

CS523 - BDT

Big Data

Technologies

Sqoop & Floom
(Ingesting Data into HDFS)

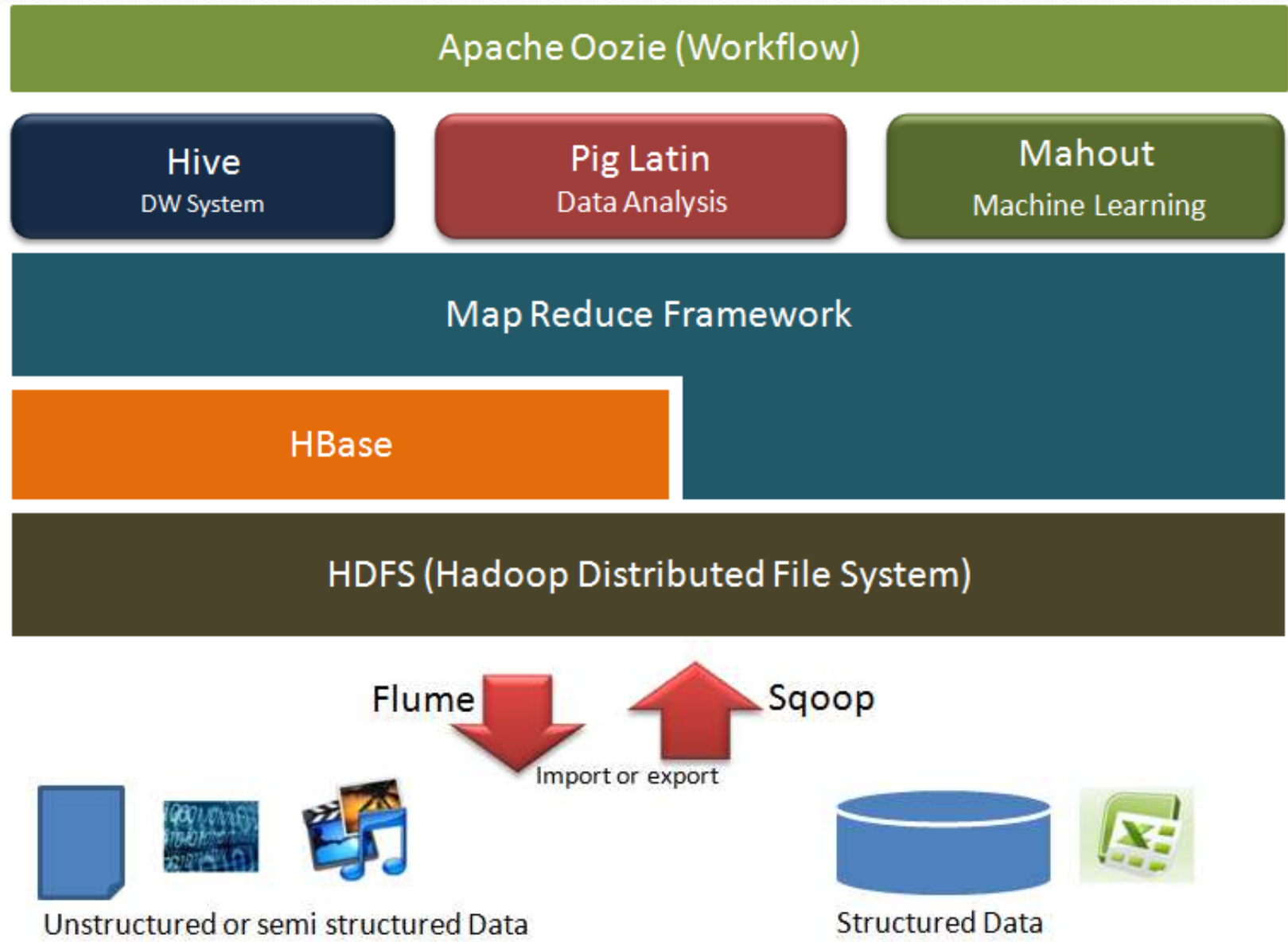
Need for Data Ingestion

- A great strength of the Hadoop platform is its ability to work with very large datasets in several different forms.
- HDFS can reliably store logs and other data formats from a plethora of sources, and MapReduce/Pig/Hive can parse diverse ad hoc data formats, extracting relevant information and combining multiple datasets into powerful results.
- It is often assumed that the data is already in HDFS, however, there are many systems that do not meet this assumption and so there is a need to ingest data from outer sources into HDFS for further processing.

Data Ingestion

- Ingestion is the process of getting data into a system with minimal processing or transformation applied during ingestion.
- **Sqoop** and **Flume** are two data integration frameworks that are heavily used for ingesting data into HDFS.
- Which one of them you use depends on from where the data is coming and how you would need to use it.

