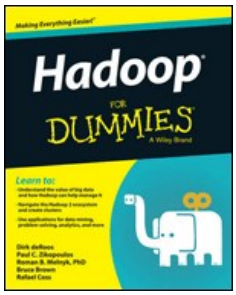

Hadoop for Dummies

by Dirk deRoos et al.

Chapter 14: Integrating Hadoop with Relational Databases Using Sqoop

In This Chapter

- Introducing Sqoop
- Looking at the nuts and bolts of Sqoop
- Importing data with Sqoop
- Exporting data with Sqoop
- Customizing your Sqoop input and output formats
- Looking ahead to Sqoop 2.0

Performing analytics on large, diverse data sets is a natural fit for Apache Hadoop. The whole point of the Hadoop File System (HDFS) is that it excels at providing a massively scalable, diverse data store that, when combined with the many analytic tools available on the Hadoop platform — from Map Reduce to Mahout and others — gives you a lean, mean, analytics machine when you hitch your data store wagon to Apache Hadoop.

This rosy picture presents a slight problem, however: It turns out that most of the world's structured data is already stored in relational database management systems (RDBMSs), and it's common practice to leverage structured query language (SQL, for short) for data transformation, processing, and analysis — and SQL is decidedly *not* a natural fit for Apache Hadoop. The Hadoop community knew what it was getting into, though, and planned to provide support for structured relational data — an SQL "fix," as it were — early on. Folks have been looking at combining and then analyzing field sensor data with the corresponding product data stored in a RDBMS or data warehouse, for example, a use case that places Apache Hive, with its SQL-like HiveQL, at its center.

It sounds like a great idea, but you may be wondering how, in this particular use case, you can get the data from the RDBMS onto the Apache Hadoop cluster, where Apache Hive can then do its magic. What's the "scoop" on that, you ask? (How's that for setting up a pun that refers to the chapter title?)

The answer, of course, is "*SQ*L to Had*oop*," or Sqoop, for short. Sqoop was first announced in 2009 by Aaron Kimball as a database import tool for Hadoop, and three years later (March 2012, to be exact), Sqoop became a top-level Apache project. The glory of Sqoop lies in the fact that it not only allows you to import relational data but also provides an export mechanism. The result is that Sqoop can provide an efficient mechanism for loading an RDBMS table by exporting data stored in HDFS, a use case perfectly suited for scenarios where you make use of Hadoop as an enterprise data warehouse (EDW) preprocessing engine. (See Chapter 11 for more on that scenario.)

Sqoop has grown a lot since its introduction in 2009. Along the way, Apache Sqoop committers have also added import support for Hive and HBase, making Sqoop a powerful addition to the Apache Hadoop ecosystem. In this chapter, you get the chance to explore the old and the new of Sqoop, from imports to exports to other, jazzier Sqoop tools. (You'll also come across a ton of hands-on examples.)

The Principles of Sqoop Design

When it comes to Sqoop, a picture is often worth a thousand words, so check out Figure 14-1, which gives you a bird's-eye view of the Sqoop architecture.

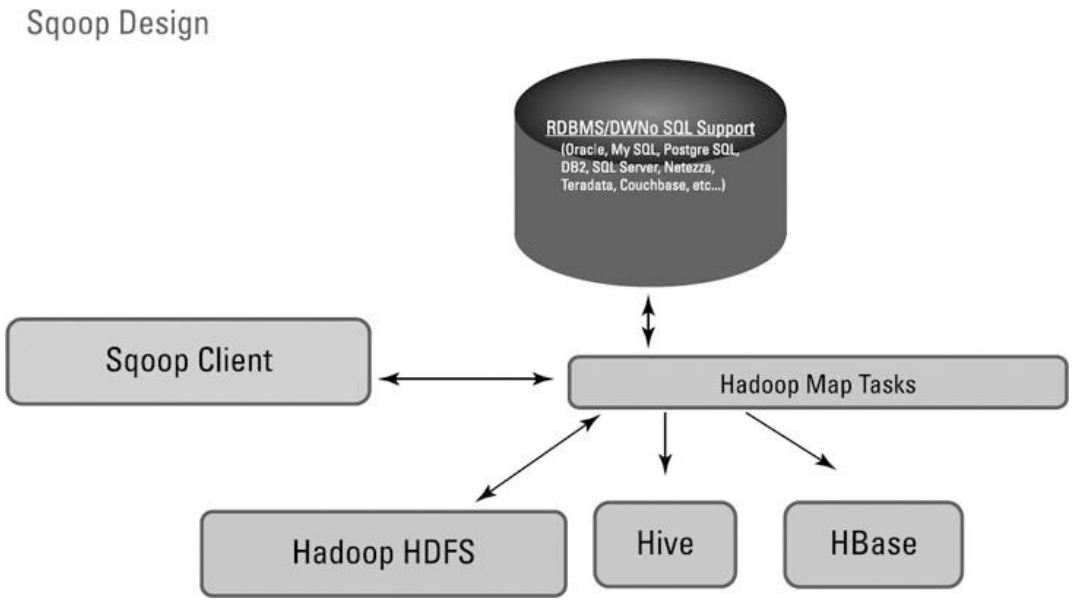

Figure 14-1: Sqoop design

The idea behind Sqoop is that it leverages *map* tasks — tasks that perform the parallel import and export of relational database tables — right from within the Hadoop MapReduce framework. This is good news because the MapReduce framework provides fault tolerance for import and export jobs along with parallel processing! You'll appreciate the fault tolerance if there is a failure during a large table import or export because the MapReduce framework will recover without requiring you to start the process all over again. (For more information on the MapReduce framework, see Chapter 6.)

REMEMBER Sqoop can import data to Hive and HBase. Note, however, that the arrows to Hive and HBase point in only one direction in Figure 14-1. Data stored in any relational database with JDBC support can be directly imported into the Hive or HBase systems with Sqoop. Exports, however, are performed from data stored in HDFS. Therefore, if you need to export your Hive tables, you point Sqoop to HDFS directories that store your Hive tables. If you need to export HBase tables, you first have to export them to HDFS and then execute the Sqoop export command.

Scooping up Data with Sqoop

Sqoop provides Hadoop with export and import capability to and from any RDBMS or data warehouse (DW) that supports the Java Database Connectivity (JDBC) application programming interface (API) suite. All major RDBMS and DW vendors generally provide JDBC-compliant drivers for their products. In addition, Sqoop releases are bundled with special connector technology for a variety of popular products. As of this writing, Sqoop version 1.4.4 provides special connectors for MySQL, PostgreSQL, Oracle, Microsoft SQL Server, DB2, and Netezza. These special connectors take advantage of specific features within the individual database systems in order to improve import/export performance and functionality. Additionally, third-party connectors are available that aren't bundled with Sqoop for other NoSQL data store and data warehouse providers (Couchbase and Teradata from Cloudera, for example). Sqoop also includes a generic JDBC connector that only supports the Java JDBC API.

Connectors and Drivers

Sqoop connectors generally go hand in hand with a JDBC driver. Sqoop does not bundle the JDBC drivers because they are usually proprietary and licensed by the RDBMS or DW vendor. So there are three possible scenarios for Sqoop, depending on the type of data management system (RDBMS, DW, or NoSQL) you are trying to interact with. Let's take a look at each one:

- **Your data management system is supported by one of the bundled Sqoop connectors listed above**. In this case, you need to acquire the JDBC driver from your data management system provider and install the `.jar` file associated with it in your `$SQOOP_HOME/lib` directory. (`$SQOOP_HOME` is an environment variable that refers to the directory pathname on your system where you install Apache Sqoop.) For the hands-on examples shown in this chapter, we installed the `mysql-connector-java-5.1.26-bin.jar` file from http://dev.mysql.com/downloads/connector in our `$SQOOP_HOME/lib` directory.
- **Sqoop does not include a connector for your database management system**. That means you need to download one from a 3rd party vendor, along with a JDBC driver if the connector requires one. (Couchbase and Teradata both do, for example.)
- **Your database management system does not provide a Sqoop connector but a JDBC driver is available**. In this case, you leverage Sqoop's generic JDBC connector and download and install your vendor's JDBC driver.

TIP For an in-depth discussion of Sqoop connectors and drivers, see the following blog entry: https://blogs.apache.org/sqoop/date/201309. For the latest release, documentation, and connector information, check out http://sqoop.apache.org.

Importing Data with Sqoop

Ready to dive into importing data with Sqoop? Start by taking a look at Figure 14-2, which illustrates the steps in a typical Sqoop import operation from an RDBMS or a data warehouse system. Nothing too complicated here — just a typical Products data table from a (typical) fictional company being imported into a typical Apache Hadoop cluster from a typical data management system (DMS).

