APACHE SQOOP

What is SQOOP:

Apache Sqoop(TM) is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured data stores such as relational databases and vice versa. This processing can be done with MapReduce programs or other higher-level tools such as Hive. (It's even possible to use Sqoop to move data from a database into HBase.) When the final results of an analytic pipeline are available, Sqoop can export these results back to the data store for consumption by other clients.

Apache Sqoop is a tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.
Sqoop imports data from external structured datastores into HDFS or related systems like Hive and HBase.

Sqoop can also be used to export data from Hadoop and export it to external structured datastores such as relational databases and enterprise data warehouses.

Sqoop works with relational databases such as: Teradata, Netezza, Oracle, MySQL, Postgres, and HSQLDB.

Why Sqoop

As more organizations deploy Hadoop to analyse vast streams of information, they may find they need to transfer large amount of data between Hadoop and their existing databases, data warehouses and other data sources.

Loading bulk data into Hadoop from production systems or accessing it from map-reduce applications running on a large cluster is a challenging task since transferring data using scripts is an inefficient and time-consuming task

Sqoop is basically an ETL Tool used to copy data between HDFS and SQL databases

Import SQL data to HDFS for archival or analysis

Export HDFS to SQL (e.g : summarized data used in a DW fact table)

Sqoop Import

The import tool imports individual tables from RDBMS to HDFS. Each row in a table is treated as a record in HDFS. All records are stored as text data in text files or as binary data in Avro and Sequence files.

Sqoop Export

The export tool exports a set of files from HDFS back to an RDBMS. The files given as input to Sqoop contain records, which are called as rows in table. Those are read and parsed into a set of records and delimited with user-specified delimiter.

Brief History:

Sqoop successfully graduated from the Incubator in March of 2012 and is now a Top-Level Apache project. Current version 1.4.6.

Features of Sqoop

Designed to efficiently transfer bulk data between Apache Hadoop and structured datastores such as relational databases, Apache Sqoop:

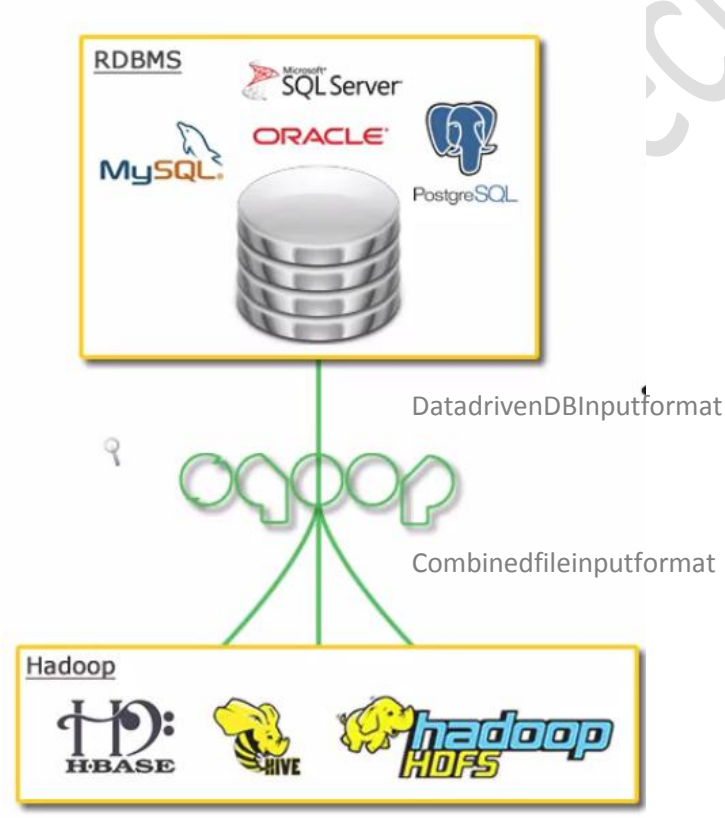
Allows data imports from external datastores and enterprise data warehouses into Hadoop

Parallelizes data transfer for fast performance and optimal system utilization

Copies data quickly from external systems to Hadoop

Makes data analysis more efficient, Mitigates excessive loads to external systems.