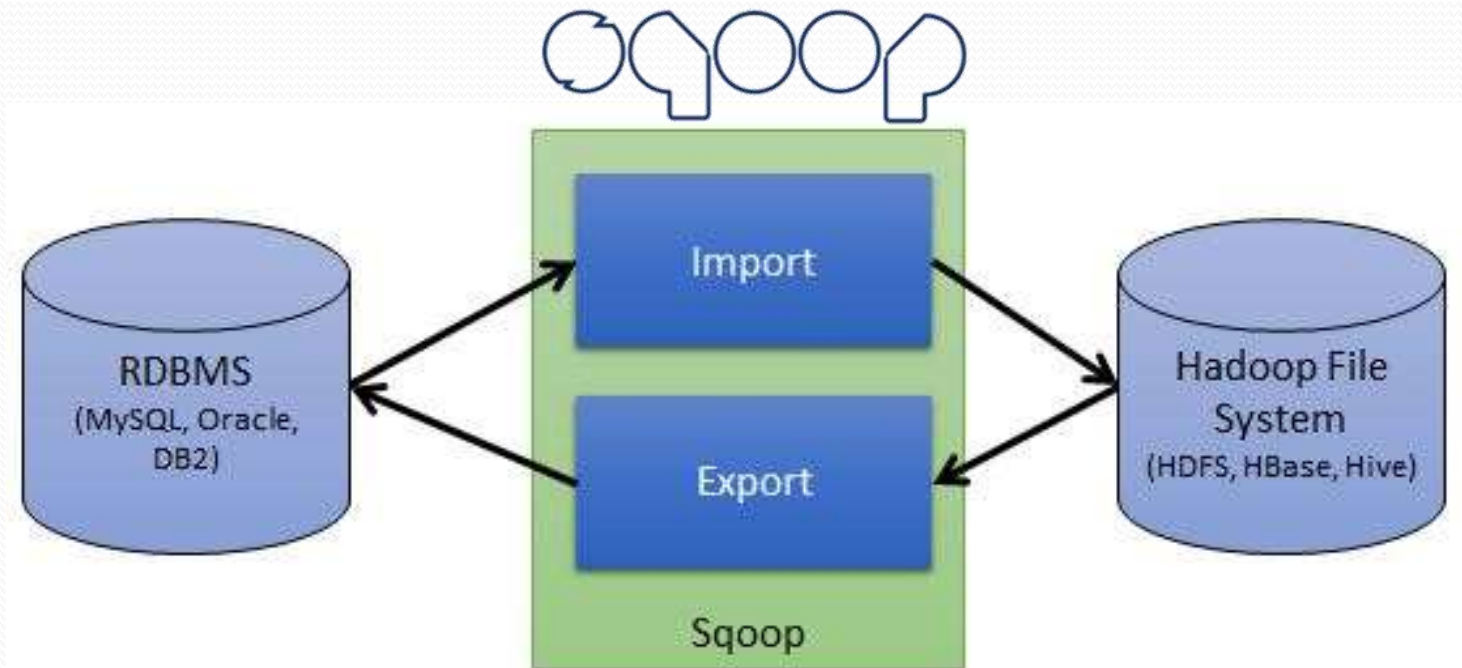
Hadoop Ecosystem



Apache Sqoop



- Often, valuable data in an organization is stored in structured data stores such as relational database management systems (RDBMSs).
- Apache Sqoop is an open source, batch-oriented program that allows users to **import data from a database into HDFS** or **export data from HDFS back to a database**.



Apache Sqoop

- Open source tool developed by Cloudera.
- Uses JDBC to talk to the DataBases and that's why can talk to virtually any DataBase in the world.
- Can import single table or all tables in DataBases
- As a developer you can specify the columns or rows that need to be imported. It also facilitates *select* query for selective import.
- Can directly create Hive tables and push data into it.
- Sqoop takes the commands from user, analyzes the tables and generates Java code that'll take care of data import to HDFS.
- After the java code generation, a map only MapReduce job is run to import the data.
- By default, 4 mappers are run with each mapper importing 25% of the data.

Apache Sqoop

- Data ingested in HDFS using Sqoop can be further used for processing using MapReduce programs or other higher-level tools such as Hive. (It's even possible to use Sqoop to move data from a database into HBase.)
- When the final results of an analytic pipeline are available, Sqoop can export these results back to the data store for consumption by other clients.
- Sqoop can create files in a variety of formats in a Hadoop cluster.
- Sqoop is a very useful tool due to the wide range of databases that it supports, but Sqoop is, by design, entirely batch-oriented.

Available Commands in Sqoop

- **codegen** Generate code to interact with database records
- **create-hive-table** Import a table definition into Hive
- **eval** Evaluate a SQL statement and display the results
- **export** Export an HDFS directory to a database table
- **help** List available commands
- **import** Import a table from a database to HDFS
- **import-all-tables** Import tables from a database to HDFS
- **list-databases** List available databases on a server
- **list-tables** List available tables in a database
- **version** Display version information

See “Sqoop help COMMAND” for information on a specific command.

Available Commands in Sqoop

- **Import**

- Full table or only a subset of a table (specific columns/rows) can be imported from RDBMS to HDFS
- The import process is performed in parallel. For this reason, the output will be in multiple files.

- **Export**

- Data can be exported from HDFS file to RDBMS table
- Sqoop's export process will read a set of delimited text files from HDFS in parallel, parse them into records, and insert them as new rows in a target database table, for consumption by external applications or users.

Incremental updates are also possible in both import and export.

Sqoop Import Command

SQOOP IMPORT

--CONNECT jdbc:mysql://10.0.2.15:3306/cs523

-USERNAME root -PASSWORD cloudera

--TABLE student

--TARGET-DIR= /user/cloudera/sqoopImport

-m 2

Sqoop Import Command

- **IMPORT**: This is the sub-command that instructs Sqoop to initiate an import.
- **--CONNECT <connect string>, --USERNAME <user name>, -PASSWORD <password>** : These are connection parameters that are used to connect with the database. This is no different from the connection parameters that you use when connecting to the database via a JDBC connection.
- **--TABLE <table name>** : This parameter specifies the table which will be imported.
- **--TARGET-DIR** : name of the directory in HDFS in which the imported data will be stored.
- **-m** : number of mappers to use for data import.

