Sqoop User Guide (v1.4.6)

Sqoop User Guide (v1.4.6)

Table of Contents

1. Introduction

Sqoop is a tool designed to transfer data between Hadoop and relational databases or mainframes. You can use Sqoop to import data from a relational database management system (RDBMS) such as MySQL or Oracle or a mainframe 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 or mainframe to 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 v1.4.6.

3. Sqoop Releases

Sqoop is an open source software product of the Apache Software Foundation.

Software development for Sqoop occurs at http://sqoop.apache.org 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

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. Sqoop is currently supporting 4 major Hadoop releases - 0.20, 0.23, 1.0 and 2.0.

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 or a mainframe into HDFS. The input to the import process is either database table or mainframe datasets. For databases, Sqoop will read the table row-by-row into HDFS. For mainframe datasets, Sqoop will read records from each mainframe dataset into HDFS. The output of this import process is a set of files containing a copy of the imported table or datasets. 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 Avro or 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. For databases, 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 Apache Bigtop) 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
  import-mainframe   Import mainframe datasets 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 `$HADOOP_COMMON_HOME` and `$HADOOP_MAPRED_HOME` environment variables.

For example:

```
$ HADOOP_COMMON_HOME=/path/to/some/hadoop \
  HADOOP_MAPRED_HOME=/path/to/some/hadoop-mapreduce \
  sqoop import --arguments...
```

or:

```
$ export HADOOP_COMMON_HOME=/some/path/to/hadoop
$ export HADOOP_MAPRED_HOME=/some/path/to/hadoop-mapreduce
$ sqoop import --arguments...
```

If either of these variables are not set, Sqoop will fall back to `$HADOOP_HOME`. If it is not set either, Sqoop will use the default installation locations for Apache Bigtop, `/usr/lib/hadoop` and `/usr/lib/hadoop-mapreduce`, respectively.

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 <class-name>     Specify connection manager class to use
   --driver <class-name>    Manually specify JDBC driver class to use
   --hadoop-mapred-home <dir>      Override $HADOOP_MAPRED_HOME
   --help                   Print usage instructions
   --password-file          Set path for file containing authentication password
   -P                       Read password from console
   --password <password>    Set authentication password
   --username <username>    Set authentication username
   --verbose                Print more information while working
   --hadoop-home <dir>     Deprecated. Override $HADOOP_HOME

[...]

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. For example, the `-D mapred.job.name=<job_name>` can be used to set the name of the MR job that Sqoop launches, if not specified, the name defaults to the jar name for the job - which is derived from the used table name.

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 as Avro or 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

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` to supply a username to the database. Sqoop provides couple of different ways to supply a password, secure and non-secure, to the database which is detailed below.

Secure way of supplying password to the database. You should save the password in a file on the users home directory with 400 permissions and specify the path to that file using the **`--password-file`** argument, and is the preferred method of entering credentials. Sqoop will then read the password from the file and pass it to the MapReduce cluster using secure means with out exposing the password in the job configuration. The file containing the password can either be on the Local FS or HDFS. For example:

```
$ sqoop import --connect jdbc:mysql://database.example.com/employees \
    --username venkatesh --password-file ${user.home}/.password
```

Warning

Sqoop will read entire content of the password file and use it as a password. This will include any trailing white space characters such as new line characters that are added by default by most of the text editors. You need to make sure that your password file contains only characters that belongs to your password. On the command line you can use command `echo` with switch `-n` to store password without any trailing white space characters. For example to store password `secret` you would call `echo -n "secret" > password.file`.

Another way of supplying passwords is using the `-P` argument which will read a password from a console prompt.

Protecting password from preying eyes. Hadoop 2.6.0 provides an API to separate password storage from applications. This API is called the credential provided API and there is a new `credential` command line tool to manage passwords and their aliases. The passwords are stored with their aliases in a keystore that is password protected. The keystore password can be the provided to a password prompt on the command line, via an environment variable or defaulted to a software defined constant. Please check the Hadoop documentation on the usage of this facility.

Once the password is stored using the Credential Provider facility and the Hadoop configuration has been suitably updated, all applications can optionally use the alias in place of the actual password and at runtime resolve the alias for the password to use.

Since the keystore or similar technology used for storing the credential provider is shared across components, passwords for various applications, various database and other passwords can be securely stored in them and only the alias needs to be exposed in configuration files, protecting the password from being visible.

Sqoop has been enhanced to allow usage of this funcionality if it is available in the underlying Hadoop version being used. One new option has been introduced to provide the alias on the command line instead of the actual password (--password-alias). The argument value this option is the alias on the storage associated with the actual password. Example usage is as follows:

```
$ sqoop import --connect jdbc:mysql://database.example.com/employees \
    --username dbuser --password-alias mydb.password.alias
```

Similarly, if the command line option is not preferred, the alias can be saved in the file provided with --password-file option. Along with this, the Sqoop configuration parameter org.apache.sqoop.credentials.loader.class should be set to the classname that provides the alias resolution: `org.apache.sqoop.util.password.CredentialProviderPasswordLoader`

Example usage is as follows (assuming .password.alias has the alias for the real password) :

```
$ sqoop import --connect jdbc:mysql://database.example.com/employees \
    --username dbuser --password-file ${user.home}/.password-alias
```

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 is the preferred method over using the `--password` argument. Credentials may still be transferred between nodes of the MapReduce cluster using insecure means. For example:

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

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> ...
```

When connecting to a database using JDBC, you can optionally specify extra JDBC parameters via a property file using the option `--connection-param-file`. The contents of this file are parsed as standard Java properties and passed into the driver while creating a connection.

Note

The parameters specified via the optional property file are only applicable to JDBC connections. Any fastpath connectors that use connections other than JDBC will ignore these parameters.

Table 2. Validation arguments More Details

Argument	Description
`--validate`	Enable validation of data copied, supports single table copy only.
`--validator <class-name>`	Specify validator class to use.
`--validation-threshold <class-name>`	Specify validation threshold class to use.
`--validation-failurehandler <class-name>`	Specify validation failure handler class to use.

Table 3. Import control arguments:

Argument	Description
`--append`	Append data to an existing dataset in HDFS
`--as-avrodatafile`	Imports data to Avro Data Files
`--as-sequencefile`	Imports data to SequenceFiles
`--as-textfile`	Imports data as plain text (default)
`--as-parquetfile`	Imports data to Parquet Files
`--boundary-query <statement>`	Boundary query to use for creating splits
`--columns <col,col,col…>`	Columns to import from table
`--delete-target-dir`	Delete the import target directory if it exists
`--direct`	Use direct connector if exists for the database
`--fetch-size <n>`	Number of entries to read from database at once.
`--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. Cannot be used with `--autoreset-to-one-mapper` option.
`--autoreset-to-one-mapper`	Import should use one mapper if a table has no primary key and no split-by column is provided. Cannot be used with `--split-by <col>` option.
`--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
`--compression-codec <c>`	Use Hadoop codec (default gzip)
`--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.

By default sqoop will use query `select min(<split-by>), max(<split-by>) from <table name>` to find out boundaries for creating splits. In some cases this query is not the most optimal so you can specify any arbitrary query returning two numeric columns using `--boundary-query` argument.

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

If you are issuing the query wrapped with double quotes ("), you will have to use `\$CONDITIONS` instead of just `$CONDITIONS` to disallow your shell from treating it as a shell variable. For example, a double quoted query may look like: `"SELECT * FROM x WHERE a='foo' AND \$CONDITIONS"`

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.

If a table does not have a primary key defined and the `--split-by <col>` is not provided, then import will fail unless the number of mappers is explicitly set to one with the `--num-mappers 1` option or the `--autoreset-to-one-mapper` option is used. The option `--autoreset-to-one-mapper` is typically used with the import-all-tables tool to automatically handle tables without a primary key in a schema.

7.2.5. Controlling Distributed Cache

Sqoop will copy the jars in $SQOOP_HOME/lib folder to job cache every time when start a Sqoop job. When launched by Oozie this is unnecessary since Oozie use its own Sqoop share lib which keeps Sqoop dependencies in the distributed cache. Oozie will do the localization on each worker node for the Sqoop dependencies only once during the first Sqoop job and reuse the jars on worker node for subsquencial jobs. Using option `--skip-dist-cache` in Sqoop command when launched by Oozie will skip the step which Sqoop copies its dependencies to job cache and save massive I/O.

7.2.6. 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.

Details about use of direct mode with each specific RDBMS, installation requirements, available options and limitations can be found in Section 25, “Notes for specific connectors”.

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.

7.2.7. Controlling transaction isolation

By default, Sqoop uses the read committed transaction isolation in the mappers to import data. This may not be the ideal in all ETL workflows and it may desired to reduce the isolation guarantees. The `--relaxed-isolation` option can be used to instruct Sqoop to use read uncommitted isolation level.

The `read-uncommitted` isolation level is not supported on all databases (for example, Oracle), so specifying the option `--relaxed-isolation` may not be supported on all databases.

7.2.8. Controlling type mapping

Sqoop is preconfigured to map most SQL types to appropriate Java or Hive representatives. However the default mapping might not be suitable for everyone and might be overridden by `--map-column-java` (for changing mapping to Java) or `--map-column-hive` (for changing Hive mapping).

Table 4. Parameters for overriding mapping

Argument	Description
`--map-column-java <mapping>`	Override mapping from SQL to Java type for configured columns.
`--map-column-hive <mapping>`	Override mapping from SQL to Hive type for configured columns.

Sqoop is expecting comma separated list of mapping in form <name of column>=<new type>. For example:

```
$ sqoop import ... --map-column-java id=String,value=Integer
```

Sqoop will rise exception in case that some configured mapping will not be used.

7.2.9. 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 5. Incremental import arguments:

Argument	Description

`--check-column (col)`	Specifies the column to be examined when determining which rows to import. (the column should not be of type CHAR/NCHAR/VARCHAR/VARNCHAR/ LONGVARCHAR/LONGNVARCHAR)
`--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.

7.2.10. 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).

Avro data files are a compact, efficient binary format that provides interoperability with applications written in other programming languages. Avro also supports versioning, so that when, e.g., columns are added or removed from a table, previously imported data files can be processed along with new ones.

By default, data is not compressed. You can compress your data by using the deflate (gzip) algorithm with the `-z` or `--compress` argument, or specify any Hadoop compression codec using the `--compression-codec` argument. This applies to SequenceFile, text, and Avro files.

7.2.11. 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 6. 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 7. 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 8. 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.
`--create-hive-table`	If set, then the job will fail if the target hive
	table exits. By default this property is false.
`--hive-table <table-name>`	Sets the table name to use when importing to Hive.
`--hive-drop-import-delims`	Drops *\n*, *\r*, and *\01* from string fields when importing to Hive.
`--hive-delims-replacement`	Replace *\n*, *\r*, and *\01* from string fields with user defined string when importing to Hive.
`--hive-partition-key`	Name of a hive field to partition are sharded on
`--hive-partition-value <v>`	String-value that serves as partition key for this imported into hive in this job.
`--map-column-hive <map>`	Override default mapping from SQL type to Hive type for configured columns.

7.2.12. 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-avrodatafile` and `--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.

Hive will have problems using Sqoop-imported data if your database's rows contain string fields that have Hive's default row delimiters (`\n` and `\r` characters) or column delimiters (`\01` characters) present in them. You can use the `--hive-drop-import-delims` option to drop those characters on import to give Hive-compatible text data. Alternatively, you can use the `--hive-delims-replacement` option to replace those characters with a user-defined string on import to give Hive-compatible text data. These options should only be used if you use Hive's default delimiters and should not be used if different delimiters are specified.

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.

Sqoop will by default import NULL values as string `null`. Hive is however using string `\N` to denote `NULL` values and therefore predicates dealing with `NULL` (like `IS NULL`) will not work correctly. You should append parameters `--null-string` and `--null-non-string` in case of import job or `--input-null-string` and `--`

`input-null-non-string` in case of an export job if you wish to properly preserve `NULL` values. Because sqoop is using those parameters in generated code, you need to properly escape value `\N` to `\\N`:

```
$ sqoop import  ... --null-string '\\N' --null-non-string '\\N'
```

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.

Hive can put data into partitions for more efficient query performance. You can tell a Sqoop job to import data for Hive into a particular partition by specifying the `--hive-partition-key` and `--hive-partition-value` arguments. The partition value must be a string. Please see the Hive documentation for more details on partitioning.

You can import compressed tables into Hive using the `--compress` and `--compression-codec` options. One downside to compressing tables imported into Hive is that many codecs cannot be split for processing by parallel map tasks. The lzop codec, however, does support splitting. When importing tables with this codec, Sqoop will automatically index the files for splitting and configuring a new Hive table with the correct InputFormat. This feature currently requires that all partitions of a table be compressed with the lzop codec.

Table 9. 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
	In case, if input table contains composite
	key, then <col> must be in the form of a
	comma-separated list of composite key
	attributes
`--hbase-table <table-name>`	Specifies an HBase table to use as the target instead of HDFS
`--hbase-bulkload`	Enables bulk loading

7.2.13. 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`.

Note

This function is incompatible with direct import (parameter `--direct`).

If the input table has composite key, the `--hbase-row-key` must be in the form of a comma-separated list of composite key attributes. In this case, the row key for HBase row will be generated by combining values of composite key attributes using underscore as a separator. NOTE: Sqoop import for a table with composite key will work only if parameter `--hbase-row-key` has been specified.

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. Sqoop will skip all rows containing null values in all columns except the row key column.

To decrease the load on hbase, Sqoop can do bulk loading as opposed to direct writes. To use bulk loading, enable it using `--hbase-bulkload`.

Table 10. Accumulo arguments:

Argument	Description
`--accumulo-table <table-nam>`	Specifies an Accumulo table to use as the target instead of HDFS
`--accumulo-column-family <family>`	Sets the target column family for the import
`--accumulo-create-table`	If specified, create missing Accumulo tables
`--accumulo-row-key <col>`	Specifies which input column to use as the row key
`--accumulo-visibility <vis>`	(Optional) Specifies a visibility token to apply to all rows inserted into Accumulo. Default is the empty string.
`--accumulo-batch-size <size>`	(Optional) Sets the size in bytes of Accumulo's write buffer. Default is 4MB.
`--accumulo-max-latency <ms>`	(Optional) Sets the max latency in milliseconds for the Accumulo batch writer. Default is 0.
`--accumulo-zookeepers <host:port>`	Comma-separated list of Zookeeper servers used by the Accumulo instance
`--accumulo-instance <table-name>`	Name of the target Accumulo instance
`--accumulo-user <username>`	Name of the Accumulo user to import as
`--accumulo-password <password>`	Password for the Accumulo user

7.2.14. Importing Data Into Accumulo

Sqoop supports importing records into a table in Accumulo

By specifying `--accumulo-table`, you instruct Sqoop to import to a table in Accumulo rather than a directory in HDFS. Sqoop will import data to the table specified as the argument to `--accumulo-table`. Each row of the input table will be transformed into an Accumulo `Mutation` 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 `--accumulo-row-key`. Each output column will be placed in the same column family, which must be specified with `--accumulo-column-family`.

Note
This function is incompatible with direct import (parameter `--direct`), and cannot be used in the same operation as an HBase import.

If the target table does not exist, the Sqoop job will exit with an error, unless the `--accumulo-create-table` parameter is specified. Otherwise, you should create the target table before running an import.

Sqoop currently serializes all values to Accumulo 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.

By default, no visibility is applied to the resulting cells in Accumulo, so the data will be visible to any Accumulo user. Use the `--accumulo-visibility` parameter to specify a visibility token to apply to all rows in the import job.

For performance tuning, use the optional `--accumulo-buffer-size\` and `--accumulo-max-latency` parameters. See Accumulo's documentation for an explanation of the effects of these parameters.

In order to connect to an Accumulo instance, you must specify the location of a Zookeeper ensemble using the `--accumulo-zookeepers` parameter, the name of the Accumulo instance (`--accumulo-instance`), and the username and password to connect with (`--accumulo-user` and `--accumulo-password` respectively).

Table 11. 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
`--map-column-java <m>`	Override default mapping from SQL type to Java type for configured columns.

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`.

7.2.15. Additional Import Configuration Properties

There are some additional properties which can be configured by modifying `conf/sqoop-site.xml`. Properties can be specified the same as in Hadoop configuration files, for example:

