MODULE 1 – LOAD SCENARIOS

Load scenarios:

- **Data at rest**
- **Data in motion**
- **Data from a web server**
- **Data from a data warehouse**

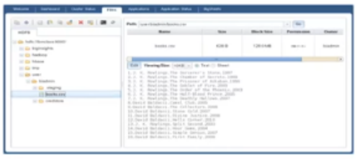
Data is at rest

Data at rest

- Data is already generated into a file or directories.
- No additional updates will be done to the file and will be transferred over "as is"

Standard HDFS shell commands

- use the **cp** or **copyFromLocal** or **put** commands



Data in motion

Data is motion

Lots of servers generating log files or other data

For example:

- A file that is getting appended
- A file that is being update

Data from multiple locations that needs to be merged into one single source

For example:

- Merge all log files from one directory into one file

Solution if data is from a data warehouse

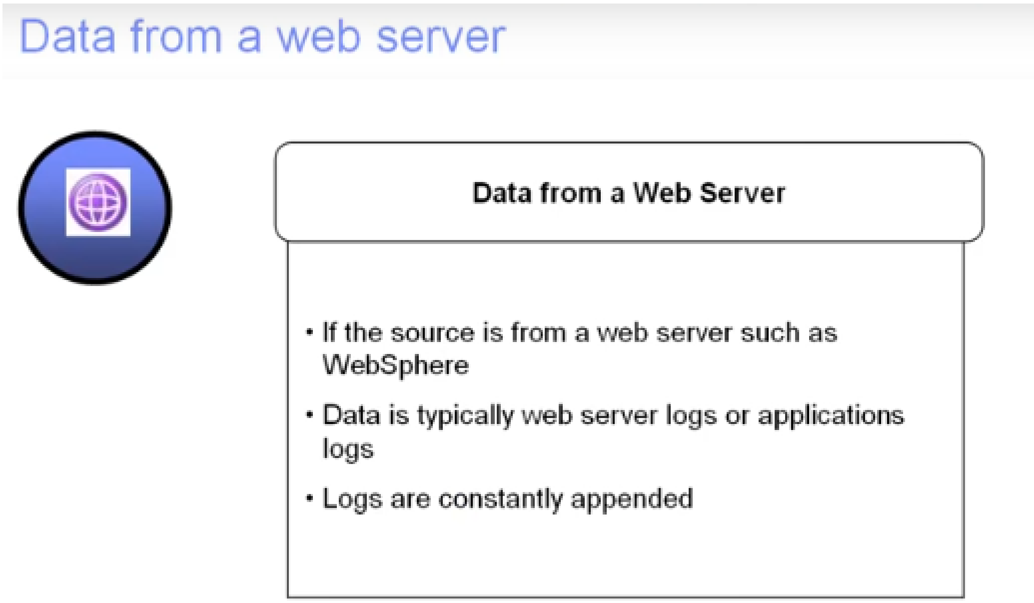
Data from a Data Warehouse

1. Using standard database export commands, export the tables into comma delimited files
2. Use Hadoop commands to import the data HDFS
3. Sqoop to load from relational systems
4. Data from DB2 and Netezza – direct import via the Jaql modules
5. Data from DB2, Netezza, and Teradata into BigSQL using the BigSQL Load

Load solution using Flume

Flume – is a distributed service for collecting data and persisting it in a centralized store

- Can be use to collect data from sources and transfer to other agents
- A number of supported sources
- A number of supported sinks
- Flume agents are placed in source and target locations



Moving Data into Hadoop: Load Scenarios
Once we have learned about the importance of putting big data into Hadoop, an early question is related to data life cycle and data movement.
How do you load data into the cluster? How do you automate the flow of this humongous amount of data? We look in this video at four different load scenarios.
After completing this topic, you wiil be able to list the different scenarios to load data into Hadoop.
We will look at four different load scenarios: Data at rest
Data in motion Data from a web server or a database log,
and Data from a data warehouse
Data at rest is data that is already in a file in some directory.
It is at rest, meaning that no additional updates are planned on this data and it can be transferred as is. the transfer can be accomplished using standard HDFS shell commands, for example., cp or copyFromLocal or put, or using the BigInsights web console
What about when data is in motion? First of all, what is meant by data in motion?
This is data that is continuously being updated: New data might be added regularly to these data sources, Data might be appended to a file, or
Discrete or different logs might be getting merged into one log.
You need to have the capability of merging the files before copying them into Hadoop.
Examples of data in motion include: Data from a web server such as WebSphere Application Server or an Apache web server Data in database server logs or application logs
When moving data from a data warehouse, or any RDBMS for that matter, we could export the data and then use Hadoop commands to import the data.
There is a separate video on Sqoop in this course.
If you are working with a Netezza system, then you can use the Jaql Netezza module to both read from and write to Netezza tables. Data can also be moved using BigSQL Load.
We also have Flume. You will see separate videos later dealing just with Flume.
Flume is a three tiered distributed service for data collection and possibly processing of the data that consists of logical nodes. The first tier, or agent tier, has Flume agents

installed at the sources of the data. These agents then send their data to the second tier, or collector tier. The collectors aggregate the data and in turn forward the data to the final storage tier such as HDFS. Each logical node has a source and a sink.
The source tells from where to collect data and the sink specifies to where the data is to be sent. Interceptors (sometimes called Decorators or Annotators) can be optionally configured to allow for some simple data processing on data it is passed through. Flume uses the concept of a physical node.
A physical node corresponds to a single Java process running on one machine in a cluster as a single JVM. Here the concepts of physical machine and node are usually synonymous. But sometimes a physical node can host multiple logical nodes.
We will see that Flume is a great tool for collecting data from a web server or from database logs. Another, alternate, approach here would be to use Java Management Extension (JMX) commands.
You have now completed this video.

MODULE 2- USING SQOOP

Workings of Sqoop

After completing this topic, you should be able to:

- Explain the use of Sqoop to
  - Import data from relational database tables into HDFS
  - Export data from HDFS into relational database tables
- Describe the basic Sqoop
  - Import statement
  - Export statement

Overview of Sqoop

- Transfers data between Hadoop and relational databases
  - Uses JDBC
  - Must copy the JDBC driver JAR files for any relational databases to $SQOOP_HOME/lib
- Uses the database to describe the schema of the data
- Uses MapReduce to import and export the data
  - Import process creates a Java class
    - Can encapsulate one row of the imported table
  - The source code of the class is provided to you
    - Can help you to quickly develop MapReduce applications that use HDFS-stored records

Sqoop connection

- Database connection requirements are the same for
  - Import
  - Export
- Specify
  - JDBC connection string
  - Username
  - Password

```
sqoop import/export --connect jdbc:db2://your.db2.com:50000/yourDB
        --username db2user --password yourpassword . . .
```

Sqoop import

- Imports data from relational tables into HDFS
  - Each row in the table becomes a separate record in HDFS
- Data can be stored
  - Text files
  - Binary files
  - Into HBase
  - Into Hive
- Imported data
  - Can be all rows of a table
  - Can limit the rows and columns
  - Can specify your own query to access relational data
- --target-dir
  - Specifies the directory in HDFS in which to place the data
    - If omitted, the name of the directory is the name of the table