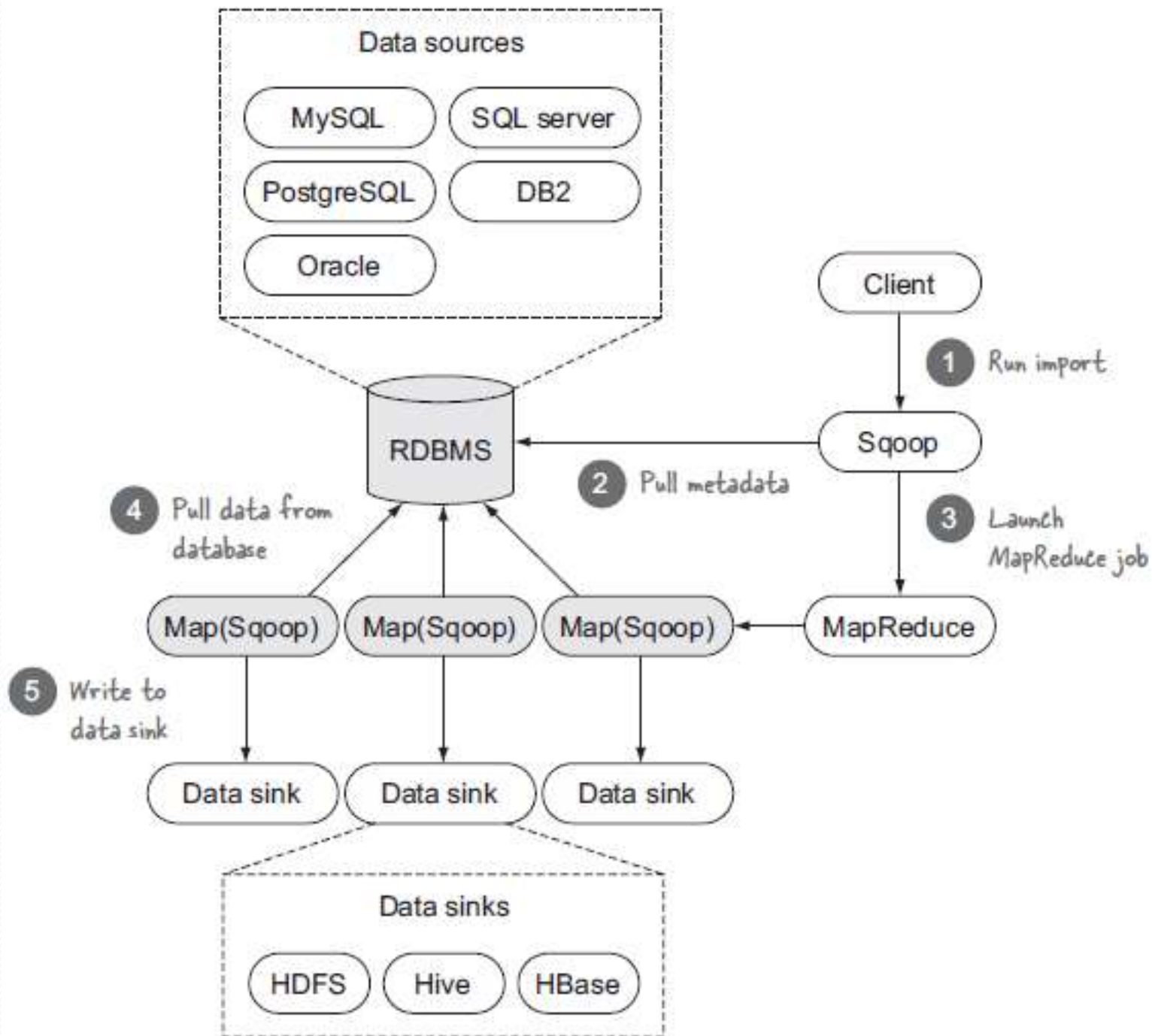
Controlling Parallelism

- Sqoop imports data in parallel from most database sources. You can specify the number of map tasks (parallel processes) to use to perform the import by using the `-m` or `--num-mappers` argument.
- Each of these arguments takes an integer value which corresponds to the degree of parallelism to employ.
- By default, 4 map tasks are used.
- When performing parallel imports, Sqoop needs a criterion by which it can split the workload.
- Sqoop uses a splitting column to split the workload. By default, Sqoop will identify the primary key column (if present) in a table and use it as the splitting column.

Controlling Parallelism

- The low and high values for the splitting column are retrieved from the database, and the map tasks operate on evenly-sized components of the total range.
- For example, if you had a table with a primary key column of `id` whose minimum value was 0 and maximum value was 1000, and Sqoop was directed to use 4 tasks, Sqoop would run four processes; each of them will execute SQL statements of the form *`SELECT * FROM sometable WHERE id >= lo AND id < hi`* with (lo, hi) set to (0, 250), (250, 500), (500, 750), and (750, 1001) in the different tasks.

Sqoop In Action



Sqoop Import Command

- By default, all columns within a table are selected for import. Imported data is written to HDFS in its "natural order".
- By default the files that are getting stored in HDFS contain comma delimited fields, with new lines separating different records.
- You can easily override the format in which data is copied over by explicitly specifying the field separator and record terminator characters.
- Sqoop also supports different data formats for importing data.
 - For example, you can easily import data in Avro data format by simply specifying the option `--as-avrodatafile` with the import command.
- There are many other options that Sqoop provides which can be used to further tune the import operation to suit your specific requirements.

Import Control Arguments

Argument	Description
<code>--append</code>	Append data to an existing dataset in HDFS
<code>--as-avrodatafile</code>	Imports data to Avro Data Files
<code>--as-sequencefile</code>	Imports data to SequenceFiles
<code>--as-textfile</code>	Imports data as plain text (default)
<code>--boundary-query <statement></code>	Boundary query to use for creating splits
<code>--columns <col,col,col...></code>	Columns to import from table
<code>--direct</code>	Use direct import fast path
<code>--direct-split-size <n></code>	Split the input stream every <i>n</i> bytes when importing in direct mode
<code>--inline-lob-limit <n></code>	Set the maximum size for an inline LOB
<code>-m,--num-mappers <n></code>	Use <i>n</i> map tasks to import in parallel
<code>-e,--query <statement></code>	Import the results of <i>statement</i> .
<code>--split-by <column-name></code>	Column of the table used to split work units
<code>--table <table-name></code>	Table to read
<code>--target-dir <dir></code>	HDFS destination dir
<code>--warehouse-dir <dir></code>	HDFS parent for table destination
<code>--where <where clause></code>	WHERE clause to use during import
<code>-z,--compress</code>	Enable compression
<code>--compression-codec <c></code>	Use Hadoop codec (default gzip)
<code>--null-string <null-string></code>	The string to be written for a null value for string columns
<code>--null-non-string <null-string></code>	The string to be written for a null value for non-string columns

Selective Import

- You can select a subset of columns and control their ordering by using the `--columns` argument. This should include a comma-delimited list of columns to import. For example: `--columns "name, student_id"`
- You can control which rows are imported by adding a SQL `WHERE` clause to the import statement.
- By default, Sqoop generates statements of the form `SELECT <column list> FROM <table name>`. You can append a `WHERE` clause to this with the `--where` argument.
 - For example: `--where "id > 400"` : Only rows where the id column has a value greater than 400 will be imported.

Import : direct argument

- By default, the import process will use JDBC which provides a reasonable cross-vendor import channel.
- Some databases can perform imports in a more high-performance fashion by using database-specific data movement tools.
- For example, MySQL provides the **mysqldump** tool which can export data from MySQL to other systems very quickly.
- By supplying the **--direct** argument, you are specifying that Sqoop should attempt the direct import channel. This channel may be higher performance than using JDBC.
- Currently, direct mode does not support imports of large object columns.