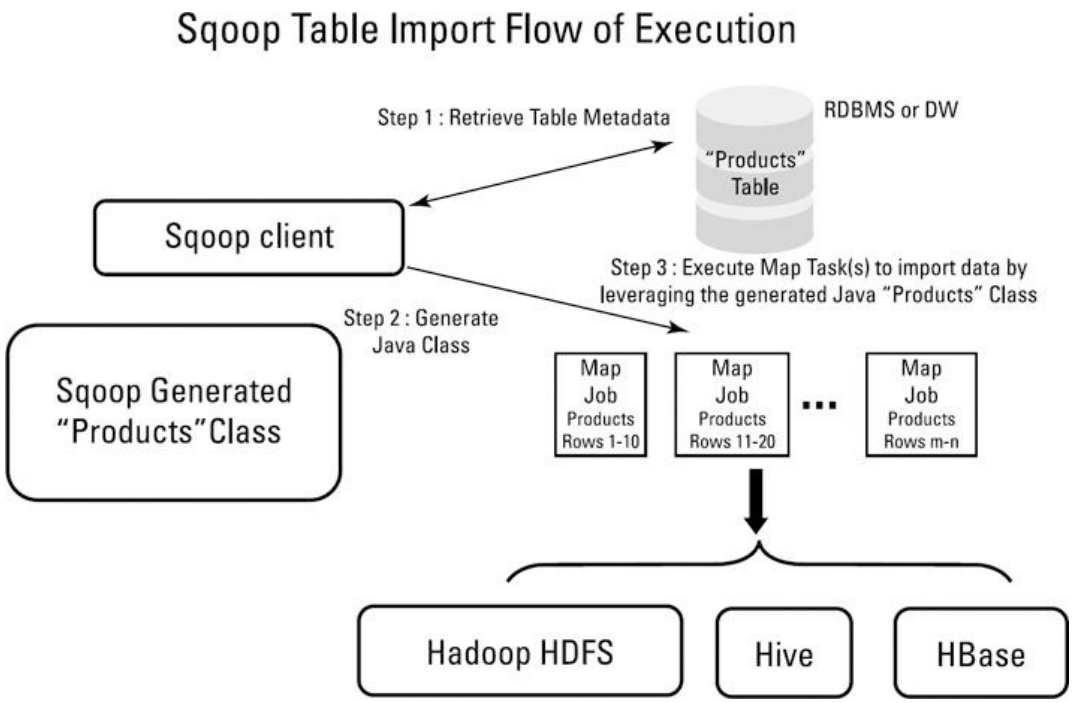

Figure 14-2: The Sqoop import flow of execution

During Step 1, Sqoop uses the appropriate connector to retrieve the Products table metadata from the target DMS. (The metadata is used to map the data types from the Products table to data types in the Java language.) Step 2 then uses this metadata to generate and compile a Java class that will be used by one or more map tasks to import the actual rows from the Products table. Sqoop saves the generated Java class to temp space or to a directory you specify so that you can leverage it for the subsequent processing of your data records.

TECHNICAL STUFF The Sqoop generated Java code that is saved for you is like the gift that keeps on giving! With this code, Sqoop imports records from the DMS and stores them to HDFS using one of three formats that you can pick: binary Avro data, binary sequence files, or delimited text files. Afterwards, this code is available to you for subsequent data processing. Sequence files are a natural choice if you're importing binary data types and you'll need the generated Java class to serialize and deserialize your data later on — perhaps for MapReduce processing or exporting. (More on exporting later — right now, we're focusing on imports.) Avro data — based on Apache's own serialization framework — is useful if you need to interact with other applications after the import to HDFS. If you choose to store your imported data in delimited text format, you may find the generated Java code valuable later on as you parse and perform data format conversions on your new data. Later in this chapter, you'll see that the generated code also helps you merge data sets after Sqoop import operations and the final example in this chapter illustrates how the generated Java code can help avoid ambiguity when processing delimited text data.

Finally, during Step 3, Sqoop divides the data records in the Products table across a number of map tasks (with the number of mappers optionally specified by the user) and imports the table data into HDFS, Hive, or HBase.

Importing Data into HDFS

Figure 14-2 gives you the big-picture view of the Sqoop data import process. Time to look at the process in a bit more detail, with the help of a few hands-on examples.

Figure 14-3 helps you imagine a relational database used by a fictional service company that has been taking (you guessed it) Apache Hadoop service calls and now wants to move some of its data onto Hadoop to run Hive queries, leverage HBase scalability and performance, and run text analytics on its customer's problem descriptions.

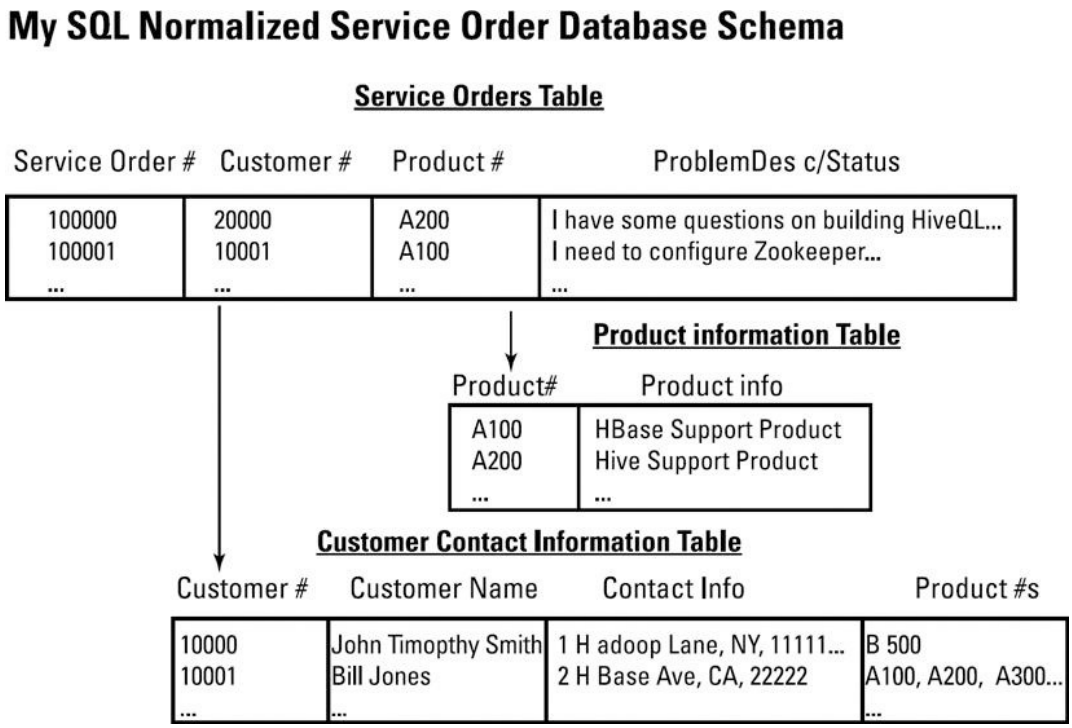

Figure 14-3: A Service Order Database schema

REMEMBER We discuss the Service Order Database in Chapter 12 and explain how it might be converted to an HBase schema. Sqoop is the tool you'll want to use to import data from relational tables into HBase tables on Hadoop.

In Listing 14-1 we show the MySQL commands we used to build the Service Order Database you see in Figure 14-3. (We filled in a couple records in the diagram shown in Figure 14-3 to make things crystal clear.) We installed a MySQL RDBMS that we could import from and export to using Sqoop. Since these commands also show you the data we load into our Service Order Database, we'll be referring back to this listing several times in this chapter to confirm that our Sqoop examples work properly.

Listing 14-1: MySQL Commands to Build the Service Order Database

```
/* Create the Service Orders Database */

CREATE DATABASE serviceorderdb;
USE serviceorderdb;

/* Create the Product Information Table */

CREATE TABLE productinfo(
productnum CHAR (4) PRIMARY KEY,
productdesc VARCHAR(100)
);

/* Create the Customer Contact Information Table */

CREATE TABLE customercontactinfo(
customernum INT PRIMARY KEY,
customername VARCHAR(100),
contactinfo VARCHAR(100),
productnums SET('A100','A200','A300','B400','B500','C500','C600','D700')
);

/* Create the Service Orders Table */

CREATE TABLE serviceorders(
serviceordernum INT PRIMARY KEY,
customernum INT,
productnum CHAR(4),
status VARCHAR(100),
FOREIGN KEY (customernum) REFERENCES customercontactinfo(customernum),
FOREIGN KEY (productnum) REFERENCES productinfo(productnum)
);
```

```
/* Insert product data into the Product Information Table */

INSERT INTO productinfo VALUES ('A100', 'HBase Support Product');
INSERT INTO productinfo VALUES ('A200', 'Hive Support Product');
INSERT INTO productinfo VALUES ('A300', 'Sqoop Support Product');
INSERT INTO productinfo VALUES ('B400', 'Ambari Support Product');
INSERT INTO productinfo VALUES ('B500', 'HDFS Support Product');
INSERT INTO productinfo VALUES ('C500', 'Mahout Support Product');
INSERT INTO productinfo VALUES ('C600', 'Zookeeper Support Product');
INSERT INTO productinfo VALUES ('D700', 'Pig Support Product');

/* Insert customer data into the Customer Contact Information Table */

INSERT INTO customercontactinfo
VALUES (10000, 'John Timothy Smith', '1 Hadoop Lane, NY, 11111,
               John.Smith@xyz.com', 'B500');

INSERT INTO customercontactinfo
VALUES (10001, 'Bill Jones', '2 HBase Ave, CA, 22222',
        'A100,A200,A300,B400,B500,C500,C600,D700');

INSERT INTO customercontactinfo
VALUES (20000, 'Jane Ann Doe', '1 Expert HBase Ave, CA, 22222',
        'A100,A200,A300');

INSERT INTO customercontactinfo
VALUES (20001, 'Joe Developer', '1 Piglatin Ave, CO, 33333', 'D700');

INSERT INTO customercontactinfo
VALUES (30000, 'Data Scientist', '1 Statistics Lane, MA, 33333', 'A300,C500');

/* Enter service orders into the Service Orders Table */

INSERT INTO serviceorders
VALUES (100000, 20000, 'A200', 'I have some questions on building HiveQL queries? My Hadoop for Dumm

INSERT INTO serviceorders
VALUES (100001, 10001, 'A100', 'I need to understand how to configure Zookeeper for my HBase Cluster

INSERT INTO serviceorders
VALUES (200000, 20001, 'D700', 'I am writing some Piglatin and I have a few questions?');

INSERT INTO serviceorders
VALUES (200001, 30000, 'A300', 'How do I merge my data sets after Sqoop incremental imports?');
```