Data Movement: Working with Sqoop
Apache Sqoop is a tool designed for efficiently transferring bulk data between Hadoop and structured data stores such as relational databases.
Sqoop has a command-line interface for transferring data. It supports incremental loads to/from
a single datanase table, or a free-form SQL query, as well as scripts that can be run whenever needed to import updates made to a database since the last import. Sqoop can be used also to populate tables in Hive or HBase.
Sqoop provides a set of high-performance open-source connectors that can be customized for your
specific external connections. Sqoop offers specific connector modules that are designed for different product types. In this video we do not intended to cover all aspects of Sqoop, but rather give you an idea of the capabilities of Sqoop and let you know where it fits in the Hadoop ecosystem. Sqoop successfully graduated from incubator status in March of 2012 and is now a top-level Apache project:
After completing this topic, you should be able to:
Explain the use of Sqoop to Import data from relational database tables
into HDFS Export data from HDFS into relational database
tables Describe the basic Sqoop
Import statement Export statement
Sqoop is designed to transfer data between relational database systems and Hadoop. It uses JDBC to access the relational systems. To use it with BigInsights, you must copy the JDBC driver JAR for the relational database to be accessed into the $SQOOP_HOME/lib directory
so that the driver can be used by the Sqoop software.
Sqoop accesses the database so that it can understand the schema of the data involved in a transfer. It then generates a MapReduce application
to import or export the data from/to the database/. When you use Sqoop to import data into Hadoop,
Sqoop generates a Java class that encapsulates one row of the imported table. You have access to the actul source code for the generated Java class. This can allow you to quickly develop other MapReduce applications that use the records that Sqoop stored into HDFS.
The connection information is the same whether you are doing an import or an export. You specify a JDBC connection string, the username, and the password.
In the example on the slide, you use the keyword import or the keyword export — just one, not both at the same time — depending on the action you want to perform.
In this example you are connecting to a DB2 system that listens on port 50000 and is running on a system with a hostname of your.db2.com. The connection is made with a userid of db2user
and a password of db2password. Note here that you follow the usual UNIX/Linux convention that single letter parameters use a single-dash, or, as in this case, double-dash because all the parameters are word parameters (for example, dash dash connect or dash dash username). Additional required and optional arguments
are not shown.
The Sqoop import command is used to extract data from a relational table and load it into Hadoop. Each row in HDFS comes from a row in the corresponding table. The resulting data in HDFS can be stored as text files or binary files, as well as imported directly into HBase or Hive. By default, all columns of all rows are imported,
however, there are arguments that allow you to specify particular columns or specify a WHERE clause to limit the rows. You can even specify your own query to access the relational data. If you want to specify the location of the
imported data, use the --target-dir argument. Otherwise the target directory name will be

the same as the table name. We will look at configuring Sqoop in a separate video where we will look in more detail at basic Sqoop import and export statements.

Sqoop import examples

- Import all rows from a table

```
sqoop import --connect jdbc:db2://your.db2.com:50000/yourDB \
    --username db2user --password db2password --table db2table \
    --target-dir sqoopdata
```

- Addition parameters:

```
--split-by tbl_primarykey
--columns "empno,empname,salary"
--where "salary > 40000"
--query 'SELECT e.empno, e.empname, d.deptname
        FROM employee e JOIN department d on (e.deptnum = d.deptnum)'
--as-textfile
--as-avrodatafile
--as-sequencefile
```

Sqoop exports

- Exports a set of files from HDFS to a relation database system
  - Table must already exist
  - Records are parsed based upon user's specifications
- Default mode is *insert*
  - Inserts rows into the table
- *Update* mode
  - Generates update statements
  - Replaces existing rows in the table
  - Does not generate an *upsert*
    - Missing rows are not inserted
    - Not detected as an error
- *Call* mode
  - Makes a stored procedure call for each record
- --export-dir
  - Specifies the directory in HDFS from which to read the data

Sqoop export examples

- Basic export from files in a directory to a table

```
sqoop export --connect jdbc:db2://your.db2.com:50000/yourDB \
        --username db2user --password db2password --table employee \
        --export-dir /employeedata/processed
```

- Example calling a stored procedure

```
sqoop export --connect jdbc:db2://your.db2.com:50000/yourDB \
        --username db2user --password db2password --call empproc \
        --export-dir /employeedata/processed
```

- Example updating a table

```
sqoop export --connect jdbc:db2://your.db2.com:50000/yourDB \
        --username db2user --password db2password --table employee \
        --update_key empno --export-dir /employeedata/processed
```

Additional export information

- Parsing data
  - Default is comma separated fields with newline separated records
  - Can provide input arguments that override the default
    - If the records to be exports loaded into HDFS using the import command
      The original generated Java class can be used to read the data

- Transactions
  - Sqoop uses multi-row insert syntax
  - Inserts up to 100 rows per statement
  - Commits work every 100 inserts
    - Commit every 10,000 rows
  - Each export map task operates as a separate transaction

Working with Sqoop (continued)
Let's look at a basic Sqoop import statement that extracts all rows and all columns from a table, db2table, that is in a DB2 database called yourDB. The results are stored in a directory in HDFS called sqoopdata. sqoop import --connect
jdbc:db2://your.db2/com:50000/yourDB
\ --username db2user --password db2password
--table db2table \ --target-dir sqoopdata
Sqoop imports the data in parallel. You can override the number of mappers that

Sqoop is to use. The default is 4. To split the data across multiple mappers,
by default, Sqoop uses the primary key of the table. It determines the minimum and maximum
values for the key and then assumes an even distribution of values. You can use the --split-by
argument to have the distribution work with a different column. If the table does not
have an index column, or has a multi-column key, then you must specify the split-by column
parameter. To only import data from a subset of columns,
use the --columns argument where the column names are comma separated. To limit the rows
use the --where argument, and supply your own query that returns the rows to be imported.
This allows for greater flexibility, for example allowing you to get data by joining tables.
By default the imported data is in delimited text format (--as-textfile). Optional parameters
allow you to import in binary format (--as-sequencefile) or as an Avro data file (as avrodatafile).
Also, you can override the default in order to have the data compressed.
The Sqoop export command reads data in Hadoop and places it into relational tables (you
export from HDFS into a database). The target table must already exist and you can specify
your own parsing specifications. By default, Sqoop inserts rows into a relational
table. This is primarily intended for loading data into a new table. If there are any errors
when doing the insert, the export process will fail.
However, there are other export modes. The update mode — the second mode — causes
Sqoop to generate update statements. To do updates, you must specify the --update-key
argument. Here you tell Sqoop which table column (or comma-separated columns) to use
in the WHERE clause of the update statement. If the update does not modify a row, it is
not considered to be an error. The condition just goes undetected.
Some database systems allow for --update-mode allowinsert to be specified — these are
databases that have an UPSERT command (UPSERT does an update if the row exists, but otherwise
inserts a new row). The third mode is call mode. With this mode,
Sqoop calls and passes the record to a stored procedure.
The --export-dir parameter defines the location of the files in HDFS that are to be exported
froom HDFS to put records into the database.
Here are a few export examples. The first reads the files in the /employeedata/processed
directory and inserts the records into the employee table in the yourDB database.
The second example accesses the same data but calls a stored procedure.
(Note the --call parameter). The third example shows an update of the employee
table from data in the same HDFS directory. The empno column is used in the WHERE clause
of the generated update statement.
Now for a couple of additional pieces of information. By default, Sqoop assumes that it is working
with comma-separated fields and that each record is terminated by a newline. Both the
import and export commands have the facility to allow you to override this behavior.
Remember, that when data is imported into Hadoop, you are given access to the Java source
for the Java class what was generated. If your data was not in the default format and,
if the data that you are exporting is in that same format, then you can use parts of that
same code to read the data. What about committing data to your database?
Transactions? When Sqoop is inserting rows into a table,
it generates a multi-row insert. Each multi-row insert handles up to 100 rows. The mapper
then does a commit after 100 statements are executed. This means that 10,000 rows are
inserted before being committed. Also, each export mapper in the generated
MapReduce program commits with separate transactions.
You have now completed this video and can work with the lab exercise.

