INGESTING DATA INTO HDFS

Ingesting Data Into HDFS

- **Hadoop Commandline**
- File System API
- Ingesting Data Streams
- Importing Data from Databases
- Exporting HDFS Data into Databases

Hadoop Commandline

- Hadoop comes with a commandline utilty, **`hadoop fs`**, which, among other things, can be used to put local data into HDFS, and get data from it, into the local file system
- It resembles some standard UNIX commands

Example:

```
# The following command creates a directory
# called /foo
# Note the hadoop fs -cmd [args] syntax

$ hadoop fs -mkdir /foo
```

Hadoop Commandline

- We are interested in **`put`**, **`get`**, and **`getmerge`**, to transfer data from the local file system to HDFS, or from HDFS to the local file system

```
# The following command is used to put local files
# (directories) into HDFS

# The syntax:
# hadoop fs -put localpath [HDFSpath]
# hadoop fs -put local1 local2 HDFSpath

# The following syntax is used for reading from stdin
# and putting it into a distributed FS path
# hadoop fs -put - HDFSpath
```

- `localpath` can be a local file, local directory, or a wildcard
- `HDFSpath` can be an HDFS directory, an HDFS file, or wildcard
- `HDFSpath` can be written relatively to `/home/<local_user>/`

Hadoop Commandline

- We are interested in **`put`**, **`get`**, and **`getmerge`**, to transfer data from the local file system to HDFS, or from HDFS to the local file system

```
# The following command is used to copy files
#(directories) in HDFS into the local file system

# The syntax:
# hadoop fs -get HDFSpath [localpath]
# hadoop fs -get HDFSpath1 HDFSpath2 localpath

# The following syntax is used for copying multiple HDFS
#files into the local files sytem as a single, merged,
#file
# hadoop fs -getmerge HDFSpath [localpath]
# hadoop fs -getmerge HDFSpath1 HDFSpath2 localpath
```

- **`HDFSpath`** can be an HDFS file, directory, or a wildcard
- **`getmerge`** is useful when fetching resulting files of a M/R job

Hadoop Commandline

- Alternatives to **`put`** and **`get`** are **`copyFromLocal`**, **`moveFromLocal`**, **`copyToLocal`** to transfer data from the local file system to HDFS, or from HDFS to the local file system
- **`moveFromLocal`** deletes the source

Ingesting Data Into HDFS

- Hadoop Commandline
- **File System API**
- Ingesting Data Streams
- Importing Data from Databases
- Exporting HDFS Data into Databases

FileSystem API

- Hadoop's `FileSystem` API (Java) provides an interface for dealing with a file system
- A local implementation is `LocalFileSystem`, and distributed implementation is `DistributedFileSystem`
- To mimic hadoop commands in Java, static methods in `org.apache.hadoop.fs.FileUtils` can be used
- A FileSystem path is denoted with a `Path` object, which can be constructed with a path string or a `URI` object
- HDFS `URI` literal: `hdfs://absolute/path`

Ingesting Data Into HDFS

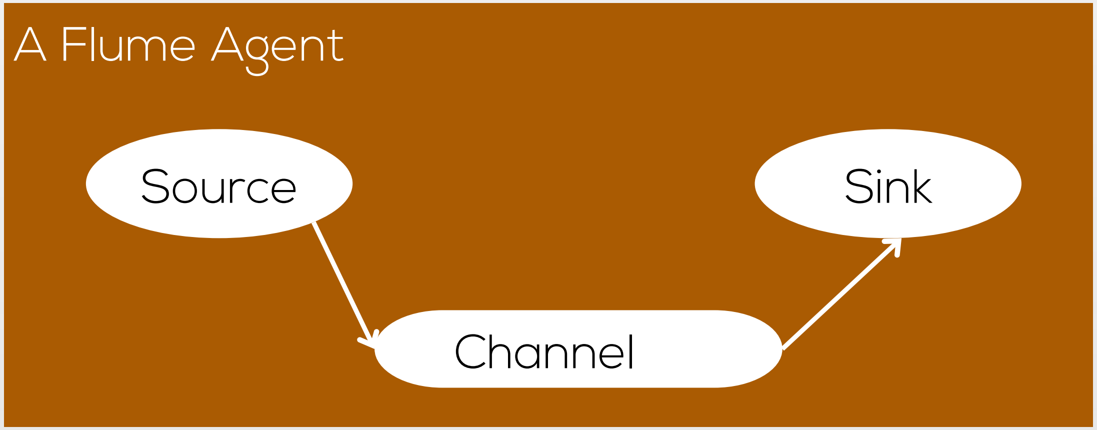
- Hadoop Commandline
- File System API
- **Ingesting Data Streams**
- Importing Data from Databases
- Exporting HDFS Data into Databases