Listing 14-2 confirms that the MySQL Service Order Database has been created using the commands in Listing 14-1, and shows you the table names that we'll import from using Sqoop.

Listing 14-2: The MySQL show tables Command

```
mysql> show tables;
+--------------------------+
| Tables_in_serviceorderdb |
+--------------------------+
| customercontactinfo      |
| productinfo              |
| serviceorders            |
+--------------------------+
3 rows in set (0.00 sec)
```

Now that you have seen the MySQL Service Order Database records that are just waiting to be exploited, it's time to turn your attention to Hadoop and run your first Sqoop command. For this example, we downloaded an Apache Hadoop distribution that provides us with Sqoop, and we already had in place an HDFS as well as Hive and HBase. (For more information on setting up your Apache Hadoop environment, see Chapter 3.)

TIP You can find a thorough list of Apache Hadoop bundles at http://wiki.apache.org/hadoop/Distributions and Commercial Support.

Note, however, that we don't pull out the trusty `import` command right off the bat. Sqoop includes several handy tools along with `import` and `export`, including the `list-databases` command, which we use in Listing 14-3. Using that command, you can confirm that you have connectivity and visibility into the MySQL database.

Listing 14-3: The Sqoop list-databases Command

```
$ sqoop list-databases --connect jdbc:mysql://localhost/ \
                       --username root -P
Enter password:
13/08/15 17:21:00 INFO manager.MySQLManager: Preparing to
          use a MySQL streaming resultset.
information_schema
mysql
performance_schema
serviceorderdb
```

The `serviceorderdb` (bolded in Listing 14-3) is shown to be available, so now you can list the tables within `serviceorderdb` by using the Sqoop `list-tables` command, as shown in Listing 14-4. Notice that now we're adding the database that we want Sqoop to access in the `jdbc:mysql URL`.

Listing 14-4: The Sqoop list-tables Command

```
$ sqoop list-tables \
         --connect jdbc:mysql://localhost/serviceorderdb \
         --username root -P
Enter password:
13/08/15 17:22:01 INFO manager.MySQLManager: Preparing to
          use a MySQL streaming resultset.
customercontactinfo
productinfo
serviceorders
```

Listing 14-3 and Listing 14-4 should assure you that Sqoop now has connectivity and can access the three tables from Figure 14-3. That means you can execute your first Sqoop `import` command and target the `serviceorders` table with a clean conscience. Sqoop `import` commands have this format:

```
sqoop import (generic arguments) (import arguments)
```

With the generic arguments, you point to your MySQL database and provide the necessary login information, just as we did with the preceding `list-tables` tool. In the import arguments, you (the user) have the ability to specify what you want to import and how you want the import to be performed. In Listing 14-5, we specify the `serviceorders` table and request that one map task be used for the import using the `-m 1` CLA. (By default, Sqoop would use four map tasks, but that would be overkill for this small table and our virtual machine.) We have also specified the `--class-name` for the generated code and specified the `--bindir` where the compiled code and `.jar` file should be located. (Without these arguments, Sqoop would place the generated Java source file in your current working directory and the compiled `.class` file and `.jar file` in `/tmp/sqoop-<username>/compile`.) The class name simply derives from the table name unless you specify a name with the help of the `--class-name` command line argument (CLA). The `--target-dir` is the location in HDFS where you want the imported table to be placed.

Listing 14-5: The Sqoop import serviceorders Table Command

```
$ sqoop import \
  --connect jdbc:mysql://localhost/serviceorderdb  \
  --username root -P \
  --table serviceorders -m 1 \
  --class-name serviceorders \
  --target-dir /usr/biadmin/serviceorders-import \
  --bindir .
Enter password:
...
13/08/25 14:43:56 INFO mapreduce.ImportJobBase:
          Transferred 356 bytes in 21.0736 seconds
          (16.8932 bytes/sec)
13/08/25 14:43:56 INFO mapreduce.ImportJobBase: Retrieved
          4 records.
```

The command ran fine, so you should have the same `serviceorders` data that's shown in Listing 14-1 now stored in your HDFS as well as the generated Java files in your current working directory. Listing 14-6 shows how you can use the `hadoop fs -cat` command to verify this.

Listing 14-6: Displaying the serviceorders Table Now Stored in HDFS and Listing the Generated Java Files

```
$ hadoop fs -cat /usr/biadmin/serviceorders-import/part-m-00000
100000,20000,A200,I have some questions on building HiveQL queries? My Hadoop
          for Dummies book has not arrived yet!
100001,10001,A100,I need to configure Zookeeper for my HBase Cluster?
200000,10001,D700,I am writing some Piglatin and I have a few questions?
200001,20000,A300,How do I merge my data sets after Sqoop incremental imports?

$ ls *.jar *.java *.class
serviceorders.class  serviceorders.jar  serviceorders.java
```

In the next two listings, we show you some additional options that can help you specify in greater detail the data you want to import. Normally, Sqoop imports the entire table or tables that you specify. However, you can control the number and order of columns using the `--columns <col1, col2, …>` command line argument. You can also provide your own `SELECT` statement after the `--query` argument. In Listing 14-7, you use the `--query` argument to specify that you want to import only the names and contact information for those customers who have open service orders. (The `WHERE $CONDITIONS` token is required by Sqoop to help the map tasks divide and conquer the import operation — at the end of this section, we explain more about how Sqoop divides an import.)

Listing 14-7: The Sqoop import Command Using the --query CLA

```
sqoop import --connect jdbc:mysql://localhost/serviceorderdb \
 --username root -P -m 2 \
 --query 'SELECT customercontactinfo.customername, customercontactinfo.
              contactinfo FROM customercontactinfo JOIN
serviceorders ON customercontactinfo.customernum = serviceorders.customernum
              WHERE $CONDITIONS' \
 --split-by serviceorders.serviceordernum \
 --boundary-query "SELECT min(serviceorders.serviceordernum),
             max(serviceorders.serviceordernum) FROM serviceorders" \
 --target-dir /usr/biadmin/customers \
 --verbose
```

This Sqoop import is somewhat complex, so we want to take the time to explain it in detail and discuss how Sqoop divides up the import job. It helps to understand that, by default, Sqoop performs the following statement to decide how to divide the table rows across the map tasks for importing:

```
SQL SELECT MIN(primary key col), MAX(primary key col) FROM
           table
```

That's the default behavior in an import operation, such as the one in Listing 14-5. The exception in that listing, of course, is that the table is very small and we used just the one map task. If it were a very large table, you would want more map tasks, to get the job done faster. Now, we made Listing 14-7 more extravagant — it uses two map tasks. In this case, Sqoop requires the `--split-by` and `--boundary-query` command line arguments because the `--table` CLA has been replaced by our own query using the `--query` CLA. So we're helping Sqoop divide the work across the two map tasks we created by specifying our own boundaries for the import. In this case, we know that the `serviceorders` table has the increasing integer primary key named `serviceordernum`, which lets Sqoop divide up the work. The `--boundary-query` command line argument lets you get creative to help Sqoop meet your table import requirements, but we keep it simple in this example.

Listings 14-8 and 14-9 confirm that our two map tasks did their job. This time we have two files to view because we used two map tasks.

Listing 14-8: Output from Map Task 1

```
$ hadoop fs -cat /usr/biadmin/customers/part-m-00000
Jane Ann Doe,1 Expert HBase Ave, CA, 22222
Bill Jones,2 HBase Ave, CA, 22222
```

Listing 14-9: Output from Map Task 2

```
$ hadoop fs -cat /usr/biadmin/customers/part-m-00001
Joe Developer,1 Piglatin Ave, CO, 33333
Data Scientist,1 Statistics Lane, MA, 33333
```

You can also control which rows are imported using the `--where` argument to provide a `WHERE` clause, as shown in Listing 14-10.