MODULE 3 – FLUME OVERVIEW

Flume

"A flume is an open artificial water channel, in the form of a gravity chute, that leads water from a diversion dam or weir completely aside a natural flow.

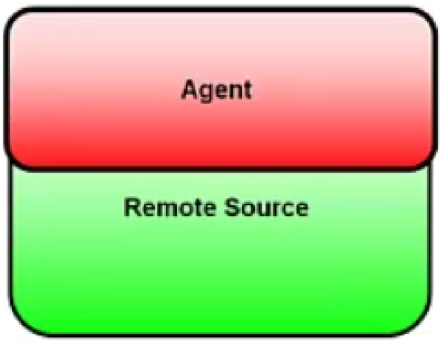
"Often, the flume is an elevated box structure (typically wood) that follows the natural contours of the land. These have been extensively used in hydraulic mining and working placer deposits for gold, tin, and other heavy minerals.

"**They are also used in the transportation of logs in the logging industry**, electric power generation, and to power various mill operations by the use of a waterwheel."

— *Wikipedia*

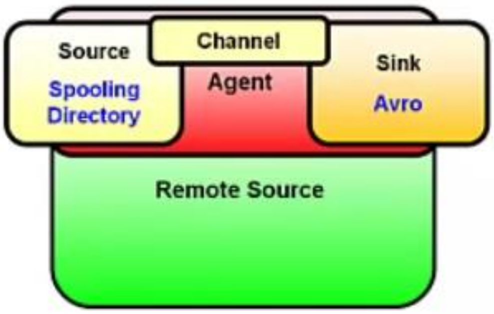
How Flume works (1 of 3)

- It is built on the concept of flows
- Flows might have different batching or reliability setup
- Flows are comprised of nodes chained together



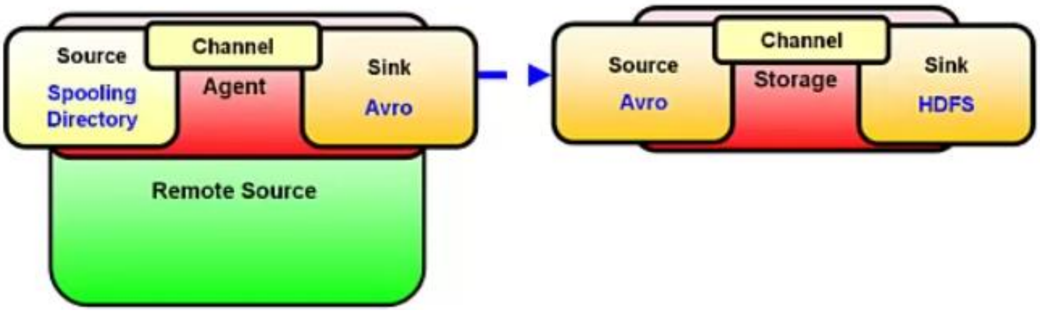
How Flume works (2 of 3)

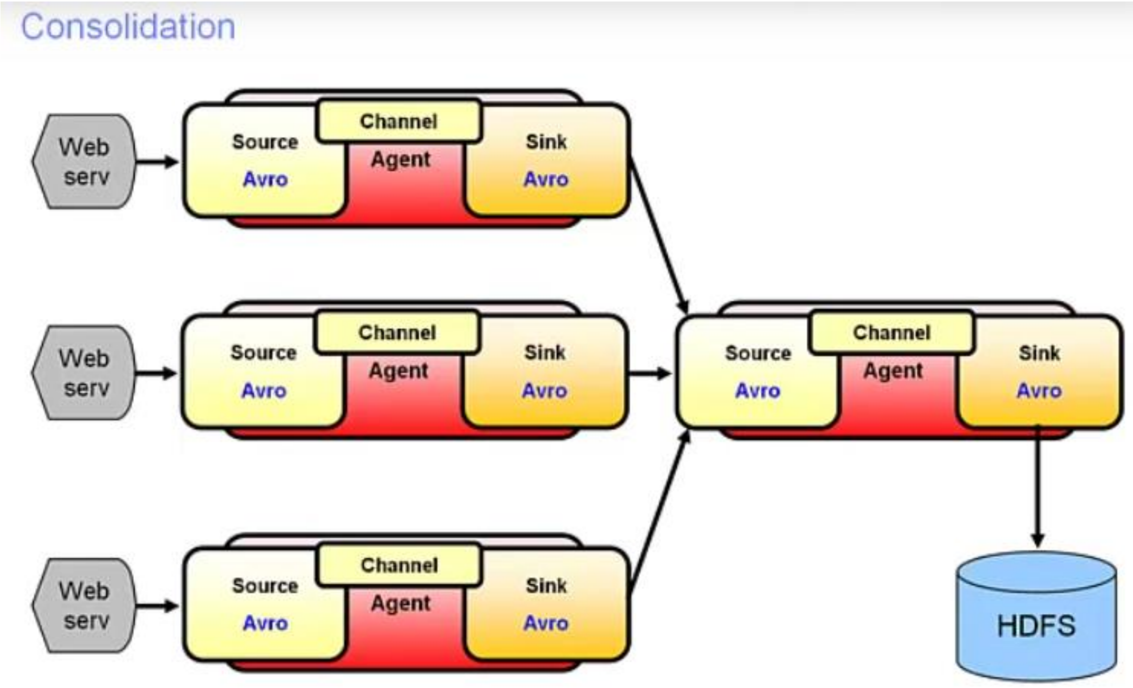
- It is built on the concept of flows
- Flows might have different batching or reliability setup
- Flows are comprised of nodes chained together
  - Each node receives data as "source", stores it in a channel, and sends it via a "sink"