Ingesting Data Streams

- We run programs of processing Big Data that is already in HDFS
- Data, however, come from other systems that actually generate data, and one of the most common data sources is distributed agents generating streams that we need to collect together
- Examples are social media sources, web server logs, …
- **Apache Flume** is a distributed, reliable, available service for efficiently collecting, aggregating, and moving large amounts of log data
- Flume can be used (and commonly used) as a data ingestion tool into HDFS

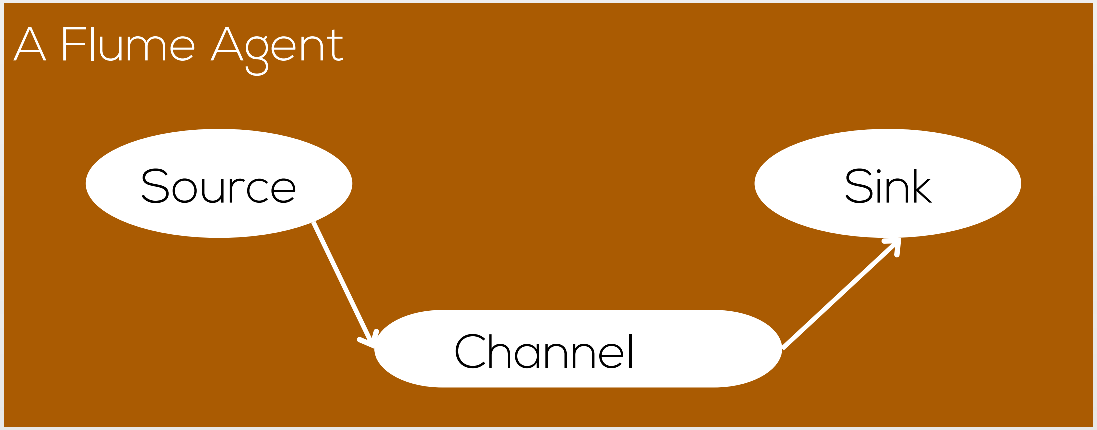
Apache Flume

- The unit of data flow in Flume is called an **event**
- The processes that host components through which the events flow from a source to the next destination are called Flume **agent**s



Apache Flume

- A Flume Source consumes events delivered to it (by an external source)
- The external source can be a data source, or another Flume agent’s sink
- The agent structure allows building complex flows



Setting up a Flume Agent

- A Flume agent can be set up using configuration files
- Configuration files are text files in a Java properties file format
- We give the agent components names and types, and then set the properties specific to the components
- Then we just start the agent with a name from its configuration

```
$ flume-ng agent -n <name> -c <conf_dir> -f <conf_file>
```

- This is a long-lived process, where a source produces events and delivers them to the channel, which is a store for events until they are forwarded to the sink

