

1. Sqoop Introduction	2
2. Sqoop Architecture	4
3. Sqoop Commands	6
4. Managing Target Directories	8
5. Working with Different File Formats	10
6. Working with Different Compressions	12
7. Conditional Imports	13
8. Split-by and Boundary Queries	15
9. Field delimiters	17
10. Incremental Appends	18
11. Sqoop Hive Import	19
12. Sqoop List Tables_Database	20
13. Sqoop Import Practice	21
14. Export from Hdfs to Mysql	23
15. Export from Hive to Mysql	24
16. Flume Introduction & Architecture	26
17. Flume Source	29
18. Flume Channel	30
19. Flume Sink	32
20. Flume Interceptors	33
21. Moving data from Twitter to HDFS	40

SQOOP

Sqoop was developed and maintained by Cloudera. Later, on 23 July 2011, it was incubated by Apache. In April 2012, the Sqoop project was promoted as Apache's top-level project.

Generally, applications interact with the relational database using RDBMS, and thus this makes relational databases one of the most important sources that generate Big Data. Such data is stored in RDB Servers in the relational structure. Here, Apache Sqoop plays an important role in the **Hadoop ecosystem**, providing feasible interaction between the relational database server and HDFS.

So, Apache Sqoop is a tool in **Hadoop ecosystem** which is designed to transfer data between **HDFS** (Hadoop storage) and relational database servers like MySQL, Oracle RDB, SQLite, Teradata, Netezza, Postgres etc. Apache Sqoop imports data from relational databases to HDFS, and exports data from HDFS to relational databases. It efficiently transfers bulk data between [Hadoop](#) and external data stores such as enterprise data warehouses, relational databases, etc.

This is how Sqoop got its name – “**SQL to Hadoop & Hadoop to SQL**”.

Additionally, Sqoop is used to import data from external datastores into Hadoop ecosystem's tools like **Hive & HBase**.

Key Features of Sqoop

Sqoop provides many salient features like:

Full Load: Apache Sqoop can load the whole table by a single command. You can also load all the tables from a database using a single command.

Incremental Load: Apache Sqoop also provides the facility of incremental load where you can load parts of table whenever it is updated.

Parallel import/export: Sqoop uses YARN framework to import and export the data, which provides fault tolerance on top of parallelism.

Import results of SQL query: You can also import the result returned from an SQL query in HDFS.

Compression: You can compress your data by using deflate(gzip) algorithm with `–compress` argument, or by specifying `–compression-codec` argument. You can also load compressed table in **Apache Hive**.

Connectors for all major RDBMS Databases: Apache Sqoop provides connectors for multiple RDBMS databases, covering almost the entire circumference.

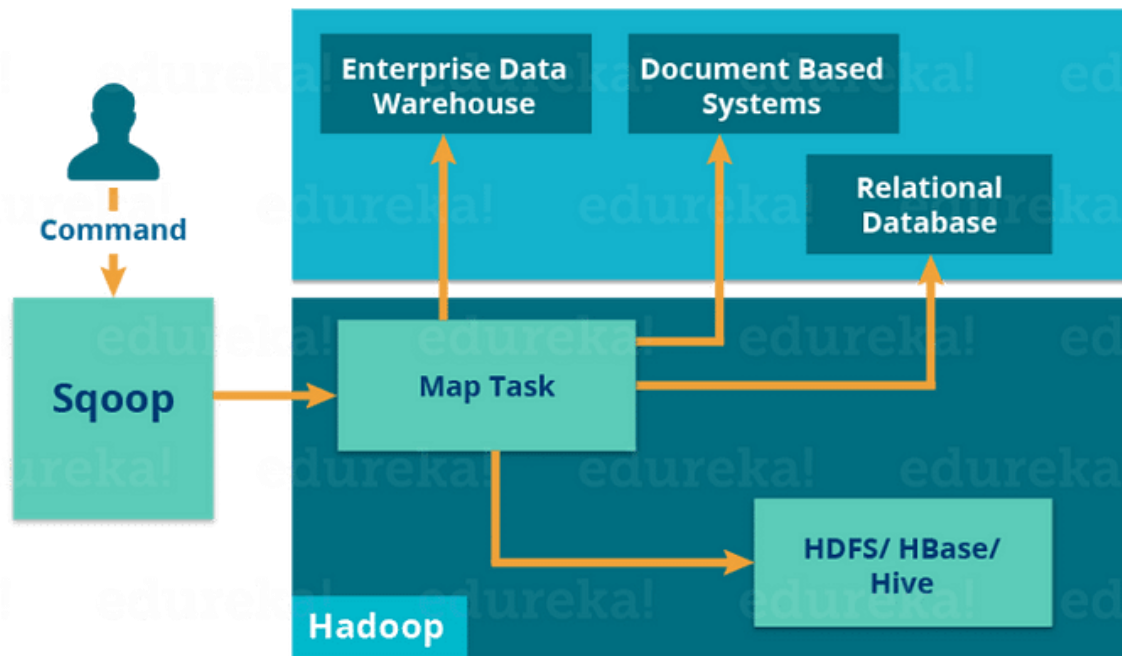
Kerberos Security Integration: Kerberos is a computer network authentication protocol which works on the basis of 'tickets' to allow nodes communicating over a non-secure