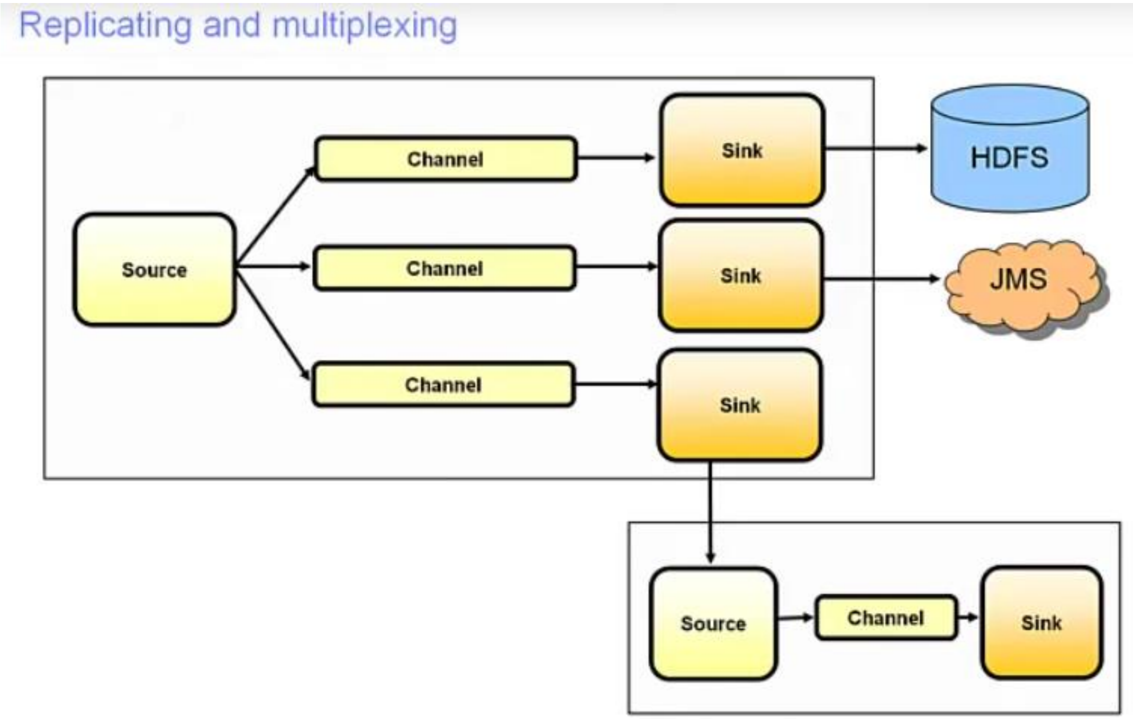


How Flume works (3 of 3)

- It is built on the concept of flows
- Flows might have different batching or reliability setup
- Flows are comprised of nodes chained together
  - Each node receives data as "source", stores it in a channel, and sends it via a "sink"







Data Movement: Workings of Flume
We will now look at yet another method for moving data into HDFS: Flume.
After completing this topic, you should be able to:
Describe Flume and how it is used with Hadoop Explain how Flume works
You should note the origin of the word Flume. It comes from Latin, via French. The Latin word flumen / flumenis is a river.
The English word Flume has the following meaning. A flume is an open artificial water channel in the form of a gravity chute that leads water from a diversion dam or weir using a

natural flow downhill. Flumes are used in the transportation of logs in the logging industry. The use of the word Flume here — with Flume software — is thus metaphorical. Flume — in the world of Hadoop — is used typically to transport web log records or database log records, and not wooden logs, from the log file where records were deposited to a central repository in the Hadoop distributed file system (HDFS) for analysis by MapReduce programs.

Flume is built on the concept of flows. The various sources of the data sent through Flume may have different batching or reliability setup. Often logs are continually being added to and what we want are the new records as they are added.

Any number of Flume agents can be chained together. You begin by starting a Flume agent where the data originates. This data then flows through a series of nodes that are chained together

Every Flume agent works with both a source and a sink. (Actually a single agent can work with multiple sources and multiple sinks.) Sources and sinks are wired together via a channel

. Each node receives data as "source," stores it in a channel, and sends it via a "sink."

An agent running on one node can pass the data to an agent on a different node. The agent on the second node also works with a source and a sink.

There are a number of different types of sources and sinks supplied with the product. (We will look at the types of source and sink in the next video.) For data to be passed from one agent to another, we can have an Avro sink on the first communicating with an Avro source on the second. Avro is a remote procedure call and serialization framework that is a separate Apache project. It uses JSON for defining data types and protocols

and it serializes data in a compact binary format.

Flume supports more that just a multi-tiered topology. This is an example of a consolidation topology. Here a single Avro source receives data from multiple Avro sinks.

Both a replication and a multiplexing topology are also supported.

In a replicating topology, log events from the source are passed to all channels connected to that source. In a multiplexing topology, data in the header area of a log event can be queried and used to distribute the event to one or more channels.

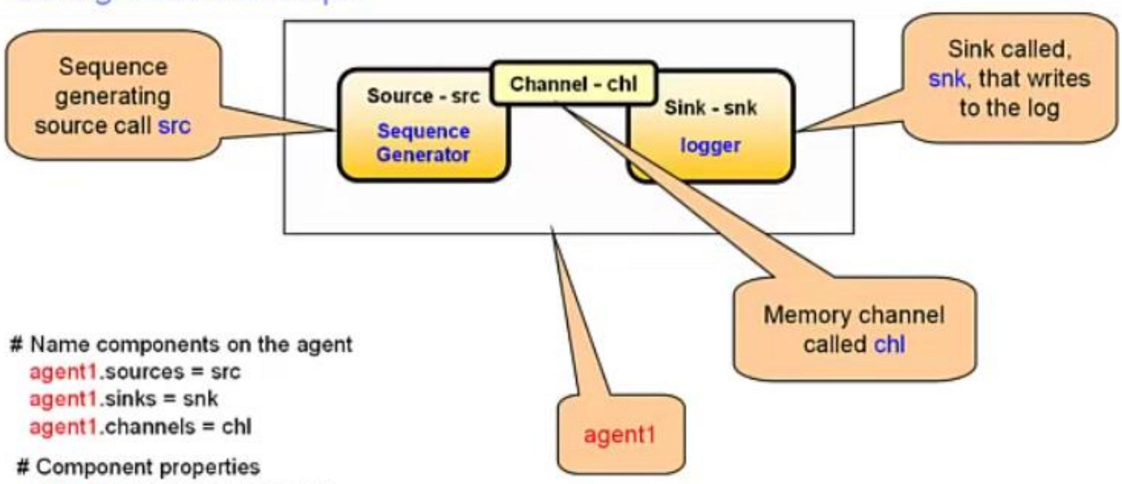
We have seen how Flume works with Hadoop at a high level.