Setting up a Flume Agent

- A Flume configuration file looks like this

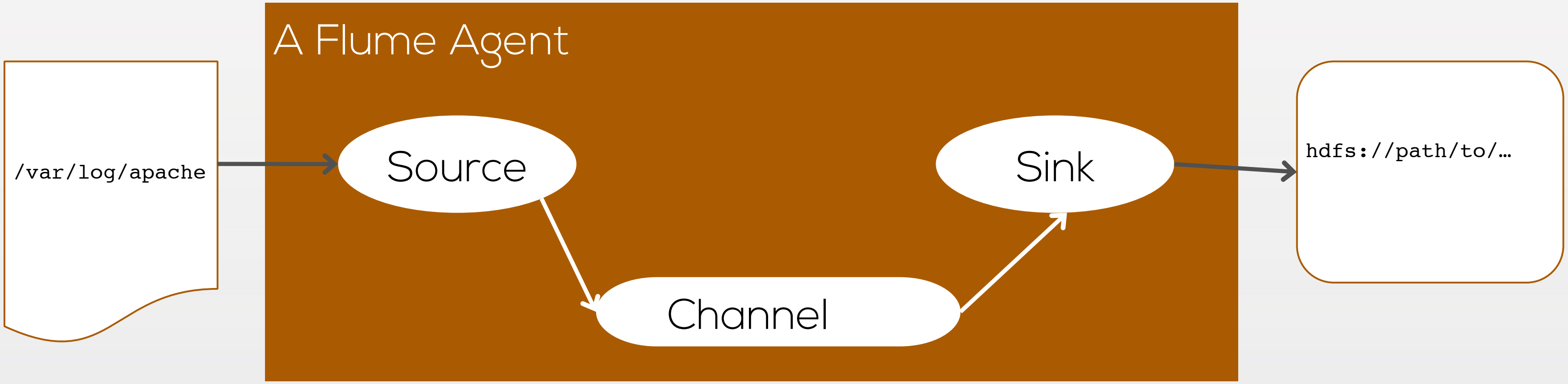
```
#This is a configuration file for agent a, with
#components my_src, my_channel and my_sink

#names
a1.sources = my_src
a1.sinks = my_sink
a1.channels = my_channel

#configure source to read from apache logs
a1.sources.my_src.type = exec
a1.sources.my_src.command = tail -F /var/log/apache
a1.sources.my_src.channels = my_channel

#configure sink to write to hdfs
a1.sinks.my_sink.type = hdfs
a1.sinks.my_sink.channel = my_channel
a1.sinks.my_sink.hdfs.path = hdfs://path/to
a1.sinks.my_sink.hdfs.inUsePrefix = _
```

Setting up a Flume Agent



Events

- In the example, each log-line added to the file is represented as an event
- An event can have a header, which we can use to selectively add events from the same source to different channels
  - The headers can be added to event by using interceptors

  The example demonstrates adding timestamp to an event, and using that information when creating HDFS directories