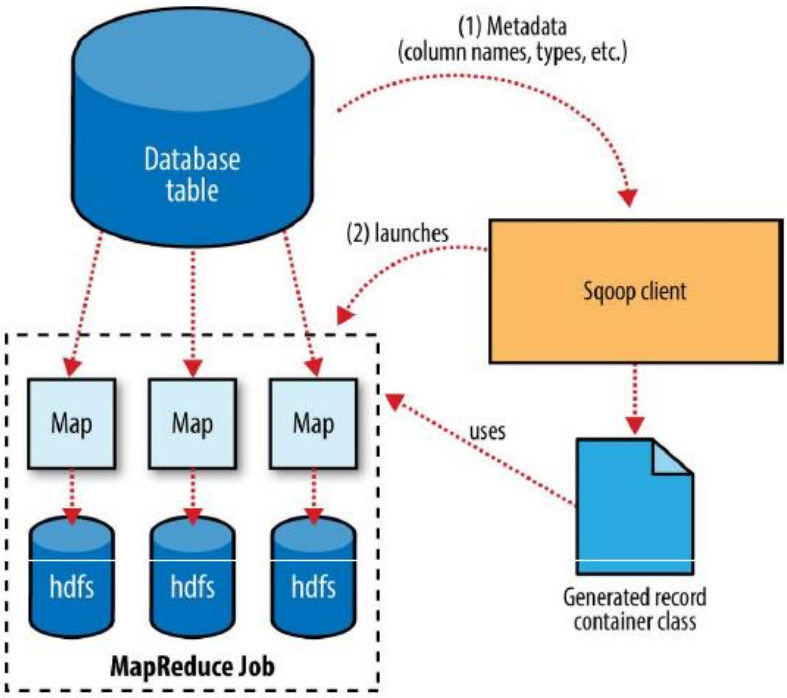
SQOOP Overview



SQOOP Properties

1. Supports only mappers.
2. Default 4 mappers.
3. We can define number of mappers with – splitby columns or m 1 or if we have primary key in the table.
4. Does not support composite key columns.
5. DatadrivenDBInputformat is the input format.
6. Imports and Consistency Performs only committed read. (READ COMMITED).
7. Incremental Imports can be done.
8. Direct-Mode Imports supported by databases such as MySQL, PostgreSQL, Oracle, and Netezza.
9. Imports Large Objects as lobfile.
10. Export works with combinedfileinputformat.
11. Performs merge operation at export.

SQOOP IMPORT ARCHITECTURE



Sqoop is written in Java. Java provides an API called Java Database Connectivity, or JDBC, that allows applications to access data stored in an RDBMS as well as to inspect the nature of this data. Most database vendors provide a JDBC *driver* that implements the JDBC API and contains the necessary code to connect to their database servers. Before the import can start, Sqoop uses JDBC to examine the table it is to import. It retrieves a list of all the columns and their SQL data types. These SQL types (VARCHAR,

INTEGER, etc.) can then be mapped to Java data types (String, Integer, etc.), which will hold the field values in MapReduce applications. Sqoop's code generator will use this information to create a table-specific class to hold a record extracted from the table.
More critical to the import system's operation, though, are the serialization methods that form the DBWritable interface, which allow the Widget class to interact with JDBC:

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

JDBC's ResultSet interface provides a cursor that retrieves records from a query; the readFields() method here will populate the fields of the Widget object with the columns from one row of the ResultSet's data. The write() method shown here allows Sqoop to insert new Widget rows into a table, a process called *exporting*. Exports are discussed in Performing an Export.
The MapReduce job launched by Sqoop uses an InputFormat that can read sections of a table from a database via JDBC. The DataDrivenDBInputFormat provided with Hadoop partitions a query's results over several map tasks.
Reading a table is typically done with a simple query such as:
SELECT *col1*,*col2*,*col3*,... FROM *tableName*
But often, better import performance can be gained by dividing this query across multiple nodes. This is done using a *splitting column*. Using metadata about the table, Sqoop will guess a good column to use for splitting the table (typically the primary key for the table, if one exists). The minimum and maximum values for the primary key column are retrieved, and then these are used in conjunction with a target number of tasks to determine the queries that each map task should issue.
For example, suppose the widgets table had 100,000 entries, with the id column containing values 0 through 99,999. When importing this table, Sqoop would determine that id is the primary key column for the table. When starting the MapReduce job, the DataDrivenDBInputFormat used to perform the import would issue a statement such as SELECT MIN(id), MAX(id) FROM widgets. These values would then be used to interpolate over the entire range of data. Assuming we specified that five map tasks should run in parallel (with **-m 5**), this would result in each map task executing queries such as
SELECT id, widget_name, ... FROM widgets WHERE id >= 0 AND id < 20000,
SELECT id, widget_name, ... FROM widgets WHERE id >= 20000 AND id < 40000,
and so on.