In the next video we will look at how one configures Flume

Configuration

- Flume components are defined in a configuration file
  - Multiple agents running on the same node can be defined in the same configuration file
- For each agent, you define the components
  - The source(s)
  - The sink(s)
  - The channel(s)
- Then define the properties for each component
- Then define the relationships between the components
- The configuration file resembles a Java properties format

Configuration example



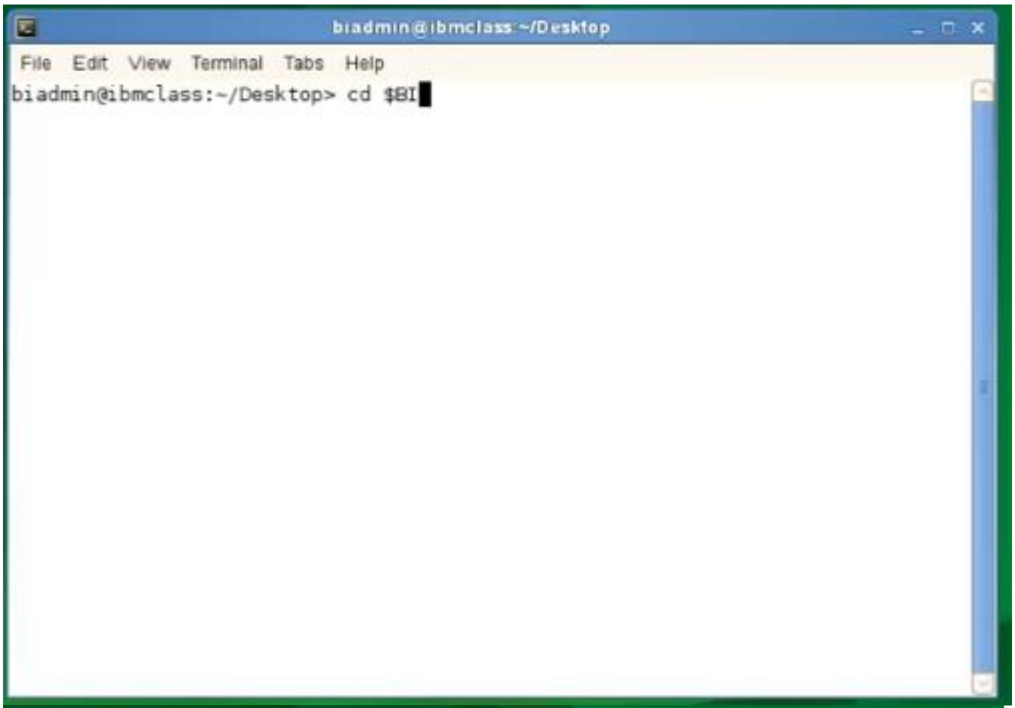

```
# Name components on the agent
  agent1.sources = src
  agent1.sinks = snk
  agent1.channels = chl
# Component properties
  agent1.sources.src.type = seq
  agent1.sinks.snk.type = logger
  agent1.channels.chl.type = memory
  agent1.channels.chl.capacity = 100
# Bind the components
  agent1.sources.src.channels = chl
  agent1.sinks.snk.channel = chl
```

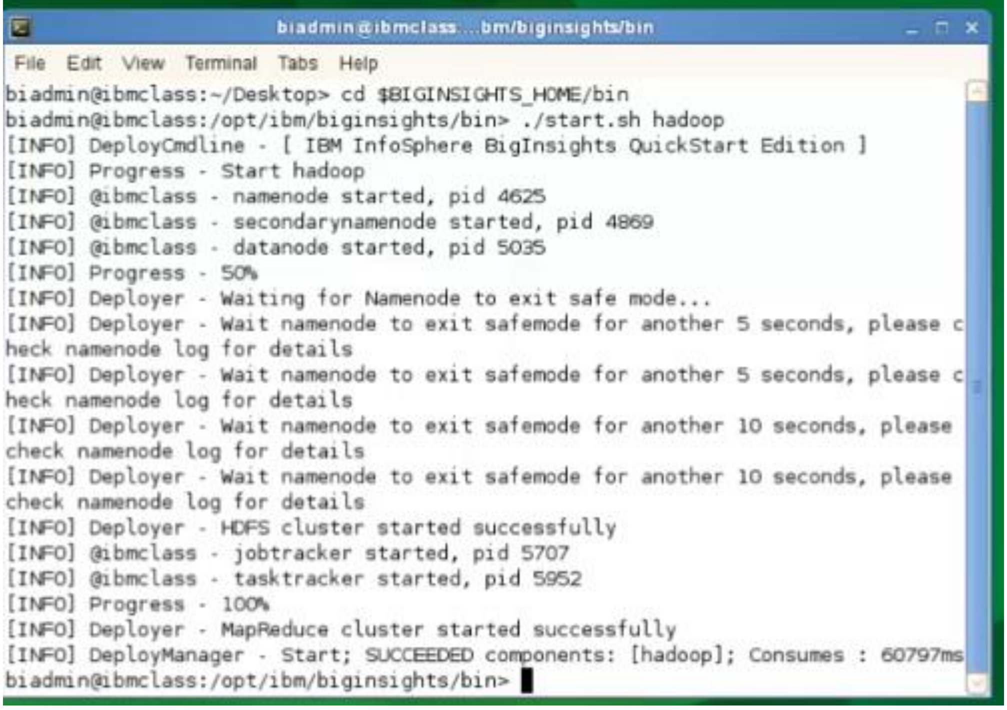
Skip to a navigable version of this video's transcript.
4:09 / 4:16
Maximum Volume.
Skip to end of transcript.