network to prove their identity to one another in a secure manner. Sqoop supports Kerberos authentication.

Load data directly into HIVE/HBase: You can load data directly into **Apache Hive** for analysis and also dump your data in HBase, which is a NoSQL database.

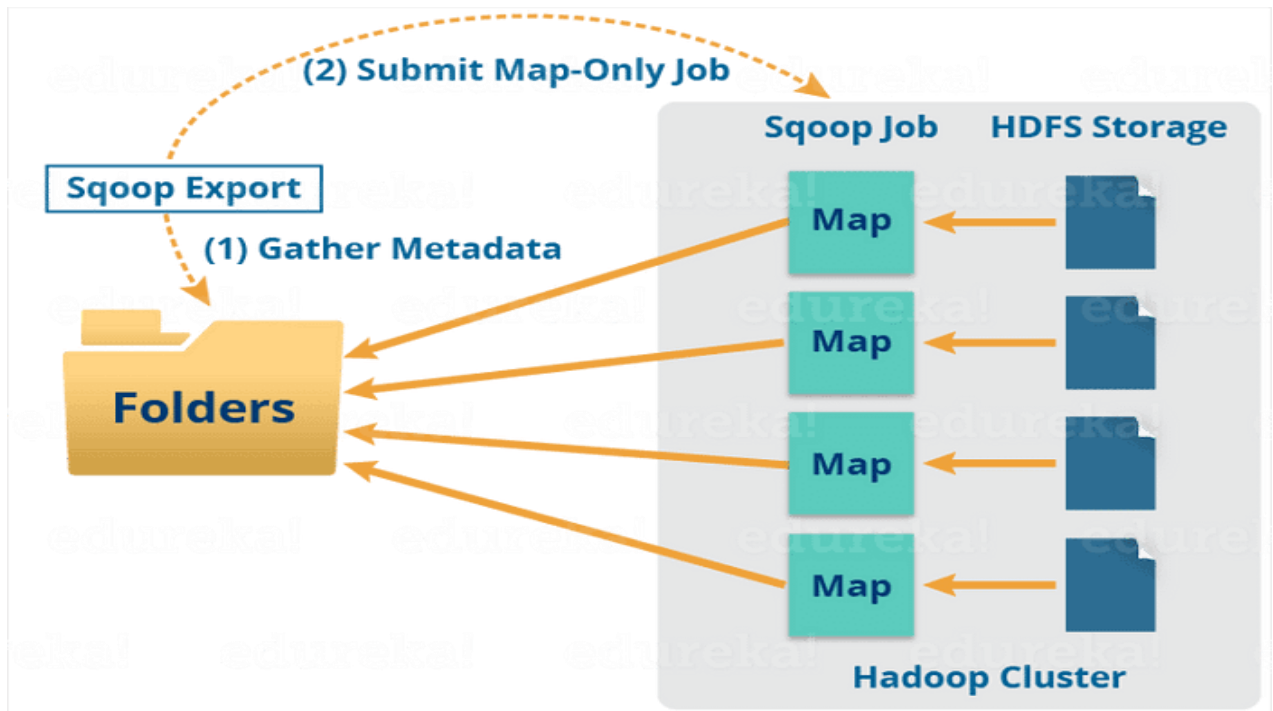
Support for Accumulo: You can also instruct Sqoop to import the table in Accumulo rather than a directory in HDFS.

