defaults.

The table name used in Hive is, by default, the same as that of the source table. You can control the output table name with the --hive-table option.

Table 7. HBase arguments:

Argument	Description
--column-family <family>	Sets the target column family for the import
--hbase-create-table	If specified, create missing HBase tables
--hbase-row-key <col>	Specifies which input column to use as the row key
--hbase-table <table-name>	Specifies an HBase table to use as the target instead of HDFS

7.2.10. Importing Data Into HBase

Sqoop supports additional import targets beyond HDFS and Hive. Sqoop can also import records into a table in HBase.

By specifying --hbase-table, you instruct Sqoop to import to a table in HBase rather than a directory in HDFS. Sqoop will import data to the table specified as the argument to --hbase-table. Each row of the input table will be transformed into an HBase Put operation to a row of the output table. The key for each row is taken from a column of the input. By default Sqoop will use the split-by column as the row key column. If that is not specified, it will try to identify the primary key column, if any, of the source table. You can manually specify the row key column with --hbase-row-key. Each output column will be placed in the same column family, which must be specified with --column-family.

If the target table and column family do not exist, the Sqoop job will exit with an error. You should create the target table and column family before running an import. If you specify --hbase-create-table, Sqoop will create the target table and column family if they do not exist, using the default parameters from your HBase configuration.

Sqoop currently serializes all values to HBase by converting each field to its string representation (as if you were importing to HDFS in text mode), and then inserts the UTF-8 bytes of this string in the target cell.

Table 8. Code generation arguments:

Argument	Description
--bindir <dir>	Output directory for compiled objects
--class-name <name>	Sets the generated class name. This overrides --package-name. When combined with --jar-file, sets the input class.

Argument	Description
--jar-file <file>	Disable code generation; use specified jar
--outdir <dir>	Output directory for generated code
--package-name <name>	Put auto-generated classes in this package

As mentioned earlier, a byproduct of importing a table to HDFS is a class which can manipulate the imported data. If the data is stored in SequenceFiles, this class will be used for the data's serialization container. Therefore, you should use this class in your subsequent MapReduce processing of the data.

The class is typically named after the table; a table named foo will generate a class named foo. You may want to override this class name. For example, if your table is named EMPLOYEES, you may want to specify --class-name Employee instead. Similarly, you can specify just the package name with --package-name. The following import generates a class named com.foocorp.SomeTable:

```
$ sqoop import --connect <connect-str> --table SomeTable --package-name com.foocorp
```

The .java source file for your class will be written to the current working directory when you run sqoop. You can control the output directory with --outdir. For example, --outdir src/generated/.

The import process compiles the source into .class and .jar files; these are ordinarily stored under /tmp. You can select an alternate target directory with --bindir. For example, --bindir /scratch.

If you already have a compiled class that can be used to perform the import and want to suppress the code-generation aspect of the import process, you can use an existing jar and class by providing the --jar-file and --class-name options. For example:

```
$ sqoop import --table SomeTable --jar-file mydatatypes.jar \
    --class-name SomeTableType
```

This command will load the SomeTableType class out of mydatatypes.jar.

7.3. Example Invocations

The following examples illustrate how to use the import tool in a variety of situations.

A basic import of a table named EMPLOYEES in the corp database:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES
```

A basic import requiring a login:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --username SomeUser -P
Enter password: (hidden)
```

Selecting specific columns from the EMPLOYEES table:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --columns "employee_id,first_name,last_name,job_title"
```

Controlling the import parallelism (using 8 parallel tasks):

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    -m 8
```

Enabling the MySQL "direct mode" fast path:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --direct
```

Storing data in SequenceFiles, and setting the generated class name to com.foocorp.Employee:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --class-name com.foocorp.Employee --as-sequencefile
```

Specifying the delimiters to use in a text-mode import:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --fields-terminated-by '\t' --lines-terminated-by '\n' \
    --optionally-enclosed-by '\"'
```

Importing the data to Hive:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
```

```
    --hive-import
```

Importing only new employees:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --where "start_date > '2010-01-01'"
```

Changing the splitting column from the default:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --split-by dept_id
```

Verifying that an import was successful:

```
$ hadoop fs -ls EMPLOYEES
Found 5 items
drwxr-xr-x   - someuser somegrp          0 2010-04-27 16:40 /user/someuser/EMPLOYEES/_logs
-rw-r--r--   1 someuser somegrp    2913511 2010-04-27 16:40 /user/someuser/EMPLOYEES/part-m-00000
-rw-r--r--   1 someuser somegrp    1683938 2010-04-27 16:40 /user/someuser/EMPLOYEES/part-m-00001
-rw-r--r--   1 someuser somegrp    7245839 2010-04-27 16:40 /user/someuser/EMPLOYEES/part-m-00002
-rw-r--r--   1 someuser somegrp    7842523 2010-04-27 16:40 /user/someuser/EMPLOYEES/part-m-00003

$ hadoop fs -cat EMPLOYEES/part-m-00000 | head -n 10
0,joe,smith,engineering
1,jane,doe,marketing
...
```