Listing 14-10: The Sqoop import Command using the --where CLA with Results

```
sqoop import \
        --connect jdbc:mysql://localhost/serviceorderdb \
        --username root -P -m 1 \
        --table customercontactinfo \
        --where 'customernum >= 20000 and customernum < 30000' \
        --target-dir /user/biadmin/customers-range
$ hadoop fs -cat /user/biadmin/customers-range/part-m-00000
20000,Jane Ann Doe,1 Expert HBase Ave, CA, 22222,A100,A200,A300
20001,Joe Developer,1 Piglatin Ave, CO, 33333,D700
```

In Listing 14-10, we're back to using the default behavior, as in Listing 14-5. But because Sqoop lets us specify a `WHERE` clause using the `--where` command line argument, we download only those customers who have IDs between 20000 and 29999.

Are you getting a sense of the power and flexibility that Sqoop brings to Apache Hadoop? Big data analytics become far more valuable when combined with existing enterprise data, and Sqoop greatly simplifies and streamlines the overall process! In the preceding example, the fictional service company can now leverage the data in the `serviceorders` table, which is now stored as a flat file in HDFS, as part of a larger Hadoop text analytics or statistical analysis application.

Importing Data into Hive

For our next example, we import all of the Service Order Database directly from MySQL into Hive and run a HiveQL query against the newly imported database on Apache Hadoop. (For more information on Hive, see Chapter 13). Listing 14-11 shows you how it's done.

Listing 14-11: Hive and Sqoop commands to import the Service Order Database into Apache Hive

```
hive> create database serviceorderdb;
OK
Time taken: 1.343 seconds
hive> use serviceorderdb;
OK
Time taken: 0.062 seconds

$ sqoop import --connect jdbc:mysql://localhost/serviceorderdb \
   --username root -P \
   --table productinfo \
   --hive-import \
   --hive-table serviceorderdb.productinfo -m 1
Enter password:
...
13/08/16 15:17:08 INFO hive.HiveImport: Hive import complete.
$ sqoop import --connect jdbc:mysql://localhost/serviceorderdb \
   --username root -P \
   --table customercontactinfo \
   --hive-import \
   --hive-table serviceorderdb.customercontactinfo -m 1
Enter password:
...
13/08/16 17:21:35 INFO hive.HiveImport: Hive import complete.
$ sqoop import --connect jdbc:mysql://localhost/serviceorderdb \
   --username root -P \
   --table serviceorders \
   --hive-import \
   --hive-table serviceorderdb.serviceorders -m 1
Enter password:
...
13/08/16 17:26:56 INFO hive.HiveImport: Hive import complete.
```

When the import operations are complete, you run the `show tables` command to list the newly imported tables (see Listing 14-12), and then run a Hive query (see Listing 14-13) to show which Apache Hadoop technologies have open service orders in the database.

Listing 14-12: Confirming the Sqoop Import Operations in Apache Hive

```
hive> show tables;
OK
customercontactinfo
```

```
productinfo
serviceorders
Time taken: 0.074 seconds
```

Listing 14-13: HiveQL Query to Determine Which Products Have Open Service Orders Against Them

```
hive> SELECT productdesc FROM productinfo
   > INNER JOIN serviceorders
   > ON productinfo.productnum = serviceorders.productnum;
...
OK
HBase Support Product
Hive Support Product
Sqoop Support Product
Pig Support Product
Time taken: 28.552 seconds
```

Based on the Service Order Database we created and populated back in Listing 14-1, you can confirm the results in Listing 14-13. We have four open service orders on the products in bold. The Sqoop Hive import operation worked, and now the service company can leverage Hive to query, analyze, and transform its service order structured data. Additionally, the company can now combine its relational data with other data types (perhaps unstructured) as part of any new Hadoop analytics applications. Many possibilities now exist with Apache Hadoop being part of the overall IT strategy!

Importing Data into HBase

Chapter 12 takes a look at how you can transform a relational database schema into an HBase schema, when appropriate. In this subsection, we demonstrate how Sqoop can be used to make that transformation much easier. Of course, our main goal here is to demonstrate how Sqoop can import data from an RDBMS or data warehouse directly into HBase, but it's always better to see how a tool is used in context versus how it's used in the abstract. Figure 14-4 shows how the Service Order Database might look after being transformed into an HBase schema.

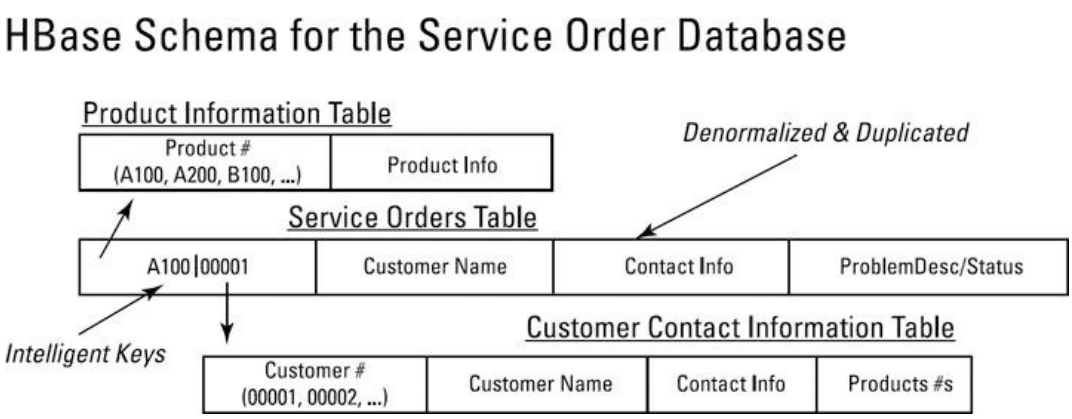

Figure 14-4: The Service Order database, translated into an HBase schema

Because we talk a lot about the process and methodology of this transformation in Chapter 12, we hold off on explaining it here. (If you desperately need to know *this instant* how this process works, of course, take a look at Chapter 12.) For more information on the Denormalization, Duplication, and Intelligent Keys (DDI) methodology of translating relational database schemas into HBase schemas, pay particular attention to the section in Chapter 12 about transitioning from an RDBMS to HBase.

TIP For this particular import example, we want to import the `customercontactinfo` table directly into an HBase table in preparation for building the HBase Service Order Database schema. (Refer to Figure 14-4.) To complete the HBase schema, you'd have to execute the same steps to import the `productinfo` table, and then the `serviceorders` table could be built with a Java MapReduce application.

Sqoop doesn't now permit you to import, all at once, a relational table directly into an HBase table having multiple column families. To work around this limitation, you create the HBase table first and then execute three Sqoop import operations to finish the task. Listing 14-14 shows the task of creating the table.

Listing 14-14: HBase customercontactinfo Table Creation Command

```
hbase(main):017:0> create 'customercontactinfo', 'CustomerName',
hbase(main):018:0*        'ContactInfo', 'ProductNums'
0 row(s) in 1.0680 seconds
```

In Listing 14-15, for each Sqoop import command, note that we have bolded the target HBase column family specified by the `--column-family` CLA and the corresponding MySQL columns specified by the `-columns` CLA. The `customernum` primary key also becomes the HBase row key, as specified by the `--hbase-row-key` CLA.

Listing 14-15: Sqoop Commands to Import the customercontactinfo Table Directly into a HBase Table

```
$ sqoop import \
    --connect jdbc:mysql://localhost/serviceorderdb \
    --username root -P \
    --table customercontactinfo \
    --columns "customernum,customername" \
    --hbase-table customercontactinfo \
    --column-family CustomerName \
    --hbase-row-key customernum -m 1
Enter password:
...
13/08/17 16:53:01 INFO mapreduce.ImportJobBase: Retrieved 5 records.
$ sqoop import \
    --connect jdbc:mysql://localhost/serviceorderdb \
    --username root -P \
    --table customercontactinfo \
    --columns "customernum,contactinfo" \
    --hbase-table customercontactinfo \
    --column-family ContactInfo \
    --hbase-row-key customernum -m 1
Enter password:
...
13/08/17 17:00:59 INFO mapreduce.ImportJobBase: Retrieved 5 records.
$ sqoop import \
    --connect jdbc:mysql://localhost/serviceorderdb \
    --username root -P \
    --table customercontactinfo \
    --columns "customernum,productnums" \
    --hbase-table customercontactinfo \
    --column-family ProductNums
    --hbase-row-key customernum -m 1
Enter password:
...
13/08/17 17:05:54 INFO mapreduce.ImportJobBase: Retrieved 5 records.
```

If you were to carry out an HBase scan of your new table (see Listing 14-16), you'd see that the import and translation from a relational database table on MySQL directly into HBase was a success. The `customercontactinfo` table in this example is rather small, but imagine the power you now have, using Sqoop and HBase, to quickly move relational tables that may be exceeding capacity on your RDBMS or data warehouse into HBase, where capacity is virtually unlimited and scalability is automatic.