Incremental Import Mode

- Sqoop provides an incremental import mode which can be used to retrieve only rows newer than some previously-imported set of rows.
- Performing an incremental import of new data, after having already imported the first 100,000 rows of a table:

```
sqoop import  
--CONNECT jdbc:mysql://10.0.2.15:3306/cs523  
--table student  
--where "id > 100000"  
--target-dir /user/cloudera/sqoopImport  
--append
```

Sqoop Export

- In some cases data processed by Hadoop pipelines may be needed in production systems to help run additional critical business functions.
- Sqoop can be used to export such data into external datastores as necessary.
- For example, data generated by the pipeline on Hadoop corresponded to the STUDENTS table in a DataBase somewhere, you could populate it using the Export command:

Sqoop Export Command

SQOOP EXPORT

```
--CONNECT jdbc:mysql://10.0.2.15:3306/cs523  
-USERNAME root -P  
--TABLE student  
--EXPORT-DIR= /user/cloudera/sqoopImport
```

- **EXPORT**: This is the sub-command that instructs Sqoop to initiate an export.
- **--TABLE** <table name>: This parameter specifies the table which will be populated.
- **--EXPORT-DIR** <directory path>: This is the directory from which data will be exported.

Sqoop Export

- The export tool exports a set of files from HDFS to RDBMS.
- The target table must already exist in the database.
- The input files are read and parsed into a set of records according to the user-specified delimiters.
- The default operation is to transform these into a set of **INSERT** statements that inject the records into the database.
- In "update mode," Sqoop will generate **UPDATE** statements that replace existing records in the database.
- If your table has constraints (e.g., a primary key column whose values must be unique) and already contains data, you must take care to avoid inserting records that violate these constraints.
- 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.

Export Control Arguments

Argument	Description
<code>--direct</code>	Use direct export fast path
<code>--export-dir <dir></code>	HDFS source path for the export
<code>-m, --num-mappers <n></code>	Use <i>n</i> map tasks to export in parallel
<code>--table <table-name></code>	Table to populate
<code>--update-key <col-name></code>	Anchor column to use for updates. Use a comma separated list of columns if there are more than one column.
<code>--update-mode <mode></code>	Specify how updates are performed when new rows are found with non-matching keys in database. Legal values for <code>mode</code> include <code>updateonly</code> (default) and <code>allowinsert</code> .
<code>--input-null-string <null-string></code>	The string to be interpreted as null for string columns
<code>--input-null-non-string <null-string></code>	The string to be interpreted as null for non-string columns
<code>--staging-table <staging-table-name></code>	The table in which data will be staged before being inserted into the destination table.
<code>--clear-staging-table</code>	Indicates that any data present in the staging table can be deleted.
<code>--batch</code>	Use batch mode for underlying statement execution.

Incremental Updates from HDFS to MySQL

SQOOP EXPORT

--CONNECT jdbc:mysql://10.0.2.15:3306/cs523

-USERNAME root -P

--TABLE student

--update-key id

--update-mode updateonly

--EXPORT-DIR= /user/cloudera/sqoopImport/

Sqoop Connectors

- Using specialized connectors, Sqoop can connect with external systems that have optimized import and export facilities, or do not support native JDBC.
- Connectors are plugin components based on Sqoop's extension framework and can be added to any existing Sqoop installation. Once a connector is installed, Sqoop can use it to efficiently transfer data between Hadoop and the external store supported by the connector.
- By default, Sqoop includes connectors for various popular databases such as MySQL, PostgreSQL, Oracle, SQL Server & DB2.
- Apart from the built-in connectors, many companies have developed their own connectors that can be plugged into Sqoop. These range from specialized connectors for enterprise data warehouse systems to NoSQL datastores.

Sqoop pros and cons

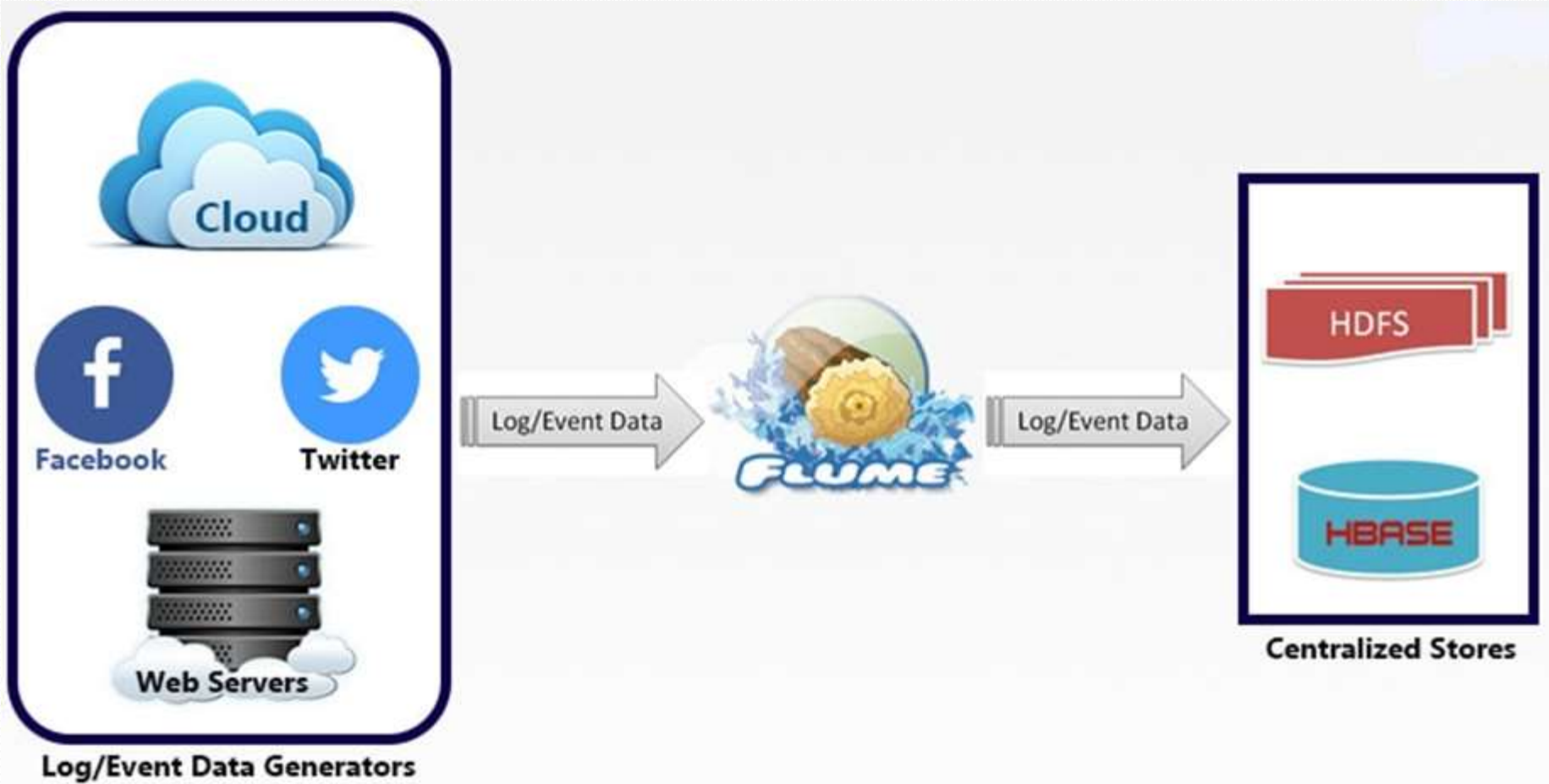
- Sqoop provides a simplified usage model compared to using the format classes (*DBInputFormat*) that are provided in MapReduce.
- But using the *DBInputFormat* classes will give you the added flexibility to transform or pre-process your data in the same MapReduce job that performs the database export.
- The advantage of Sqoop is that it doesn't require you to write any code, and it has some useful notions, such as staging, to help you achieve your idempotent goals.
 - Some connectors support staging tables that help isolate production tables from possible corruption in case of job failures due to any reason. Staging tables are first populated by the map tasks and then merged into the target table once all of the data has been delivered to it.

