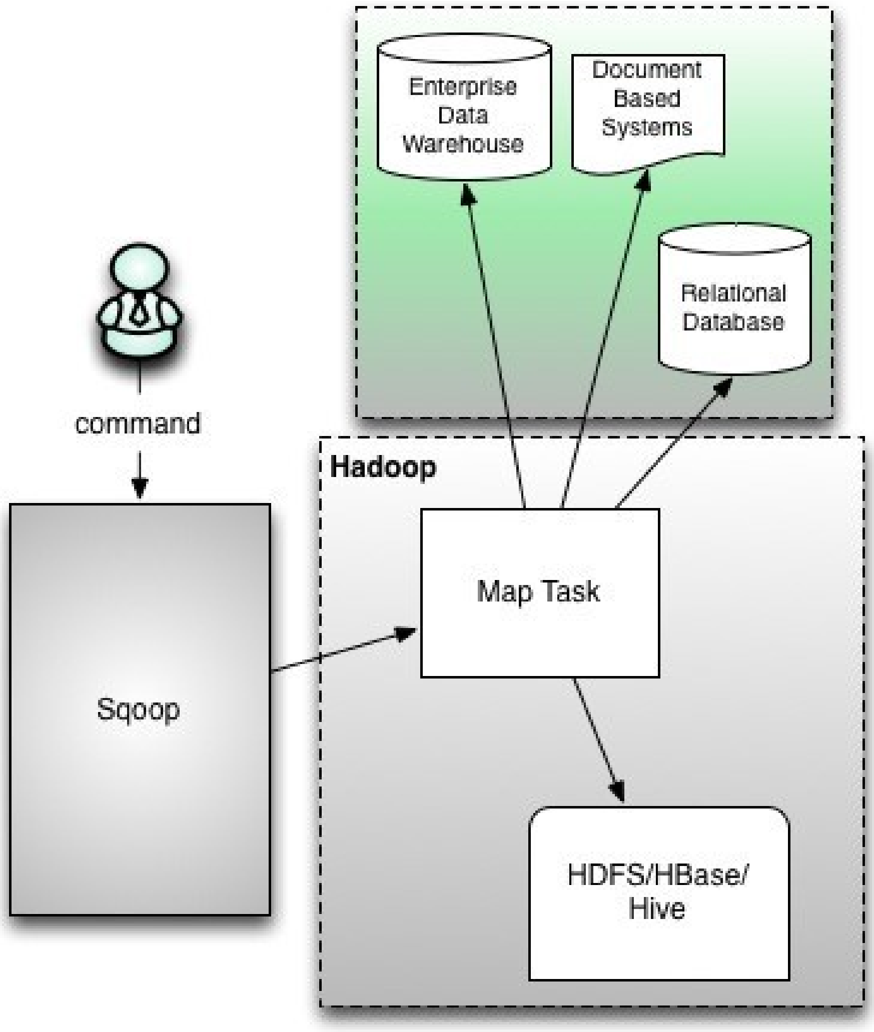
# SQOOP

- Apache Sqoop is a tool designed for efficiently transferring bulk data in a distributed manner between Apache Hadoop and structured datastores such as relational databases, enterprise data warehouses, and NoSQL systems.
- Sqoop can be used to import data into HBase, HDFS and Hive and out of it into RDBMS, in an automated fashion, leveraging Oozie for scheduling.
- It has a connector based architecture that supports plugins that provide connectivity to new external systems.

# SQOOP

- Sqoop automates most of the process, depends on the database to describe the schema of the data to be imported. Sqoop uses MapReduce framework to import and export the data, which provides parallel mechanism as well as fault tolerance.
- Behind the scenes, the dataset being transferred is split into partitions and map only jobs are launched for each partition with the mappers managing transferring the dataset assigned to it.  Sqoop uses the database metadata to infer the types, and handles the data in a type safe manner.
- Sqoop makes developers life easy by providing command line interface. Developers just need to provide basic information like source, destination and database authentication details in the sqoop command. Sqoop takes care of remaining part.