Listing 14-16: HBase Scan of the New customercontactinfo Table Confirming Success

```
hbase(main):033:0> scan 'customercontactinfo'
ROW         COLUMN+CELL
 10000      column=ContactInfo:contactinfo,
            timestamp=1376773256317, value=1 Hadoop Lane,
            NY, 11111, John.Smith@xyz.com
 10000      column=CustomerName:customername,
            timestamp=1376772776684, value=John Timothy
            Smith
 10000      column=ProductNums:productnums,
            timestamp=1376773551221, value=B500
 10001      column=ContactInfo:contactinfo,
            timestamp=1376773256317, value=2 HBase Ave, CA,
            22222
 10001      column=CustomerName:customername,
            timestamp=1376772776684, value=Bill Jones
 10001      column=ProductNums:productnums,
            timestamp=1376773551221,
            value=A100,A200,A300,B400,B500,C500,C600,D700
 20000      column=ContactInfo:contactinfo,
            timestamp=1376773256317, value=1 Expert HBase
```

```
            Ave, CA, 22222
20000       column=CustomerName:customername,
            timestamp=1376772776684, value=Jane Ann Doe
20000       column=ProductNums:productnums,
            timestamp=1376773551221, value=A100,A200,A300
20001       column=ContactInfo:contactinfo,
            timestamp=1376773256317, value=1 Piglatin Ave,
            CO, 33333
20001       column=CustomerName:customername,
            timestamp=1376772776684, value=Joe Developer
20001       column=ProductNums:productnums,
            timestamp=1376773551221, value=D700
30000       column=ContactInfo:contactinfo,
            timestamp=1376773256317, value=1 Statistics
            Lane, MA, 33333
30000       column=CustomerName:customername,
            timestamp=1376772776684, value=Data Scientist
30000       column=ProductNums:productnums,
            timestamp=1376773551221, value=C500
5 row(s) in 0.1120 seconds
```

TIP Importing existing relational data via Sqoop into Hive and HBase tables can potentially enable a wide range of new and exciting data analysis workflows. If this feature is of interest to you, check out the Apache Sqoop documentation for additional Hive and HBase command line arguments and features.

Importing Incrementally

If the tables you're planning to import into Hadoop are changing or growing (which means that you may be planning more than one import or perhaps continual imports), be sure to check out Sqoop's Incremental Import feature. Sqoop provides several options and tools to make incremental import operations flexible and straightforward.

Incremental Import Append Mode

When you have a table that is receiving new rows and it has a column with a continually increasing value (like the `customernum` from our `customercontactinfo` table), you can leverage incremental append mode. Below we show how you can incrementally import all new customers from the fictional service company that have been appended to our MySQL `customercontactinfo` table since the last import operation.

First you need to know the number of the last customer in our MySQL `customercontactinfo` table. A quick review of Listing 14-1 shows that our last customer, Mr. Data Scientist, was given a customer number of 30000.

In the next step, you need to add three customers to our MySQL `customercontactinfo` table for the example to work properly. The SQL statements in Listing 14-17 will get the job done.

Listing 14-17: Insert Commands in the MySQL customercontactinfo Table

```
INSERT INTO customercontactinfo VALUES (40000, 'Isaac
          Newton', '1 Gravity Lane, London, United
          Kingdom', 'C500');
INSERT INTO customercontactinfo VALUES (50000, 'Johann
          Kepler', '1 Astronomy Street, Wrttemberg,
          Germany', 'A100,B500,C500');
INSERT INTO customercontactinfo VALUES (60000, 'Louis
          Pasteur', '1 Bacteriology Ave, Dole, France',
          'A100,A200,A300,B500,C600');
```

At this point, you're ready to let Sqoop do the work and import all new customers with a customer number greater than 30000. Listing 14-18 provides the command you can use.

Listing 14-18: Sqoop Incremental Import Command to Pick Up New Customers in Hive

```
sqoop import \
         --connect jdbc:mysql://localhost/serviceorderdb \
         --username root -P \
         --table customercontactinfo -m 1 \
         --incremental append \
         --check-column customernum \
         --last-value 30000
```

```
Enter password:
...
13/08/24 14:15:28 INFO tool.ImportTool:  --incremental append
13/08/24 14:15:28 INFO tool.ImportTool:   --check-column customernum
(A) 13/08/24 14:15:28 INFO tool.ImportTool:   --last-value 60000
(B) 13/08/24 14:15:28 INFO tool.ImportTool: (Consider saving this with 'sqoop job --create')
```

Listing 14-19 confirms our success. You now have three new customers stored in your HDFS file.

Listing 14-19: New Customers Now Stored in HDFS after Sqoop Incremental Import

```
$ hadoop fs -cat /user/biadmin/customercontactinfo/part-m-00000
40000,Isaac Newton,1 Gravity Lane, London, United Kingdom,C500
50000,Johann Kepler,1 Astronomy Street, W_rttemberg, Germany,A100,B500,C500
60000,Louis Pasteur,1 Bacteriology Ave, Dole, France,A100,A200,A300,B500,C600
```

Note the last two lines of output from Listing 14-18. The line labeled A lets you know that, of the customer records that were imported, the last new record had the customer ID 60000 (Louis Pasteur). This handy bookkeeping feature in Sqoop gets even better! Line B suggests that you save the value for the next incremental import and consider using the sqoop-job tool to make the task easier. The `sqoop job --create` command works hand in hand with incremental imports. Using the `sqoop-job` tool, you can create a job that you can run as often as you need to, and Sqoop's metastore keeps track of the vital information — like `last-value`, in this case. Listing 14-20 creates a Sqoop job that you can call every time you need to import new customers into your HDFS. (We call our job *load-new-customers* but you call it whatever makes sense for your application.)

Listing 14-20: The sqoop job --create Command and Subsequent sqoop job --list to Confirm Results

```
$ sqoop job --create load-new-customers -- \
        import \
         --connect jdbc:mysql://localhost/serviceorderdb \
         --username root -P \
         --table customercontactinfo -m 1 \
         --incremental append \
         --check-column customernum \
         --last-value 60000
Enter password:
$ sqoop job --list
Available jobs:
  load-new-customers
```

Additionally, you can leverage another Sqoop tool — `sqoop-metastore` — to create an HSQLDB instance that can be accessed by other users on your network; now your Sqoop meta data can be shared by others on your team!

TECHNICAL STUFF HSQLDB, which stands for *H*yper*SQL D*ata*B*ase, is an SQL database written in Java. For more information on HSQLDB, go to http://hsqldb.org. For the metastore thing to work, you also need to add some information to your `$SQOOP_HOME/conf/sqoop-site.xml` file.

After running the `sqoop-metastore` command, your team can leverage it in the `sqoop job --create` command by adding a `--meta-connect` command line argument, as shown in this example:

```
sqoop job
  --create load-new-customers \
  --meta-connect jdbc.hsqldb:hsql://<servername>:<port>/sqoop \
  --import \
  --table xyz \
   ...
```

Incremental Import Lastmodified Mode

In addition to incremental append mode, Sqoop provides last modified mode. You can use this mode to incrementally import updates from a table to HDFS. For example, to import to HDFS any changes in the `customercontactinfo` table that took place yesterday, you would have to modify the table to include a `LastUpdate` column that would hold the timestamp for each update. With a new `LastUpdate` column, you could create this `Sqoop` command:

```
sqoop import
        --connect jdbc:mysql://localhost/serviceorderdb \
        --username root -P \
        --table customerinfo -m 1 \
        --incremental lastmodified \
```

```
        --check-column LastUpdate \
        --last-value "2013-08-23 00:00:00"
```

Note that, as with the incremental append mode option, the `sqoop-job` tool can come in quite handy for saving the `last-value` timestamp for subsequent `incremental lastmodified` imports. Speaking of subsequent imports, what do you suppose happens when you run the same command again (or job, if you created one) on, say, the next day to pick up more potential `customercontactinfo` table changes? The answer is that you get another file under the directory `customercontactinfo` in your HDFS with the `customercontactinfo` table modifications. So how do you merge these files? You use the `sqoop-merge` command, of course, which is the subject of the next subsection.

The Sqoop Merge Tool

The `sqoop merge` tool works hand in hand with the incremental import lastmodified mode. Each import creates a new file, so if you want to keep the table data together in one file, you use the merge tool. The `sqoop merge` tool combines a newer data set with an older data set by overwriting rows from the older data set with the rows from the new dataset when the primary keys match. The `sqoop merge` command shown in the following example illustrates how this would look when using new and old `customercontactinfo` incremental imports:

REMEMBER The generated Java class file from the previous import (specified with `--jar-file customercontactinfo.jar`) is required to parse the records for this merge example. If you don't keep it around, you'll need to use the `codegen` tool to recreate it.

```
sqoop merge
      --new-data \
        /user/biadmin/customercontactinfo/part-m-00001 \
      --onto \
        /user/biadmin/customercontactinfo/part-m-00000 \
      --target-dir /user/biadmin/merged-customers \
      --jar-file customercontactinfo.jar \
      --class-name customercontactinfo \
      --merge-key customernum
```

Benefiting from Additional Sqoop Import Features

With the hands-on examples from the preceding section in mind, we'd like to describe some additional import features that you should know about. It's beyond the scope of this chapter to cover every Sqoop feature in detail, but Table 14-1 exposes you to its more significant features. Also note that the Sqoop community is always innovating and adding functionality to Sqoop, so you should watch the community documentation pages under http://sqoop.apache.org/docs for the latest features and new Sqoop command options.