```
  <property>
    <name>property.name</name>
    <value>property.value</value>
  </property>
```

They can also be specified on the command line in the generic arguments, for example:

```
sqoop import -D property.name=property.value ...
```

Table 12. Additional import configuration properties:

Argument	Description

sqoop.bigdecimal.format.string	Controls how BigDecimal columns will formatted when stored as a String. A value of true (default) will use toPlainString to store them without an exponent component (0.0000001); while a value of false will use toString which may include an exponent (1E-7)
sqoop.hbase.add.row.key	When set to false (default), Sqoop will not add the column used as a row key into the row data in HBase. When set to true, the column used as a row key will be added to the row data in HBase.

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
```

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
```

An import of a table named `EMPLOYEES` in the `corp` database that uses validation to validate the import using the table row count and number of rows copied into HDFS: More Details

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

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 or `--autoreset-to-one-mapper` option must be used.
- 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 13. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

Table 14. Import control arguments:

Argument	Description
`--as-avrodatafile`	Imports data to Avro Data Files
`--as-sequencefile`	Imports data to SequenceFiles
`--as-textfile`	Imports data as plain text (default)
`--as-parquetfile`	Imports data to Parquet Files
`--direct`	Use direct import fast path
`--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
`--compression-codec <c>`	Use Hadoop codec (default gzip)
`--exclude-tables <tables>`	Comma separated list of tables to exclude from import process
`--autoreset-to-one-mapper`	Import should use one mapper if a table with no primary key is encountered

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`. The `--exclude-tables` argument is for `+sqoop-import-all-tables` only.

Table 15. 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 16. 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 17. 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.
`--create-hive-table`	If set, then the job will fail if the target hive
	table exits. By default this property is false.
`--hive-table <table-name>`	Sets the table name to use when importing to Hive.
`--hive-drop-import-delims`	Drops *\n*, *\r*, and *\01* from string fields when importing to Hive.
`--hive-delims-replacement`	Replace *\n*, *\r*, and *\01* from string fields with user defined string when importing to Hive.

Argument	Description
`--hive-partition-key`	Name of a hive field to partition are sharded on
`--hive-partition-value <v>`	String-value that serves as partition key for this imported into hive in this job.
`--map-column-hive <map>`	Override default mapping from SQL type to Hive type for configured columns.

Table 18. 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-import-mainframe

9.1. Purpose

The `import-mainframe` tool imports all sequential datasets in a partitioned dataset(PDS) on a mainframe to HDFS. A PDS is akin to a directory on the open systems. The records in a dataset can contain only character data. Records will be stored with the entire record as a single text field.

9.2. Syntax

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

Table 19. Common arguments

Argument	Description
`--connect <hostname>`	Specify mainframe host to connect
`--connection-manager <class-name>`	Specify connection manager class to use
`--hadoop-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters

9.2.1. Connecting to a Mainframe

Sqoop is designed to import mainframe datasets into HDFS. To do so, you must specify a mainframe host name in the Sqoop `--connect` argument.

```
$ sqoop import-mainframe --connect z390
```

This will connect to the mainframe host z390 via ftp.

You might need to authenticate against the mainframe host to access it. You can use the `--username` to supply a username to the mainframe. Sqoop provides couple of different ways to supply a password, secure and non-secure, to the mainframe which is detailed below.

Secure way of supplying password to the mainframe. You should save the password in a file on the users home directory with 400 permissions and specify the path to that file using the **`--password-file`** argument, and is the preferred method of entering credentials. Sqoop will then read the password from the file and pass it to the MapReduce cluster using secure means with out exposing the password in the job configuration. The file containing the password can either be on the Local FS or HDFS.

Example:

```
$ sqoop import-mainframe --connect z390 \
    --username david --password-file ${user.home}/.password
```

Another way of supplying passwords is using the `-P` argument which will read a password from a console prompt.

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 is the preferred method over using the `--password` argument. Credentials may still be transferred between nodes of the MapReduce cluster using insecure means.

Example:

```
$ sqoop import-mainframe --connect z390 --username david --password 12345
```

Table 20. Import control arguments:

Argument	Description
`--as-avrodatafile`	Imports data to Avro Data Files
`--as-sequencefile`	Imports data to SequenceFiles
`--as-textfile`	Imports data as plain text (default)
`--as-parquetfile`	Imports data to Parquet Files
`--delete-target-dir`	Delete the import target directory if it exists
`-m,--num-mappers <n>`	Use *n* map tasks to import in parallel
`--target-dir <dir>`	HDFS destination dir
`--warehouse-dir <dir>`	HDFS parent for table destination
`-z,--compress`	Enable compression
`--compression-codec <c>`	Use Hadoop codec (default gzip)

9.2.2. Selecting the Files to Import

You can use the `--dataset` argument to specify a partitioned dataset name. All sequential datasets in the partitioned dataset will be imported.

9.2.3. Controlling Parallelism

Sqoop imports data in parallel by making multiple ftp connections to the mainframe to transfer multiple files simultaneously. 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. You can adjust this value to maximize the data transfer rate from the mainframe.

9.2.4. Controlling Distributed Cache

Sqoop will copy the jars in $SQOOP_HOME/lib folder to job cache every time when start a Sqoop job. When launched by Oozie this is unnecessary since Oozie use its own Sqoop share lib which keeps Sqoop dependencies in the distributed cache. Oozie will do the localization on each worker node for the Sqoop dependencies only once during the first Sqoop job and reuse the jars on worker node for subsquencial jobs. Using option `--skip-dist-cache` in Sqoop command when launched by Oozie will skip the step which Sqoop copies its dependencies to job cache and save massive I/O.

9.2.5. Controlling the Import Process

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

```
$ sqoop import-mainframe --connnect <host> --dataset foo --warehouse-dir /shared \
    ...
```

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

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

```
$ sqoop import-mainframe --connnect <host> --dataset foo --target-dir /dest \
    ...
```

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

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.

9.2.6. File Formats

By default, each record in a dataset is stored as a text record with a newline at the end. Each record is assumed to contain a single text field with the name DEFAULT_COLUMN. When Sqoop imports data to HDFS, it generates a Java class which can reinterpret the text files that it creates.

You can also import mainframe records to Sequence, Avro, or Parquet files.

By default, data is not compressed. You can compress your data by using the deflate (gzip) algorithm with the `-z` or `--compress` argument, or specify any Hadoop compression codec using the `--compression-codec` argument.

Table 21. 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

Since mainframe record contains only one field, importing to delimited files will not contain any field delimiter. However, the field may be enclosed with enclosing character or escaped by an escaping character.

Table 22. 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 23. 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.
`--create-hive-table`	If set, then the job will fail if the target hive
	table exits. By default this property is false.
`--hive-table <table-name>`	Sets the table name to use when importing to Hive.

Argument	Description
`--hive-drop-import-delims`	Drops *\n*, *\r*, and *\01* from string fields when importing to Hive.
`--hive-delims-replacement`	Replace *\n*, *\r*, and *\01* from string fields with user defined string when importing to Hive.
`--hive-partition-key`	Name of a hive field to partition are sharded on
`--hive-partition-value <v>`	String-value that serves as partition key for this imported into hive in this job.
`--map-column-hive <map>`	Override default mapping from SQL type to Hive type for configured columns.

9.2.7. 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-avrodatafile` and `--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.

Hive will have problems using Sqoop-imported data if your database's rows contain string fields that have Hive's default row delimiters (`\n` and `\r` characters) or column delimiters (`\01` characters) present in them. You can use the `--hive-drop-import-delims` option to drop those characters on import to give Hive-compatible text data. Alternatively, you can use the `--hive-delims-replacement` option to replace those characters with a user-defined string on import to give Hive-compatible text data. These options should only be used if you use Hive's default delimiters and should not be used if different delimiters are specified.

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.

Sqoop will by default import NULL values as string `null`. Hive is however using string `\N` to denote `NULL` values and therefore predicates dealing with `NULL` (like `IS NULL`) will not work correctly. You should append parameters `--null-string` and `--null-non-string` in case of import job or `--input-null-string` and `--input-null-non-string` in case of an export job if you wish to properly preserve `NULL` values. Because sqoop is using those parameters in generated code, you need to properly escape value `\N` to `\\N`:

```
$ sqoop import  ... --null-string '\\N' --null-non-string '\\N'
```

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.

Hive can put data into partitions for more efficient query performance. You can tell a Sqoop job to import data for Hive into a particular partition by specifying the `--hive-partition-key` and `--hive-partition-`

`value` arguments. The partition value must be a string. Please see the Hive documentation for more details on partitioning.

You can import compressed tables into Hive using the `--compress` and `--compression-codec` options. One downside to compressing tables imported into Hive is that many codecs cannot be split for processing by parallel map tasks. The lzop codec, however, does support splitting. When importing tables with this codec, Sqoop will automatically index the files for splitting and configuring a new Hive table with the correct InputFormat. This feature currently requires that all partitions of a table be compressed with the lzop codec.

Table 24. 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
	In case, if input table contains composite
	key, then <col> must be in the form of a
	comma-separated list of composite key
	attributes
`--hbase-table <table-name>`	Specifies an HBase table to use as the target instead of HDFS
`--hbase-bulkload`	Enables bulk loading

9.2.8. 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`.

Note

This function is incompatible with direct import (parameter `--direct`).

If the input table has composite key, the `--hbase-row-key` must be in the form of a comma-separated list of composite key attributes. In this case, the row key for HBase row will be generated by combining values of composite key attributes using underscore as a separator. NOTE: Sqoop import for a table with composite key will work only if parameter `--hbase-row-key` has been specified.

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. Sqoop will skip all rows containing null values in all columns except the row key column.

To decrease the load on hbase, Sqoop can do bulk loading as opposed to direct writes. To use bulk loading, enable it using `--hbase-bulkload`.

Table 25. Accumulo arguments:

Argument	Description

`--accumulo-table <table-nam>`	Specifies an Accumulo table to use as the target instead of HDFS
`--accumulo-column-family <family>`	Sets the target column family for the import
`--accumulo-create-table`	If specified, create missing Accumulo tables
`--accumulo-row-key <col>`	Specifies which input column to use as the row key
`--accumulo-visibility <vis>`	(Optional) Specifies a visibility token to apply to all rows inserted into Accumulo. Default is the empty string.
`--accumulo-batch-size <size>`	(Optional) Sets the size in bytes of Accumulo's write buffer. Default is 4MB.
`--accumulo-max-latency <ms>`	(Optional) Sets the max latency in milliseconds for the Accumulo batch writer. Default is 0.
`--accumulo-zookeepers <host:port>`	Comma-separated list of Zookeeper servers used by the Accumulo instance
`--accumulo-instance <table-name>`	Name of the target Accumulo instance
`--accumulo-user <username>`	Name of the Accumulo user to import as
`--accumulo-password <password>`	Password for the Accumulo user

9.2.9. Importing Data Into Accumulo

Sqoop supports importing records into a table in Accumulo

By specifying `--accumulo-table`, you instruct Sqoop to import to a table in Accumulo rather than a directory in HDFS. Sqoop will import data to the table specified as the argument to `--accumulo-table`. Each row of the input table will be transformed into an Accumulo `Mutation` 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 `--accumulo-row-key`. Each output column will be placed in the same column family, which must be specified with `--accumulo-column-family`.

Note

This function is incompatible with direct import (parameter `--direct`), and cannot be used in the same operation as an HBase import.

If the target table does not exist, the Sqoop job will exit with an error, unless the `--accumulo-create-table` parameter is specified. Otherwise, you should create the target table before running an import.

Sqoop currently serializes all values to Accumulo 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.

By default, no visibility is applied to the resulting cells in Accumulo, so the data will be visible to any Accumulo user. Use the `--accumulo-visibility` parameter to specify a visibility token to apply to all rows in the import job.

For performance tuning, use the optional `--accumulo-buffer-size\` and `--accumulo-max-latency` parameters. See Accumulo's documentation for an explanation of the effects of these parameters.

In order to connect to an Accumulo instance, you must specify the location of a Zookeeper ensemble using the `--accumulo-zookeepers` parameter, the name of the Accumulo instance (`--accumulo-instance`), and the username and password to connect with (`--accumulo-user` and `--accumulo-password` respectively).

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`. 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
`--map-column-java <m>`	Override default mapping from SQL type to Java type for configured columns.

As mentioned earlier, a byproduct of importing a table to HDFS is a class which can manipulate the imported data. You should use this class in your subsequent MapReduce processing of the data.

The class is typically named after the partitioned dataset name; a partitioned dataset named `foo` will generate a class named `foo`. You may want to override this class name. For example, if your partitioned dataset 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.SomePDS`:

```
$ sqoop import-mainframe --connect <host> --dataset SomePDS --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-mainframe --dataset SomePDS --jar-file mydatatypes.jar \
    --class-name SomePDSType
```

This command will load the `SomePDSType` class out of `mydatatypes.jar`.

9.2.10. Additional Import Configuration Properties

There are some additional properties which can be configured by modifying `conf/sqoop-site.xml`. Properties can be specified the same as in Hadoop configuration files, for example:

```
  <property>
    <name>property.name</name>
    <value>property.value</value>
  </property>
```

They can also be specified on the command line in the generic arguments, for example:

```
sqoop import -D property.name=property.value ...
```

9.3. Example Invocations

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

A basic import of all sequential files in a partitioned dataset named `EMPLOYEES` in the mainframe host z390:

```
$ sqoop import-mainframe --connect z390 --dataset EMPLOYEES \
    --username SomeUser -P
Enter password: (hidden)
```

Controlling the import parallelism (using 8 parallel tasks):

```
$ sqoop import-mainframe --connect z390 --dataset EMPLOYEES \
    --username SomeUser --password-file mypassword -m 8
```

Importing the data to Hive:

```
$ sqoop import-mainframe --connect z390 --dataset EMPLOYEES \
    --hive-import
```

10. sqoop-export

10.1. 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, and in "call mode" Sqoop will make a stored procedure call for each record.

10.2. 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 27. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

Table 28. Validation arguments More Details

Argument	Description
`--validate`	Enable validation of data copied, supports single table copy only.
`--validator <class-name>`	Specify validator class to use.
`--validation-threshold <class-name>`	Specify validation threshold class to use.
`--validation-failurehandler <class-name>`	Specify validation failure handler class to use.

Table 29. Export control arguments:

Argument	Description
`--columns <col,col,col…>`	Columns to export to table
`--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
`--call <stored-proc-name>`	Stored Procedure to call
`--update-key <col-name>`	Anchor column to use for updates. Use a comma separated list of columns if there are more than one column.
`--update-mode <mode>`	Specify how updates are performed when new rows are found with non-matching keys in database.
	Legal values for `mode` include `updateonly` (default) and `allowinsert`.
`--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.
`--batch`	Use batch mode for underlying statement execution.

The `--export-dir` argument and one of `--table` or `--call` are required. These specify the table to populate in the database (or the stored procedure to call), and the directory in HDFS that contains the source data.

By default, all columns within a table are selected for export. 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 export. For example: `--columns "col1,col2,col3"`. Note that columns that are not included in the `--columns` parameter need to have either defined default value or allow `NULL` values. Otherwise your database will reject the imported data which in turn will make Sqoop job fail.

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.

Some databases provides a direct mode for exports as well. Use the `--direct` argument to specify this codepath. This may be higher-performance than the standard JDBC codepath. Details about use of direct mode with each specific RDBMS, installation requirements, available options and limitations can be found in Section 25, “Notes for specific connectors”.

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 always available for `--direct` exports. It is also not available when export is invoked using the `--update-key` option for updating existing data, and when stored procedures are used to insert the data. It is best to check the Section 25, "Notes for specific connectors" section to validate.

10.3. 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(s) 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.

The argument `--update-key` can also be given a comma separated list of column names. In which case, Sqoop will match all keys from this list before updating any existing record.

Depending on the target database, you may also specify the `--update-mode` argument with `allowinsert` mode if you want to update rows if they exist in the database already or insert rows if they do not exist yet.

Table 30. 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 31. 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

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 32. 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
`--map-column-java <m>`	Override default mapping from SQL type to Java type for configured columns.

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.

10.4. 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.

10.5. 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.

10.6. 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.

Alternatively, you can specify the columns to be exported by providing `--columns "col1,col2,col3"`. Please note that columns that are not included in the `--columns` parameter need to have either defined default value or allow `NULL` values. Otherwise your database will reject the imported data which in turn will make Sqoop job fail.

Another basic export to populate a table named `bar` with validation enabled: More Details

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

An export that calls a stored procedure named `barproc` for every record in `/results/bar_data` would look like:

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

11. validation

11.1. Purpose

Validate the data copied, either import or export by comparing the row counts from the source and the target post copy.

11.2. Introduction

There are 3 basic interfaces: ValidationThreshold - Determines if the error margin between the source and target are acceptable: Absolute, Percentage Tolerant, etc. Default implementation is AbsoluteValidationThreshold which ensures the row counts from source and targets are the same.

ValidationFailureHandler - Responsible for handling failures: log an error/warning, abort, etc. Default implementation is LogOnFailureHandler that logs a warning message to the configured logger.

Validator - Drives the validation logic by delegating the decision to ValidationThreshold and delegating failure handling to ValidationFailureHandler. The default implementation is RowCountValidator which validates the row counts from source and the target.

11.3. Syntax

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

Validation arguments are part of import and export arguments.

11.4. Configuration

The validation framework is extensible and pluggable. It comes with default implementations but the interfaces can be extended to allow custom implementations by passing them as part of the command line arguments as described below.

Validator.

```
Property:         validator
Description:      Driver for validation,
                  must implement org.apache.sqoop.validation.Validator
Supported values: The value has to be a fully qualified class name.
Default value:    org.apache.sqoop.validation.RowCountValidator
```

Validation Threshold.

```
Property:         validation-threshold
Description:      Drives the decision based on the validation meeting the
                  threshold or not. Must implement
                  org.apache.sqoop.validation.ValidationThreshold
Supported values: The value has to be a fully qualified class name.
Default value:    org.apache.sqoop.validation.AbsoluteValidationThreshold
```

Validation Failure Handler.

```
Property:         validation-failurehandler
Description:      Responsible for handling failures, must implement
                  org.apache.sqoop.validation.ValidationFailureHandler
Supported values: The value has to be a fully qualified class name.
Default value:    org.apache.sqoop.validation.AbortOnFailureHandler
```

11.5. Limitations

Validation currently only validates data copied from a single table into HDFS. The following are the limitations in the current implementation:

- all-tables option
- free-form query option
- Data imported into Hive, HBase or Accumulo
- table import with --where argument
- incremental imports

11.6. Example Invocations

A basic import of a table named `EMPLOYEES` in the `corp` database that uses validation to validate the row counts:

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

A basic export to populate a table named `bar` with validation enabled:

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

Another example that overrides the validation args:

```
$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \
    --validate --validator org.apache.sqoop.validation.RowCountValidator \
    --validation-threshold \
          org.apache.sqoop.validation.AbsoluteValidationThreshold \
    --validation-failurehandler \
          org.apache.sqoop.validation.AbortOnFailureHandler
```

12. 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.

13. sqoop-job

13.1. 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.

13.2. 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 33. 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.
`--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 34. 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 35. Common options:

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

13.3. 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.

Note that you have to set `sqoop.metastore.client.record.password` to `true` if you are executing saved jobs via Oozie because Sqoop cannot prompt the user to enter passwords while being executed as Oozie tasks.

13.4. 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.

14. sqoop-metastore

14.1. 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.

14.2. 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 36. 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.

15. sqoop-merge

15.1. 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.

15.2. 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 37. 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-, Avro- and text-based incremental imports. The file types of the newer and older datasets must be the same.

16. sqoop-codegen

16.1. 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.

16.2. 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 38. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

Table 39. 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
`--map-column-java <m>`	Override default mapping from SQL type to Java type for configured columns.

Table 40. 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 41. Input parsing arguments:

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

Argument	Description
`--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 42. 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.
`--create-hive-table`	If set, then the job will fail if the target hive
	table exits. By default this property is false.
`--hive-table <table-name>`	Sets the table name to use when importing to Hive.
`--hive-drop-import-delims`	Drops *\n*, *\r*, and *\01* from string fields when importing to Hive.
`--hive-delims-replacement`	Replace *\n*, *\r*, and *\01* from string fields with user defined string when importing to Hive.
`--hive-partition-key`	Name of a hive field to partition are sharded on
`--hive-partition-value <v>`	String-value that serves as partition key for this imported into hive in this job.
`--map-column-hive <map>`	Override default mapping from SQL type to Hive type for configured columns.

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.

16.3. 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
```

17. sqoop-create-hive-table

17.1. 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.

17.2. 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 43. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

Table 44. Hive arguments:

Argument	Description
`--hive-home <dir>`	Override `$HIVE_HOME`
`--hive-overwrite`	Overwrite existing data in the Hive table.
`--create-hive-table`	If set, then the job will fail if the target hive
	table exits. By default this property is false.
`--hive-table <table-name>`	Sets the table name to use when importing to Hive.
`--table`	The database table to read the definition from.

Table 45. 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

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

17.3. 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
```

18. sqoop-eval

18.1. 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.

Warning
The `eval` tool is provided for evaluation purpose only. You can use it to verify database connection from within the Sqoop or to test simple queries. It's not suppose to be used in production workflows.

18.2. 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 46. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

Table 47. SQL evaluation arguments:

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

18.3. 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')"
```

19. sqoop-list-databases

19.1. Purpose

List database schemas present on a server.

19.2. 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 48. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

19.3. 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.

20. sqoop-list-tables

20.1. Purpose

List tables present in a database.

20.2. 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 49. 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-mapred-home <dir>`	Override $HADOOP_MAPRED_HOME
`--help`	Print usage instructions
`--password-file`	Set path for a file containing the authentication password
`-P`	Read password from console
`--password <password>`	Set authentication password
`--username <username>`	Set authentication username
`--verbose`	Print more information while working
`--connection-param-file <filename>`	Optional properties file that provides connection parameters
`--relaxed-isolation`	Set connection transaction isolation to read uncommitted for the mappers.

20.3. 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
```

In case of postgresql, list tables command with common arguments fetches only "public" schema. For custom schema, use --schema argument to list tables of particular schema Example

```
$ sqoop list-tables --connect jdbc:postgresql://localhost/corp  --username name -P -- --schema payrolldept
employees
expenses
```

21. sqoop-help

21.1. Purpose

List tools available in Sqoop and explain their usage.

21.2. 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.

21.3. 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-mapred-home <dir>            Override $HADOOP_MAPRED_HOME
   --help                   Print usage instructions
   --password-file          Set path for file containing authentication password
   -P                       Read password from console
   --password <password>    Set authentication password
   --username <username>    Set authentication username
   --verbose                Print more information while working
   --hadoop-home <dir>      Deprecated. Override $HADOOP_HOME

Import control arguments:
   --as-avrodatafile            Imports data to Avro Data Files
   --as-sequencefile            Imports data to SequenceFiles
   --as-textfile                Imports data as plain text (default)
   --as-parquetfile             Imports data to Parquet Data Files
...
```

22. sqoop-version

22.1. Purpose

Display version information for Sqoop.