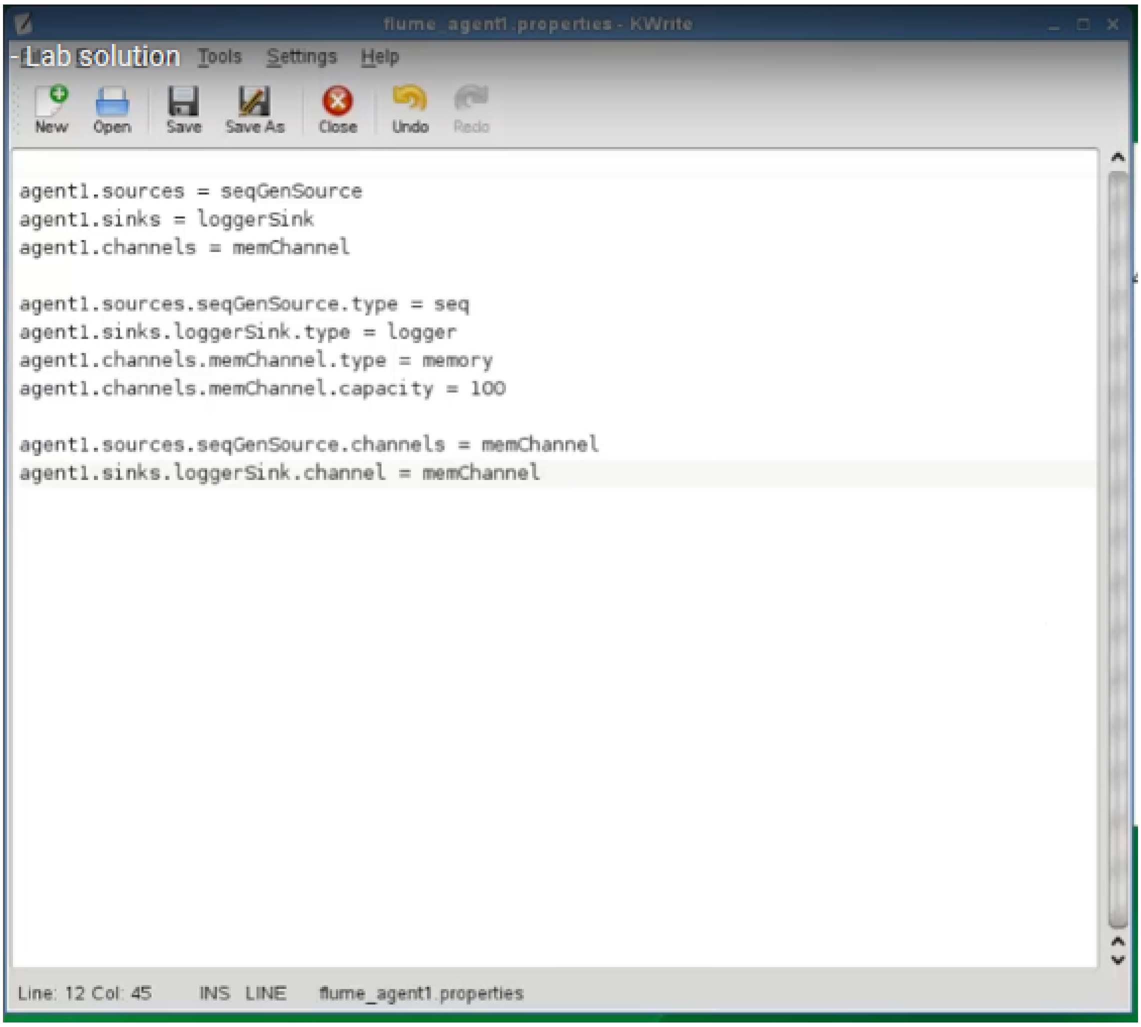
Data Movement: Configuring Flume
In this video we will look at some aspects of configuring Flume.
After completing this topic, you will be able to:
List the Flume configuration components Describe how interceptors can be used to modify or drop events Explain how sources and sinks are linked using

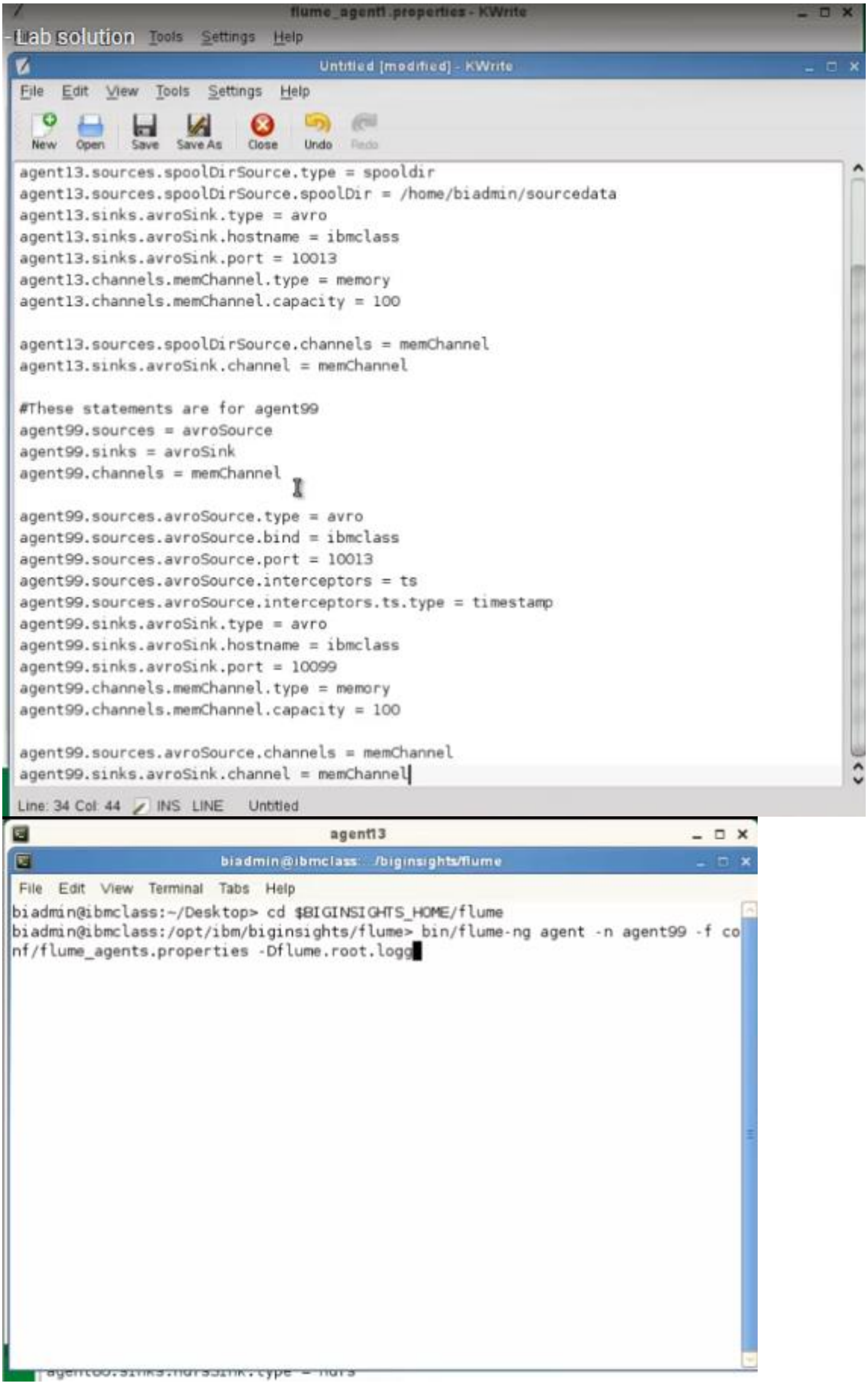
channels Describe how to start a Flume agent
Flume components are defined in a configuration file.
Where there multiple agents running on the same node, they can all be defined in the one configuration file. For each agent, you define the components:
The source(s) The sink(s)
The channel(s For an agent to run, it must have a source from which to get data, a sink that writes to a target, and a channel which specifies where the data is to be temporarily held until it is written by the sink. This information is supplied to the agent via the configuration file that the agent loads at startup.
Within the configuration file each source, sink, and channel are given names. Properties are assigned to each component and you define the relationships between the components.
The configuration file is similar to a Java properties file.
Let's start out by looking at a configuration example. The intention here is not to go into a lot of detail but to begin to lay a foundation so that the following information makes a little sense. By the way, there is no required order in defining components, specifying properties, and defining relationships. The definitions file is declarative in nature and not procedural. In this visual, the name of the agent to run on this system is agent1. (More than one agent can run on a system.
This example, for ease of learning, only has one.)
The source for agent1 is going to be a sequence generator with a name of src.
(A sequence generator continuously generates events and is used for testing.)
The sink, in this case, is a logger sink with a name of snk.
(A logger sink logs events at the INFO level and is also used for testing.)
The source and the sink are wired together via a channel named chl.
Next look at the configuration file definitions. Note that all statements begin with agent1. (agent1 dot). These are attributes of agent agent1. This is how an agent detects applicable configuration statements. As stated before, it is possible to run multiple agents on a system, and all of those agents can be configured the same configuration file.
Agents only pay attention to configuration statements that are prefixed by their name.
The three sections: Name the components on the agent
Provide component properties Bind the components
The third of these — "bind the components" — is the set of statements that wire the source and the sink together. Since both src and snk are connected to the same channel, they are wired together. We will continue Configuring Flume in the next video, where we will get into more detail about the components.