Table 14-1: Miscellaneous Sqoop Import Options

Command Line Arguments	*Description*
Generic	
`--driver <class-name>--connection-manager <manager-name>`	Earlier in the chapter under the subsection entitled "Connectors and Drivers" we explain three approaches for using Sqoop depending on which data management system you are interfacing with. If you need to download and install your own connector, then you'll need to use the `--connection-manager` CLA and possibly the `--driver` CLA as well. If you find yourself needing to use the generic JDBC connector, then you have to specify that with the `--connection-manager` CLA and your vendor specific JDBC driver with the `--driver` CLA.
Import	
`--append`	You can append imported data to an existing dataset stored in HDFS. Without the `--append` CLA, if you try to import to an existing HDFS directory, the import fails. With the `--append` CLA, the import data is written to a new file in the same HDFS directory and is given a name that doesn't conflict with the existing file(s).
`–as-avrodatafile, --as-sequencefile, --as-textfile`	These three arguments let you specify the import data format when it's stored on HDFS. The default import format is `textfile`.
`--direct`	Some of the Sqoop-supported databases offer high-performance tools for data movement that exceed the performance of their respective JDBC drivers. As of this writing, both MySQL and PostregSQL provide these high-performance tools, and you can leverage them by using the `--direct` argument along with the table-split-size argument via `--direct-split-size <n>`. Beware that there may be certain limitations in direct mode (e.g. large objects may not be supported) so consult your database documentation.
`--map-column-java <mapping>, --map-column-hive <mapping>`	Sqoop lets you explicitly specify the Java type mapping for imports into HDFS and Hive.
`--inline-lob-limit <size>`	As you might expect, Sqoop can import large objects (BLOBs and CLOBs, in RDBMS terms). After all, Apache Hadoop is all about big data! As long as the large object doesn't exceed the size of the `--inline-lob-limit <size>` CLA, Sqoop stores the large object in line with the rest of the data in HDFS. However, if the large object exceeds the aforementioned limit specified by the CLA, it's stored in the subdirectory named `_lobs`, off the main HDFS import directory.
`--compress, --compression-codec <c>`	By default, data isn't compressed, but you can leverage gzip by specifying the `--compress` argument or your own algorithm using the `--compression-codec` argument. All three of the supported file types (text, sequence, and

	Avro) can be compressed.

Sending Data Elsewhere with Sqoop

Sqoop export operations are quite similar to import operations, with a couple of notable exceptions. First, Sqoop cannot determine the correct data types for your relational tables. SQL data types are numerous and rich, so it makes far more sense for you to first decide how you want to map your Hadoop data into relational database types, and then complete the export. In other words, you need to create the target table in your RDBMS or data warehouse first to hold the data you want to export. Second, when you execute the Sqoop export command, you specify the HDFS directory where the export data is stored. You cannot specify a Hive or HBase table name for exports, as you can with imports.

Figure 14-5 illustrates the steps involved in a Sqoop export from HDFS to an RDBMS or data warehouse system.

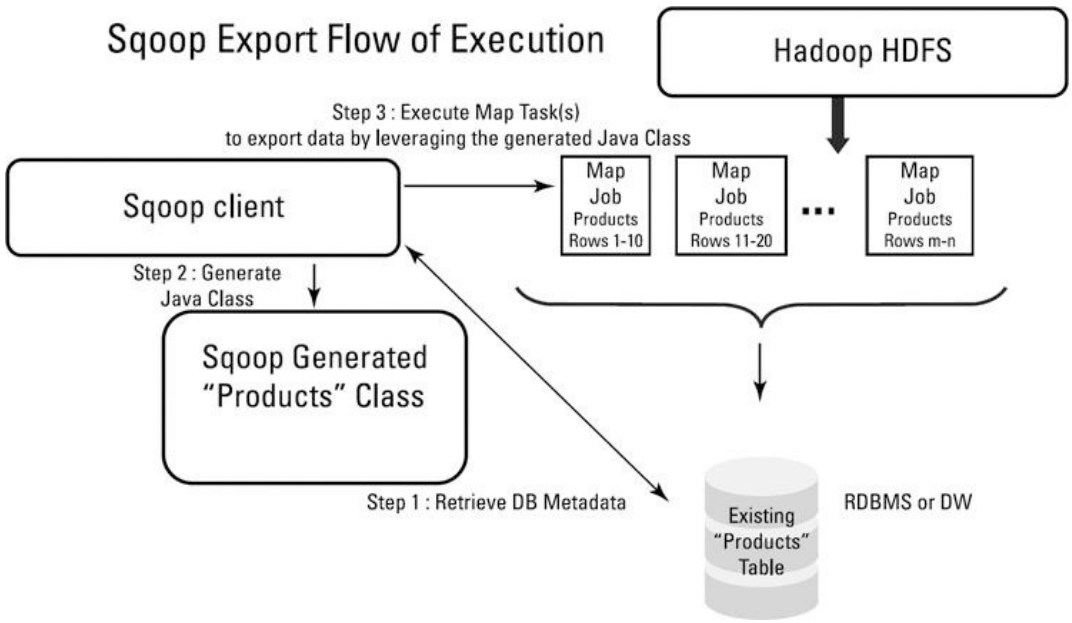

Figure 14-5: The Sqoop export flow of execution

As you can see, the Sqoop export flow of execution is similar to the import flow. Figure 14-5 focuses on the export of a potentially large Products file from HDFS into a similar Products data table in a data management system. Three map tasks are depicted to parallelize the process, but more or less could be specified by the user, based on the dataset size and the size of the Hadoop cluster. Carefully consider specifying the number of map tasks, in terms of both exports and imports. Too many map tasks can take *longer* if sufficient resources don't exist on your Hadoop cluster, and, similarly, too many map tasks can overwhelm the data management system as well.

Exporting Data from HDFS

The following hands-on example demonstrates an export of a Hive table called `sev1_serviceorders.` A fictional service company has derived the table from the original `serviceorders` table that we show you how to import from the MySQL `serviceorderdb` earlier in this chapter. It was decided, after leveraging text analytics on the Apache Hadoop cluster against the database, that service orders for customer number 20000 should be treated with a severity level of 1 and be exported back to the MySQL database for report generation. (This example is contrived but still illustrative of a typical joint use case for Apache Hadoop and the RDBMS or data warehouse.)

Right off the bat, make sure that the MySQL `serviceorderdb` has an appropriate table to receive your Sqoop export. The data definition language to create the table is given in Listing 14-21.

Listing 14-21: MySQL Create Table Statement

```
CREATE TABLE sev1_serviceorders(
serviceordernum INT PRIMARY KEY,
customernum INT,
productnum CHAR(4),
status VARCHAR(100),
FOREIGN KEY (customernum) REFERENCES
          customercontactinfo(customernum),
FOREIGN KEY (productnum) REFERENCES
          productinfo(productnum)
);
```

The Hive `sev1_serviceorders` table can be created and displayed in several different ways, but for the sake of illustration, we've included

a pair of possible HiveQL statements in Listing 14-22 and Listing 14-23.

Listing 14-22: HiveQL Create Table Statement with INSERT Command to Load Data

```
hive> CREATE TABLE sev1_serviceorders(
    > serviceordernum INT,
    > customernum INT,
    > productnum STRING,
    > status STRING);
OK
Time taken: 0.7 seconds
hive> INSERT OVERWRITE TABLE sev1_serviceorders
    > SELECT * FROM serviceorders WHERE customernum =
                20000;
...
Total MapReduce CPU Time Spent: 1 seconds 30 msec
OK
Time taken: 26.836 seconds
```

Listing 14-23: HiveQL SELECT Command to Display the Contents of the New Table

```
hive> SELECT * FROM sev1_serviceorders;
OK
100000  20000   A200    I have some questions on building
          HiveQL queries? My Hadoop for Dummies book has
          not arrived yet!
Time taken: 0.167 seconds
```

Listing 14-23 confirms that everything is in place to perform the Sqoop export operation. Sqoop export commands are similar to import commands, as you can see in this example:

```
sqoop export (generic arguments) (export arguments)
```

In the export arguments, you specify in your HDFS the pathname to the Hive warehouse where the `sev1_serviceorders` table is stored. In addition, you specify the field delimiter that you want to use for your table, because Hive allows many different types of delimiters. Listing 14-24 shows a possible scenario, and Listing 14-25 shows the results.

Listing 14-24: Sqoop export Command from HDFS to MySQL

```
$ sqoop export \
--connect jdbc:mysql://localhost/serviceorderdb \
--username root -P -m 1 \
--table sev1_serviceorders \
--export-dir /biginsights/hive/warehouse/serviceorderdb.db/sev1_serviceorders \
--input-fields-terminated-by '\0x0001'
Enter password:
...
13/08/18 19:08:27 INFO mapreduce.ExportJobBase: Exported 1 records.
```

Listing 14-25: MySQL Export Results

```
mysql> select * from sev1_serviceorders;
| serviceordernum | customernum | productnum | status
|          100000 |       20000 | A200       | I have some questions on building HiveQL queries? My
1 row in set (0.00 sec)
```

Listing 14-25 confirms that the export was successful and the record you expected to be inserted into the `sev1_serviceorders` table in the MySQL database has in fact been inserted.