Sqoop Architecture & Working



The import tool imports individual tables from RDBMS to HDFS. Each row in a table is treated as a record in HDFS.

When we submit Sqoop command, our main task gets divided into subtasks which is handled by individual Map Task internally. Map Task is the subtask, which imports part of data to the Hadoop Ecosystem. Collectively, all Map tasks imports the whole data.



Export also works in a similar manner. The export tool exports a set of files from HDFS back to an RDBMS. The files given as input to Sqoop contain records, which are called as rows in the table.

When we submit our Job, it is mapped into Map Tasks which brings the chunk of data from HDFS. These chunks are exported to a structured data destination. Combining all these exported chunks of data, we receive the whole data at the destination, which in most of the cases is an RDBMS (MYSQL/Oracle/SQL Server).

Reduce phase is required in case of aggregations. But, Apache Sqoop just imports and exports the data; it does not perform any aggregations. Map job launch multiple mappers depending on the number defined by the user. For Sqoop import, each mapper task will be assigned with a part of data to be imported. Sqoop distributes the input data among the mappers equally to get high performance. Then each mapper creates a connection with the database using JDBC and fetches the part of data assigned by Sqoop and writes it into HDFS or Hive or HBase based on the arguments provided in the CLI.

Sqoop Commands

First connect to the MYSQL database

Mysql -uroot -ppassword -hlocalhost

Use retail_db;

Show tables;

Describe customers;

sqoop import \

--connect jdbc:mysql://localhost/retail_db \

--username root \

--password cloudera \

--table customers

When we command sqoop it will create java classes those java classes use to import data from mysql to hdfs. It uses parallel processing. Four threads are running simultaneously. It provide fault tolerant. It uses map reduce. Beginning code generation from there java class are started and uses limit 1 to get meta information about data.

```
> --table customers
Warning: /usr/lib/sqoop/./accumulo does not exist! Accumulo imports will fail.
Please set $ACCUMULO_HOME to the root of your Accumulo installation.
19/01/16 08:13:44 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6-cdh5.12.0
19/01/16 08:13:44 WARN tool.BaseSqoopTool: Setting your password on the command-
line is insecure. Consider using -P instead.
19/01/16 08:13:46 INFO manager.MySQLManager: Preparing to use a MySQL streaming
resultset.
19/01/16 08:13:46 INFO tool.CodeGenTool: Beginning code generation
19/01/16 08:13:49 INFO manager.SqlManager: Executing SQL statement: SELECT t.* F
ROM `customers` AS t LIMIT 1
19/01/16 08:13:50 INFO manager.SqlManager: Executing SQL statement: SELECT t.* F
ROM `customers` AS t LIMIT 1
19/01/16 08:13:50 INFO orm.CompilationManager: HADOOP_MAPRED_HOME is /usr/lib/ha
doop-mapreduce
Note: /tmp/sqoop-cloudera/compile/078a729bf21674490824cc93471b3e7a/customers.jav
a uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
19/01/16 08:14:05 INFO orm.CompilationManager: Writing jar file: /tmp/sqoop-clou
dera/compile/078a729bf21674490824cc93471b3e7a/customers.jar
19/01/16 08:14:05 WARN manager.MySQLManager: It looks like you are importing fro
m mysql.
19/01/16 08:14:05 WARN manager.MySQLManager: This transfer can be faster! Use th
e --direct
19/01/16 08:14:05 WARN manager.MySQLManager: option to exercise a MySQL-specific
fast path.
19/01/16 08:14:05 INFO manager.MySQLManager: Setting zero DATETIME behavior to c
onvertToNull (mysql)
19/01/16 08:14:05 INFO mapreduce.ImportJobBase: Beginning import of customers
19/01/16 08:14:05 INFO Configuration.deprecation: mapred.job.tracker is deprecate
d. Instead, use mapreduce.jobtracker.address
```