Performing an incremental import of new data, after having already imported the first 100,000 rows of a table:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/somedb --table sometable \
    --where "id > 100000" --target-dir /incremental_dataset --append
```

8. sqoop-import-all-tables

8.1. Purpose

The import-all-tables tool imports a set of tables from an RDBMS to HDFS. Data from each table is stored in a separate

directory in HDFS.

For the import-all-tables tool to be useful, the following conditions must be met:

- Each table must have a single-column primary key.
- You must intend to import all columns of each table.
- You must not intend to use non-default splitting column, nor impose any conditions via a WHERE clause.

8.2. Syntax

```
$ sqoop import-all-tables (generic-args) (import-args)
$ sqoop-import-all-tables (generic-args) (import-args)
```

Although the Hadoop generic arguments must preceed any import arguments, the import arguments can be entered in any order with respect to one another.

Table 9. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

Table 10. Import control arguments:

Argument	Description
--as-sequencefile	Imports data to SequenceFiles
--as-textfile	Imports data as plain text (default)

Argument	Description
--direct	Use direct import fast path
--direct-split-size <n>	Split the input stream every *n* bytes when importing in direct mode
--inline-lob-limit <n>	Set the maximum size for an inline LOB
-m,--num-mappers <n>	Use *n* map tasks to import in parallel
--warehouse-dir <dir>	HDFS parent for table destination
-z,--compress	Enable compression

These arguments behave in the same manner as they do when used for the sqoop-import tool, but the --table, --split-by, --columns, and --where arguments are invalid for sqoop-import-all-tables.

Table 11. Output line formatting arguments:

Argument	Description
--enclosed-by <char>	Sets a required field enclosing character
--escaped-by <char>	Sets the escape character
--fields-terminated-by <char>	Sets the field separator character
--lines-terminated-by <char>	Sets the end-of-line character
--mysql-delimiters	Uses MySQL's default delimiter set: fields: , lines: \n escaped-by: \ optionally-enclosed-by: '
--optionally-enclosed-by <char>	Sets a field enclosing character

Table 12. Input parsing arguments:

Argument	Description
--input-enclosed-by <char>	Sets a required field encloser
--input-escaped-by <char>	Sets the input escape character
--input-fields-terminated-by <char>	Sets the input field separator
--input-lines-terminated-by <char>	Sets the input end-of-line character
--input-optionally-enclosed-by <char>	Sets a field enclosing character

Table 13. Hive arguments:

Argument	Description
--hive-home <dir>	Override $HIVE_HOME
--hive-import	Import tables into Hive (Uses Hive's default delimiters if none are set.)
--hive-overwrite	Overwrite existing data in the Hive table.
--hive-table <table-name>	Sets the table name to use when importing to Hive.

Table 14. Code generation arguments:

Argument	Description
--bindir <dir>	Output directory for compiled objects
--jar-file <file>	Disable code generation; use specified jar
--outdir <dir>	Output directory for generated code
--package-name <name>	Put auto-generated classes in this package

The import-all-tables tool does not support the --class-name argument. You may, however, specify a package with --package-name in which all generated classes will be placed.

8.3. Example Invocations

Import all tables from the corp database:

```
$ sqoop import-all-tables --connect jdbc:mysql://db.foo.com/corp
```

Verifying that it worked:

```
$ hadoop fs -ls
Found 4 items
drwxr-xr-x   - someuser somegrp       0 2010-04-27 17:15 /user/someuser/EMPLOYEES
drwxr-xr-x   - someuser somegrp       0 2010-04-27 17:15 /user/someuser/PAYCHECKS
drwxr-xr-x   - someuser somegrp       0 2010-04-27 17:15 /user/someuser/DEPARTMENTS
drwxr-xr-x   - someuser somegrp       0 2010-04-27 17:15 /user/someuser/OFFICE_SUPPLIES
```

9. sqoop-export

9.1. Purpose

The export tool exports a set of files from HDFS back to an RDBMS. The target table must already exist in the database. The input files are read and parsed into a set of records according to the user-specified delimiters.

The default operation is to transform these into a set of INSERT statements that inject the records into the database. In "update mode," Sqoop will generate UPDATE statements that replace existing records in the database.

9.2. Syntax