Apache Flume



- There are many systems which produce streams of data that we would like to aggregate, store, and analyze using Hadoop.
- E.g. A bunch of log files are being produced by multiple applications and systems across multiple servers. There's no doubt that there's valuable information to be analyzed from these logs, but your first challenge is how to move these logs into your Hadoop cluster so that you can perform some analysis.
- So to push all of your production server's system log files into HDFS, you'll use **Flume**.
- Flume is a service that basically lets you ingest streaming data (typically file data) into HDFS.
- Flume is developed by Cloudera for high-volume ingestion of event-based data into Hadoop.

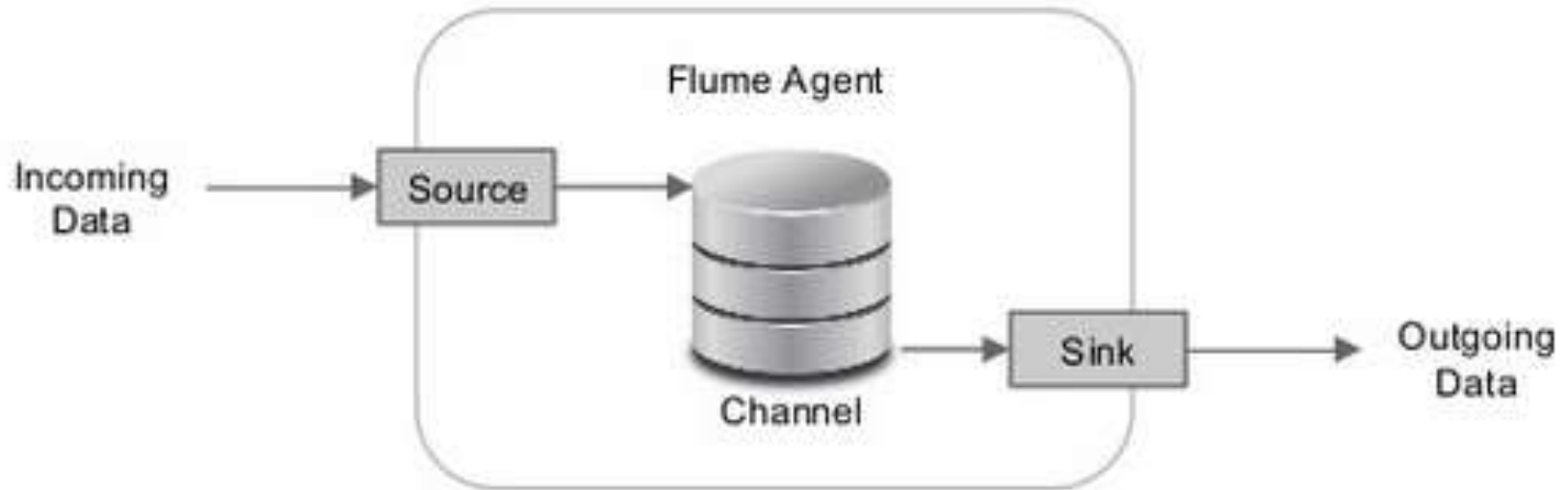
Apache Flume



Apache Flume

- Flume is a log file collection and distribution system, and collecting system logs and transporting them to HDFS is its bread and butter.
- Flume is highly fault tolerant with many failover and recovery mechanisms which makes it a one-stop solution for data collection of all formats!
- To use Flume, we need to run a **Flume agent**, which is a long-lived Java process that runs **sources** and **sinks**, connected by **channels** through which events flow from an external source to the next destination (hop).
- A **source** in Flume produces events and delivers them to the **channel**, which stores the events until they are forwarded to the **sink**.
- You can think of the **source-channel-sink** combination as a basic Flume building block.