The choice of splitting column is essential to parallelizing work efficiently. If the id column were not uniformly distributed (perhaps there are no widgets with IDs between 50,000 and 75,000), then some map tasks might have little or no work to perform, whereas others would have a great deal. Users can specify a particular splitting column when running an import job (via the --split-by argument), to tune the job to the data's actual distribution. If an import job is run as a single (sequential) task with **-m 1**, this split process is not performed.
After generating the deserialization code and configuring the InputFormat, Sqoop sends the job to the MapReduce cluster. Map tasks execute the queries and deserialize rows from the ResultSet into instances of the generated class, which are either stored directly in SequenceFiles or transformed into delimited text before being written to HDFS.

Text and Binary File Formats

Sqoop is capable of importing into a few different file formats. Text files (the default) offer a human-readable representation of data, platform independence, and the simplest structure. However, they cannot hold binary fields (such as database columns of type VARBINARY), and distinguishing between null values and String-based fields containing

the value "null" can be problematic (although using the --null-string import option allows you to control the representation of null values).
To handle these conditions, Sqoop also supports SequenceFiles, Avro datafiles, and Parquet files. These binary formats provide the most precise representation possible of the imported data. They also allow data to be compressed while retaining MapReduce's ability to process different sections of the same file in parallel. However, current versions of Sqoop cannot load Avro datafiles or SequenceFiles into Hive (although you can load Avro into Hive manually, and Parquet can be loaded directly into Hive by Sqoop). Another disadvantage of SequenceFiles is that they are Java specific, whereas Avro and Parquet files can be processed by a wide range of languages.

Controlling the Import

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

Imports and Consistency

When importing data to HDFS, it is important that you ensure access to a consistent snapshot of the source data. (Map tasks reading from a database in parallel are running in separate processes. Thus, they cannot share a single database transaction.) The best way to do this is to ensure that any processes that update existing rows of a table are disabled during the import.

Incremental Imports

It's common to run imports on a periodic basis so that the data in HDFS is kept synchronized with the data stored in the database. To do this, there needs to be some way of identifying the new data. Sqoop will import rows that have a column value (for the column specified with --check-column) that is greater than some specified value (set via --last-value).
The value specified as --last-value can be a row ID that is strictly increasing, such as an AUTO_INCREMENT primary key in MySQL. This is suitable for the case where new rows are added to the database table, but existing rows are not updated. This mode is called *append* mode, and is activated via --incremental append. Another option is time-based incremental imports (specified by --incremental lastmodified), which is appropriate when existing rows may be updated, and there is a column (the check column) that records the last modified time of the update.
At the end of an incremental import, Sqoop will print out the value to be specified as --last-value on the next import. This is useful when running incremental imports manually, but for running periodic imports it is better to use Sqoop's saved job facility, which automatically stores the last value and uses it on the next job run. Type sqoop job