22.2. Syntax

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

22.3. 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
```

23. Sqoop-HCatalog Integration

23.1. HCatalog Background

HCatalog is a table and storage management service for Hadoop that enables users with different data processing tools Pig, MapReduce, and Hive to more easily read and write data on the grid. HCatalog's table abstraction presents users with a relational view of data in the Hadoop distributed file system (HDFS) and ensures that users need not worry about where or in what format their data is stored: RCFile format, text files, or SequenceFiles.

HCatalog supports reading and writing files in any format for which a Hive SerDe (serializer-deserializer) has been written. By default, HCatalog supports RCFile, CSV, JSON, and SequenceFile formats. To use a custom format, you must provide the InputFormat and OutputFormat as well as the SerDe.

The ability of HCatalog to abstract various storage formats is used in providing the RCFile (and future file types) support to Sqoop.

23.2. Exposing HCatalog Tables to Sqoop

HCatalog integration with Sqoop is patterned on an existing feature set that supports Avro and Hive tables. Seven new command line options are introduced, and some command line options defined for Hive have been reused.

23.2.1. New Command Line Options

`--hcatalog-database`
: Specifies the database name for the HCatalog table. If not specified, the default database name `default` is used. Providing the `--hcatalog-database` option without `--hcatalog-table` is an error. This is not a required option.

`--hcatalog-table`
: The argument value for this option is the HCatalog tablename. The presence of the `--hcatalog-table` option signifies that the import or export job is done using HCatalog tables, and it is a required option for HCatalog jobs.

`--hcatalog-home`
: The home directory for the HCatalog installation. The directory is expected to have a `lib` subdirectory and a `share/hcatalog` subdirectory with necessary HCatalog libraries. If not specified, the system property `hcatalog.home` will be checked and failing that, a system environment variable `HCAT_HOME` will be checked. If none of these are set, the default value will be used and currently the default is set to `/usr/lib/hcatalog`. This is not a required option.

`--create-hcatalog-table`
: This option specifies whether an HCatalog table should be created automatically when importing data. By default, HCatalog tables are assumed to exist. The table name will be the same as the database table name translated to lower case. Further described in `Automatic Table Creation` below.

`--hcatalog-storage-stanza`
: This option specifies the storage stanza to be appended to the table. Further described in `Automatic Table Creation` below.

`--hcatalog-partition-keys` and `--hcatalog-partition-values`
: These two options are used to specify multiple static partition key/value pairs. In the prior releases, `--hive-partition-key` and `--hive-partition-value` options were used to specify the static partition key/value pair, but only one level of static partition keys could be provided. The options `--hcatalog-partition-keys` and `--hcatalog-partition-values` allow multiple keys and values to be provided as static partitioning keys. Multiple option values are to be separated by , (comma).

For example, if the hive partition keys for the table to export/import from are defined with partition key names year, month and date and a specific partition with year=1999, month=12, day=31 is the desired partition, then the values for the two options will be as follows:

- `--hcatalog-partition-keys` year,month,day
- `--hcatalog-partition-values` 1999,12,31

To provide backward compatibility, if `--hcatalog-partition-keys` or `--hcatalog-partition-values` options are not provided, then `--hive-partitition-key` and `--hive-partition-value` will be used if provided.

It is an error to specify only one of `--hcatalog-partition-keys` or `--hcatalog-partition-values` options. Either both of the options should be provided or neither of the options should be provided.

23.2.2. Supported Sqoop Hive Options

The following Sqoop options are also used along with the `--hcatalog-table` option to provide additional input to the HCatalog jobs. Some of the existing Hive import job options are reused with HCatalog jobs instead of creating HCatalog-specific options for the same purpose.

`--map-column-hive`
: This option maps a database column to HCatalog with a specific HCatalog type.

`--hive-home`
: The Hive home location.

`--hive-partition-key`
: Used for static partitioning filter. The partitioning key should be of type STRING. There can be only one static partitioning key. Please see the discussion about `--hcatalog-partition-keys` and `--hcatalog-partition-values` options.

`--hive-partition-value`
: The value associated with the partition. Please see the discussion about `--hcatalog-partition-keys` and `--hcatalog-partition-values` options.

23.2.3. Direct Mode support

HCatalog integration in Sqoop has been enhanced to support direct mode connectors (which are high performance connectors specific to a database). Netezza direct mode connector has been enhanced to take advatange of this feature.

Important
Only Netezza direct mode connector is currently enabled to work with HCatalog.

23.2.4. Unsupported Sqoop Options

23.2.4.1. Unsupported Sqoop Hive Import Options

The following Sqoop Hive import options are not supported with HCatalog jobs.

- `--hive-import`
- `--hive-overwrite`

23.2.4.2. Unsupported Sqoop Export and Import Options

The following Sqoop export and import options are not supported with HCatalog jobs.

- `--export-dir`
- `--target-dir`
- `--warehouse-dir`
- `--append`
- `--as-sequencefile`
- `--as-avrodatafile`
- `--as-parquetfile`

23.2.5. Ignored Sqoop Options

The following options are ignored with HCatalog jobs.

- All input delimiter options are ignored.
- Output delimiters are generally ignored unless either `--hive-drop-import-delims` or `--hive-delims-replacement` is used. When the `--hive-drop-import-delims` or `--hive-delims-replacement` option is specified, all `CHAR` type database table columns will be post-processed to either remove or replace the delimiters, respectively. See `Delimited Text Formats and Field and Line Delimiter Characters` below. This is only needed if the HCatalog table uses text formats.

23.3. Automatic Table Creation

One of the key features of Sqoop is to manage and create the table metadata when importing into Hadoop. HCatalog import jobs also provide for this feature with the option `--create-hcatalog-table`. Furthermore, one of the important benefits of the HCatalog integration is to provide storage agnosticism to Sqoop data movement jobs. To provide for that feature, HCatalog import jobs provide an option that lets a user specifiy the storage format for the created table.

The option `--create-hcatalog-table` is used as an indicator that a table has to be created as part of the HCatalog import job. If the option `--create-hcatalog-table` is specified and the table exists, then the table creation will fail and the job will be aborted.

The option `--hcatalog-storage-stanza` can be used to specify the storage format of the newly created table. The default value for this option is `stored as rcfile`. The value specified for this option is assumed to be a valid Hive storage format expression. It will be appended to the `create table` command generated by the HCatalog import job as part of automatic table creation. Any error in the storage stanza will cause the table creation to fail and the import job will be aborted.

Any additional resources needed to support the storage format referenced in the option `--hcatalog-storage-stanza` should be provided to the job either by placing them in `$HIVE_HOME/lib` or by providing them in `HADOOP_CLASSPATH` and `LIBJAR` files.

If the option `--hive-partition-key` is specified, then the value of this option is used as the partitioning key for the newly created table. Only one partitioning key can be specified with this option.

Object names are mapped to the lowercase equivalents as specified below when mapped to an HCatalog table. This includes the table name (which is the same as the external store table name converted to lower case) and field names.

23.4. Delimited Text Formats and Field and Line Delimiter Characters

HCatalog supports delimited text format as one of the table storage formats. But when delimited text is used and the imported data has fields that contain those delimiters, then the data may be parsed into a different number of fields and records by Hive, thereby losing data fidelity.

For this case, one of these existing Sqoop import options can be used:

- `--hive-delims-replacement`
- `--hive-drop-import-delims`

If either of these options is provided for import, then any column of type STRING will be formatted with the Hive delimiter processing and then written to the HCatalog table.

23.5. HCatalog Table Requirements

The HCatalog table should be created before using it as part of a Sqoop job if the default table creation options (with optional storage stanza) are not sufficient. All storage formats supported by HCatalog can be used with the creation of the HCatalog tables. This makes this feature readily adopt new storage formats that come into the Hive project, such as ORC files.

23.6. Support for Partitioning

The Sqoop HCatalog feature supports the following table types:

- Unpartitioned tables
- Partitioned tables with a static partitioning key specified
- Partitioned tables with dynamic partition keys from the database result set
- Partitioned tables with a combination of a static key and additional dynamic partitioning keys

23.7. Schema Mapping

Sqoop currently does not support column name mapping. However, the user is allowed to override the type mapping. Type mapping loosely follows the Hive type mapping already present in Sqoop except that SQL types FLOAT and REAL are mapped to HCatalog type float. In the Sqoop type mapping for Hive, these two are mapped to double. Type mapping is primarily used for checking the column definition correctness only and can be overridden with the --map-column-hive option.

All types except binary are assignable to a String type.

Any field of number type (int, shortint, tinyint, bigint and bigdecimal, float and double) is assignable to another field of any number type during exports and imports. Depending on the precision and scale of the target type of assignment, truncations can occur.

Furthermore, date/time/timestamps are mapped to date/timestamp hive types. (the full date/time/timestamp representation). Date/time/timstamp columns can also be mapped to bigint Hive type in which case the value will be the number of milliseconds since epoch.

BLOBs and CLOBs are only supported for imports. The BLOB/CLOB objects when imported are stored in a Sqoop-specific format and knowledge of this format is needed for processing these objects in a Pig/Hive job or another Map Reduce job.

Database column names are mapped to their lowercase equivalents when mapped to the HCatalog fields. Currently, case-sensitive database object names are not supported.

Projection of a set of columns from a table to an HCatalog table or loading to a column projection is allowed, subject to table constraints. The dynamic partitioning columns, if any, must be part of the projection when importing data into HCatalog tables.

Dynamic partitioning fields should be mapped to database columns that are defined with the NOT NULL attribute (although this is not enforced during schema mapping). A null value during import for a dynamic partitioning column will abort the Sqoop job.

23.8. Support for HCatalog Data Types

All the primitive Hive types that are part of Hive 0.13 version are supported. Currently all the complex HCatalog types are not supported.

BLOB/CLOB database types are only supported for imports.

23.9. Providing Hive and HCatalog Libraries for the Sqoop Job

With the support for HCatalog added to Sqoop, any HCatalog job depends on a set of jar files being available both on the Sqoop client host and where the Map/Reduce tasks run. To run HCatalog jobs,

the environment variable `HADOOP_CLASSPATH` must be set up as shown below before launching the Sqoop HCatalog jobs.

```
HADOOP_CLASSPATH=$(hcat -classpath) export HADOOP_CLASSPATH
```

The necessary HCatalog dependencies will be copied to the distributed cache automatically by the Sqoop job.

23.10. Examples

Create an HCatalog table, such as:

```
hcat -e "create table txn(txn_date string, cust_id string, amount float, store_id int) partitioned by (cust_id string) stored as rcfile;"
```

Then Sqoop import and export of the "txn" HCatalog table can be invoked as follows:

23.11. Import

```
$SQOOP_HOME/bin/sqoop import --connect <jdbc-url> -table <table-name> --hcatalog-table txn <other sqoop options>
```

23.12. Export

```
$SQOOP_HOME/bin/sqoop export --connect <jdbc-url> -table <table-name> --hcatalog-table txn <other sqoop options>
```

24. 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.

24.1. 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://`
CUBRID	9.2+	NO	`jdbc:cubrid:*`

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.

24.2. MySQL

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`.

24.2.1. 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
```

24.2.2. `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`.

24.2.3. `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.

24.2.4. Importing views in direct mode

Sqoop is currently not supporting import from view in direct mode. Use JDBC based (non direct) mode in case that you need to import view (simply omit `--direct` parameter).

24.3. PostgreSQL

Sqoop supports JDBC-based connector for PostgreSQL: http://jdbc.postgresql.org/

The connector has been tested using JDBC driver version "9.1-903 JDBC 4" with PostgreSQL server 9.1.

24.3.1. Importing views in direct mode

Sqoop is currently not supporting import from view in direct mode. Use JDBC based (non direct) mode in case that you need to import view (simply omit `--direct` parameter).

24.4. 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.

24.4.1. 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.

24.5. 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.

24.6. CUBRID

Sqoop supports JDBC-based connector for Cubrid: http://www.cubrid.org/?mid=downloads&item=jdbc_driver

The connector has been tested using JDBC driver version "JDBC-9.2.0.0155-cubrid.jar" with Cubrid 9.2.

25. Notes for specific connectors

25.1. MySQL JDBC Connector

This section contains information specific to MySQL JDBC Connector.

25.1.1. Upsert functionality

MySQL JDBC Connector is supporting upsert functionality using argument `--update-mode allowinsert`. To achieve that Sqoop is using MySQL clause INSERT INTO … ON DUPLICATE KEY UPDATE. This clause do not allow user to specify which columns should be used to distinct whether we should update existing row or add new row. Instead this clause relies on table's unique keys (primary key belongs to this set). MySQL will try to insert new row and if the insertion fails with duplicate unique key error it will update appropriate row instead. As a result, Sqoop is ignoring values specified in parameter `--update-key`, however user needs to specify at least one valid column to turn on update mode itself.

25.2. MySQL Direct Connector

MySQL Direct Connector allows faster import and export to/from MySQL using `mysqldump` and `mysqlimport` tools functionality instead of SQL selects and inserts.

To use the MySQL Direct Connector, specify the `--direct` argument for your import or export job.

Example:

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

Passing additional parameters to mysqldump:

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

25.2.1. Requirements

Utilities `mysqldump` and `mysqlimport` should be present in the shell path of the user running the Sqoop command on all nodes. To validate SSH as this user to all nodes and execute these commands. If you get an error, so will Sqoop.

25.2.2. Limitations

- Currently the direct connector does not support import of large object columns (BLOB and CLOB).
- Importing to HBase and Accumulo is not supported
- Use of a staging table when exporting data is not supported
- Import of views is not supported

25.2.3. 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.

Sometimes you need to export large data with Sqoop to a live MySQL cluster that is under a high load serving random queries from the users of your application. While data consistency issues during the export can be easily solved with a staging table, there is still a problem with the performance impact caused by the heavy export.

First off, the resources of MySQL dedicated to the import process can affect the performance of the live product, both on the master and on the slaves. Second, even if the servers can handle the import with no significant performance impact (mysqlimport should be relatively "cheap"), importing big tables can cause serious replication lag in the cluster risking data inconsistency.

With `-D sqoop.mysql.export.sleep.ms=time`, where *time* is a value in milliseconds, you can let the server relax between checkpoints and the replicas catch up by pausing the export process after transferring the number of bytes specified in `sqoop.mysql.export.checkpoint.bytes`. Experiment with different settings of these two parameters to archieve an export pace that doesn't endanger the stability of your MySQL cluster.

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). Don't forget that these parameters are only supported with the `--direct` flag set.

25.3. Microsoft SQL Connector

25.3.1. Extra arguments

List of all extra arguments supported by Microsoft SQL Connector is shown below:

Table 50. Supported Microsoft SQL Connector extra arguments:

Argument	Description
+--identity-insert	Set IDENTITY_INSERT to ON before export insert.
`--non-resilient`	Don't attempt to recover failed export operations.
`--schema <name>`	Scheme name that sqoop should use. Default is "dbo".
`--table-hints <hints>`	Table hints that Sqoop should use for data movement.

25.3.2. Allow identity inserts

You can allow inserts on columns that have identity. For example:

```
$ sqoop export ... --export-dir custom_dir --table custom_table -- --identity-insert
```

25.3.3. Non-resilient operations

You can override the default and not use resilient operations during export. This will avoid retrying failed operations. For example:

```
$ sqoop export ... --export-dir custom_dir --table custom_table -- --non-resilient
```

25.3.4. Schema support

If you need to work with tables that are located in non-default schemas, you can specify schema names via the `--schema` argument. Custom schemas are supported for both import and export jobs. For example:

```
$ sqoop import ... --table custom_table -- --schema custom_schema
```

25.3.5. Table hints

Sqoop supports table hints in both import and export jobs. Table hints are used only for queries that move data from/to Microsoft SQL Server, but they cannot be used for meta data queries. You can specify a comma-separated list of table hints in the `--table-hints` argument. For example:

```
$ sqoop import ... --table custom_table -- --table-hints NOLOCK
```

25.4. PostgreSQL Connector

25.4.1. Extra arguments

List of all extra arguments supported by PostgreSQL Connector is shown below:

Table 51. Supported PostgreSQL extra arguments:

Argument	Description
`--schema <name>`	Scheme name that sqoop should use. Default is "public".

25.4.2. Schema support

If you need to work with table that is located in schema other than default one, you need to specify extra argument `--schema`. Custom schemas are supported for both import and export job (optional staging table however must be present in the same schema as target table). Example invocation:

```
$ sqoop import ... --table custom_table -- --schema custom_schema
```

25.5. PostgreSQL Direct Connector

PostgreSQL Direct Connector allows faster import and export to/from PostgresSQL "COPY" command.

To use the PostgreSQL Direct Connector, specify the `--direct` argument for your import or export job.

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.

The direct connector offers also additional extra arguments:

Table 52. Additional supported PostgreSQL extra arguments in direct mode:

Argument	Description
`--boolean-true-string <str>`	String that will be used to encode `true` value of `boolean` columns.
	Default is "TRUE".
`--boolean-false-string <str>`	String that will be used to encode `false` value of `boolean` columns.
	Default is "FALSE".

25.5.1. Requirements

Utility `psql` should be present in the shell path of the user running the Sqoop command on all nodes. To validate SSH as this user to all nodes and execute these commands. If you get an error, so will Sqoop.

25.5.2. Limitations

- Currently the direct connector does not support import of large object columns (BLOB and CLOB).
- Importing to HBase and Accumulo is not supported
- Import of views is not supported

25.6. pg_bulkload connector

25.6.1. Purpose

pg_bulkload connector is a direct connector for exporting data into PostgreSQL. This connector uses pg_bulkload. Users benefit from functionality of pg_bulkload such as fast exports bypassing shared bufferes and WAL, flexible error records handling, and ETL feature with filter functions.

25.6.2. Requirements

pg_bulkload connector requires following conditions for export job execution:

- The pg_bulkload must be installed on DB server and all slave nodes. RPM for RedHat or CentOS is available in then download page.
- The PostgreSQL JDBC is required on client node.
- Superuser role of PostgreSQL database is required for execution of pg_bulkload.