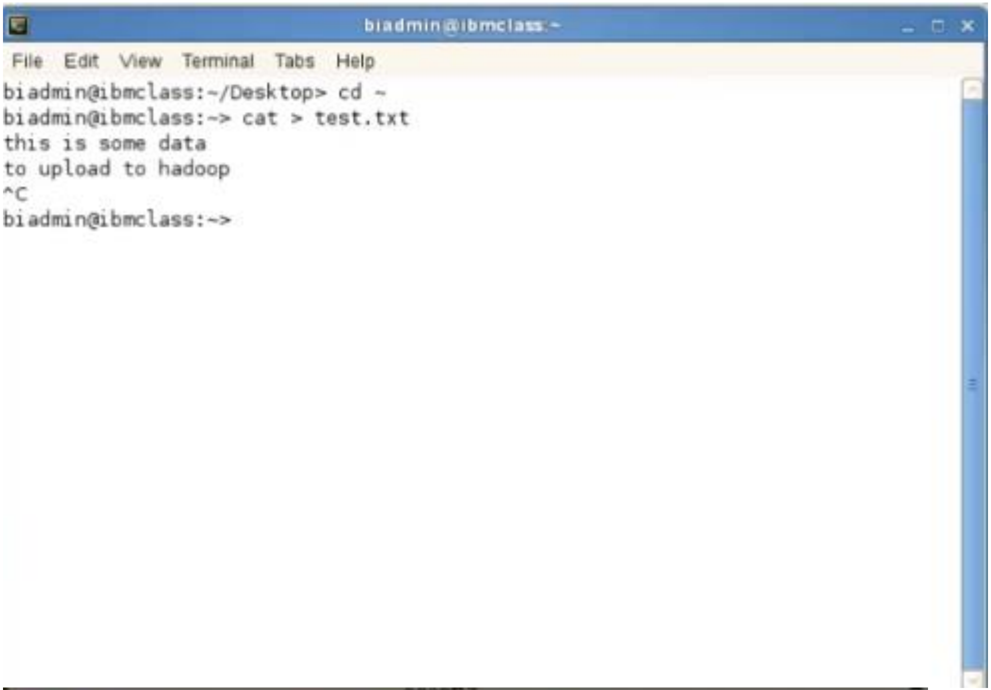
>>Lab:

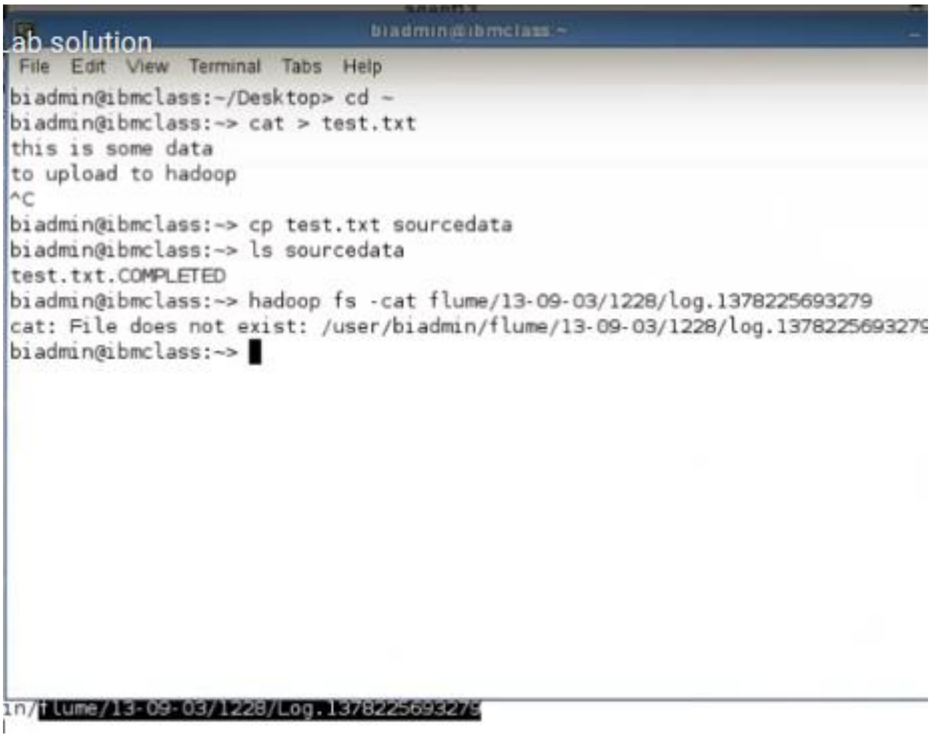

biadmin@ibmclass:~/Desktop
File Edit View Terminal Tabs Help
biadmin@ibmclass:~/Desktop> cd $BI


biadmin@ibmclass:...bm/biginsights/bin
File Edit View Terminal Tabs Help
biadmin@ibmclass:~/Desktop> cd $BIGINSIGHTS_HOME/bin
biadmin@ibmclass:/opt/ibm/biginsights/bin> ./start.sh hadoop
[INFO] DeployCmdline - [IBM InfoSphere BigInsights QuickStart Edition]
[INFO] Progress - Start hadoop
[INFO] @ibmclass - namenode started, pid 4625
[INFO] @ibmclass - secondarynamenode started, pid 4869
[INFO] @ibmclass - datanode started, pid 5035
[INFO] Progress - 50%
[INFO] Deployer - Waiting for Namenode to exit safe mode...
[INFO] Deployer - Wait namenode to exit safemode for another 5 seconds, please check namenode log for details
[INFO] Deployer - Wait namenode to exit safemode for another 5 seconds, please check namenode log for details
[INFO] Deployer - Wait namenode to exit safemode for another 10 seconds, please check namenode log for details
[INFO] Deployer - Wait namenode to exit safemode for another 10 seconds, please check namenode log for details
[INFO] Deployer - HDFS cluster started successfully
[INFO] @ibmclass - jobtracker started, pid 5707
[INFO] @ibmclass - tasktracker started, pid 5952
[INFO] Progress - 100%
[INFO] Deployer - MapReduce cluster started successfully
[INFO] DeployManager - Start; SUCCEEDED components: [hadoop]; Consumes : 60797ms
biadmin@ibmclass:/opt/ibm/biginsights/bin>