Flume Building Blocks



Source

- Accepts incoming Data
- Scales as required
- Writes data to Channel

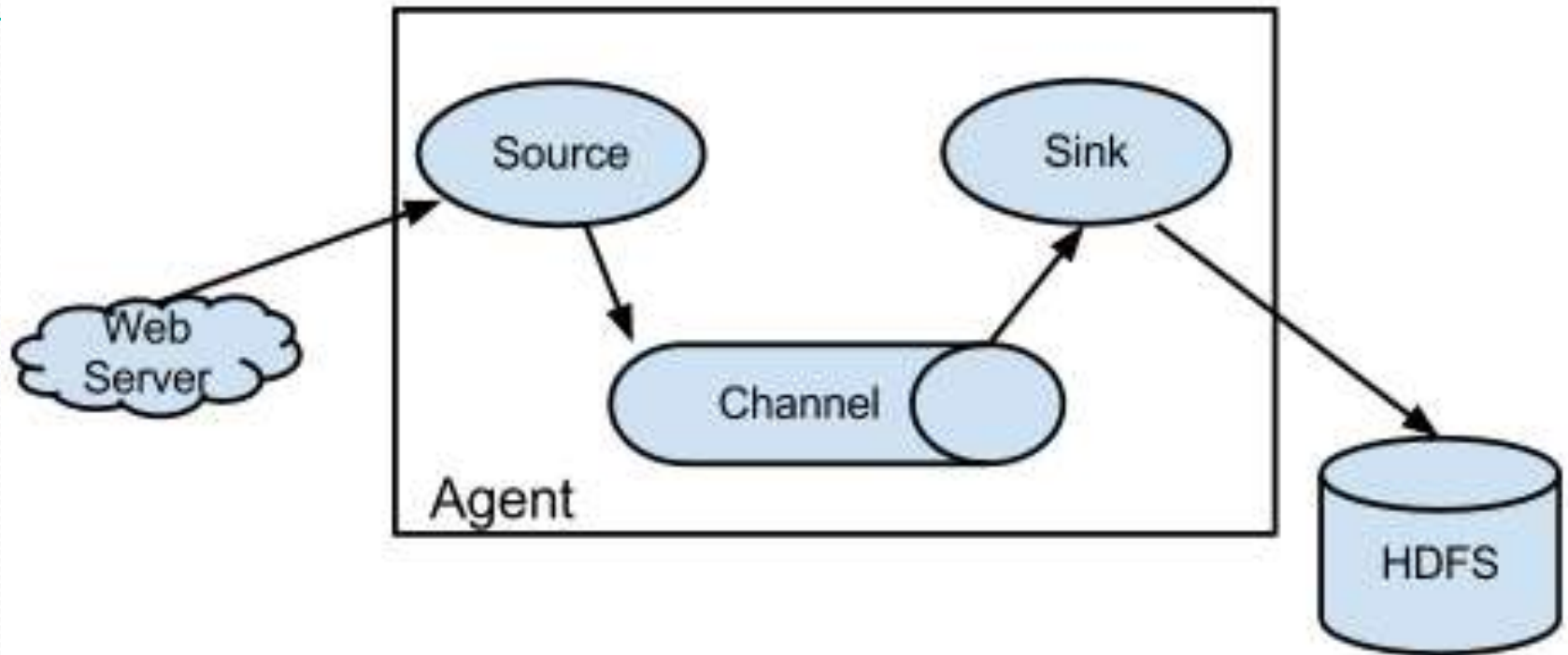
Channel

- Stores data in the order received

Sink

- Removes data from Channel
- Sends data to downstream Agent or Destination

Data Flow Model

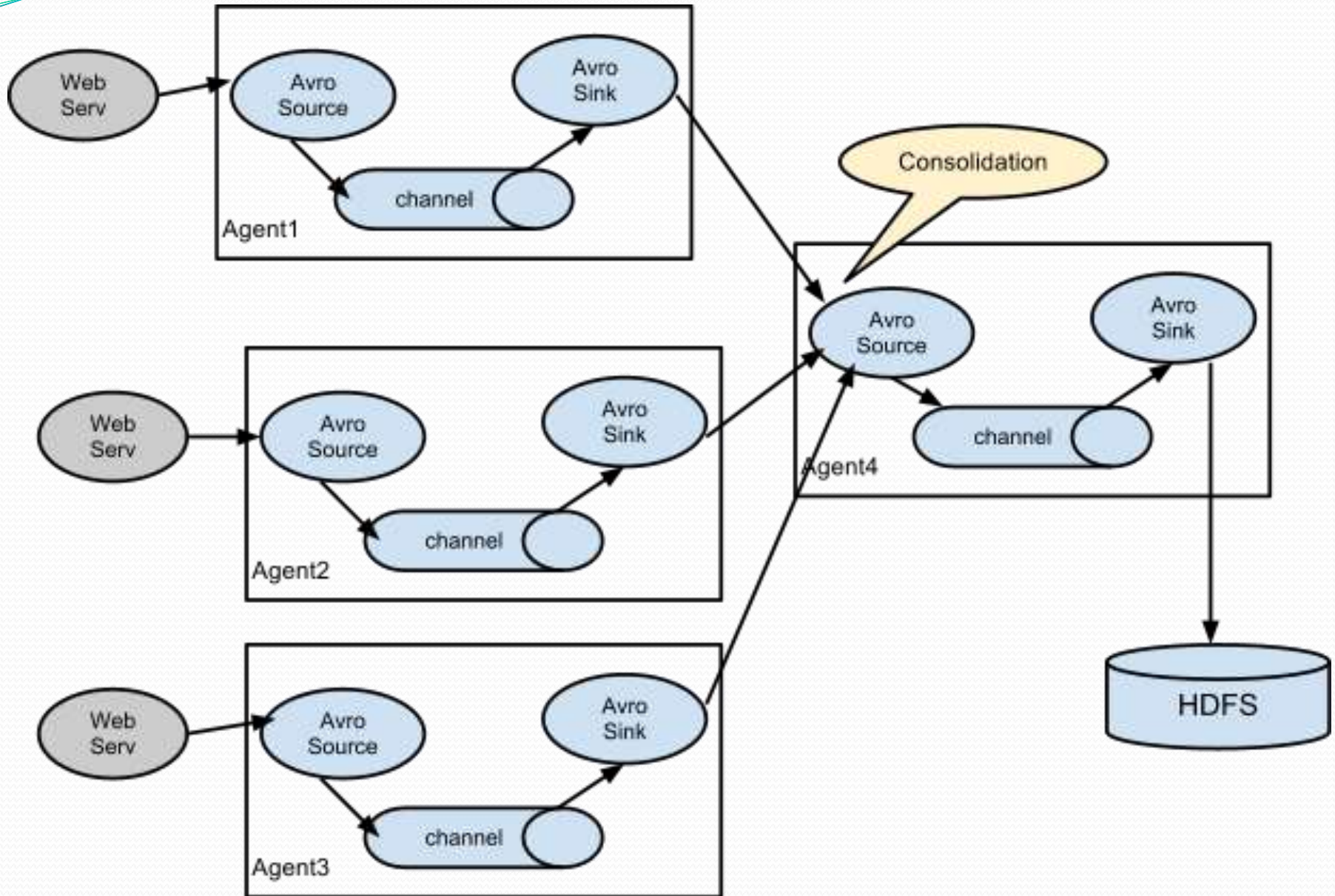


- A Flume source consumes *events* delivered to it by an external source like a web server.
 - A *Flume event* is defined as a unit of data flow having a byte payload and an optional set of string attributes.
- The external source sends *events* to Flume in a format that is recognized by the target Flume source.