25.6.3. Syntax

Use `--connection-manager` option to specify connection manager classname.

```
$ sqoop export (generic-args) --connection-manager org.apache.sqoop.manager.PGBulkloadManager (export-args)
$ sqoop-export (generic-args) --connection-manager org.apache.sqoop.manager.PGBulkloadManager (export-args)
```

This connector supports export arguments shown below.

Table 53. Supported export control arguments:

Argument	Description
`--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
`--input-null-string <null-string>`	The string to be interpreted as null for string columns

There are additional configuration for pg_bulkload execution specified via Hadoop Configuration properties which can be given with `-D <property=value>` option. Because Hadoop Configuration properties are generic arguments of the sqoop, it must preceed any export control arguments.

Table 54. Supported export control properties:

Property	Description
mapred.reduce.tasks	Number of reduce tasks for staging. The defalt value is 1. Each tasks do staging in a single transaction.
pgbulkload.bin	Path of the pg_bulkoad binary installed on each slave nodes.
pgbulkload.check.constraints	Specify whether CHECK constraints are checked during the loading. The default value is YES.
pgbulkload.parse.errors	The maximum number of ingored records that cause errors during parsing, encoding, filtering, constraints checking, and data type conversion. Error records are recorded in the PARSE BADFILE. The default value is INFINITE.
pgbulkload.duplicate.errors	Number of ingored records that violate unique constraints. Duplicated records are recorded in the DUPLICATE BADFILE on DB server. The default value is INFINITE.
pgbulkload.filter	Specify the filter function to convert each row in the input file. See the pg_bulkload documentation to know how to write FILTER functions.
pgbulkload.clear.staging.table	Indicates that any data present in the staging table can be dropped.

Here is a example of complete command line.

```
$ sqoop export \
    -Dmapred.reduce.tasks=2
    -Dpgbulkload.bin="/usr/local/bin/pg_bulkload" \
    -Dpgbulkload.input.field.delim=$'\t' \
    -Dpgbulkload.check.constraints="YES" \
    -Dpgbulkload.parse.errors="INFINITE" \
    -Dpgbulkload.duplicate.errors="INFINITE" \
    --connect jdbc:postgresql://pgsql.example.net:5432/sqooptest \
    --connection-manager org.apache.sqoop.manager.PGBulkloadManager \
    --table test --username sqooptest --export-dir=/test -m 2
```

25.6.4. Data Staging

Each map tasks of pg_bulkload connector's export job create their own staging table on the fly. The Name of staging tables is decided based on the destination table and the task attempt ids. For example, the name of staging table for the "test" table is like `test_attempt_1345021837431_0001_m_000000_0` .

Staging tables are automatically dropped if tasks successfully complete or map tasks fail. When reduce task fails, staging table for the task are left for manual retry and users must take care of it.

25.7. Netezza Connector

25.7.1. Extra arguments

List of all extra arguments supported by Netezza Connector is shown below:

Table 55. Supported Netezza extra arguments:

Argument	Description
`--partitioned-access`	Whether each mapper acts on a subset of data slices of a table or all Default is "false" for standard mode and "true" for direct mode.
`--max-errors`	Applicable only in direct mode. This option specifies the error threshold per mapper while transferring data. If the number of errors encountered exceed this threshold then the job will fail.
	Default value is 1.
`--log-dir`	Applicable only in direct mode. Specifies the directory where Netezza external table operation logs are stored on the hadoop filesystem. Logs are stored under this directory with one directory for the job and sub-directories for each task number and attempt. Default value is the user home directory.
`--trunc-string`	Applicable only in direct mode. Specifies whether the system truncates strings to the declared storage and loads the data. By default truncation of strings is reported as an error.
`--ctrl-chars`	Applicable only in direct mode. Specifies whether control characters (ASCII chars 1 - 31) can be allowed to be part of char/nchar/varchar/nvarchar columns. Default is false.

25.7.2. Direct Mode

Netezza connector supports an optimized data transfer facility using the Netezza external tables feature. Each map tasks of Netezza connector's import job will work on a subset of the Netezza partitions and transparently create and use an external table to transport data. Similarly, export jobs will use the external table to push data fast onto the NZ system. Direct mode does not support staging tables, upsert options etc.

Here is an example of complete command line for import using the Netezza external table feature.

```
$ sqoop import \
    --direct \
    --connect jdbc:netezza://nzhost:5480/sqoop \
    --table nztable \
    --username nzuser \
```

```
    --password nzpass \
    --target-dir hdfsdir
```

Here is an example of complete command line for export with tab as the field terminator character.

```
$ sqoop export \
    --direct \
    --connect jdbc:netezza://nzhost:5480/sqoop \
    --table nztable \
    --username nzuser \
    --password nzpass \
    --export-dir hdfsdir \
    --input-fields-terminated-by "\t"
```

25.7.3. Null string handling

Netezza direct connector supports the null-string features of Sqoop. The null string values are converted to appropriate external table options during export and import operations.

Table 56. Supported export control arguments:

Argument	Description
`--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.

In the case of Netezza direct mode connector, both the arguments must be left to the default values or explicitly set to the same value. Furthermore the null string value is restricted to 0-4 utf8 characters.

On export, for non-string columns, if the chosen null value is a valid representation in the column domain, then the column might not be loaded as null. For example, if the null string value is specified as "1", then on export, any occurrence of "1" in the input file will be loaded as value 1 instead of NULL for int columns.

It is suggested that the null value be specified as empty string for performance and consistency.

Table 57. Supported import control arguments:

Argument	Description
`--null-string <null-string>`	The string to be interpreted as null for string columns.
`--null-non-string <null-string>`	The string to be interpreted as null for non string columns.

In the case of Netezza direct mode connector, both the arguments must be left to the default values or explicitly set to the same value. Furthermore the null string value is restricted to 0-4 utf8 characters.

On import, for non-string columns, the chosen null value in current implementations the null value representation is ignored for non character columns. For example, if the null string value is specified as "\N", then on import, any occurrence of NULL for non-char columns in the table will be imported as an empty string instead of *\N*, the chosen null string representation.

It is suggested that the null value be specified as empty string for performance and consistency.

25.8. Data Connector for Oracle and Hadoop

25.8.1. About

The Data Connector for Oracle and Hadoop is now included in Sqoop.

It can be enabled by specifying the `--direct` argument for your import or export job.

25.8.1.1. Jobs

The Data Connector for Oracle and Hadoop inspects each Sqoop job and assumes responsibility for the ones it can perform better than the Oracle manager built into Sqoop.

Data Connector for Oracle and Hadoop accepts responsibility for the following Sqoop Job types:

- **Import** jobs that are **Non-Incremental**.
- **Export** jobs
- Data Connector for Oracle and Hadoop does not accept responsibility for other Sqoop job types. For example Data Connector for Oracle and Hadoop does not accept **eval** jobs etc.

Data Connector for Oracle and Hadoop accepts responsibility for those Sqoop Jobs with the following attributes:

- Oracle-related
- Table-Based - Jobs where the table argument is used and the specified object is a table.

> **Note**
> Data Connector for Oracle and Hadoop does not process index-organized tables unless the table is partitioned and `oraoop.chunk.method` is set to `PARTITION`

- There are at least 2 mappers — Jobs where the Sqoop command-line does not include: `--num-mappers 1`

25.8.1.2. How The Standard Oracle Manager Works for Imports

The Oracle manager built into Sqoop uses a range-based query for each mapper. Each mapper executes a query of the form:

```
SELECT * FROM sometable WHERE id >= lo AND id < hi
```

The **lo** and **hi** values are based on the number of mappers and the minimum and maximum values of the data in the column the table is being split by.

If no suitable index exists on the table then these queries result in full table-scans within Oracle. Even with a suitable index, multiple mappers may fetch data stored within the same Oracle blocks, resulting in redundant IO calls.

25.8.1.3. How The Data Connector for Oracle and Hadoop Works for Imports

The Data Connector for Oracle and Hadoop generates queries for the mappers of the form:

```
SELECT *
  FROM sometable
 WHERE rowid >= dbms_rowid.rowid_create(1, 893, 1, 279, 0) AND
       rowid <= dbms_rowid.rowid_create(1, 893, 1, 286, 32767)
```

The Data Connector for Oracle and Hadoop queries ensure that:

- No two mappers read data from the same Oracle block. This minimizes redundant IO.
- The table does not require indexes.
- The Sqoop command line does not need to specify a `--split-by` column.

25.8.1.4. Data Connector for Oracle and Hadoop Exports

Benefits of the Data Connector for Oracle and Hadoop:

- **Merge-Export facility** - Update Oracle tables by modifying changed rows AND inserting rows from the HDFS file that did not previously exist in the Oracle table. The Connector for Oracle and Hadoop's Merge-Export is unique - there is no Sqoop equivalent.
- **Lower impact on the Oracle database** - Update the rows in the Oracle table that have changed, not all rows in the Oracle table. This has performance benefits and reduces the impact of the query on Oracle (for example, the Oracle redo logs).
- **Improved performance** - With partitioned tables, mappers utilize temporary Oracle tables which allow parallel inserts and direct path writes.

25.8.2. Requirements

25.8.2.1. Ensure The Oracle Database JDBC Driver Is Setup Correctly

You may want to ensure the Oracle Database 11g Release 2 JDBC driver is setup correctly on your system. This driver is required for Sqoop to work with Oracle.

The Oracle Database 11g Release 2 JDBC driver file is `ojdbc6.jar` (3.2Mb).

If this file is not on your system then download it from:
http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html

This file should be put into the `$SQOOP_HOME/lib` directory.

25.8.2.2. Oracle Roles and Privileges

The Oracle user for The Data Connector for Oracle and Hadoop requires the following roles and privileges:

- `create session`

In addition, the user must have the select any dictionary privilege or select_catalog_role role or all of the following object privileges:

- `select on v_$instance`
- `select on dba_tables`
- `select on dba_tab_columns`
- `select on dba_objects`
- `select on dba_extents`
- `select on dba_segments` — Required for Sqoop imports only
- `select on dba_constraints` — Required for Sqoop imports only
- `select on v_$database` — Required for Sqoop imports only
- `select on v_$parameter` — Required for Sqoop imports only

Note
The user also requires the alter session privilege to make use of session tracing functionality. See "oraoop.oracle.session.initialization.statements" for more information.

25.8.2.3. Additional Oracle Roles And Privileges Required for Export

The Oracle user for Data Connector for Oracle and Hadoop requires:

- Quota on the tablespace in which the Oracle export tables are located.

  An example Oracle command to achieve this is

  ```
  alter user username quota unlimited on tablespace
  ```

- The following privileges:

Type of Export	Privileges Required
All Export	`create table`
	`select on dba_tab_partitions`
	`select on dba_tab_subpartitions`
	`select on dba_indexes`
	`select on dba_ind_columns`
Insert-Export with a template table into another schema	`select any table`
	`create any table`
	`insert any table`
	`alter any table` (partitioning)
Insert-Export without a template table into another schema	`select,insert on table` (no partitioning)
	`select,alter on table` (partitioning)
Update-Export into another schema	`select,update on table` (no partitioning)
	`select,delete,alter,insert on table` (partitioning)
Merge-Export into another schema	`select,insert,update on table` (no partitioning)
	`select,insert,delete,alter on table` (partitioning)

25.8.2.4. Supported Data Types

The following Oracle data types are supported by the Data Connector for Oracle and Hadoop:

BINARY_DOUBLE	NCLOB
BINARY_FLOAT	NUMBER
BLOB	NVARCHAR2
CHAR	RAW
CLOB	ROWID
DATE	TIMESTAMP
FLOAT	TIMESTAMP WITH TIME ZONE
INTERVAL DAY TO SECOND	TIMESTAMP WITH LOCAL TIME ZONE
INTERVAL YEAR TO MONTH	URITYPE
LONG	VARCHAR2
NCHAR	

All other Oracle column types are NOT supported. Example Oracle column types NOT supported by Data Connector for Oracle and Hadoop include:

All of the ANY types	BFILE
All of the MEDIA types	LONG RAW
All of the SPATIAL types	MLSLABEL
Any type referred to as UNDEFINED	UROWID
All custom (user-defined) URI types	XMLTYPE

Note

Data types RAW, LONG and LOB (BLOB, CLOB and NCLOB) are supported for Data Connector for Oracle and Hadoop imports. They are not supported for Data Connector for Oracle and Hadoop exports.

25.8.3. Execute Sqoop With Data Connector for Oracle and Hadoop

25.8.3.1. Connect to Oracle / Oracle RAC

The Sqoop `--connect` parameter defines the Oracle instance or Oracle RAC to connect to. It is required with all Sqoop import and export commands.

Data Connector for Oracle and Hadoop expects the associated connection string to be of a specific format dependent on whether the Oracle SID, Service or TNS name is defined. The TNS name based URL scheme can be used to enable authentication using Oracle wallets.

```
--connect jdbc:oracle:thin:@OracleServer:OraclePort:OracleSID

--connect jdbc:oracle:thin:@//OracleServer:OraclePort/OracleService

--connect jdbc:oracle:thin:@TNSName
```

25.8.3.2. Connect to An Oracle Database Instance

Parameter / Component	Description
`jdbc:oracle:thin`	The Data Connector for Oracle and Hadoop requires the connection string starts with jdbc:oracle.
	The Data Connector for Oracle and Hadoop has been tested with the thin driver however it should work equally well with other drivers such as OCI.
`OracleServer`	The host name of the Oracle server.
`OraclePort`	The port to connect to the Oracle server.
`OracleSID`	The Oracle instance.
`OracleService`	The Oracle Service.
`TNSName`	The TNS name for the entry describing the connection to the Oracle server.

Note

The Hadoop mappers connect to the Oracle database using a dynamically generated JDBC URL. This is designed to improve performance however it can be disabled by specifying:

```
-D oraoop.jdbc.url.verbatim=true
```

25.8.3.3. Connect to An Oracle RAC

Use the `--connect` parameter as above. The connection string should point to one instance of the Oracle RAC. The listener of the host of this Oracle instance will locate the other instances of the Oracle RAC.

Note

To improve performance, The Data Connector for Oracle and Hadoop identifies the active instances of the Oracle RAC and connects each Hadoop mapper to them in a roundrobin manner.