```
$ sqoop export (generic-args) (export-args)
$ sqoop-export (generic-args) (export-args)
```

Although the Hadoop generic arguments must preceed any export arguments, the export arguments can be entered in any order with respect to one another.

Table 15. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

Table 16. Export control arguments:

Argument	Description
--direct	Use direct export fast path
--export-dir <dir>	HDFS source path for the export
-m,--num-mappers <n>	Use *n* map tasks to export in parallel
--table <table-name>	Table to populate
--update-key <col-name>	Anchor column to use for updates
--input-null-string <null-string>	The string to be interpreted as null for string columns
--input-null-non-string <null-string>	The string to be interpreted as null for non-string columns
--staging-table <staging-table-name>	The table in which data will be staged before being inserted into the destination table.
--clear-staging-table	Indicates that any data present in the staging table can be deleted.

The --table and --export-dir arguments are required. These specify the table to populate in the database, and the directory in HDFS that contains the source data.

You can control the number of mappers independently from the number of files present in the directory. Export performance depends on the degree of parallelism. By default, Sqoop will use four tasks in parallel for the export process. This may not be optimal; you will need to experiment with your own particular setup. Additional tasks may offer better concurrency, but if the database is already bottlenecked on updating indices, invoking triggers, and so on, then additional load may decrease performance. The --num-mappers or -m arguments control the number of map tasks, which is the degree of parallelism used.

MySQL provides a direct mode for exports as well, using the mysqlimport tool. When exporting to MySQL, use the --direct argument to specify this codepath. This may be higher-performance than the standard JDBC codepath.

Note

When using export in direct mode with MySQL, the MySQL bulk utility mysqlimport must be available in the shell path of the task process.

The --input-null-string and --input-null-non-string arguments are optional. If --input-null-string is not specified, then the string

"null" will be interpreted as null for string-type columns. If --input-null-non-string is not specified, then both the string "null" and the empty string will be interpreted as null for non-string columns. Note that, the empty string will be always interpreted as null for non-string columns, in addition to other string if specified by --input-null-non-string.

Since Sqoop breaks down export process into multiple transactions, it is possible that a failed export job may result in partial data being committed to the database. This can further lead to subsequent jobs failing due to insert collisions in some cases, or lead to duplicated data in others. You can overcome this problem by specifying a staging table via the --staging-table option which acts as an auxiliary table that is used to stage exported data. The staged data is finally moved to the destination table in a single transaction.

In order to use the staging facility, you must create the staging table prior to running the export job. This table must be structurally identical to the target table. This table should either be empty before the export job runs, or the --clear-staging-table option must be specified. If the staging table contains data and the --clear-staging-table option is specified, Sqoop will delete all of the data before starting the export job.

Note

Support for staging data prior to pushing it into the destination table is not available for --direct exports. It is also not available when export is invoked using the --update-key option for updating existing data.

9.3. Inserts vs. Updates

By default, sqoop-export appends new rows to a table; each input record is transformed into an INSERT statement that adds a row to the target database table. If your table has constraints (e.g., a primary key column whose values must be unique) and already contains data, you must take care to avoid inserting records that violate these constraints. The export process will fail if an INSERT statement fails. This mode is primarily intended for exporting records to a new, empty table intended to receive these results.

If you specify the --update-key argument, Sqoop will instead modify an existing dataset in the database. Each input record is treated as an UPDATE statement that modifies an existing row. The row a statement modifies is determined by the column name specified with --update-key. For example, consider the following table definition:

```
CREATE TABLE foo(
    id INT NOT NULL PRIMARY KEY,
    msg VARCHAR(32),
    bar INT);
```

Consider also a dataset in HDFS containing records like these:

```
0,this is a test,42
1,some more data,100
...
```

Running `sqoop-export --table foo --update-key id --export-dir /path/to/data --connect …` will run an export job that executes SQL statements based on the data like so:

```
UPDATE foo SET msg='this is a test', bar=42 WHERE id=0;
UPDATE foo SET msg='some more data', bar=100 WHERE id=1;
...
```

If an `UPDATE` statement modifies no rows, this is not considered an error; the export will silently continue. (In effect, this means that an update-based export will not insert new rows into the database.) Likewise, if the column specified with `--update-key` does not uniquely identify rows and multiple rows are updated by a single statement, this condition is also undetected.

Table 17. Input parsing arguments:

Argument	Description
--input-enclosed-by <char>	Sets a required field encloser
--input-escaped-by <char>	Sets the input escape character
--input-fields-terminated-by <char>	Sets the input field separator
--input-lines-terminated-by <char>	Sets the input end-of-line character
--input-optionally-enclosed-by <char>	Sets a field enclosing character

Table 18. Output line formatting arguments:

Argument	Description
--enclosed-by <char>	Sets a required field enclosing character
--escaped-by <char>	Sets the escape character
--fields-terminated-by <char>	Sets the field separator character
--lines-terminated-by <char>	Sets the end-of-line character
--mysql-delimiters	Uses MySQL's default delimiter set: fields: , lines: \n escaped-by: \ optionally-enclosed-by: '

Argument	Description
--optionally-enclosed-by <char>	Sets a field enclosing character

Sqoop automatically generates code to parse and interpret records of the files containing the data to be exported back to the database. If these files were created with non-default delimiters (comma-separated fields with newline-separated records), you should specify the same delimiters again so that Sqoop can parse your files.

If you specify incorrect delimiters, Sqoop will fail to find enough columns per line. This will cause export map tasks to fail by throwing ParseExceptions.

Table 19. Code generation arguments:

Argument	Description
--bindir <dir>	Output directory for compiled objects
--class-name <name>	Sets the generated class name. This overrides --package-name. When combined with --jar-file, sets the input class.
--jar-file <file>	Disable code generation; use specified jar
--outdir <dir>	Output directory for generated code
--package-name <name>	Put auto-generated classes in this package

If the records to be exported were generated as the result of a previous import, then the original generated class can be used to read the data back. Specifying --jar-file and --class-name obviate the need to specify delimiters in this case.

The use of existing generated code is incompatible with --update-key; an update-mode export requires new code generation to perform the update. You cannot use --jar-file, and must fully specify any non-default delimiters.

9.4. Exports and Transactions

Exports are performed by multiple writers in parallel. Each writer uses a separate connection to the database; these have separate transactions from one another. Sqoop uses the multi-row INSERT syntax to insert up to 100 records per statement. Every 100 statements, the current transaction within a writer task is committed, causing a commit every 10,000 rows. This ensures that transaction buffers do not grow without bound, and cause out-of-memory conditions. Therefore, an export is not an atomic process. Partial results from the export will become visible before the export is

complete.

9.5. Failed Exports

Exports may fail for a number of reasons:

- Loss of connectivity from the Hadoop cluster to the database (either due to hardware fault, or server software crashes)
- Attempting to INSERT a row which violates a consistency constraint (for example, inserting a duplicate primary key value)
- Attempting to parse an incomplete or malformed record from the HDFS source data
- Attempting to parse records using incorrect delimiters
- Capacity issues (such as insufficient RAM or disk space)

If an export map task fails due to these or other reasons, it will cause the export job to fail. The results of a failed export are undefined. Each export map task operates in a separate transaction. Furthermore, individual map tasks commit their current transaction periodically. If a task fails, the current transaction will be rolled back. Any previously-committed transactions will remain durable in the database, leading to a partially-complete export.

9.6. Example Invocations

A basic export to populate a table named bar:

```
$ sqoop export --connect jdbc:mysql://db.example.com/foo --table bar  \
    --export-dir /results/bar_data
```

This example takes the files in /results/bar_data and injects their contents in to the bar table in the foo database on db.example.com. The target table must already exist in the database. Sqoop performs a set of INSERT INTO operations, without regard for existing content. If Sqoop attempts to insert rows which violate constraints in the database (for example, a particular primary key value already exists), then the export fails.

10. Saved Jobs

Imports and exports can be repeatedly performed by issuing the same command multiple times. Especially when using the incremental import capability, this is an expected scenario.

Sqoop allows you to define *saved jobs* which make this process easier. A saved job records the configuration information required to execute a Sqoop command at a later time. The section on the sqoop-job tool describes how to create and work with saved jobs.

By default, job descriptions are saved to a private repository stored in $HOME/.sqoop/. You can configure Sqoop to instead use a shared *metastore*, which makes saved jobs available to multiple users across a shared cluster. Starting the metastore is covered by the section on the sqoop-metastore tool.

11. sqoop-job

11.1. Purpose

The job tool allows you to create and work with saved jobs. Saved jobs remember the parameters used to specify a job, so they can be re-executed by invoking the job by its handle.

If a saved job is configured to perform an incremental import, state regarding the most recently imported rows is updated in the saved job to allow the job to continually import only the newest rows.

11.2. Syntax

```
$ sqoop job (generic-args) (job-args) [-- [subtool-name] (subtool-args)]
$ sqoop-job (generic-args) (job-args) [-- [subtool-name] (subtool-args)]
```

Although the Hadoop generic arguments must preceed any job arguments, the job arguments can be entered in any order with respect to one another.

Table 20. Job management options:

Argument	Description
--create <job-id>	Define a new saved job with the specified job-id (name). A second Sqoop command-line, separated by a -- should be specified; this defines the saved job.

Argument	Description
--delete <job-id>	Delete a saved job.
--exec <job-id>	Given a job defined with --create, run the saved job.
--show <job-id>	Show the parameters for a saved job.
--list	List all saved jobs

Creating saved jobs is done with the --create action. This operation requires a -- followed by a tool name and its arguments. The tool and its arguments will form the basis of the saved job. Consider:

```
$ sqoop job --create myjob -- import --connect jdbc:mysql://example.com/db \
    --table mytable
```

This creates a job named myjob which can be executed later. The job is not run. This job is now available in the list of saved jobs:

```
$ sqoop job --list
Available jobs:
  myjob
```

We can inspect the configuration of a job with the show action:

```
 $ sqoop job --show myjob
 Job: myjob
 Tool: import
 Options:
 ----------------------------
 direct.import = false
 codegen.input.delimiters.record = 0
 hdfs.append.dir = false
 db.table = mytable
 ...
```

And if we are satisfied with it, we can run the job with exec:

```
$ sqoop job --exec myjob
10/08/19 13:08:45 INFO tool.CodeGenTool: Beginning code generation
...
```

The exec action allows you to override arguments of the saved job by supplying them after a --. For example, if the database were changed to require a username, we could specify the username and password with:

```
$ sqoop job --exec myjob -- --username someuser -P
Enter password:
...
```

Table 21. Metastore connection options:

Argument	Description
--meta-connect <jdbc-uri>	Specifies the JDBC connect string used to connect to the metastore

By default, a private metastore is instantiated in $HOME/.sqoop. If you have configured a hosted metastore with the sqoop-metastore tool, you can connect to it by specifying the --meta-connect argument. This is a JDBC connect string just like the ones used to connect to databases for import.

In conf/sqoop-site.xml, you can configure sqoop.metastore.client.autoconnect.url with this address, so you do not have to supply --meta-connect to use a remote metastore. This parameter can also be modified to move the private metastore to a location on your filesystem other than your home directory.

If you configure sqoop.metastore.client.enable.autoconnect with the value false, then you must explicitly supply --meta-connect.

Table 22. Common options:

Argument	Description
--help	Print usage instructions
--verbose	Print more information while working

11.3. Saved jobs and passwords

The Sqoop metastore is not a secure resource. Multiple users can access its contents. For this reason, Sqoop does not store passwords in the metastore. If you create a job that requires a password, you will be prompted for that password each time you execute the job.

You can enable passwords in the metastore by setting sqoop.metastore.client.record.password to true in the configuration.

11.4. Saved jobs and incremental imports

Incremental imports are performed by comparing the values in a *check column* against a reference value for the most recent import. For example, if the --incremental append argument was specified, along with --check-column id and --last-value 100, all rows with id > 100 will be imported. If an incremental import is run from the command line, the value which should be specified as --last-value in a subsequent incremental import will be printed to the screen for your reference. If an incremental import is run from a saved job, this value will be retained in the saved job. Subsequent runs of sqoop job --exec someIncrementalJob will continue to import only newer rows than those previously imported.

12. sqoop-metastore

12.1. Purpose

The metastore tool configures Sqoop to host a shared metadata repository. Multiple users and/or remote users can define and execute saved jobs (created with sqoop job) defined in this metastore.

Clients must be configured to connect to the metastore in sqoop-site.xml or with the --meta-connect argument.

12.2. Syntax

```
$ sqoop metastore (generic-args) (metastore-args)
$ sqoop-metastore (generic-args) (metastore-args)
```

Although the Hadoop generic arguments must preceed any metastore arguments, the metastore arguments can be entered in any order with respect to one another.

Table 23. Metastore management options:

Argument	Description
--shutdown	Shuts down a running metastore instance on the same machine.

Running sqoop-metastore launches a shared HSQLDB database instance on the current machine. Clients can connect to this metastore and create jobs which can be shared between users for execution.

The location of the metastore's files on disk is controlled by the sqoop.metastore.server.location property in conf/sqoop-site.xml. This should point to a directory on the local filesystem.

The metastore is available over TCP/IP. The port is controlled by the sqoop.metastore.server.port configuration parameter, and defaults to 16000.

Clients should connect to the metastore by specifying sqoop.metastore.client.autoconnect.url or --meta-connect with the value jdbc:hsqldb:hsql://<server-name>:<port>/sqoop. For example, jdbc:hsqldb:hsql://metaserver.example.com:16000/sqoop.

This metastore may be hosted on a machine within the Hadoop cluster, or elsewhere on the network.

13. sqoop-merge

13.1. Purpose

The merge tool allows you to combine two datasets where entries in one dataset should overwrite entries of an older dataset. For example, an incremental import run in last-modified mode will generate multiple datasets in HDFS where successively newer data appears in each dataset. The merge tool will "flatten" two datasets into one, taking the newest available records for each primary key.

13.2. Syntax

```
$ sqoop merge (generic-args) (merge-args)
$ sqoop-merge (generic-args) (merge-args)
```

Although the Hadoop generic arguments must preceed any merge arguments, the job arguments can be entered in any order with respect to one another.

Table 24. Merge options:

Argument	Description
--class-name <class>	Specify the name of the record-specific class to use during the merge job.
--jar-file <file>	Specify the name of the jar to load the record class from.
--merge-key <col>	Specify the name of a column to use as the merge key.
--new-data <path>	Specify the path of the newer dataset.
--onto <path>	Specify the path of the older dataset.
--target-dir <path>	Specify the target path for the output of the merge job.

The merge tool runs a MapReduce job that takes two directories as input: a newer dataset, and an older one. These are specified with --new-data and --onto respectively. The output of the MapReduce job will be placed in the directory in HDFS specified by --target-dir.

When merging the datasets, it is assumed that there is a unique primary key value in each record. The column for the primary key is specified with --merge-key. Multiple rows in the same dataset should not have the same primary key, or else data loss may occur.

To parse the dataset and extract the key column, the auto-generated class from a previous import must be used. You should specify the class name and jar file with --class-name and --jar-file. If this is not availab,e you can recreate the class using the codegen tool.

The merge tool is typically run after an incremental import with the date-last-modified mode (sqoop import --incremental lastmodified ...).

Supposing two incremental imports were performed, where some older data is in an HDFS directory named older and newer data is in an HDFS directory named newer, these could be merged like so:

```
$ sqoop merge --new-data newer --onto older --target-dir merged \
    --jar-file datatypes.jar --class-name Foo --merge-key id
```

This would run a MapReduce job where the value in the id column of each row is used to join rows; rows in the newer dataset will be used in preference to rows in the older dataset.

This can be used with both SequenceFile- and text-based incremental imports. The file types of the newer and older datasets must be the same.

14. sqoop-codegen

14.1. Purpose

The codegen tool generates Java classes which encapsulate and interpret imported records. The Java definition of a record is instantiated as part of the import process, but can also be performed separately. For example, if Java source is lost, it can be recreated. New versions of a class can be created which use different delimiters between fields, and so on.

14.2. Syntax

```
$ sqoop codegen (generic-args) (codegen-args)
$ sqoop-codegen (generic-args) (codegen-args)
```

Although the Hadoop generic arguments must preceed any codegen arguments, the codegen arguments can be entered in any order with respect to one another.

Table 25. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

Table 26. Code generation arguments:

Argument	Description
--bindir <dir>	Output directory for compiled objects
--class-name <name>	Sets the generated class name. This overrides --package-name.
--outdir <dir>	Output directory for generated code
--package-name <name>	Put auto-generated classes in this package
--table <table-name>	Name of the table to generate code for.

Table 27. Output line formatting arguments:

Argument	Description
--enclosed-by <char>	Sets a required field enclosing character
--escaped-by <char>	Sets the escape character
--fields-terminated-by <char>	Sets the field separator character
--lines-terminated-by <char>	Sets the end-of-line character
--mysql-delimiters	Uses MySQL's default delimiter set: fields: , lines: \n escaped-by: \ optionally-enclosed-by: '
--optionally-enclosed-by <char>	Sets a field enclosing character

Table 28. Input parsing arguments:

Argument	Description
--input-enclosed-by <char>	Sets a required field encloser
--input-escaped-by <char>	Sets the input escape character
--input-fields-terminated-by <char>	Sets the input field separator
--input-lines-terminated-by <char>	Sets the input end-of-line character
--input-optionally-enclosed-by <char>	Sets a field enclosing character

Table 29. Hive arguments:

Argument	Description

Argument	Description
--hive-home <dir>	Override $HIVE_HOME
--hive-import	Import tables into Hive (Uses Hive's default delimiters if none are set.)
--hive-overwrite	Overwrite existing data in the Hive table.
--hive-table <table-name>	Sets the table name to use when importing to Hive.

If Hive arguments are provided to the code generation tool, Sqoop generates a file containing the HQL statements to create a table and load data.

14.3. Example Invocations

Recreate the record interpretation code for the employees table of a corporate database:

```
$ sqoop codegen --connect jdbc:mysql://db.example.com/corp \
    --table employees
```

15. sqoop-create-hive-table

15.1. Purpose

The create-hive-table tool populates a Hive metastore with a definition for a table based on a database table previously imported to HDFS, or one planned to be imported. This effectively performs the "--hive-import" step of sqoop-import without running the preceeding import.

If data was already loaded to HDFS, you can use this tool to finish the pipeline of importing the data to Hive. You can also create Hive tables with this tool; data then can be imported and populated into the target after a preprocessing step run by the user.

15.2. Syntax

```
$ sqoop create-hive-table (generic-args) (create-hive-table-args)
$ sqoop-create-hive-table (generic-args) (create-hive-table-args)
```

Although the Hadoop generic arguments must preceed any create-hive-table arguments, the create-hive-table arguments can be entered in any order with respect to one another.

Table 30. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

Table 31. Hive arguments:

Argument	Description
--hive-home <dir>	Override $HIVE_HOME
--hive-overwrite	Overwrite existing data in the Hive table.
--hive-table <table-name>	Sets the table name to use when importing to Hive.
--table	The database table to read the definition from.

Table 32. Output line formatting arguments:

Argument	Description
--enclosed-by <char>	Sets a required field enclosing character
--escaped-by <char>	Sets the escape character

Argument	Description
--fields-terminated-by <char>	Sets the field separator character
--lines-terminated-by <char>	Sets the end-of-line character
--mysql-delimiters	Uses MySQL's default delimiter set: fields: , lines: \n escaped-by: \ optionally-enclosed-by: '
--optionally-enclosed-by <char>	Sets a field enclosing character

Do not use enclosed-by or escaped-by delimiters with output formatting arguments used to import to Hive. Hive cannot currently parse them.

15.3. Example Invocations

Define in Hive a table named emps with a definition based on a database table named employees:

```
$ sqoop create-hive-table --connect jdbc:mysql://db.example.com/corp \
    --table employees --hive-table emps
```

16. sqoop-eval

16.1. Purpose

The eval tool allows users to quickly run simple SQL queries against a database; results are printed to the console. This allows users to preview their import queries to ensure they import the data they expect.

16.2. Syntax