Just because we authors value thoroughness, we show you four distinct export approaches in this section: insert, update, update insert, and call procedures. The preceding example used the insert approach. In the following four sections, we explain each export approach (yes, even insert again) and their various options.

Sqoop Exports Using the Insert Approach

In the hands-on export example in the previous section, the rows are exported from the Hive data warehouse (stored in HDFS) with the help of

SQL `INSERT` statements in the MySQL RDBMS. The export operation was a small one, for the sake of illustration, but often, exports include very large tables with *millions* of rows. Sqoop handles large export use cases by way of batching techniques and by leveraging multiple map tasks to write the data in parallel. (As with imports, Sqoop uses four map tasks by default with exports.) The idea behind batching is to execute a group of SQL `INSERT` statements together instead of the serial approach of executing them one by one. The idea is straightforward, but the approach for batching differs from one database technology to another. The Sqoop designers knew this, so they made some good, educated guesses on batch default parameters and then gave us different options for adapting to, and tuning for, our database of choice.

This list describes two techniques that Sqoop users can leverage to batch export operations:

- **The `--batch` command line argument**: This argument allows Sqoop to batch together SQL `INSERT` statements using the JDBC PreparedStatement interface. So the Sqoop client creates a batch of the following statements using the JDBC APIs:

  ```
  INSERT INTO table VALUES (col1,col2,...);
  INSERT INTO table VALUES (col1,col2,...);
  INSERT INTO table VALUES (col1,col2,...)
  ```

  In theory, this technique should result in better export throughput because Sqoop's map task writers avoid sending individual `INSERT` statements and instead batch them together.

- **The `-D <property=value>argument`**: If you were to issue the `sqoop help export` command, you'd see a command line argument that begins with `-D` to allow you to set properties for Sqoop that would otherwise have to be set in the `$SQOOP_HOME/conf/sqoop-site.xml` file. If you leverage the `-D <property=value>` argument, you can set the `sqoop.export.records.per.statement` property to a value that determines the number of records per `INSERT` statement. For example, setting the aforementioned property to 3 would generate the `INSERT` statement

  ```
  INSERT INTO "table" VALUES (x,y,z,...), (x,y,z,...),
         (x,y,z,...);
  ```

  You can also set the `sqoop.export.statements.per.transaction` property to a value that specifies the number of `INSERT` statements to be executed before you commit the transaction.

Which option should you use? Well, it depends on your chosen database technology. The `--batch` command line argument may work fine, but it depends on how the JDBC driver was implemented. As of this writing, the default behavior for Sqoop 1.4.4 is to leverage the `-D <property=value>` argument, with records per statement set to 100 and statements per transaction set to 100. Therefore, every 10,000 rows, Sqoop commits your batch `INSERT` operations. By causing a commit every 10,000 rows, Sqoop avoids out-of-memory errors. We don't mean that the `-D <property=value>` argument works with every database technology — it just happens to be what the Sqoop designers chose, based on certain assumptions. Consult your vendor, or review the database documentation before executing batch Sqoop export commands to see which options are supported.

Sqoop Exports Using the Update and Update Insert Approach

With insert mode, records exported by Sqoop are appended to the end of the target table. Sqoop also provides an update mode that you can use by providing the `--update-key <column(s)>` command line argument. This action causes Sqoop to generate a SQL `UPDATE` statement to run on the RDBMS or data warehouse. Assume that you want to update a three-column table with data stored in the HDFS file `/user/my-hdfs-file`. The file contains this data:

```
100, 1000, 2000
```

The following abbreviated Sqoop export command generates the corresponding SQL `UPDATE` statement on your database system:

```
$ sqoop export (Generic Arguments)
  --table target-relational-table \
  --update-key column1
  --export-dir /user/my-hdfs-file
  ...

Generates => UPDATE target-relational-table SET
                    column2=1000,column3=2000
               WHERE column1=100;
```

With the preceding export command, if the `target-relational-table` on your RDBMS or data warehouse system has no record with the matching value in `column1`, nothing is changed in `target-relational-table`. However, you may also include another argument that inserts or appends your data to `target-table` if no matching records are found. Think of it this way: `If exists UPDATE else INSERT`. This technique is often referred to as *upsert* in the database vernacular or as `MERGE` in other implementations. The argument for upsert mode is `--update-mode <mode>`, where `updateonly` is the default and `allowinsert` activates upsert mode. Check your database documentation or consult with your vender to determine whether upsert mode is supported with Apache Sqoop.

Sqoop Exports Using Call Stored Procedures

Sqoop can also export HDFS data by calling a stored procedure in your RDBMS or data warehouse using the `--call <stored procedure>` command line argument. The following abbreviated Sqoop export command illustrates this approach:

```
sqoop export (Generic Arguments)
  --call my-stored-procedure \
  --export-dir /user/my-hdfs-export-data
```

In this example, Sqoop calls the `my-stored-procedure` for every record in the `/user/my-hdfs-export-data` file. Many use cases can leverage this feature. A classic example is that you already have existing stored procedures that you use to import data into your RDBMS or data warehouse.

TECHNICAL STUFF A *stored procedure* is a subroutine that's stored in the RDBMS or data warehouse. It can centralize common logic that would otherwise have to exist at the application level.

Sqoop Exports and Transactions

The beauty of Sqoop is that it can export massive data sets to an RDBMS or data warehouse by batching SQL statements and leveraging parallel map writer tasks. However, the export operation is not *atomic* — it isn't an all-or-nothing entity, in other words. Individual writer tasks can fail, leaving the Sqoop export operation in a partially completed state. If this happens, your table data is corrupt and you're unlikely to be a "happy Hadooper." Sqoop solves this problem with the help of staging tables.

The idea here is that you can first export data to a staging table and after the export successfully completes, move your staging table to the final table in one atomic transaction. Use the command line argument `--staging-table <table name>` to specify your staging table, and use `--clear-staging-table` to clear the staging table before each subsequent export.

TIP Staging tables aren't supported when using the `--direct` option, update mode, update insert mode, or called procedures. Staging tables are only available with the insert approach discussed above and demonstrated in Listing 14-25.

Looking at Your Sqoop Input and Output Formatting Options

In the earlier subsection "Importing Data with Sqoop," we talk about Sqoop's code generation feature. A bit later in the chapter — at Listing 14-5 or thereabouts — we also leverage code generation command-line arguments to demonstrate how you can control the code generation process and results. (Then you can use the `.jar` file for subsequent applications where you need to process the data now stored in HDFS.) Finally, in Listing 14-24, we use an `--input-fields-terminated-by '\0x0001'` command line argument to instruct the Sqoop export tool how to read and parse records managed by Hive before exporting to MySQL. Hive uses control-A characters ('\0x0001' in Listing 14-24) rather than the default comma for field termination. In this section, we help you take a closer look at input parsing CLAs as well as output line formatting CLAs. When you choose to import or export delimited text, you often need these CLAs.

Table 14-2 lists the input parsing CLAs which begin with `--input`, and the output line formatting CLAs. You'll probably notice that these CLAs are just opposites of each other.

Table 14-2: Sqoop Output Line Formatting and Input Parsing CLAs

Command Line Argument	*What It Does*
`--enclosed-by <char>` `--input-enclosed-by <char>`	Specifies a field-enclosing character (double quotes, for example).
`--optionally-enclosed-by <char>` `--input-optionally-enclosed-by <char>`	Specifies that if the data includes the `enclosed-by <char>`, say double quotes ("), then the double quotes should be written; otherwise, double quotes are optional — don't write them. So for example, if Sqoop imports a string field enclosed in double quotes then it will be written to HDFS with double quotes. Otherwise, other fields would not be written to HDFS with double quotes.
`--escaped-by <char>` `--input-escaped-by <char>`	Specifies an escape character to avoid ambiguity when parsing or writing records to HDFS. As an example, you might make the `--escaped-by` character a backslash (\) which would allow you to import a string with double quotes inside the string. When Sqoop writes the field to HDFS, the double quotes within the string would be preceded with a backslash. In a similar way, if you use the generated Java code to parse a string with quotes inside the string, specifying a backslash (\) with the `--input-escaped-by` CLA would save you from losing data because Sqoop would see the backslash, skip over the quotes and continue looking for the enclosing quotes.
`--fields-terminated-by <char>` `--input-fields-terminated-by <char>`	Specifies the field-termination character (a comma, for example).
`--lines-`	Specifies the record- or line-termination character (a new-line character for example).