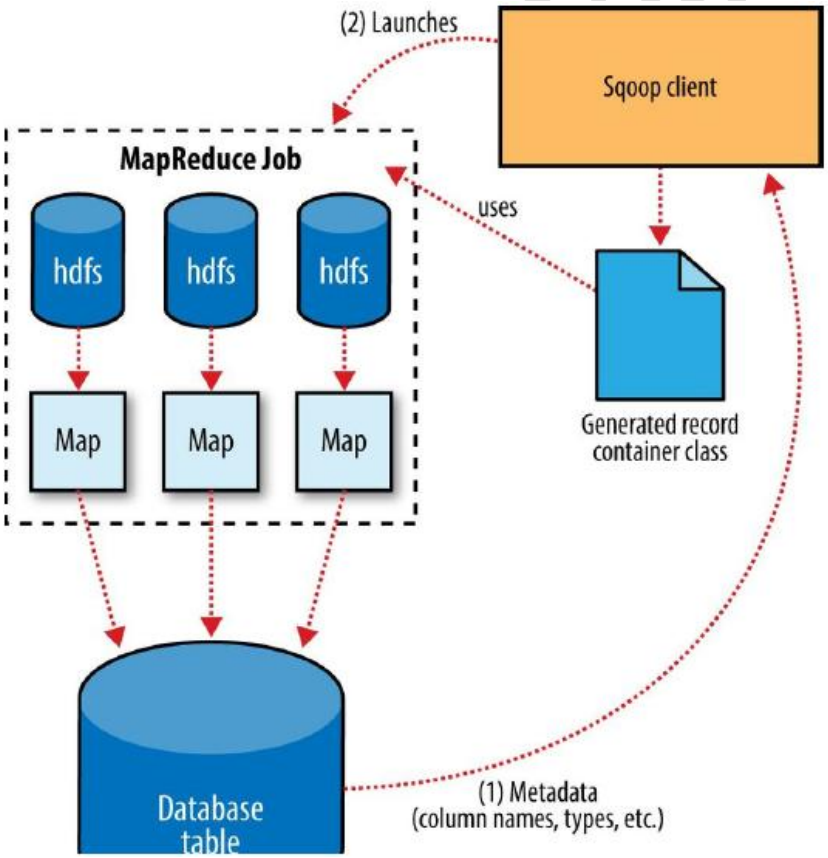
--help for usage instructions for saved jobs.

Direct-Mode Imports

Sqoop's architecture allows it to choose from multiple available strategies for performing an import. Most databases will use the DataDrivenDBInputFormat-based approach described earlier. Some databases, however, offer specific tools designed to extract data quickly. For example, MySQL's mysqldump application can read from a table with greater throughput than a JDBC channel. The use of these external tools is referred to as *direct mode* in Sqoop's documentation. Direct mode must be specifically enabled by the user (via the --direct argument), as it is not as general purpose as the JDBC approach. (For example, MySQL's direct mode cannot handle large objects, such as CLOB or BLOB columns, and that's why Sqoop needs to use a JDBC-specific API to load these columns into HDFS.)

For databases that provide such tools, Sqoop can use these to great effect. A direct-mode import from MySQL is usually much more efficient (in terms of map tasks and time required) than a comparable JDBC-based import. Sqoop will still launch multiple map tasks in parallel. These tasks will then spawn instances of the mysqldump program and read its output. Sqoop can also perform direct-mode imports from PostgreSQL, Oracle, and Netezza. Even when direct mode is used to access the contents of a database, the metadata is still queried through JDBC.

SQOOP EXPORT:



The Sqoop performs exports is very similar in nature to how Sqoop performs imports Before performing the export, Sqoop picks a strategy based on the database connect string. For most systems, Sqoop uses JDBC. Sqoop then generates a Java class based on the target table definition. This generated class has the ability to parse records from text files and insert values of the appropriate types into a table (in addition to the ability to read the columns from a ResultSet). A MapReduce job is then launched that reads the source datafiles from HDFS, parses the records using the generated class, and executes the chosen export strategy. The JDBC-based export strategy builds up batch INSERT statements that will each add multiple records to the target table. Inserting many records per statement performs much better than executing many single-row INSERT statements on most database systems. Separate threads are used to read from HDFS and communicate with the database, to ensure that I/O operations involving different systems are overlapped as much as possible.

For MySQL, Sqoop can employ a direct-mode strategy using mysqlimport. Each map task spawns a mysqlimport process that it communicates with via a named FIFO file on the local filesystem. Data is then streamed into mysqlimport via the FIFO channel, and from there into the database.
Whereas most MapReduce jobs reading from HDFS pick the degree of parallelism (number of map tasks) based on the number and size of the files to process, Sqoop's export system allows users explicit control over the number of tasks. The performance of the export can be affected by the number of parallel writers to the database, so Sqoop uses the CombineFileInputFormat class to group the input files into a smaller number of map tasks.