If services are defined for this Oracle RAC then use the following parameter to specify the service name:

```
-D oraoop.oracle.rac.service.name=ServiceName
```

Parameter / Component	Description
`OracleServer:OraclePort:OracleInstance`	Name one instance of the Oracle RAC. The Data Connector for Oracle and Hadoop assumes the same port number for all instances of the Oracle RAC.
	The listener of the host of this Oracle instance is used to locate other instances of the Oracle RAC. For more information enter this command on the host command line:
	`lsnrctl status`
`-D oraoop.oracle.rac.service.name=ServiceName`	The service to connect to in the Oracle RAC.
	A connection is made to all instances of the Oracle RAC associated with the service given by `ServiceName`.
	If omitted, a connection is made to all instances of the Oracle RAC.
	The listener of the host of this Oracle instance needs to know the `ServiceName` and all instances of the Oracle RAC. For more information enter this command on the host command line:
	`lsnrctl status`

25.8.3.4. Login to The Oracle Instance

Login to the Oracle instance on the Sqoop command line:

```
--connect jdbc:oracle:thin:@OracleServer:OraclePort:OracleInstance --username UserName -P
```

Parameter / Component	Description
`--username UserName`	The username to login to the Oracle instance (SID).
`-P`	You will be prompted for the password to login to the Oracle instance.

25.8.3.5. Kill Data Connector for Oracle and Hadoop Jobs

Use the Hadoop Job Tracker to kill the Sqoop job, just as you would kill any other Map-Reduce job.

```
$ hadoop job -kill jobid
```

To allow an Oracle DBA to kill a Data Connector for Oracle and Hadoop job (via killing the sessions in Oracle) you need to prevent Map-Reduce from re-attempting failed jobs. This is done via the following Sqoop command-line switch:

```
-D mapred.map.max.attempts=1
```

This sends instructions similar to the following to the console:

```
14/07/07 15:24:51 INFO oracle.OraOopManagerFactory:
Note: This Data Connector for Oracle and Hadoop job can be killed via Oracle
by executing the following statement:
  begin
    for row in (select sid,serial# from v$session where
                module='Data Connector for Oracle and Hadoop' and
                action='import 20140707152451EST') loop
      execute immediate 'alter system kill session ''' || row.sid ||
                        ',' || row.serial# || '''';
    end loop;
  end;
```

25.8.4. Import Data from Oracle

25.8.4.3. Consistent Read: All Mappers Read From The Same Point In Time

Execute Sqoop. Following is an example command:

```
$ sqoop import --direct --connect … --table OracleTableName
```

If The Data Connector for Oracle and Hadoop accepts the job then the following text is output:

```
**********************************************
*** Using Data Connector for Oracle and Hadoop ***
**********************************************
```

Note

- More information is available on the `--connect` parameter. See "Connect to Oracle / Oracle RAC" for more information.
- If Java runs out of memory the workaround is to specify each mapper's JVM memory allocation. Add the following parameter for example to allocate 4GB:

  ```
  -Dmapred.child.java.opts=-Xmx4000M
  ```

- An Oracle optimizer hint is included in the SELECT statement by default. See "oraoop.import.hint" for more information.

  You can alter the hint on the command line as follows:

  ```
  -Doraoop.import.hint="NO_INDEX(t)"
  ```

  You can turn off the hint on the command line as follows (notice the space between the double quotes):

  ```
  -Doraoop.import.hint=" "
  ```

25.8.4.1. Match Hadoop Files to Oracle Table Partitions

```
-Doraoop.chunk.method={ROWID|PARTITION}
```

To import data from a partitioned table in such a way that the resulting HDFS folder structure in Hadoop will match the table's partitions, set the chunk method to PARTITION. The alternative (default) chunk method is ROWID.

Note

- For the number of Hadoop files to match the number of Oracle partitions, set the number of mappers to be greater than or equal to the number of partitions.
- If the table is not partitioned then value PARTITION will lead to an error.

25.8.4.2. Specify The Partitions To Import

```
-Doraoop.import.partitions=PartitionA,PartitionB --table OracleTableName
```

Imports `PartitionA` and `PartitionB` of `OracleTableName`.

Note

- You can enclose an individual partition name in double quotes to retain the letter case or if the name has special characters.

  ```
  -Doraoop.import.partitions='"PartitionA",PartitionB' --table OracleTableName
  ```

  If the partition name is not double quoted then its name will be automatically converted to upper case, PARTITIONB for above.

  When using double quotes the entire list of partition names must be enclosed in single quotes.

If the last partition name in the list is double quoted then there must be a comma at the end of the list.

```
-Doraoop.import.partitions='"PartitionA","PartitionB",' --table OracleTableName
```

- Name each partition to be included. There is no facility to provide a range of partition names.
- There is no facility to define sub partitions. The entire partition is included/excluded as per the filter.

25.8.4.3. Consistent Read: All Mappers Read From The Same Point In Time

```
-Doraoop.import.consistent.read={true|false}
```

When set to `false` (by default) each mapper runs a select query. This will return potentially inconsistent data if there are a lot of DML operations on the table at the time of import.

Set to `true` to ensure all mappers read from the same point in time. The System Change Number (SCN) is passed down to all mappers, which use the Oracle Flashback Query to query the table as at that SCN.

Note

- Values `true` | `false` are case sensitive.
- By default the SCN is taken from V$database. You can specify the SCN in the following command

  ```
  -Doraoop.import.consistent.read.scn=12345
  ```

25.8.5. Export Data into Oracle

Execute Sqoop. Following is an example command:

```
$ sqoop export --direct --connect … --table OracleTableName --export-dir /user/username/tablename
```

The Data Connector for Oracle and Hadoop accepts all jobs that export data to Oracle. You can verify The Data Connector for Oracle and Hadoop is in use by checking the following text is output:

```
**********************************************
*** Using Data Connector for Oracle and Hadoop ***
**********************************************
```

Note

- `OracleTableName` is the Oracle table the data will export into.
- `OracleTableName` can be in a schema other than that for the connecting user. Prefix the table name with the schema, for example `SchemaName.OracleTableName`.
- Hadoop tables are picked up from the `/user/username/tablename` directory.
- The export will fail if the Hadoop file contains any fields of a data type not supported by The Data Connector for Oracle and Hadoop. See "Supported Data Types" for more information.
- The export will fail if the column definitions in the Hadoop table do not exactly match the column definitions in the Oracle table.

- The Data Connector for Oracle and Hadoop indicates if it finds temporary tables that it created more than a day ago that still exist. Usually these tables can be dropped. The only circumstance when these tables should not be dropped is when an The Data Connector for Oracle and Hadoop job has been running for more than 24 hours and is still running.
- More information is available on the `--connect` parameter. See "Connect to Oracle / Oracle RAC" for more information.

25.8.5.1. Insert-Export

Appends data to `OracleTableName`. It does not modify existing data in `OracleTableName`.

Insert-Export is the default method, executed in the absence of the `--update-key parameter`. All rows in the HDFS file in `/user/UserName/TableName` are inserted into `OracleTableName`. No change is made to pre-existing data in `OracleTableName`.

```
$ sqoop export --direct --connect … --table OracleTableName --export-dir /user/username/tablename
```

Note

- If `OracleTableName` was previously created by The Data Connector for Oracle and Hadoop with partitions then this export will create a new partition for the data being inserted.
- When creating `OracleTableName` specify a template. See "Create Oracle Tables" for more information.

25.8.5.2. Update-Export

```
--update-key OBJECT
```

Updates existing rows in `OracleTableName`.

Rows in the HDFS file in `/user/UserName/TableName` are matched to rows in `OracleTableName` by the `OBJECT` column. Rows that match are copied from the HDFS file to the Oracle table. No action is taken on rows that do not match.

```
$ sqoop export --direct --connect … --update-key OBJECT --table OracleTableName --export-dir
/user/username/tablename
```

Note

- If `OracleTableName` was previously created by The Data Connector for Oracle and Hadoop with partitions then this export will create a new partition for the data being inserted. Updated rows will be moved to the new partition that was created for the export.
- For performance reasons it is strongly recommended that where more than a few rows are involved column `OBJECT` be an index column of `OracleTableName`.
- Ensure the column name defined with `--update-key OBJECT` is specified in the correct letter case. Sqoop will show an error if the letter case is incorrect.
- It is possible to match rows via multiple columns. See "Match Rows Via Multiple Columns" for more information.

25.8.5.3. Merge-Export

```
--update-key OBJECT -Doraoop.export.merge=true
```

Updates existing rows in `OracleTableName`. Copies across rows from the HDFS file that do not exist within the Oracle table.

Rows in the HDFS file in `/user/UserName/TableName` are matched to rows in `OracleTableName` by the `OBJECT` column. Rows that match are copied from the HDFS file to the Oracle table. Rows in the HDFS file that do not exist in `OracleTableName` are added to `OracleTableName`.

```
$ sqoop export --direct --connect … --update-key OBJECT -Doraoop.export.merge=true --table OracleTableName --export-dir /user/username/tablename
```

Note

- Merge-Export is unique to The Data Connector for Oracle and Hadoop. It is not a standard Sqoop feature.
- If `OracleTableName` was previously created by The Data Connector for Oracle and Hadoop with partitions, then this export will create a new partition for the data being inserted. Updated rows will be moved to the new partition that was created for the export.
- For performance reasons it is strongly recommended that where more than a few rows are involved column `OBJECT` be an index column of `OracleTableName`.
- Ensure the column name defined with `--update-key OBJECT` is specified in the correct letter case. Sqoop will show an error if the letter case is incorrect.
- It is possible to match rows via multiple columns. See "Match Rows Via Multiple Columns" for more information.

25.8.5.4. Create Oracle Tables

`-Doraoop.template.table=TemplateTableName`

Creates `OracleTableName` by replicating the structure and data types of `TemplateTableName`. `TemplateTableName` is a table that exists in Oracle prior to executing the Sqoop command.

Note

- The export will fail if the Hadoop file contains any fields of a data type not supported by The Data Connector for Oracle and Hadoop. See "Supported Data Types" for more information.
- The export will fail if the column definitions in the Hadoop table do not exactly match the column definitions in the Oracle table.
- This parameter is specific to creating an Oracle table. The export will fail if `OracleTableName` already exists in Oracle.

Example command:

```
$ sqoop export --direct --connect.. --table OracleTableName --export-dir /user/username/tablename -Doraoop.template.table=TemplateTableName
```

25.8.5.5. NOLOGGING

`-Doraoop.nologging=true`

Assigns the NOLOGGING option to `OracleTableName`.

NOLOGGING may enhance performance but you will be unable to backup the table.

25.8.5.6. Partitioning

`-Doraoop.partitioned=true`

Partitions the table with the following benefits:

- The speed of the export is improved by allowing each mapper to insert data into a separate Oracle table using direct path writes. (An alter table exchange subpartition SQL statement is subsequently executed to swap the data into the export table.)
- You can selectively query or delete the data inserted by each Sqoop export job. For example, you can delete old data by dropping old partitions from the table.

The partition value is the SYSDATE of when Sqoop export job was performed.

The partitioned table created by The Data Connector for Oracle and Hadoop includes the following columns that don't exist in the template table:

- `oraoop_export_sysdate` - This is the Oracle SYSDATE when the Sqoop export job was performed. The created table will be partitioned by this column.
- `oraoop_mapper_id` - This is the id of the Hadoop mapper that was used to process the rows from the HDFS file. Each partition is subpartitioned by this column. This column exists merely to facilitate the exchange subpartition mechanism that is performed by each mapper during the export process.
- `oraoop_mapper_row` - A unique row id within the mapper / partition.

Note

If a unique row id is required for the table it can be formed by a combination of oraoop_export_sysdate, oraoop_mapper_id and oraoop_mapper_row.

25.8.5.7. Match Rows Via Multiple Columns

```
-Doraoop.update.key.extra.columns="ColumnA,ColumnB"
```

Used with Update-Export and Merge-Export to match on more than one column. The first column to be matched on is `--update-key OBJECT`. To match on additional columns, specify those columns on this parameter.

Note

- Letter case for the column names on this parameter is not important.
- All columns used for matching should be indexed. The first three items on the index should be `ColumnA`, `ColumnB` and the column specified on `--update-key` - but the order in which the columns are specified is not important.

25.8.5.8. Storage Clauses

```
-Doraoop.temporary.table.storage.clause="StorageClause"

-Doraoop.table.storage.clause="StorageClause"
```

Use to customize storage with Oracle clauses as in TABLESPACE or COMPRESS

`-Doraoop.table.storage.clause` applies to the export table that is created from `-Doraoop.template.table`. See "Create Oracle Tables" for more information. `-Doraoop.temporary.table.storage.clause` applies to all other working tables that are created during the export process and then dropped at the end of the export job.

25.8.6. Manage Date And Timestamp Data Types

25.8.6.1. Import Date And Timestamp Data Types from Oracle

This section lists known differences in the data obtained by performing an Data Connector for Oracle and Hadoop import of an Oracle table versus a native Sqoop import of the same table.

25.8.6.2. The Data Connector for Oracle and Hadoop Does Not Apply A Time Zone to DATE / TIMESTAMP Data Types

Data stored in a DATE or TIMESTAMP column of an Oracle table is not associated with a time zone. Sqoop without the Data Connector for Oracle and Hadoop inappropriately applies time zone information to this data.

Take for example the following timestamp in an Oracle DATE or TIMESTAMP column: `2am on 3rd October, 2010`.

Request Sqoop without the Data Connector for Oracle and Hadoop import this data using a system located in Melbourne Australia. The data is adjusted to Melbourne Daylight Saving Time. The data is imported into Hadoop as: `3am on 3rd October, 2010.`

The Data Connector for Oracle and Hadoop does not apply time zone information to these Oracle data-types. Even from a system located in Melbourne Australia, The Data Connector for Oracle and Hadoop ensures the Oracle and Hadoop timestamps match. The Data Connector for Oracle and Hadoop correctly imports this timestamp as: `2am on 3rd October, 2010`.

Note

In order for The Data Connector for Oracle and Hadoop to ensure data accuracy, Oracle DATE and TIMESTAMP values must be represented by a String, even when `--as-sequencefile` is used on the Sqoop command-line to produce a binary file in Hadoop.

25.8.6.3. The Data Connector for Oracle and Hadoop Retains Time Zone Information in TIMEZONE Data Types

Data stored in a TIMESTAMP WITH TIME ZONE column of an Oracle table is associated with a time zone. This data consists of two distinct parts: when the event occurred and where the event occurred.