`terminated-by <char>` `--input-lines-terminated-by <char>`	
`--mysql-delimeters <char>`	For output line formatting only, this CLA indicates that the default MySQL delimiters should be used for outputting records to HDFS. MySQL's default delimiter set is the following: fields:, lines: \n escaped-by: \ optionally-enclosed-by: '

TIP If you accidentally delete Java files generated by `sqoop-import` or `sqoop-export`, you can use the `sqoop-codegen` tool later to reproduce the files. The Sqoop codegen tool accepts the same CLAs in Table 14-2. You can also use `sqoop-codegen` independently and specify the jar file and class name for your `sqoop-import` or `sqoop-export` commands.

Getting down to Brass Tacks: An Example of Output Line-Formatting and Input-Parsing

To ensure that this whole output line formatting / input parsing feature in Sqoop is clear, we close this discussion with an example using our old standby, the Service Order Database. Imagine a call center operator from our fictional service company taking calls from customers and inputting their comments into the MySQL `serviceorderdb` that was used in earlier examples. You might imagine an operator entering commas in the problem description, in an attempt to keep the prose as clear as possible for the engineer, who would later try to solve the issue for the customer. However, unbeknownst to the call center operator, commas are the default field-termination characters for Sqoop — so later, when the IT staff decides to import part or all of the `serviceorderdb` into Hadoop for analysis, we have a problem. It could happen like this: the call center operator takes a service call from a customer and the MySQL system inserts the following record into the `serviceorderdb`.

```
INSERT INTO serviceorders VALUES (100000, 20000, 'A200',
          'I have some questions, on building HiveQL
          queries? My Hadoop for Dummies book has not
          arrived yet!');
```

Later on, the IT staff imports the `serviceorders` table into Apache Hadoop using this familiar command:

```
sqoop import \
  --connect jdbc:mysql://localhost/serviceorderdb \
  --username root -P \
  --table serviceorders -m 1
```

At this point, everything is good; even though a comma appears in the problem description, it's imported into HDFS verbatim. However, suppose that the IT staff decides to export the data from Hadoop back into a MySQL table later on, using this command:

```
sqoop export \
  --connect jdbc:mysql://localhost/serviceorderdb \
  --username root -P \
  --export-dir /user/biadmin/serviceorders \
  --table serviceorders -m 1
```

After the export operation, the MySQL database administrator looks at the new table and sees the following records:

```
mysql> select * from serviceorders;
...
| serviceordernum | customernum | productnum | status
|          100000 |       20000 | A200       | I have some questions
|          100001 |       10001 | A100       | I need to understand how to configure Zookeeper for m
...
```

The Sqoop export command has interpreted the operator's comma as a field delimiter, and some vital data was lost. We're in danger of losing an important customer because we can't address the problem without an embarrassing return phone call to solve the data loss problem! It sounds bad, so what's the solution? The solution is output line formatting and input parsing CLAs. Two commands (one `import` and one `export`— see Listing 14-26) would solve the problem.

Listing 14-26: An Output Line Formatting and Input Parsing Example

```
sqoop import \
   --connect jdbc:mysql://localhost/serviceorderdb \
   --username root -P -m 1 \
   --table serviceorders \
   --target-dir /user/biadmin/serviceorders-test \
   --escaped-by \\ \
   --input-escaped-by \\ \
   --class-name serviceorderstest \
   --bindir /home/biadmin/serviceorders-test
sqoop export \
```

```
  --connect jdbc:mysql://localhost/serviceorderdb \
  --username root -P -m 1 \
  --table serviceorders \
  --export-dir /user/biadmin/serviceorders-test \
  --class-name serviceorderstest \
  --jar-file /home/biadmin/serviceorders-test/serviceorderstest.jar
```

Because this topic is important, we walk you through each step. First, in the import, we're specifying an output line formatting escape character (\). This character causes the generated code (which we're naming `serviceorderstest`) to place a backslash (because of the `--escaped by \` CLA) before the operator's comma in the HDFS records file. Then when the `serviceorders` records are exported from HDFS back to the MySQL `serviceorders` table (or another table like it), we'll reuse the generated code, which we saved in the `/home/biadmin/serviceorders-test` directory with the `--bindir` CLA. This generated code has an input parse method that knows how to read the problem description, so whenever it sees the backslash and comma (because of the `--input-escaped-by \` CLA in the import command), it continues reading and exporting the whole problem description until it finds the final field-enclosing comma. Now, when the MySQL database administrator from the service company issues the `SELECT` statement, he or she sees the whole problem description.

TECHNICAL STUFF The Linux shell uses the backslash (\) as a line continuation character so you can just keep on typing with a whole new line. (This is a pretty important little technique with Sqoop and its long command structures.) That's why we have three backslashes on the lines in Listing 14-26 where we are specifying the backslash as an escape character. We're escaping our *escape character* and continuing our line.

REMEMBER If you had chosen to import binary data from a data management system (DMS) with Sqoop, and store that data in HDFS using a sequence file (with the `--as-sequencefile` CLA), then you should save your generated Java class (like we did in Listing 14-26) so you can point to it (using the `--class-name` and `--jar-file` CLAs) if you need to export the data back to the DMS.

Sqoop 2.0 Preview

With all the success surrounding Sqoop 1.x upon its graduation from the Apache incubator, Sqoop has momentum! So, as you might expect, Sqoop 2.0 is in the works with exciting new features on the way. If you haven't already, we suggest checking out http://sqoop.apache.org for the full story. As of this writing, you can see that Sqoop 1.99.3 is downloadable, complete with documentation. We'd bet that you're wondering (like we are) how many 1.99.x releases will be available before the big 2.0 hits http://sqoop.apache.org. Well, our crystal ball only works part-time so the answer is "not yet."

We can still dream, right? And while we're dreaming, we can still provide you with a preview of Sqoop 2.0 features. However, you know the drill: The situation can change leading up to the 2.0 release, so we keep our description at a relatively high level of generality.

Figure 14-6 illustrates (documented) design plans for Sqoop 2.0.

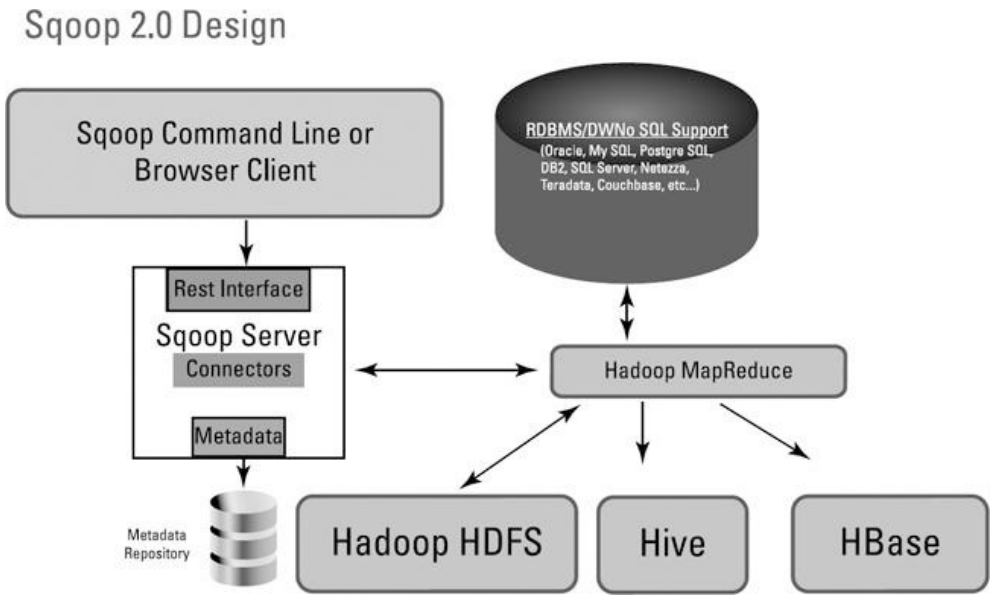

Figure 14-6: Sqoop 2.0 design plans

As you can see, the big change in the works is that Sqoop 2.0 will have a separate server, which is good news for a number of reasons. First, you won't have to do so much work. The Sqoop connector and JDBC driver will be installed once by the system administrator for your cluster instead of once per Sqoop client. If you happen to be the system administrator, we extend our condolences. You still have to do the work, but maybe you'll like the next benefit: Sqoop 2.0 will be more secure! With a Sqoop server as part of the architecture, sensitive operations such as connecting to the database servers only have to happen on the Sqoop server and you'll have role-based access control. Additionally, Sqoop clients can leverage Sqoop from anywhere on the network (thanks to the new rest interface), and they will enjoy a new graphical user interface

(GUI). We think you'll agree that the command line options are necessary and powerful for scripting purposes, but we all like a cool GUI from time to time. Sqoop requires many command line options, which can be error-prone without a GUI to guide you.

We'll leave this preview as is for now because we don't want to discuss features that might change. We would bet that you've noticed MapReduce (instead of just map tasks) proudly displayed in Figure 14-6. We've inserted it on purpose, but we'll wait to add our two cents until after we hear the exact details on how reducers are leveraged when the 2.0 announcement hits the community page. Until then, enjoy Sqoop 1.x and start experimenting with 1.99.x.

TIP You can read more about the Sqoop 2 goals and architecture on this web site:
https://cwiki.apache.org/confluence/display/SQOOP/Sqoop+2