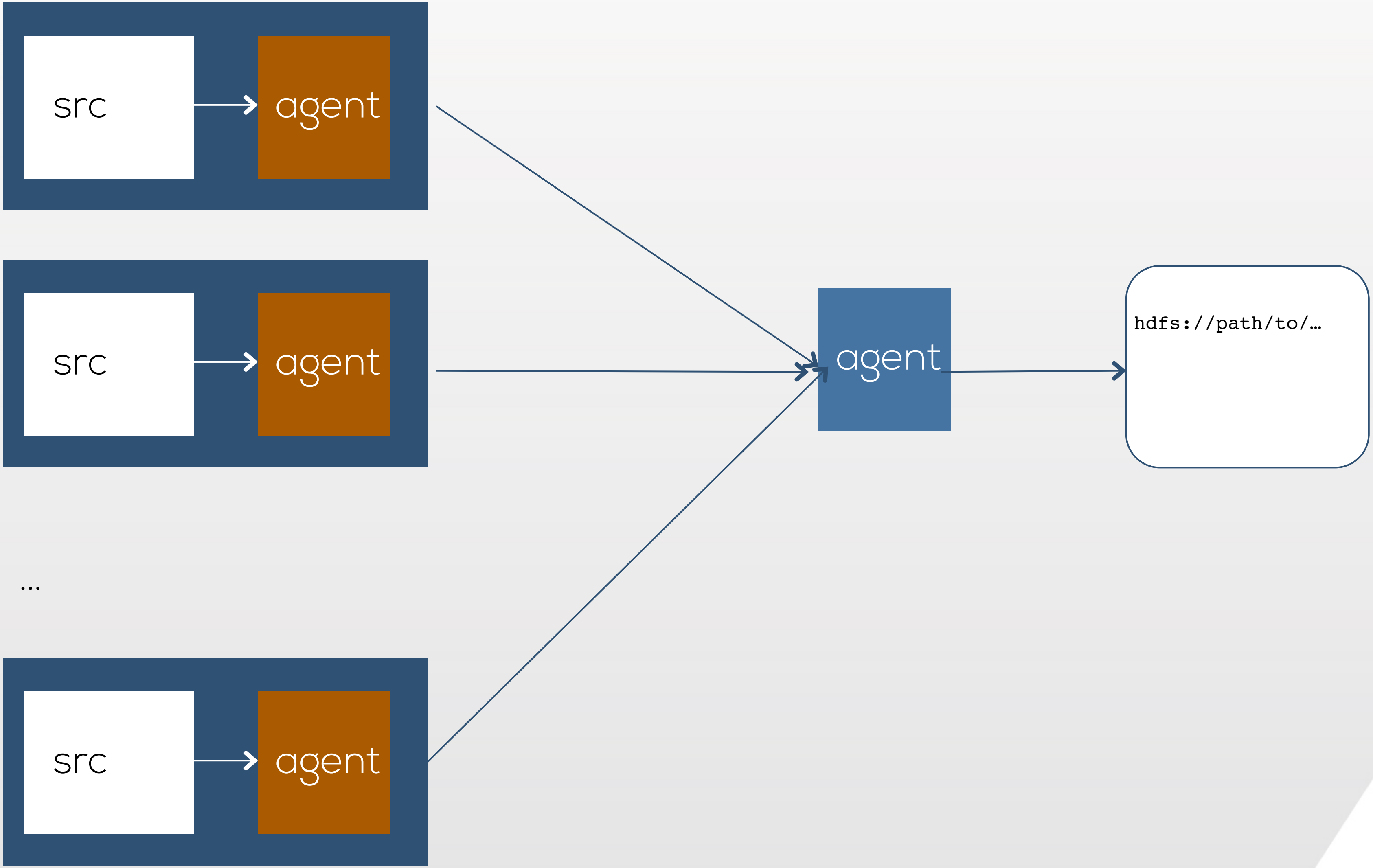
```
...
a2.sources.source1.interceptors = interceptor1
a2.sources.source1.interceptors.interceptor1.type = timestamp

#configure sink to write to hdfs
a2.sinks.my_sink.type = hdfs
a2.sinks.my_sink.channel = my_channel
a2.sinks.my_sink.hdfs.path = hdfs://path/to/year=%Y/%m/
a2.sinks.my_sink.hdfs.inUsePrefix = _
...
```

Designing a Flow

- Using Flume is setting up agents in different nodes, and designing a flow representing a topology
- There are multiple topologies one can create in Flume, for example
  - Consolidating
    - Using one agent for consolidating events generated by multiple other agents
  - Multiplexing
    - Within an agent, delivering events from a source to multiple channels, selectively based on header
  - Replicating
    - Within an agent, replicating events from a source to multiple agents

Consolidating



Designing a Flow

- Wiring two agents is done by using an Avro (or Thrift) sink in the first one, and an Avro (or Thrift) source in the second one

Flume Sources, Sinks, Channels

- Flume has many built-in source and sink definitions, as well as channels, for example:
  - Exec, Avro, Thrift, JMS, Spooling, Twitter, Kafka, NetCat, Syslog, HTTP sources
  - HDFS, Hive, Logger, Avro, Thrift, File Roll, Elastic Search, Kafka sinks
  - Memory, JDBC, Kafka, File channels