Exports and Transactionality

Due to the parallel nature of the process, often an export is not an atomic operation. Sqoop will spawn multiple tasks to export slices of the data in parallel. These tasks can complete at different times, meaning that even though transactions are used inside tasks, results from one task may be visible before the results of another task. Moreover, databases often use fixed-size buffers to store transactions. As a result, one transaction cannot necessarily contain the entire set of operations performed by a task. Sqoop commits results every few thousand rows, to ensure that it does not run out of memory. These intermediate results are visible while the export continues. Applications that will use the results of an export should not be started until the export process is complete, or they may see partial results. To solve this problem, Sqoop can export to a temporary staging table and then, at the end of the job — if the export has succeeded — move the staged data into the destination table in a single transaction. You can specify a staging table with the --staging-table option.The staging table must already exist and have the same schema as the destination. It must also be empty, unless the --clear-staging-table option is also supplied.

SQOOP Best Practices and Performance tuning

Import:

1. **Definate number of mappers: --num-mappers**
    a. More mappers can lead to faster jobs, but only up to a saturation point. This varies per table, job parameters, time of day and server availability.
    b. Too many mappers will increase the number of parallel sessions on the database, hence affect source DB performance affecting the regular workload of the DB.

2. **Use Direct mode for all available DBs.**

a. Rather than using the JDBC interface for transferring data, the direct mode delegatesthe job of transferring data to the native utilities provided by the database vendor. For Eg. In the case of MySQL, the mysqldump and mysqlimport will be used for retrieving data from the database server or moving data back.
b. Escape characters, type mapping, column and row delimiters may not be supported. Binary formats don't work.

3. Splitting Data --split-by: Boundary Queries --boundary-query

a. By default, the primary key is used. Prior to starting the transfer, Sqoop will retrieve the min/max values for this column. Changed column with the --split-by parameter
b. Boundary Queries - What if your split-by column is skewed, table is not indexed or can be retrieved from another table?
If --split-by is not giving you the optimal performance you can use this to improve the performance further to Use a boundary query to create the splits using the option --boundary-query

Eg. sqoop import \
--connect 'jdbc:mysql://.../...' \
**--direct **
--username uname --password pword \
--hive-import \
--hive-table query_import \
**--boundary-query 'SELECT 0, MAX(id) FROM a' **
--query 'SELECT a.id, a.name, b.id, b.name FROM a, b WHERE a.id = b.id AND **$CONDITIONS'**\
--num-mappers 3
**--split-by a.id **
--target-dir /data/import \

4. **Using $CONDITIONS**

a. $CONDITIONS is used by Sqoop process, it will replace with a unique condition expression internally to get the data-set. If you run a parallel import, the map tasks will execute your query with different values substituted in for $CONDITIONS. For Eg. Above query will execute parallel like this.
SELECT a.id, a.name, b.id, b.name FROM a, b WHERE a.id = b.id AND **a.id BETWEEN 0 AND 10**;
SELECT a.id, a.name, b.id, b.name FROM a, b WHERE a.id = b.id AND **a.id BETWEEN 11 AND 20;**
SELECT a.id, a.name, b.id, b.name FROM a, b WHERE a.id = b.id AND **a.id BETWEEN 21 AND 30;**

Export:

5. BATCH mode --batch

a. Sqoop performs export row by row if we don't leverage batch mode option.
b. Enabling batch mode will export more than one row at a time as batch of rows.

6. Specify the number of records to export -Dsqoop.export.records.per.statement=10

a. The above option will define how many number of rows should be used in each insert statements.

7. Specify the number of records per transaction - -Dsqoop.export.statements.per.transaction=10

a. The above option will define how many number of rows should be used in each transactions.

8. Data Consistency --staging-table

a. In order to provide the consistent data access for the users in end database, using a staging table, Sqoop will first export all data into this staging table instead of the main table that is present in the parameter --table. Sqoop opens a new transaction to move data from the staging table to the final destination, if and only if all
parallel tasks successfully transfer data.