Data Flow Model Contd..

- When a Flume source receives an event, it stores it into one or more channels.
- The channel is a passive store that keeps the event until it's consumed by a Flume sink.
 - The file channel is one example – it is backed by the local filesystem.
- The sink removes the event from the channel and puts it into an external repository like HDFS (via Flume HDFS sink) or forwards it to the Flume source of the next Flume agent (next hop) in the flow.
- The source and sink within the given agent run asynchronously with the events staged in the channel.

Complex Data Flows



Complex Data Flows

- A Flume installation can be made up of a collection of connected agents running in a distributed topology.
- Agents on the edge of the system (co-located on web server machines, for example) collect data and forward it to agents that are responsible for aggregating and then storing the data in its final destination.
- Agents are configured to run a collection of particular sources and sinks, so using Flume is mainly a configuration exercise in wiring the pieces together.

Transactions in Flume

- Flume is a transactional system. Transactions are essentially batches of events written into a channel atomically.
- Multiple events can be either *Put* or *Taken* in a single transaction.
- A Flume transaction consists of either *Puts* or *Takes*, but not both, and either a *commit* or a *rollback*.
- Each transaction implements both a *Put* and *Take* method.
- Sources do *Puts* onto the channel and Sinks do Takes from the channel.
- Channels are transactional in nature. Each write to a channel and each read from a channel happens within the context of a transaction.
- Only once a write transaction is committed will the events from that transaction be readable by any sinks.
- Also, if a sink has successfully *taken* an event, the event is not available for other sinks to take unless and until the sink rolls back the transaction.

Transactions in Flume

- Transactions are important to provide guarantees of exactly when an event is written to or removed from the channel.
- For example, a sink could take an event from the channel and attempt to write it to HDFS and fail. In this case, the event should go back to the channel and be available for this sink or another one to take and write to HDFS.
- Making sure that "*takes*" cause events to be removed only on transaction commit, guarantees that events are not lost even if the write fails once, at which point the sink can just roll back this transaction.
- Transactions could have one event or many events, but for performance reasons it is always recommended to have a reasonably large number of events per transaction.

Reliability in Flume

- An Event is staged in a Flume agent's Channel. Then it's the Sink's responsibility to deliver the Event to the next agent or terminal repository (like HDFS) in the flow.
- The Sink removes an Event from the Channel only after the Event is stored into the Channel of the next agent or stored in the terminal repository.
- This is how the single-hop message delivery semantics in Flume provide end-to-end reliability of the flow.
- In the case of a multi-hop flow, the Sink from the previous hop and the Source of the next hop both have their Transactions open to ensure that the Event data is safely stored in the Channel of the next hop.
- Flume uses a **transactional** approach to guarantee the reliable delivery of the Events.
- **The Sources and Sinks encapsulate the storage/retrieval of the Events in a Transaction provided by the Channel.**
- This ensures that the set of Events are reliably passed from point to point in the flow.

Flume Agent Setup

- Flume agent configuration is stored in a local configuration file. This is a text file that follows the Java properties file format.
- Configurations for one or more agents can be specified in the same configuration file.
- The configuration file includes properties of each source, sink and channel in an agent and how they are wired together to form data flows.
- Each component (source, sink or channel) in the flow has a name, type, and set of properties that are specific to the type and instantiation.
- For example, an HDFS sink needs to know the file system URI, path to create files, frequency of file rotation ("hdfs.rollInterval") etc. All such attributes of a component needs to be set in the properties file of the hosting Flume agent.

Flume Configuration File

- Let's see an example of a flume configuration file which will setup all the components for the flume agent.
- The configuration file is for a simple sequence generator that continuously generates events with a counter that starts from 0, increments by 1 and stops at *totalEvents*.
- It retries when it can't send events to the channel.
- Useful mainly for testing.

Flume Configuration File

The configuration file needs to define the sources, the channels and the sinks.
Sources, channels and sinks are defined per agent, in this case called 'myAgent'.

myAgent.sources = seqGenSrc

myAgent.channels = memoryChannel

myAgent.sinks = loggerSink

For each one of the sources, the type is defined

myAgent.sources.seqGenSrc.type = seq

The channel for the source can be defined as follows

myAgent.sources.seqGenSrc.channels = memoryChannel

Each sink's type must be defined – logger sink for logging events to the console

myAgent.sinks.loggerSink.type = logger

#Specify the channel the sink should use

myAgent.sinks.loggerSink.channel = memoryChannel

Each channel's type is defined.

myAgent.channels.memoryChannel.type = memory

Other config values specific to each type of channel(sink or source) can be defined as well. In this case, it specifies the capacity of the memory channel

myAgent.channels.memoryChannel.capacity = 100

Examples of Source, Channel & Sink

Source

Channel

Sink

Flume Sources
Avro Source
Thrift Source
Exec Source
JMS Source
Spooling Directory Source
Twitter Source
Kafka Source
NetCat Source
Sequence Generator Source
Syslog Sources
Syslog TCP Source
Multiport Syslog TCP Source
Syslog UDP Source
HTTP Source
Stress Source
Legacy Sources
Avro Legacy Source
Thrift Legacy Source
Custom Source
Scribe Source

Flume Channels
Memory Channel
JDBC Channel
Kafka Channel
File Channel
Spillable Memory Channel
Pseudo Transaction Channel
Custom Channel
Custom Channel Selector

Flume Sinks
HDFS Sink
Hive Sink
Logger Sink
Avro Sink
Thrift Sink
IRC Sink
File Roll Sink
Null Sink
Hbase Sinks
Hbase Sink
AsyncHBaseSink
MorphlineSolrSink
ElasticSearchSink
Kite Dataset Sink
Kafka Sink
Custom Sink

Sources

- Flume *sources* are responsible for reading data from external clients or from other Flume sinks.
- A unit of data in Flume is defined as an *event*, which is essentially a payload and optional set of metadata.
- A Flume source sends these events to one or more Flume channels, which deal with storage and buffering.
- Flume has an extensive set of built-in sources, including HTTP, JMS, and RPC. The complete set of flume sources is available at <http://flume.apache.org/FlumeUserGuide.html#flume-sources>
- Let's take a look at the source-specific configuration properties that need to be set for an application that writes to a log file on disk and Flume tails the file, sending each line as an event.