```
$ sqoop eval (generic-args) (eval-args)
$ sqoop-eval (generic-args) (eval-args)
```

Although the Hadoop generic arguments must preceed any eval arguments, the eval arguments can be entered in any order with respect to one another.

Table 33. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

Table 34. SQL evaluation arguments:

Argument	Description
-e,--query <statement>	Execute statement in SQL.

16.3. Example Invocations

Select ten records from the employees table:

```
$ sqoop eval --connect jdbc:mysql://db.example.com/corp \
    --query "SELECT * FROM employees LIMIT 10"
```

Insert a row into the foo table:

```
$ sqoop eval --connect jdbc:mysql://db.example.com/corp \
    -e "INSERT INTO foo VALUES(42, 'bar')"
```

17. sqoop-list-databases

17.1. Purpose

17.1. Purpose

List database schemas present on a server.

17.2. Syntax

```
$ sqoop list-databases (generic-args) (list-databases-args)
$ sqoop-list-databases (generic-args) (list-databases-args)
```

Although the Hadoop generic arguments must preceed any list-databases arguments, the list-databases arguments can be entered in any order with respect to one another.

Table 35. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

17.3. Example Invocations

List database schemas available on a MySQL server:

```
$ sqoop list-databases --connect jdbc:mysql://database.example.com/
information_schema
```

```
employees
```

Note

This only works with HSQLDB, MySQL and Oracle. When using with Oracle, it is necessary that the user connecting to the database has DBA privileges.

18. sqoop-list-tables

18.1. Purpose

List tables present in a database.

18.2. Syntax

```
$ sqoop list-tables (generic-args) (list-tables-args)
$ sqoop-list-tables (generic-args) (list-tables-args)
```

Although the Hadoop generic arguments must preceed any list-tables arguments, the list-tables arguments can be entered in any order with respect to one another.

Table 36. Common arguments

Argument	Description
--connect <jdbc-uri>	Specify JDBC connect string
--connection-manager <class-name>	Specify connection manager class to use
--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

Argument	Description
--username <username>	Set authentication username
--verbose	Print more information while working

18.3. Example Invocations

List tables available in the "corp" database:

```
$ sqoop list-tables --connect jdbc:mysql://database.example.com/corp
employees
payroll_checks
job_descriptions
office_supplies
```

19. sqoop-help

19.1. Purpose

List tools available in Sqoop and explain their usage.

19.2. Syntax

```
$ sqoop help [tool-name]
$ sqoop-help [tool-name]
```

If no tool name is provided (for example, the user runs sqoop help), then the available tools are listed. With a tool name, the usage instructions for that specific tool are presented on the console.

19.3. Example Invocations

List available tools:

```
$ 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

...

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

Display usage instructions for the import tool:

```
$ bin/sqoop help import
usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]

Common arguments:
   --connect <jdbc-uri>      Specify JDBC connect string
   --connection-manager <class-name>     Specify connection manager class to use
   --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

Import control arguments:
   --as-sequencefile         Imports data to SequenceFiles
   --as-textfile             Imports data as plain text (default)
...
```

20. sqoop-version

20.1. Purpose

Display version information for Sqoop.

20.2. Syntax

```
$ sqoop version
$ sqoop-version
```

20.3. Example Invocations

Display the version:

```
$ sqoop version
Sqoop {revnumber}
git commit id 46b3e06b79a8411320d77c984c3030db47dd1c22
Compiled by aaron@jargon on Mon May 17 13:43:22 PDT 2010
```

21. Compatibility Notes

Sqoop uses JDBC to connect to databases and adheres to published standards as much as possible. For databases which do not support standards-compliant SQL, Sqoop uses alternate codepaths to provide functionality. In general, Sqoop is believed to be compatible with a large number of databases, but it is tested with only a few.

Nonetheless, several database-specific decisions were made in the implementation of Sqoop, and some databases offer additional settings which are extensions to the standard.

This section describes the databases tested with Sqoop, any exceptions in Sqoop's handling of each database relative

to the norm, and any database-specific settings available in Sqoop.

21.1. Supported Databases

While JDBC is a compatibility layer that allows a program to access many different databases through a common API, slight differences in the SQL language spoken by each database may mean that Sqoop can't use every database out of the box, or that some databases may be used in an inefficient manner.

When you provide a connect string to Sqoop, it inspects the protocol scheme to determine appropriate vendor-specific logic to use. If Sqoop knows about a given database, it will work automatically. If not, you may need to specify the driver class to load via `--driver`. This will use a generic code path which will use standard SQL to access the database. Sqoop provides some databases with faster, non-JDBC-based access mechanisms. These can be enabled by specfying the `--direct` parameter.

Sqoop includes vendor-specific support for the following databases:

Database	version	`--direct` support?	connect string matches
HSQLDB	1.8.0+	No	jdbc:hsqldb:*//
MySQL	5.0+	Yes	jdbc:mysql://
Oracle	10.2.0+	No	jdbc:oracle:*//
PostgreSQL	8.3+	Yes (import only)	jdbc:postgresql://

Sqoop may work with older versions of the databases listed, but we have only tested it with the versions specified above.