When Sqoop without The Data Connector for Oracle and Hadoop is used to import data it converts the timestamp to the time zone of the system running Sqoop and omits the component of the data that specifies where the event occurred.

Take for example the following timestamps (with time zone) in an Oracle TIMESTAMP WITH TIME ZONE column:

```
2:59:00 am on 4th April, 2010. Australia/Melbourne
2:59:00 am on 4th April, 2010. America/New York
```

Request Sqoop without The Data Connector for Oracle and Hadoop import this data using a system located in Melbourne Australia. From the data imported into Hadoop we know when the events occurred, assuming we know the Sqoop command was run from a system located in the Australia/Melbourne time zone, but we have lost the information regarding where the event occurred.

```
2010-04-04 02:59:00.0
2010-04-04 16:59:00.0
```

Sqoop with The Data Connector for Oracle and Hadoop imports the example timestamps as follows. The Data Connector for Oracle and Hadoop retains the time zone portion of the data.

```
2010-04-04 02:59:00.0 Australia/Melbourne
2010-04-04 02:59:00.0 America/New_York
```

25.8.6.4. Data Connector for Oracle and Hadoop Explicitly States Time Zone for LOCAL TIMEZONE Data Types

Data stored in a TIMESTAMP WITH LOCAL TIME ZONE column of an Oracle table is associated with a time zone. Multiple end-users in differing time zones (locales) will each have that data expressed as a timestamp within their respective locale.

When Sqoop without the Data Connector for Oracle and Hadoop is used to import data it converts the timestamp to the time zone of the system running Sqoop and omits the component of the data that specifies location.

Take for example the following two timestamps (with time zone) in an Oracle TIMESTAMP WITH LOCAL TIME ZONE column:

```
2:59:00 am on 4th April, 2010. Australia/Melbourne
2:59:00 am on 4th April, 2010. America/New York
```

Request Sqoop without the Data Connector for Oracle and Hadoop import this data using a system located in Melbourne Australia. The timestamps are imported correctly but the local time zone has to be guessed. If multiple systems in different locale were executing the Sqoop import it would be very difficult to diagnose the cause of the data corruption.

```
2010-04-04 02:59:00.0
2010-04-04 16:59:00.0
```

Sqoop with the Data Connector for Oracle and Hadoop explicitly states the time zone portion of the data imported into Hadoop. The local time zone is GMT by default. You can set the local time zone with parameter:

```
-Doracle.sessionTimeZone=Australia/Melbourne
```

The Data Connector for Oracle and Hadoop would import these two timestamps as:

```
2010-04-04 02:59:00.0 Australia/Melbourne
2010-04-04 16:59:00.0 Australia/Melbourne
```

25.8.6.5. java.sql.Timestamp

To use Sqoop's handling of date and timestamp data types when importing data from Oracle use the following parameter:

```
-Doraoop.timestamp.string=false
```

Note

Sqoop's handling of date and timestamp data types does not store the timezone. However, some developers may prefer Sqoop's handling as the Data Connector for Oracle and Hadoop converts date and timestamp data types to string. This may not work for some developers as the string will require parsing later in the workflow.

25.8.6.6. Export Date And Timestamp Data Types into Oracle

Ensure the data in the HDFS file fits the required format exactly before using Sqoop to export the data into Oracle.

Note

- The Sqoop export command will fail if the data is not in the required format.
- ff = Fractional second
- TZR = Time Zone Region

Oracle Data Type	Required Format of The Data in the HDFS File
DATE	yyyy-mm-dd hh24:mi:ss
TIMESTAMP	yyyy-mm-dd hh24:mi:ss.ff

Oracle Data Type	Required Format of The Data in the HDFS File
TIMESTAMPTZ	`yyyy-mm-dd hh24:mi:ss.ff TZR`
TIMESTAMPLTZ	`yyyy-mm-dd hh24:mi:ss.ff TZR`

25.8.7. Configure The Data Connector for Oracle and Hadoop

25.8.7.1. oraoop-site-template.xml

The oraoop-site-template.xml file is supplied with the Data Connector for Oracle and Hadoop. It contains a number of ALTER SESSION statements that are used to initialize the Oracle sessions created by the Data Connector for Oracle and Hadoop.

If you need to customize these initializations to your environment then:

1. Find `oraoop-site-template.xml` in the Sqoop configuration directory.
2. Copy `oraoop-site-template.xml` to `oraoop-site.xml`.
3. Edit the `ALTER SESSION` statements in `oraoop-site.xml`.

25.8.7.2. oraoop.oracle.session.initialization.statements

The value of this property is a semicolon-delimited list of Oracle SQL statements. These statements are executed, in order, for each Oracle session created by the Data Connector for Oracle and Hadoop.

The default statements include:

```
alter session set time_zone = '{oracle.sessionTimeZone|GMT}';
```

This statement initializes the timezone of the JDBC client. This ensures that data from columns of type TIMESTAMP WITH LOCAL TIMEZONE are correctly adjusted into the timezone of the client and not kept in the timezone of the Oracle database.

Note

- There is an explanation to the text within the curly-braces. See "Expressions in oraoop-site.xml" for more information..
- A list of the time zones supported by your Oracle database is available by executing the following query: `SELECT TZNAME FROM V$TIMEZONE_NAMES;`

```
alter session disable parallel query;
```

This statement instructs Oracle to not parallelize SQL statements executed by the Data Connector for Oracle and Hadoop sessions. This Oracle feature is disabled because the Map/Reduce job launched by Sqoop is the mechanism used for parallelization.

It is recommended that you not enable parallel query because it can have an adverse effect the load on the Oracle instance and on the balance between the Data Connector for Oracle and Hadoop mappers.

Some export operations are performed in parallel where deemed appropriate by the Data Connector for Oracle and Hadoop. See "Parallelization" for more information.

`alter session set "_serial_direct_read"=true;`

This statement instructs Oracle to bypass the buffer cache. This is used to prevent Oracle from filling its buffers with the data being read by the Data Connector for Oracle and Hadoop, therefore diminishing its capacity to cache higher prioritized data. Hence, this statement is intended to minimize the Data Connector for Oracle and Hadoop's impact on the immediate future performance of the Oracle database.

`--alter session set events '10046 trace name context forever, level 8';`

This statement has been commented-out. To allow tracing, remove the comment token "--" from the start of the line.

Note

- These statements are placed on separate lines for readability. They do not need to be placed on separate lines.
- A statement can be commented-out via the standard Oracle double-hyphen token: "--". The comment takes effect until the next semicolon.

25.8.7.3. oraoop.table.import.where.clause.location

SUBSPLIT (default)

When set to this value, the where clause is applied to each subquery used to retrieve data from the Oracle table.

A Sqoop command like:

```
sqoop import -D oraoop.table.import.where.clause.location=SUBSPLIT --table JUNK --where "owner like 'G%'"
```

Generates SQL query of the form:

```
SELECT OWNER,OBJECT_NAME
  FROM JUNK
 WHERE ((rowid >=
          dbms_rowid.rowid_create(1, 113320, 1024, 4223664, 0)
        AND rowid <=
            dbms_rowid.rowid_create(1, 113320, 1024, 4223671, 32767)))
   AND (owner like 'G%')
UNION ALL
SELECT OWNER,OBJECT_NAME
  FROM JUNK
 WHERE ((rowid >=
          dbms_rowid.rowid_create(1, 113320, 1024, 4223672, 0)
        AND rowid <=
          dbms_rowid.rowid_create(1, 113320, 1024, 4223679, 32767)))
   AND (owner like 'G%')
```

SPLIT

When set to this value, the where clause is applied to the entire SQL statement used by each split/mapper.

A Sqoop command like:

```
sqoop import -D oraoop.table.import.where.clause.location=SPLIT --table JUNK --where "rownum <= 10"
```

Generates SQL query of the form:

```
SELECT OWNER,OBJECT_NAME
  FROM (
        SELECT OWNER,OBJECT_NAME
          FROM JUNK
         WHERE ((rowid >=
                  dbms_rowid.rowid_create(1, 113320, 1024, 4223664, 0)
                AND rowid <=
                  dbms_rowid.rowid_create(1, 113320, 1024, 4223671, 32767)))
        UNION ALL
        SELECT OWNER,OBJECT_NAME
          FROM JUNK
         WHERE ((rowid >=
                  dbms_rowid.rowid_create(1, 113320, 1024, 4223672, 0)
                AND rowid <=
                  dbms_rowid.rowid_create(1, 113320, 1024, 4223679,32767)))
```

```
      )
WHERE rownum <= 10
```

Note

- In this example, there are up to 10 rows imported per mapper.
- The SPLIT clause may result in greater overhead than the SUBSPLIT clause because the UNION statements need to be fully materialized before the data can be streamed to the mappers. However, you may wish to use SPLIT in the case where you want to limit the total number of rows processed by each mapper.

25.8.7.4. oracle.row.fetch.size

The value of this property is an integer specifying the number of rows the Oracle JDBC driver should fetch in each network round-trip to the database. The default value is 5000.

If you alter this setting, confirmation of the change is displayed in the logs of the mappers during the Map-Reduce job.

25.8.7.5. oraoop.import.hint

The Oracle optimizer hint is added to the SELECT statement for IMPORT jobs as follows:

```
SELECT /*+ NO_INDEX(t) */ * FROM employees;
```

The default hint is `NO_INDEX(t)`

Note

- The hint can be added to the command line. See "Import Data from Oracle" for more information.
- See the Oracle Database Performance Tuning Guide (Using Optimizer Hints) for more information on Oracle optimizer hints.
- To turn the hint off, insert a space between the <value> elements.

```
<property>
  <name>oraoop.import.hint</name>
  <value> </value>
</property>
```

25.8.7.6. oraoop.oracle.append.values.hint.usage

The value of this property is one of: AUTO / ON / OFF.

AUTO

AUTO is the default value.

Currently AUTO is equivalent to OFF.

ON

During export the Data Connector for Oracle and Hadoop uses direct path writes to populate the target Oracle table, bypassing the buffer cache. Oracle only allows a single session to perform direct writes against a specific table at any time, so this has the effect of serializing the writes to the table. This may reduce throughput, especially if the number of mappers is high. However, for databases where DBWR is very busy, or where the IO bandwidth to the underlying table is narrow (table resides on a single disk spindle for instance), then setting `oraoop.oracle.append.values.hint.usage` to ON may reduce the load on the Oracle database and possibly increase throughput.

OFF

During export the Data Connector for Oracle and Hadoop does not use the `APPEND_VALUES` Oracle hint.

Note

This parameter is only effective on Oracle 11g Release 2 and above.

25.8.7.7. mapred.map.tasks.speculative.execution

By default speculative execution is disabled for the Data Connector for Oracle and Hadoop. This avoids placing redundant load on the Oracle database.

If Speculative execution is enabled, then Hadoop may initiate multiple mappers to read the same blocks of data, increasing the overall load on the database.

25.8.7.8. oraoop.block.allocation

This setting determines how Oracle's data-blocks are assigned to Map-Reduce mappers.

Note

Applicable to import. Not applicable to export.

ROUNDROBIN (default)

Each chunk of Oracle blocks is allocated to the mappers in a roundrobin manner. This helps prevent one of the mappers from being allocated a large proportion of typically small-sized blocks from the start of Oracle data-files. In doing so it also helps prevent one of the other mappers from being allocated a large proportion of typically larger-sized blocks from the end of the Oracle data-files.

Use this method to help ensure all the mappers are allocated a similar amount of work.

RANDOM

The list of Oracle blocks is randomized before being allocated to the mappers via a round-robin approach. This has the benefit of increasing the chance that, at any given instant in time, each mapper is reading from a different Oracle data-file. If the Oracle data-files are located on separate spindles, this should increase the overall IO throughput.

SEQUENTIAL

Each chunk of Oracle blocks is allocated to the mappers sequentially. This produces the tendency for each mapper to sequentially read a large, contiguous proportion of an Oracle data-file. It is unlikely for the performance of this method to exceed that of the round-robin method and it is more likely to allocate a large difference in the work between the mappers.

Use of this method is generally not recommended.

25.8.7.9. oraoop.import.omit.lobs.and.long

This setting can be used to omit all LOB columns (BLOB, CLOB and NCLOB) and LONG column from an Oracle table being imported. This is advantageous in troubleshooting, as it provides a convenient way to exclude all LOB-based data from the import.

25.8.7.10. oraoop.locations

Note

Applicable to import. Not applicable to export.

By default, four mappers are used for a Sqoop import job. The number of mappers can be altered via the Sqoop `--num-mappers` parameter.

If the data-nodes in your Hadoop cluster have 4 task-slots (that is they are 4-CPU core machines) it is likely for all four mappers to execute on the same machine. Therefore, IO may be concentrated between the Oracle database and a single machine.

This setting allows you to control which DataNodes in your Hadoop cluster each mapper executes on. By assigning each mapper to a separate machine you may improve the overall IO performance for the job. This will also have the side-effect of the imported data being more diluted across the machines in the cluster. (HDFS replication will dilute the data across the cluster anyway.)

Specify the machine names as a comma separated list. The locations are allocated to each of the mappers in a round-robin manner.

If using EC2, specify the internal name of the machines. Here is an example of using this parameter from the Sqoop command-line:

```
$ sqoop import -D oraoop.locations=ip-10-250-23-225.ec2.internal,ip-10-250-107-32.ec2.internal,ip-10-250-207-2.ec2.internal,ip-10-250-27-114.ec2.internal --direct --connect…
```

25.8.7.11. sqoop.connection.factories

This setting determines behavior if the Data Connector for Oracle and Hadoop cannot accept the job. By default Sqoop accepts the jobs that the Data Connector for Oracle and Hadoop rejects.

Set the value to `org.apache.sqoop.manager.oracle.OraOopManagerFactory` when you want the job to fail if the Data Connector for Oracle and Hadoop cannot accept the job.

25.8.7.12. Expressions in oraoop-site.xml

Text contained within curly-braces { and } are expressions to be evaluated prior to the SQL statement being executed. The expression contains the name of the configuration property optionally followed by a default value to use if the property has not been set. A pipe | character is used to delimit the property name and the default value.

For example:

When this Sqoop command is executed

```
$ sqoop import -D oracle.sessionTimeZone=US/Hawaii --direct --connect
```

The statement within oraoop-site.xml

```
alter session set time_zone ='{oracle.sessionTimeZone|GMT}';
```

Becomes

```
alter session set time_zone = 'US/Hawaii'
```

If the oracle.sessionTimeZone property had not been set, then this statement would use the specified default value and would become

```
alter session set time_zone = 'GMT'
```

Note

The `oracle.sessionTimeZone` property can be specified within the `sqoop-site.xml` file if you want this setting to be used all the time.