Once reading all metadata writing jar file beginning execution to import and it will started importing.

```
cloudera@quickstart:~$ cat /dev/null
File Edit View Search Terminal Help
at java.lang.Object.wait(Native Method)
at java.lang.Thread.join(Thread.java:1252)
at java.lang.Thread.join(Thread.java:1326)
at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.closeResponder(DFS
OutputStream.java:952)
at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.endBlock(DFSOutpu
Stream.java:690)
at org.apache.hadoop.hdfs.DFSOutputStream$DataStreamer.run(DFSOutputStre
m.java:879)
9/01/16 08:14:25 INFO mapreduce.JobSubmitter: number of splits:4
9/01/16 08:14:26 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_15
7642205074_0002
9/01/16 08:14:29 INFO impl.YarnClientImpl: Submitted application application_15
7642205074_0002
9/01/16 08:14:30 INFO mapreduce.Job: The url to track the job: http://quickstar
.cloudera:8088/proxy/application_1547642205074_0002/
9/01/16 08:14:30 INFO mapreduce.Job: Running job: job_1547642205074_0002
9/01/16 08:15:08 INFO mapreduce.Job: Job job_1547642205074_0002 running in uber
mode : false
9/01/16 08:15:08 INFO mapreduce.Job: map 0% reduce 0%
9/01/16 08:16:15 INFO mapreduce.Job: map 75% reduce 0%
9/01/16 08:16:16 INFO mapreduce.Job: map 100% reduce 0%
9/01/16 08:16:17 INFO mapreduce.Job: Job job_1547642205074_0002 completed succe
ssfully
9/01/16 08:16:18 INFO mapreduce.Job: Counters: 30
File System Counters
FILE: Number of bytes read=0
FILE: Number of bytes written=606172
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=487
HDFS: Number of bytes written=953525
HDFS: Number of read operations=16
```

Number of splits : 4

Four parallel processing is running. Table data divided into 4 parts that will be processed by 4 parallel threads. Each part given to one of the threads

Map Tasks for job_1547642205074_0002 - Mozilla Firefox

quickstart.cloudera:19888/jobhistory/tasks/job_1547642205074_0002/m

Queue: root.cloudera
State: SUCCEEDED
Uberized: false
Submitted: Wed Jan 16 08:14:29 PST 2019
Started: Wed Jan 16 08:15:04 PST 2019
Finished: Wed Jan 16 08:16:15 PST 2019
Elapsed: 1mins, 10sec
Diagnostics:
Average Map Time: 1mins, 5sec

ApplicationMaster			
Attempt Number	Start Time	Node	Logs
1	Wed Jan 16 08:14:42 PST 2019	quickstart.cloudera:8042	logs

Task Type	Total	Complete	
Map	4	4	
Reduce	0	0	
Attempt Type			
	Failed	Killed	Successful
Maps	0	0	4
Reduces	0	0	0

Managing Target Directories

Specifying Mappers

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers  
-m 2
```

Two mappers specified explicitly

Defining warehouse directory

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--warehouse-dir /user/cloudera/new-warehouse
```

To create parent directory

Defining target directory

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-new
```

Whatever path will be given it will create on given path

If you want to check then you can run `hdfs dfs -ls /user/cloudera/customer-new`

Delete target directory if already exists

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-new \  
--delete-target-dir
```