flume_agent1.properties - KWrite
Lab solution
Tools
Settings
Help
New
Open
Save
Save As
Close
Undo
Redo
agent1.sources = seqGenSource
agent1.sinks = loggerSink
agent1.channels = memChannel
agent1.sources.seqGenSource.type = seq
agent1.sinks.loggerSink.type = logger
agent1.channels.memChannel.type = memory
agent1.channels.memChannel.capacity = 100
agent1.sources.seqGenSource.channels = memChannel
agent1.sinks.loggerSink.channel = memChannel
Line: 12 Col: 45
INS LINE
flume_agent1.properties


flume_agent1.properties - KWrite
Lab solution
Tools
Settings
Help
Untitled [modified] - KWrite
File
Edit
View
Tools
Settings
Help
New
Open
Save
Save As
Close
Undo
Redo
agent13.sources.spoolDirSource.type = spooldir
agent13.sources.spoolDirSource.spoolDir = /home/biadmin/sourcedata
agent13.sinks.avroSink.type = avro
agent13.sinks.avroSink.hostname = ibmclass
agent13.sinks.avroSink.port = 10013
agent13.channels.memChannel.type = memory
agent13.channels.memChannel.capacity = 100
agent13.sources.spoolDirSource.channels = memChannel
agent13.sinks.avroSink.channel = memChannel
#These statements are for agent99
agent99.sources = avroSource
agent99.sinks = avroSink
agent99.channels = memChannel
agent99.sources.avroSource.type = avro
agent99.sources.avroSource.bind = ibmclass
agent99.sources.avroSource.port = 10013
agent99.sources.avroSource.interceptors = ts
agent99.sources.avroSource.interceptors.ts.type = timestamp
agent99.sinks.avroSink.type = avro
agent99.sinks.avroSink.hostname = ibmclass
agent99.sinks.avroSink.port = 10099
agent99.channels.memChannel.type = memory
agent99.channels.memChannel.capacity = 100
agent99.sources.avroSource.channels = memChannel
agent99.sinks.avroSink.channel = memChannel
Line: 34 Col: 44
INS
LINE
Untitled
agent13
biadmin@ibmclass: /biginsights/flume
File
Edit
View
Terminal
Tabs
Help
biadmin@ibmclass:~/Desktop> cd $BIGINSIGHTS_HOME/flume
biadmin@ibmclass:/opt/ibm/biginsights/flume> bin/flume-ng agent -n agent99 -f co
nf/flume_agents.properties -Dflume.root.logg





Since
Hadoop was stopped in a previous exercise
I'm going to start it right now
and this is so we'll be able to copy data
using flume from our local filesystem
into Hadoop initially
we're just going to do some testing
with Flume we're going to use
one of the sources that comes with Flume that creates
sequential data and we will use a sink
that logs or writes the

output out to a log and we're gonna have the log directed to the console
just to kind of see if we can get this working start in editor
to simplify things
I'm going to copy in the
configuration statements we have a source for agent1
defined called seqGenSource
we have a sink for agent2 called loggerSink
and a channel called memChannel
the next thing we want to do is
define properties for these so for the
seqGenSource it
its type is seq meaning it's going to
utilize the sequential generator the type for
loggerSink is logger it's gonna write output
to the log memChannel
type is memory the events will be stored in memory
and we're defining a capacity of 100 events
we're wiring the
seqGenSource to the
memChannel and we're going to do the same for the loggerSink
which basically means that seqGenSource
is connected to the logger sink via the memory
channel let's save our file
we're going to save it in the flume
configuration directory
we'll call it
flume_agent1.properties
and then we will start an agent
that is going to
read it that configuration file that we just created
so we'll change to the flume directory
execute out of the bin directory underneath the flume directory
flume -ng we're gonna
invoke an agent the name of that agent is going to be
agent1 hence the reason all over our configuration commands began
with agent1 and the configuration file is found in the conf directory
underneath flume and it was called flume_ agent1.properties
we're going to
override and specify that our logging
is going to information logging is gonna go to the console
we're executing this and there you see
sequential records being
written to the log so basically what this says is
we got it to work we created a little test
scenario and we are able to get
a flume agent to do what we wanted to do
so the next step is to go
into something a little bit more complicated and in this case
we're going to find three agents we're going to
pretend like we got data on one system
which is going to be transferred to a second system where it's gonna be
manipulated and then

sent to a third system where it will be written into
HDFS you'll notice on the first system we've got
for agent13 a source called spoolDirSource
source a sink called avroSink and a
channel called manChannel for our
spoolDirSource it is a spool directory type
and it is going to read from the source
data directory in BI admin the sink type is avro
it's going to be listening on ibmclass port
10013 we've defined
a second set of configuration statements for agent99
its source is avroSouce its sink is avroSink
and it has a channel memChannel notice
avroSink, memChannel those only have to be unique within a particular
agent the names
the avroSource is going to
be listening on the same port as the previous
avroSink was writing out to we are utilizing interceptors
we're gonna put a timestamp into the header information of our events
and we're gonna write out using an avro
sink and it's going to
be written out on port 10099
and then we're going to read that avro data in again
using a source called avroSource
we're gonna write out to a sink called hdfsSink
type HDFS means it's gonna go into Hadoop
there is the path notice the %y
%m %d those are year month day hour minute
and that information is extracted from the header
of the event which was put in by the previous agent
format is going to be text and the file is going to be prefixed by
log so we'll save this
configuration file
I'm going to start-up
an agent
agent13 I always
have to switch to the flume directory and then I'm executing
out of the bin directory all three
of these agents will be reading from
the same configuration file there's nothing wrong with that
because the statements for each
agent are all prefix by the agent's
name we're starting
agent13
now it's going to be monitoring a directory called source data
and it will be trying to write out
to an avroSink well right then it
went by quickly but the avroSink
wasn't working yet and the reason is because it needs
a complimenting avro
source so now we're going to start
agent99 agent99 the

avro source for agent99
works with the avroSink for agent13
so once we start agent99
we're gonna see that the
sink for agent13 was able to
start working properly right so we got connected and there we see that the avroSink
is now being able to work with port
10013 however
agent99 is in that same situation its avroSink
is still looking for an avroSource
and that will come about when we start agent86
now agent86 it's sink
is going to write out into HDFS
we will see that once
agent86 get started that the
after a sink for agent99
is able to be connected and we will have
all over agents running the way we want
notice that agent99's
sink is now connected gonna open another
command line and I am
going to create a file called
test.txt just gonna have two lines
of text in it this is some data
the second line to be uploaded are to upload to Hadoop
I'm now gonna
copy that file into the sourcedata
directory this is the one is being monitored by
agent13 and if we do a list of that directory we're gonna see that
test.txt is now completed
meaning it's been processed and if we look at agent86 we can see
where the data is being written into
HDFS so now I'm gonna try to
display that I'm gonna take
the name of the file if you'll notice it's flume/ and then that
%y %d was translated into 13-09-03
in a time of 12:28 and I'm going to do a cat
of that data and we can see
that our data was uploaded into
Hadoop