25.8.8. Troubleshooting The Data Connector for Oracle and Hadoop

25.8.8.1. Quote Oracle Owners And Tables

If the owner of the Oracle table needs to be quoted, use:	`$ sqoop import … --table "\"\"Scott\".customers\""`

	This is the equivalent of: "Scott".customers
If the Oracle table needs to be quoted, use:	$ `sqoop import … --table "\"scott.\"Customers\"\""` This is the equivalent of: scott."Customers"
If both the owner of the Oracle table and the table itself needs to be quoted, use:	$ `sqoop import … --table "\"\"Scott\".\"Customers\"\""` This is the equivalent of: "Scott"."Customers"

Note

- The HDFS output directory is called something like: /user/username/"Scott"."Customers"
- If a table name contains a $ character, it may need to be escaped within your Unix shell. For example, the dr$object table in the ctxsys schema would be referred to as: $ `sqoop import … --table "ctxsys.dr\$object"`

25.8.8.2. Quote Oracle Columns

If a column name of an Oracle table needs to be quoted, use

$ `sqoop import … --table customers --columns "\"\"first name\"\""`

This is the equivalent of: `select "first name" from customers`

25.8.8.3. Confirm The Data Connector for Oracle and Hadoop Can Initialize The Oracle Session

If the Sqoop output includes feedback such as the following then the configuration properties contained within `oraoop-site-template.xml` and `oraoop-site.xml` have been loaded by Hadoop and can be accessed by the Data Connector for Oracle and Hadoop.

```
14/07/08 15:21:13 INFO oracle.OracleConnectionFactory: Initializing Oracle session with SQL
```

25.8.8.4. Check The Sqoop Debug Logs for Error Messages

For more information about any errors encountered during the Sqoop import, refer to the log files generated by each of the (by default 4) mappers that performed the import.

The logs can be obtained via your Map-Reduce Job Tracker's web page.

Include these log files with any requests you make for assistance on the Sqoop User Group web site.

25.8.8.5. Export: Check Tables Are Compatible

Check tables particularly in the case of a parsing error.

- Ensure the fields contained with the HDFS file and the columns within the Oracle table are identical. If they are not identical, the Java code dynamically generated by Sqoop to parse the HDFS file will throw an error when reading the file – causing the export to fail. When creating a table in Oracle ensure the definitions for the table template are identical to the definitions for the HDFS file.
- Ensure the data types in the table are supported. See "Supported Data Types" for more information.
- Are date and time zone based data types used? See "Export Date And Timestamp Data Types into Oracle" for more information.

25.8.8.6. Export: Parallelization

```
-D oraoop.export.oracle.parallelization.enabled=false
```

If you see a parallelization error you may decide to disable parallelization on Oracle queries.

25.8.8.7. Export: Check oraoop.oracle.append.values.hint.usage

The oraoop.oracle.append.values.hint.usage parameter should not be set to ON if the Oracle table contains either a BINARY_DOUBLE or BINARY_FLOAT column and the HDFS file being exported contains a NULL value in either of these column types. Doing so will result in the error: `ORA-12838: cannot read/modify an object after modifying it in parallel`.

25.8.8.8. Turn On Verbose

Turn on verbose on the Sqoop command line.

```
--verbose
```

Check Sqoop stdout (standard output) and the mapper logs for information as to where the problem may be.

26. Getting Support

Some general information is available at the http://sqoop.apache.org/

Report bugs in Sqoop to the issue tracker at https://issues.apache.org/jira/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`).

27. Troubleshooting

27.1. General Troubleshooting Process

The following steps should be followed to troubleshoot any failure that you encounter while running Sqoop.

- Turn on verbose output by executing the same command again and specifying the `--verbose` option. This produces more debug output on the console which can be inspected to identify any obvious errors.
- Look at the task logs from Hadoop to see if there are any specific failures recorded there. It is possible that the failure that occurs while task execution is not relayed correctly to the console.
- Make sure that the necessary input files or input/output tables are present and can be accessed by the user that Sqoop is executing as or connecting to the database as. It is possible that the necessary files or tables are present but the specific user that Sqoop connects as does not have the necessary permissions to access these files.

- If you are doing a compound action such as populating a Hive table or partition, try breaking the job into two separate actions to see where the problem really occurs. For example if an import that creates and populates a Hive table is failing, you can break it down into two steps - first for doing the import alone, and the second to create a Hive table without the import using the `create-hive-table` tool. While this does not address the original use-case of populating the Hive table, it does help narrow down the problem to either regular import or during the creation and population of Hive table.
- Search the mailing lists archives and JIRA for keywords relating to the problem. It is possible that you may find a solution discussed there that will help you solve or work-around your problem.

27.2. Specific Troubleshooting Tips

27.2.1. Oracle: Connection Reset Errors

Problem: When using the default Sqoop connector for Oracle, some data does get transferred, but during the map-reduce job a lot of errors are reported as below:

```
11/05/26 16:23:47 INFO mapred.JobClient: Task Id : attempt_201105261333_0002_m_000002_0, Status : FAILED
java.lang.RuntimeException: java.lang.RuntimeException: java.sql.SQLRecoverableException: IO Error: Connection reset
at com.cloudera.sqoop.mapreduce.db.DBInputFormat.setConf(DBInputFormat.java:164)
at org.apache.hadoop.util.ReflectionUtils.setConf(ReflectionUtils.java:62)
at org.apache.hadoop.util.ReflectionUtils.newInstance(ReflectionUtils.java:117)
at org.apache.hadoop.mapred.MapTask.runNewMapper(MapTask.java:605)
at org.apache.hadoop.mapred.MapTask.run(MapTask.java:322)
at org.apache.hadoop.mapred.Child$4.run(Child.java:268)
at java.security.AccessController.doPrivileged(Native Method)
at javax.security.auth.Subject.doAs(Subject.java:396)
at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1115)
at org.apache.hadoop.mapred.Child.main(Child.java:262)
Caused by: java.lang.RuntimeException: java.sql.SQLRecoverableException: IO Error: Connection reset
at com.cloudera.sqoop.mapreduce.db.DBInputFormat.getConnection(DBInputFormat.java:190)
at com.cloudera.sqoop.mapreduce.db.DBInputFormat.setConf(DBInputFormat.java:159)
... 9 more
Caused by: java.sql.SQLRecoverableException: IO Error: Connection reset
at oracle.jdbc.driver.T4CConnection.logon(T4CConnection.java:428)
at oracle.jdbc.driver.PhysicalConnection.<init>(PhysicalConnection.java:536)
at oracle.jdbc.driver.T4CConnection.<init>(T4CConnection.java:228)
at oracle.jdbc.driver.T4CDriverExtension.getConnection(T4CDriverExtension.java:32)
at oracle.jdbc.driver.OracleDriver.connect(OracleDriver.java:521)
at java.sql.DriverManager.getConnection(DriverManager.java:582)
at java.sql.DriverManager.getConnection(DriverManager.java:185)
at com.cloudera.sqoop.mapreduce.db.DBConfiguration.getConnection(DBConfiguration.java:152)
at com.cloudera.sqoop.mapreduce.db.DBInputFormat.getConnection(DBInputFormat.java:184)
... 10 more
Caused by: java.net.SocketException: Connection reset
at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:96)
at java.net.SocketOutputStream.write(SocketOutputStream.java:136)
at oracle.net.ns.DataPacket.send(DataPacket.java:199)
at oracle.net.ns.NetOutputStream.flush(NetOutputStream.java:211)
at oracle.net.ns.NetInputStream.getNextPacket(NetInputStream.java:227)
at oracle.net.ns.NetInputStream.read(NetInputStream.java:175)
at oracle.net.ns.NetInputStream.read(NetInputStream.java:100)
at oracle.net.ns.NetInputStream.read(NetInputStream.java:85)
at oracle.jdbc.driver.T4CSocketInputStreamWrapper.readNextPacket(T4CSocketInputStreamWrapper.java:123)
at oracle.jdbc.driver.T4CSocketInputStreamWrapper.read(T4CSocketInputStreamWrapper.java:79)
at oracle.jdbc.driver.T4CMAREngine.unmarshalUB1(T4CMAREngine.java:1122)
at oracle.jdbc.driver.T4CMAREngine.unmarshalSB1(T4CMAREngine.java:1099)
at oracle.jdbc.driver.T4CTTIfun.receive(T4CTTIfun.java:288)
at oracle.jdbc.driver.T4CTTIfun.doRPC(T4CTTIfun.java:191)
at oracle.jdbc.driver.T4CTTIoauthenticate.doOAUTH(T4CTTIoauthenticate.java:366)
at oracle.jdbc.driver.T4CTTIoauthenticate.doOAUTH(T4CTTIoauthenticate.java:752)
at oracle.jdbc.driver.T4CConnection.logon(T4CConnection.java:366)
... 18 more
```

Solution: This problem occurs primarily due to the lack of a fast random number generation device on the host where the map tasks execute. On typical Linux systems this can be addressed by setting the following property in the `java.security` file:

```
securerandom.source=file:/dev/../dev/urandom
```

The `java.security` file can be found under `$JAVA_HOME/jre/lib/security` directory. Alternatively, this property can also be specified on the command line via:

```
-D mapred.child.java.opts="-Djava.security.egd=file:/dev/../dev/urandom"
```

Please note that it's very important to specify this weird path `/dev/../dev/urandom` as it is due to a Java bug 6202721, or `/dev/urandom` will be ignored and substituted by `/dev/random`.

27.2.2. Oracle: Case-Sensitive Catalog Query Errors

Problem: While working with Oracle you may encounter problems when Sqoop can not figure out column names. This happens because the catalog queries that Sqoop uses for Oracle expect the correct case to be specified for the user name and table name.

One example, using --hive-import and resulting in a NullPointerException:

```
1/09/21 17:18:49 INFO manager.OracleManager: Time zone has been set to
GMT
11/09/21 17:18:49 DEBUG manager.SqlManager: Using fetchSize for next
query: 1000
11/09/21 17:18:49 INFO manager.SqlManager: Executing SQL statement:
SELECT t.* FROM addlabel_pris t WHERE 1=0
11/09/21 17:18:49 DEBUG manager.OracleManager$ConnCache: Caching
released connection for jdbc:oracle:thin:
11/09/21 17:18:49 ERROR sqoop.Sqoop: Got exception running Sqoop:
java.lang.NullPointerException
java.lang.NullPointerException
at com.cloudera.sqoop.hive.TableDefWriter.getCreateTableStmt(TableDefWriter.java:148)
at com.cloudera.sqoop.hive.HiveImport.importTable(HiveImport.java:187)
at com.cloudera.sqoop.tool.ImportTool.importTable(ImportTool.java:362)
at com.cloudera.sqoop.tool.ImportTool.run(ImportTool.java:423)
at com.cloudera.sqoop.Sqoop.run(Sqoop.java:144)
at org.apache.hadoop.util.ToolRunner.run(ToolRunner.java:65)
at com.cloudera.sqoop.Sqoop.runSqoop(Sqoop.java:180)
at com.cloudera.sqoop.Sqoop.runTool(Sqoop.java:219)
at com.cloudera.sqoop.Sqoop.runTool(Sqoop.java:228)
at com.cloudera.sqoop.Sqoop.main(Sqoop.java:237)
```

Solution:

1. Specify the user name, which Sqoop is connecting as, in upper case (unless it was created with mixed/lower case within quotes).
2. Specify the table name, which you are working with, in upper case (unless it was created with mixed/lower case within quotes).

27.2.3. MySQL: Connection Failure

Problem: While importing a MySQL table into Sqoop, if you do not have the necessary permissions to access your MySQL database over the network, you may get the below connection failure.

```
Caused by: com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: Communications link failure
```

Solution: First, verify that you can connect to the database from the node where you are running Sqoop:

```
$ mysql --host=<IP Address> --database=test --user=<username> --password=<password>
```

If this works, it rules out any problem with the client network configuration or security/authentication configuration.

Add the network port for the server to your my.cnf file `/etc/my.cnf`:

```
[mysqld]
port = xxxx
```

Set up a user account to connect via Sqoop. Grant permissions to the user to access the database over the network: (1.) Log into MySQL as root `mysql -u root -p<ThisIsMyPassword>`. (2.) Issue the following command:

```
mysql> grant all privileges on test.* to 'testuser'@'%' identified by 'testpassword'
```

Note that doing this will enable the testuser to connect to the MySQL server from any IP address. While this will work, it is not advisable for a production environment. We advise consulting with your DBA to grant the necessary privileges based on the setup topology.

If the database server's IP address changes, unless it is bound to a static hostname in your server, the connect string passed into Sqoop will also need to be changed.

27.2.4. Oracle: ORA-00933 error (SQL command not properly ended)

Problem: While working with Oracle you may encounter the below problem when the Sqoop command explicitly specifies the --driver <driver name> option. When the driver option is included in the Sqoop command, the built-in connection manager selection defaults to the generic connection manager, which causes this issue with Oracle. If the driver option is not specified, the built-in connection manager selection mechanism selects the Oracle specific connection manager which generates valid SQL for Oracle and uses the driver "oracle.jdbc.OracleDriver".

```
ERROR manager.SqlManager: Error executing statement:
java.sql.SQLSyntaxErrorException: ORA-00933: SQL command not properly ended
```

Solution: Omit the option --driver oracle.jdbc.driver.OracleDriver and then re-run the Sqoop command.

27.2.5. MySQL: Import of TINYINT(1) from MySQL behaves strangely

Problem: Sqoop is treating TINYINT(1) columns as booleans, which is for example causing issues with HIVE import. This is because by default the MySQL JDBC connector maps the TINYINT(1) to java.sql.Types.BIT, which Sqoop by default maps to Boolean.

Solution: A more clean solution is to force MySQL JDBC Connector to stop converting TINYINT(1) to java.sql.Types.BIT by adding `tinyInt1isBit=false` into your JDBC path (to create something like `jdbc:mysql://localhost/test?tinyInt1isBit=false`). Another solution would be to explicitly override the column mapping for the datatype TINYINT(1) column. For example, if the column name is foo, then pass the following option to Sqoop during import: --map-column-hive foo=tinyint. In the case of non-Hive imports to HDFS, use --map-column-java foo=integer.