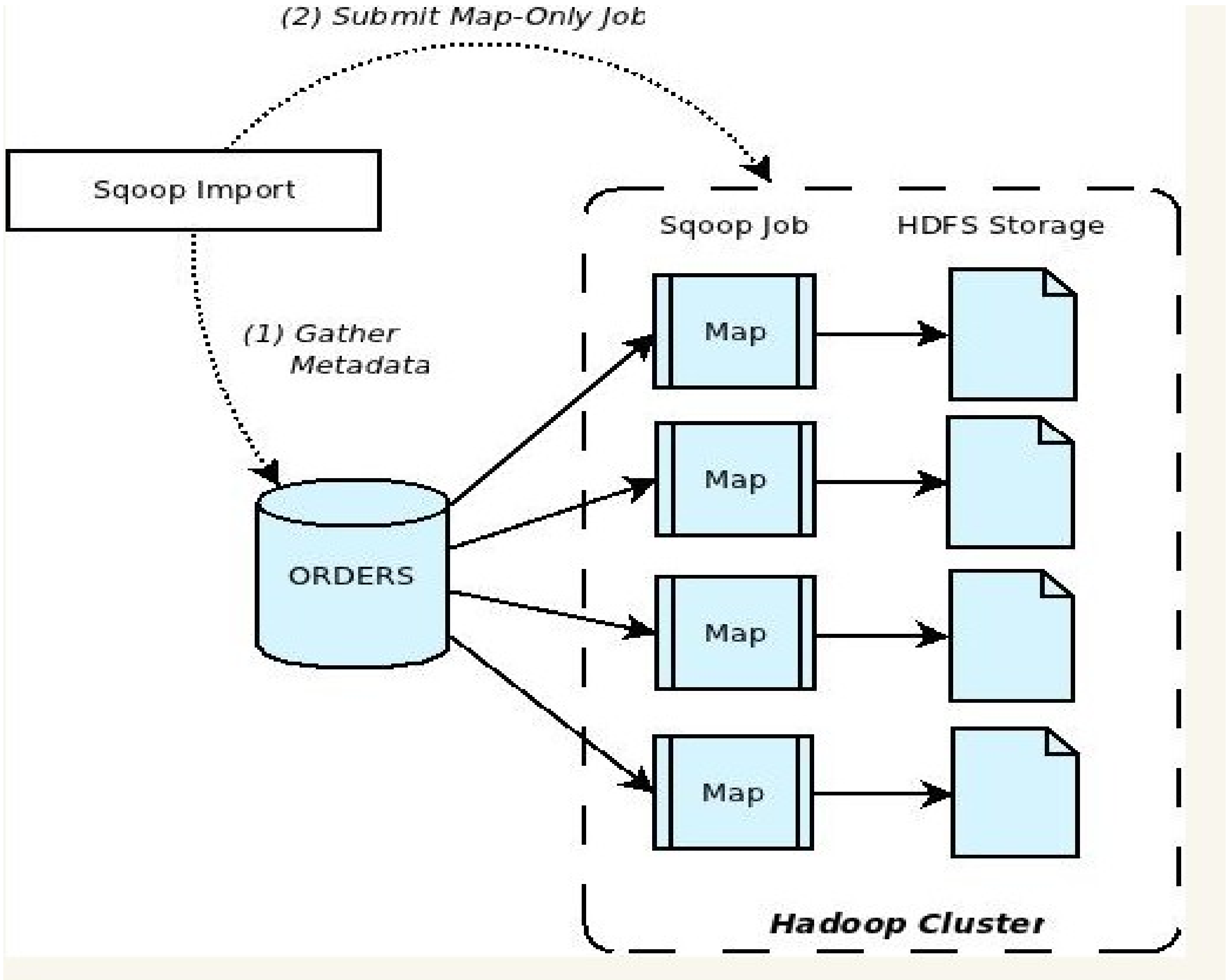
# SQOOP ARCHITECTURE



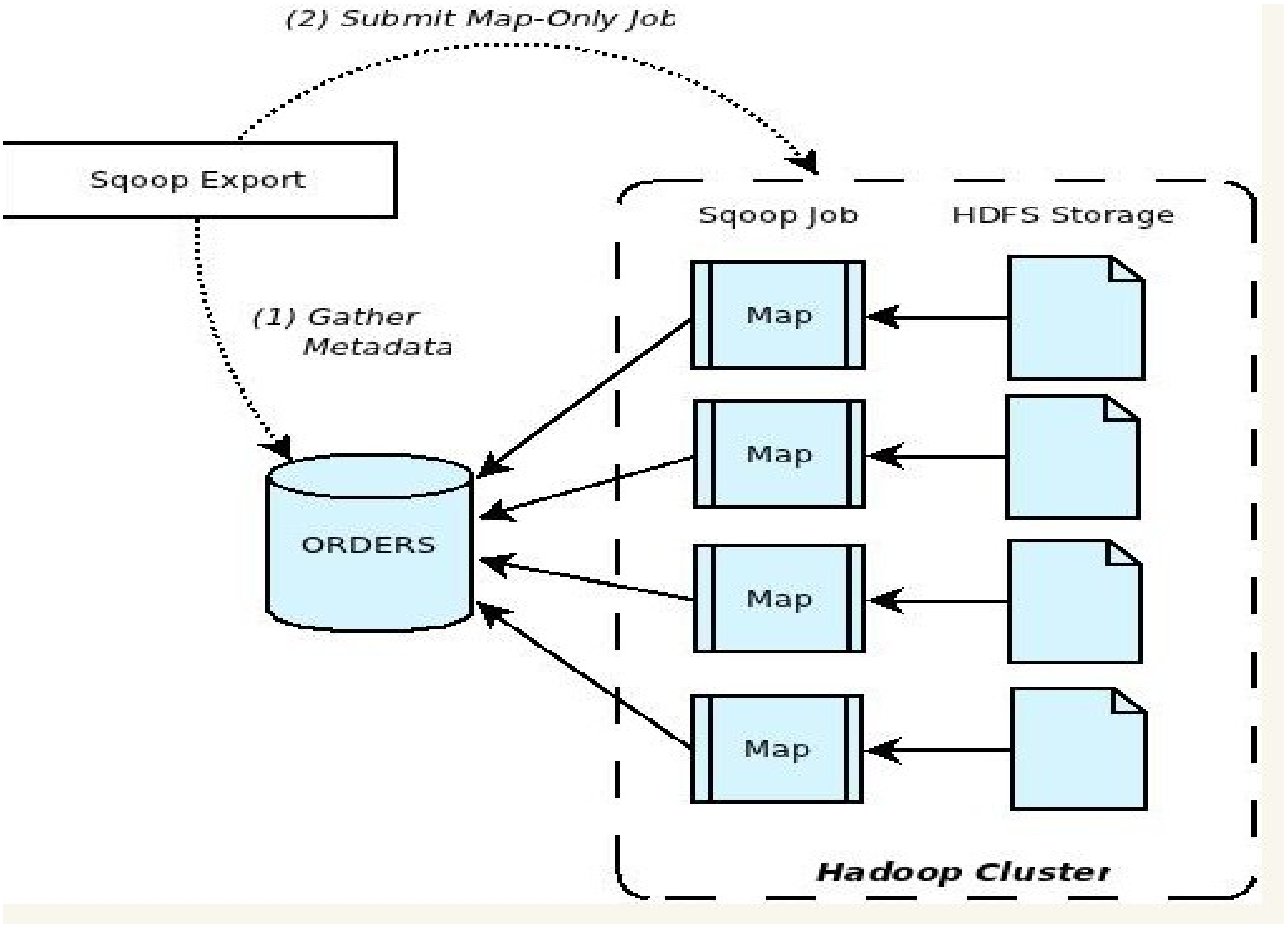
## SQOOP IMPORT & EXPORT

- 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.
- **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.

# IMPORT


(2) Submit Map-Only Job
Sqoop Import
(1) Gather Metadata
ORDERS
Sqoop Job
HDFS Storage
Map
Map
Map
Map
Hadoop Cluster

# EXPORT


(2) Submit Map-Only Job
Sqoop Export
(1) Gather Metadata
ORDERS
Sqoop Job
HDFS Storage
Map
Map
Map
Map
Hadoop Cluster

## TOOLS

- To List Commands in sqoop

```
$ sqoop help
```

- To List Generic Commands and Specific Arguments

```
$ sqoop import help
```

## TOOLS-GENERAL

- Options Files to Pass Arguments
- General Import Command

```
$ sqoop import --connect jdbc:mysql://localhost/sqoopdb --username root --table emp
```

  – Rather than repeat the import command along with connection related input required, each time, you can pass an options file as an argument to sqoop. Create a text file, as follows, and save it someplace, locally on the node you are running the sqoop client on.

```
$ sqoop import --options-file
```

- where the options file /users/geouser/import.txt contains the following:

```
import
--connect
jdbc:mysql://localhost/sqoopdb
--username
root
```

- Sqoop invocation for import can be specified by

```
$ sqoop --options-file /users/geouser/import.txt --table emp
```

## COMMAND OVERVIEW

- --connect Argument
  - 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
  - *$ sqoop import --connect jdbc:mysql://localhost/sqoopdb*
    - This string will connect to a MySQL database named sqoopdb on the host database.example.com. It’s important that you do use the URL localhost if you intend to use Sqoop with a single node system.