Even if Sqoop supports a database internally, you may still need to install the database vendor's JDBC driver in your `$SQOOP_HOME/lib` path on your client. Sqoop can load classes from any jars in `$SQOOP_HOME/lib` on the client and will use them as part of any MapReduce jobs it runs; unlike older versions, you no longer need to install JDBC jars in the Hadoop library path on your servers.

21.2. MySQL

21.2.4. Direct-mode Transactions

JDBC Driver: MySQL Connector/J

MySQL v5.0 and above offers very thorough coverage by Sqoop. Sqoop has been tested with mysql-connector-java-5.1.13-bin.jar.

21.2.1. zeroDateTimeBehavior

MySQL allows values of '0000-00-00\' for DATE columns, which is a non-standard extension to SQL. When communicated via JDBC, these values are handled in one of three different ways:

- Convert to NULL.
- Throw an exception in the client.
- Round to the nearest legal date ('0001-01-01\').

You specify the behavior by using the zeroDateTimeBehavior property of the connect string. If a zeroDateTimeBehavior property is not specified, Sqoop uses the convertToNull behavior.

You can override this behavior. For example:

```
$ sqoop import --table foo \
    --connect jdbc:mysql://db.example.com/someDb?zeroDateTimeBehavior=round
```

21.2.2. UNSIGNED columns

Columns with type UNSIGNED in MySQL can hold values between 0 and 2^32 (4294967295), but the database will report the data type to Sqoop as INTEGER, which will can hold values between -2147483648 and \+2147483647. Sqoop cannot currently import UNSIGNED values above 2147483647.

21.2.3. BLOB and CLOB columns

Sqoop's direct mode does not support imports of BLOB, CLOB, or LONGVARBINARY columns. Use JDBC-based imports for these columns; do not supply the --direct argument to the import tool.

21.2.4. Direct-mode Transactions

For performance, each writer will commit the current transaction approximately every 32 MB of exported data. You can control this by specifying the following argument *before* any tool-specific arguments: `-D sqoop.mysql.export.checkpoint.bytes=size`, where *size* is a value in bytes. Set *size* to 0 to disable intermediate checkpoints, but individual files being exported will continue to be committed independently of one another.

Important

Note that any arguments to Sqoop that are of the form `-D parameter=value` are Hadoop *generic arguments* and must appear before any tool-specific arguments (for example, `--connect`, `--table`, etc).

21.3. Oracle

JDBC Driver: Oracle JDBC Thin Driver - Sqoop is compatible with `ojdbc6.jar`.

Sqoop has been tested with Oracle 10.2.0 Express Edition. Oracle is notable in its different approach to SQL from the ANSI standard, and its non-standard JDBC driver. Therefore, several features work differently.

21.3.1. Dates and Times

Oracle JDBC represents `DATE` and `TIME` SQL types as `TIMESTAMP` values. Any `DATE` columns in an Oracle database will be imported as a `TIMESTAMP` in Sqoop, and Sqoop-generated code will store these values in `java.sql.Timestamp` fields.

When exporting data back to a database, Sqoop parses text fields as `TIMESTAMP` types (with the form `yyyy-mm-dd HH:MM:SS.ffffffff`) even if you expect these fields to be formatted with the JDBC date escape format of `yyyy-mm-dd`. Dates exported to Oracle should be formatted as full timestamps.

Oracle also includes the additional date/time types `TIMESTAMP WITH TIMEZONE` and `TIMESTAMP WITH LOCAL TIMEZONE`. To support these types, the user's session timezone must be specified. By default, Sqoop will specify the timezone `"GMT"` to Oracle. You can override this setting by specifying a Hadoop property `oracle.sessionTimeZone` on the command-line when running a Sqoop job. For example:

```
$ sqoop import -D oracle.sessionTimeZone=America/Los_Angeles \
    --connect jdbc:oracle:thin:@//db.example.com/foo --table bar
```

Note that Hadoop parameters (-D ...) are *generic arguments* and must appear before the tool-specific arguments (--connect, --table, and so on).

Legal values for the session timezone string are enumerated at http://download-west.oracle.com/docs/cd/B19306_01/server.102/b14225/applocaledata.htm#i637736.

21.4. Schema Definition in Hive

Hive users will note that there is not a one-to-one mapping between SQL types and Hive types. In general, SQL types that do not have a direct mapping (for example, DATE, TIME, and TIMESTAMP) will be coerced to STRING in Hive. The NUMERIC and DECIMAL SQL types will be coerced to DOUBLE. In these cases, Sqoop will emit a warning in its log messages informing you of the loss of precision.

22. Getting Support

Some general information is available at the Sqoop wiki.

Report bugs in Sqoop to the issue tracker at https://issues.cloudera.org/browse/SQOOP.

Questions and discussion regarding the usage of Sqoop should be directed to the sqoop-user mailing list.

Before contacting either forum, run your Sqoop job with the --verbose flag to acquire as much debugging information as possible. Also report the string returned by sqoop version as well as the version of Hadoop you are running (hadoop version).

This document was built from Sqoop source available at http://github.com/cloudera/sqoop.