Flume Sources, Sinks, Channels

- Writing new sources and sinks (and also channels) are possible,
  - In configuration, the names for custom sources are sinks are the fully qualified class names of them
  - Custom sources and sinks need to implement the `Source` and `Sink` interfaces, respectively
  - They should be added to Flume's classpath to be used, which is as easy as adding appropriate jars under `$FLUME_HOME/plugins.d` directory

Demo

Collecting Log Data into HDFS

Ingesting Data Into HDFS

- Hadoop Commandline
- File System API
- Ingesting Data Streams
- **Importing Data from Databases**
- Exporting HDFS Data into Databases

Importing Data from Databases

- Often, we want to join the data stored in relational databases with Big Data stored in HDFS
- Or, we want to perform more advanced analytics on the data residing in structured data stores
- To extract data from a relational database into Hadoop, Apache Sqoop, *another top-level Apache project* can be used
  - The Hadoop target can be HDFS, Hive, even HBase

Apache Sqoop

- Sqoop is available at sqoop.apache.org
- It is a client utility (in its current architecture), that can effectively (by running a MapReduce job) import the results from a query (or an entire table) into HDFS
- Once it is installed, the **`sqoop import`** utility can be used to import data from a database
- With built-in Sqoop Connectors, data from MySQL, PostreSQL, Oracle, SQL Server, DB2, Netezza, … can be imported in parallel, utilizing the MapReduce cluster, with Map-only jobs
- Various 3rd party connectors, as well as a built-in generic JDBC connector are also available

Apache Sqoop

- The `import` tool interprets each row from the source table as a separate record
- In HDFS, records can be stored
  - As text files (one record per line)
  - In binary representation (SequenceFile)

sqoop import Utility

```
#Several example sqoop import runs
#Also add --username and --password options if necessary
#-P option can be used for password to be read from console

#Import the entire table
$ sqoop import \
        --connect jdbc:mysql://<host>/<db>
        --table <table_name>

#Import certain columns
$ sqoop import \
        --connect jdbc:mysql://<host>/<db> \
        --table <table_name> \
        --columns “c1,c2”

#Appending a WHERE clause
$ sqoop import \
        --connect jdbc:mysql://<host>/<db> \
        --table <table_name> \
        --columns “c1,c2”\
        --where “c1 > 100”
```

Controlling Parallelism

- Sqoop imports data in parallel, that is:
  - Sqoop submits a Map-only M/R job with a number of Map tasks (this is by default 4, and it can be changed using **`-m`** argument),
  - Where each Map task queries the table with a WHERE clause, which is based on the splitting column
  - Splitting column is picked by default (the primary key), or the user can pick it manually using **`--split-by <col>`**
  - If the table neither has a primary key, nor a splitting column is specified, the import fails (unless number of mappers is set to 1)

Controlling Parallelism

- For instance, if an auto-increment id column exists in the table, MAX(id) = 1000 and MIN(id) = 0, and 4 map tasks are created, each Map task imports data using one of the following SQL statements:

```
SELECT * FROM t WHERE id>=0 AND id<250
SELECT * FROM t WHERE id>=250 AND id<500
SELECT * FROM t WHERE id>=500 AND id<750
SELECT * FROM t WHERE id>=750 AND id<1001
```

Controlling Parallelism

```
#This is an example sqoop import run

#Import the entire table, using 10 mappers, and split by c1
$ sqoop import \
        --connect jdbc:mysql://<host>/<db> \
        --table <table_name> \
        --split-by c1 \
        -m 10
```

Using Native Connectors

- Some databases provide tools for importing data in a more efficient way, such as the **`mysqldump`** utility of MySQL
- Sqoop can take advantage of such tools, when **`--direct`** argument is supplied