Source Configuration

```
agent1.sources = tail_source1  
# define tail source  
agent1.sources.tail_source1.type = exec  
agent1.sources.tail_source1.command = tail -F /cs523/log/messages  
agent1.sources.tail_source1.channels = ch1  
agent1.sources.tail_source1.shell = /bin/bash -c  
agent1.sources.tail_source1.interceptors = ts  
agent1.sources.tail_source1.interceptors.ts.type = timestamp
```

- The **exec** source allows you to execute a Unix command, and each line emitted in standard o/p is captured as an event (standard error is ignored by default).
- In the above example, the **tail -F** command is used to capture system messages as they are produced.

Interceptors in Sources

- **Interceptors** allow you to add metadata to events. They also give Flume the capability to modify/drop events in the flow.
- They are attached to sources, and are run on events before the events have been placed in a channel.
- **Host interceptor:**
 - This interceptor inserts the hostname or IP address of the host that this agent is running on.
 - It inserts a header with key host or a configured key whose value is the hostname or IP address of the host, based on configuration.

Timestamp Interceptors

- This interceptor inserts into the event headers, the time in millis at which it processes the event.
- It inserts a header with key *timestamp* whose value is the relevant timestamp.
- It can preserve an existing timestamp if it is already present in the configuration.
- The following extra configuration lines add a timestamp interceptor to source1, which adds a timestamp header to every event produced by the source:

```
agent1.sources.source1.interceptors = interceptor1  
agent1.sources.source1.interceptors.interceptor1.type =  
timestamp
```

Timestamp Interceptors

- Using the timestamp interceptor ensures that the timestamps closely reflect the times at which the events were created.
- For some applications, using a timestamp for when the event was written to HDFS might be sufficient—although, be aware that when there are multiple tiers of Flume agents there can be a significant difference between creation time and write time, especially in the event of agent downtime.
- For these cases, the HDFS sink has a setting, *hdfs.useLocalTimeStamp*, that will use a timestamp generated by the Flume agent running the HDFS sink.

Channels

- Flume Channels are buffers that sit in between sources and sinks providing data storage facilities inside an agent.
- As such, channels allow sources and sinks to operate at different rates. Channels are key to Flume's guarantees of not losing data (of course, when configured properly)
- Sources add events to a channel, and sinks remove events from a channel.
- A sink can read only from one channel, while multiple sinks can read from the same channel for better performance.
- Channels provide durability properties inside Flume, and you pick a channel based on which level of durability and throughput you need for your application.

Channels

- There are three main channels bundled with Flume:
 - **Memory channels** store events in an in-memory queue. This is very useful for high-throughput data flows, but they have no durability guarantees, meaning that if an agent goes down, you'll lose data.
 - **File channels** persist events to disk. The implementation uses an efficient write-ahead log and has strong durability properties.
 - **JDBC channels** store events in a database. This provides the strongest durability and recoverability properties, but at a cost to performance.
- Additional details on Flume channels can be seen at <http://flume.apache.org/FlumeUserGuide.html#flume-channels>

Channel Configuration

```
agent1.channels = ch1
# define in-memory channel
agent1.channels.ch1.type = memory
agent1.channels.ch1.capacity = 100000
agent1.channels.ch1.transactionCapacity = 1000
```

- The above configuration defined an in-memory channel and capped the number of events that it would store at 100,000.
- Once the max number of events is reached in a memory channel, it will start refusing additional requests from sources to add more events. Depending on the type of source, this means that the source will either retry or drop the event (*exec* source will drop the event).
- *TransactionCapacity* which is the maximum number of events the channel will take from a source or give to a sink per transaction is capped at 1000.

Sinks

- A Flume sink drains events out of one or more Flume channels and will either forward these events to another Flume source (in a multihop flow), or handle the events in a sink-specific manner.
- There are a number of sinks built into Flume, including HDFS, HBase, Solr, and ElasticSearch.

Sink Configuration

```
agent1.sinks = hdfs_sink1
# define HDFS sink properties
agent1.sinks.hdfs_sink1.type = hdfs
agent1.sinks.hdfs_sink1.hdfs.path = /flume/%y%m%d/%H%M%S
agent1.sinks.hdfs_sink1.hdfs.fileType = DataStream
agent1.sinks.hdfs_sink1.channel = ch1
```

- You configured the sink to write files based on a timestamp (note the %y and other timestamp aliases). You could do this because you decorated the events with a timestamp interceptor in the exec source.
- In fact, you can use any header value to determine the output location for events (e.g, you can add a host interceptor and then write files according to which host produced the event).



HDFS Sink

- The HDFS sink can be configured in various ways to determine how files are rolled.
 - When a sink reads the first event, it will open a new file (if one isn't already open) and write to it.
 - By default, the sink will continue to keep the file open and write events into it for 30 seconds, after which it will close it.
 - The rolling behaviour can be changed with the properties shown on the next slide.
- The default HDFS sink settings shouldn't be used in production, as they'll result in a large number of potentially small files. It's recommended that you either bump up the values or use a downstream compaction job to merge these small files.

Rollover properties for Flume's HDFS Sinks

Name	Default	Description
<code>hdfs.rollInterval</code>	30	Number of seconds to wait before rolling current file (0 = never roll based on time interval)
<code>hdfs.rollSize</code>	1024	File size to trigger roll, in bytes (0: never roll based on file size)
<code>hdfs.rollCount</code>	10	Number of events written to file before it rolled (0 = never roll based on number of events)
<code>hdfs.idleTimeout</code>	0	Timeout after which inactive files get closed (0 = disable automatic closing of idle files)
<code>hdfs.batchSize</code>	100	number of events written to file before it is flushed to HDFS

Starting an Agent

- An agent is started using a shell script called `flume-ng` which is located in the bin directory of the Flume distribution.
- You need to specify the agent name, the config directory, and the config file on the command line:

```
$flume-ng agent -n $agent_name -c conf -f conf/flume-conf.properties.template
```

- Now the agent will start running source and sinks configured in the given properties file.