## COMMAND OVERVIEW

- --username Argument
    - We might need to authenticate against the database before you can access it. We can use the --username to supply a username to the database
- Database Password
    - Sqoop provides couple of different ways to supply a password, secure and non-secure, to the database
  - Secure way of supplying password to the database
    - --password-file
  - Protecting password from preying eyes
    - -P
  - --password Argument.

# COMMAND OVERVIEW

- --pasword-file
  - 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.
- -P
  - Hadoop 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
- --password
  - 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
  -

## TOOLS-LIST

- Sqoop list commands
  - List Databases

```
$ sqoop list-databases
--connect jdbc:mysql://localhost/
--username root
--password password
```

  - List Tables

```
$ sqoop list-tables
--connect jdbc:mysql://localhost/sqoopdb
--username root
--password password
```

# TOOLS-IMPORT

- Import:
  - 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.
  - 
    ```
    $ sqoop import \
    --connect jdbc:mysql://localhost/sqoopdb \
    --username root
    --password password \
    --table emp \
     -m 1 \
    --target-dir /user/geouser/sqoop/import
    ```

# IMPORT ARGUMENTS

- --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

# IMPORT ARGUMENTS cont..

- --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.

# IMPORT ARGUMENTS cont..

- --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

# IMPORT COLUMNS

- Import all rows of a table in mySQL, but specific columns of the table

```
 $ sqoop --options-file /user/geouser/user.txt \
--table  emp \
--columns "EMP_NO,DEPT_NO,FROM_DATE,TO_DATE" \
-m 1 \
--target-dir /user/geouser/sqoop/emp/columns
```

# IMPORT - WHERE

- Sqoop Import with Where Clause

```
sqoop import \
--connect jdbc:mysql://localhost/sqoopdb \
--username root \
--password password \
--table emp \
--where “EMP_NO=5”
```

## IMPORT - WHERE

- We 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>.
- We 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.

## IMPORT - QUERY

- 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 ***--quer***y 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***

## IMPORT - QUERY

- Sqoop Import with Free Form Query
  - *sqoop import*

    *--connect jdbc:mysql://localhost/sqoopdb*

    *--username root*

    *--password password*

    *--query “select * from emp where EMP_NO= 5 and $CONDITIONS*

    *--m 1*

    *--target-dir '/user/geouser/sqoop/query'*

## 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.
- 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.
- 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 EMP_NO***.

# Controlling Import

- 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.
- Sqoop will import a table named emp to a directory named employee inside your home directory in HDFS. For example, if your username is geouser, then the import tool will write to /user/geouser/employee/(files). You can adjust the parent directory of the import with the ***--warehouse-dir*** argument.
- **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)
  - *$ sqoop import ... --map-column-java id=String,value=Integer*

## Incremental Imports

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

## File Formats

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

# Sequence & Avro File Format

- **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.
- **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.

## Delimiters

- 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.
- --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-enclosedby:**'**
- --optionally-enclosed-by <char>
  - Sets a field enclosing character

# Delimiters cont.,

- 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.

# Import 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.
- 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.

# Hive arguments

- --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.

## Import 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***.

## Import 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***.

## HBase Arguments

- --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

# Import – All Tables

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

## Tools - Export

- 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.
- *$ sqoop export (generic-args) (export-args)*

## Tools - Export

- Export Command

```
sqoop export \
--connect jdbc:mysql://localhost/sqoopdb \
--username root \
--password password \
--table employees_export
--staging-table employees_exp_stg \
--clear-staging-table \
--export-dir /user/geouser/sqoopl/emp
```

## Export Arguments

- --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

# Export Arguments cont..

- --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.

## Command Overview - Export

- 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.
- The ***--num-mappers*** or ***-m*** arguments control the number of map tasks, which is the degree of parallelism used.
- ***--direct*** argument to specify this codepath. This may be higher-performance than the standard JDBC codepath.
- 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.

## Commands Overview - Export

- 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.
- 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.

## 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.
- 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***.
- 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.

## Sqoop Job

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

# Sqoop Job

- Job Arguments:
  - --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

## Sqoop Job

- Sqoop Job Command

```
sqoop job  \
--create test1
-- \
--connect jdbc:mysql://localhost/sqoop_ex \
--username root \
--password password \
--table course_view \
--hive-import \
--incremental append
--check-column course_code
--last-value 5
--m 1
--target-dir '/user/hive/warehouse/hive_import_db'
```

- To Execute the Job

```
$ sqoop job --exec test1;
```