Importing from Custom Queries

- Using Sqoop, resulting rows of an arbitrary SQL statement can also be imported, by supplying the query with **`--query <statement>`** (instead of passing a table)
- When importing from queries
  - the HDFS destination must be specified with **`--target-dir <HDFS Path>`**
  - the splitting column must be specified with **`--split-by <col>`**
  - a **`WHERE $CONDITIONS`** token should be included for Sqoop to run the query in parallel, based on the parallelism and the splitting column

Importing from Custom Queries

- Using Sqoop, resulting rows of an arbitrary SQL statement can also be imported, by supplying the query with **`--query <statement>`** (instead of passing a table)

```
#This is an example sqoop import run

#Import using a custom query, use 10 mappers, split by t1.c1
$ sqoop import \
        --connect jdbc:mysql://<host>/<db> \
        --query ‘SELECT * FROM t1 JOIN t2 ON (t1.c1 = t2.c2) \
                WHERE $CONDITIONS’ \
        --split-by t1.c1 \
        --target-dir /user/foo/joined \
        -m 10
```

Incremental Import

- Sqoop can perform incremental imports, in two modes:
  - append mode
  - lastmodified mode
- **append** mode is used when there is an auto-increment column, **lastmodified** is used when there is a column of timestamp
- The mode, the column for checking the last-value, and the last-value are supplied using the **`--incremental <mode>`**, **`--check-column <col>`**, and **`--last-value <val>`** arguments, respectively
- *"You should specify **append** mode when importing a table where new rows are continually being added with increasing row id (or modification time) values. You specify the column to check with* **`--check-column`***. Sqoop imports rows where the check column has a value greater (or more recent) than the one specified with* **`--last-value`***."*

File Formats

- By default, the import tool creates text files
  - The `--as-textfile` argument is default
  - This is a delimited text file where each line represents a row of the input table (default comma, can be overridden by `--fields-terminated-by <char>`)
- Importing into SequenceFiles can be performed by supplying `--as-sequencefile` argument

Demo

Importing into HDFS from Relational Databases

Ingesting Data Into HDFS

- Hadoop Commandline
- File System API
- Ingesting Data Streams
- Importing Data from Databases
- **Exporting HDFS Data into Databases**

Exporting HDFS Data into Databases

- Just like `sqoop import` tool can be used to import from databases into HDFS, `sqoop export` tool can be used to export HDFS data into databases
- The generic arguments (`--connect`, `--username`, `--password`, etc.) are used with the exact purpose as the import tool
- The target table **must** exist for export to work
- Target table is specified with `--table <target>`
- The HDFS path to be exported is specified with `--export-dir <dir>`
- Target columns (all columns are selected by default) can be specified with `--columns <c1, c2, …>`
- `--direct` can also be used here, for using native export tools

Exporting with INSERTs or UPDATEs

- Input records are translated to `INSERT` statements by default
  - The export process will fail if an `INSERT` statement fails. This mode is primarily intended for exporting records to a new, empty table intended to receive these results.
- If an `--update-key <col>` is specified, records are translated to `UPDATE` statements, where the row to be updated is selected based on the update key
  - An update based export does not add any new rows to the table, and new records (records with no matching target row based on the update key) are simply ignored
- To have sqoop to try updating first and if it fails inserting a new row, `--update-mode allowinsert` can be used

sqoop export Utility

```
#Several example sqoop export runs

#Populate table
$ sqoop export \
        --connect jdbc:mysql://<host>/<db>
        --table <table_name>
        --export-dir <hdfs_path>

#Export certain columns
$ sqoop export \
        --connect jdbc:mysql://<host>/<db> \
        --table <table_name> \
        --columns “c1,c2” \
        --export-dir <hdfs_path>

#UPDATE first, and INSERT if that fails
$ sqoop export \
        --connect jdbc:mysql://<host>/<db> \
        --table <table_name> \
        --update-key “c1” \
        --update-mode allowinsert
        --export-dir <hdfs_path>
```

Demo

Exporting from HDFS into Relational Databases

Ingesting Data into HDFS

End of Chapter