Working with Different File Formats

Managing destination directory

Importing as avro files

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-avro \  
--as-avrodatafile
```

Importing as parquet files

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-parquet \  
--as-parquetfile
```

```
Found 6 items  
drwxr-xr-x - cloudera supergroup 0 2019-01-16 22:14 /user/cloudera/customer_parquet/.metadata  
drwxr-xr-x - cloudera supergroup 0 2019-01-16 22:15 /user/cloudera/customer_parquet/.signals  
-rw-r--r-- 1 cloudera supergroup 88944 2019-01-16 22:15 /user/cloudera/customer_parquet/23775b52-764b-4f11-811f-2fefaa57ac3c.parquet  
-rw-r--r-- 1 cloudera supergroup 89163 2019-01-16 22:15 /user/cloudera/customer_parquet/6137d256-fcc9-444f-b9fc-cecdfad40d25.parquet  
-rw-r--r-- 1 cloudera supergroup 88762 2019-01-16 22:15 /user/cloudera/customer_parquet/6aac2061-3597-4e2d-974d-2d65b32965ed.parquet  
-rw-r--r-- 1 cloudera supergroup 89047 2019-01-16 22:15 /user/cloudera/customer_parquet/7ea3c00f-1f53-420c-97fa-990cba83561a.parquet
```

Importing as Sequence files

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-sequence \  
--as-sequencefile
```

Sequence files are generally the flat files. But data is stored in binary format.

Working with Different Compressions

Working with File Formats

Gzip Compressed

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer_gzip \  
--compress
```

By default the format is gzip format

Snappy Compressed

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer_snappy \  
--compress \  
--compression-codec snappy
```

Deflate Compressed

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer_deflate \  
--compress \  
--compression-codec deflate
```

Working with Compression Types

Bzip Compressed

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer_bzip \  
--compress \  
--compression-codec bzip2
```

Lz4 Compressed

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer_lz4 \  
--compress \  
--compression-codec lz4
```

```
-rw-r--r-- 1 cloudera supergroup 110165 2019-01-17 09:46 /user/clouder
/customer-snappy/part-m-00000.snappy
-rw-r--r-- 1 cloudera supergroup 109884 2019-01-17 09:46 /user/clouder
/customer-snappy/part-m-00001.snappy
-rw-r--r-- 1 cloudera supergroup 110479 2019-01-17 09:46 /user/clouder
/customer-snappy/part-m-00002.snappy
-rw-r--r-- 1 cloudera supergroup 110616 2019-01-17 09:46 /user/clouder
/customer-snappy/part-m-00003.snappy
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/customer-deflate
Found 5 items
-rw-r--r-- 1 cloudera supergroup 0 2019-01-17 09:49 /user/clouder
/customer-deflate/_SUCCESS
-rw-r--r-- 1 cloudera supergroup 63925 2019-01-17 09:49 /user/clouder
/customer-deflate/part-m-00000.deflate
-rw-r--r-- 1 cloudera supergroup 63696 2019-01-17 09:49 /user/clouder
/customer-deflate/part-m-00001.deflate
-rw-r--r-- 1 cloudera supergroup 64159 2019-01-17 09:49 /user/clouder
/customer-deflate/part-m-00002.deflate
-rw-r--r-- 1 cloudera supergroup 63893 2019-01-17 09:49 /user/clouder
/customer-deflate/part-m-00003.deflate
[cloudera@quickstart ~]$ hdfs dfs -ls /user/cloudera/customer-lz4
Found 5 items
-rw-r--r-- 1 cloudera supergroup 0 2019-01-17 09:56 /user/clouder
/customer-lz4/_SUCCESS
-rw-r--r-- 1 cloudera supergroup 97878 2019-01-17 09:55 /user/clouder
/customer-lz4/part-m-00000.lz4
-rw-r--r-- 1 cloudera supergroup 97610 2019-01-17 09:55 /user/clouder
/customer-lz4/part-m-00001.lz4
-rw-r--r-- 1 cloudera supergroup 98155 2019-01-17 09:56 /user/clouder
/customer-lz4/part-m-00002.lz4
-rw-r--r-- 1 cloudera supergroup 98349 2019-01-17 09:56 /user/clouder
/customer-lz4/part-m-00003.lz4
```

Conditional/Selective Imports

Conditional Imports

```
sqoop import \
--connect jdbc:mysql://localhost/retail_db \
--username root --password cloudera \
--table customers \
--target-dir /user/cloudera/customer-name-m \
--where "customer_fname='Mary'"
```

Go to target directory and use command `hdfs dfs -tail path`

Selective Column Imports

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-selected \  
--columns  
"customer_fname,customer_lname,customer_city"
```

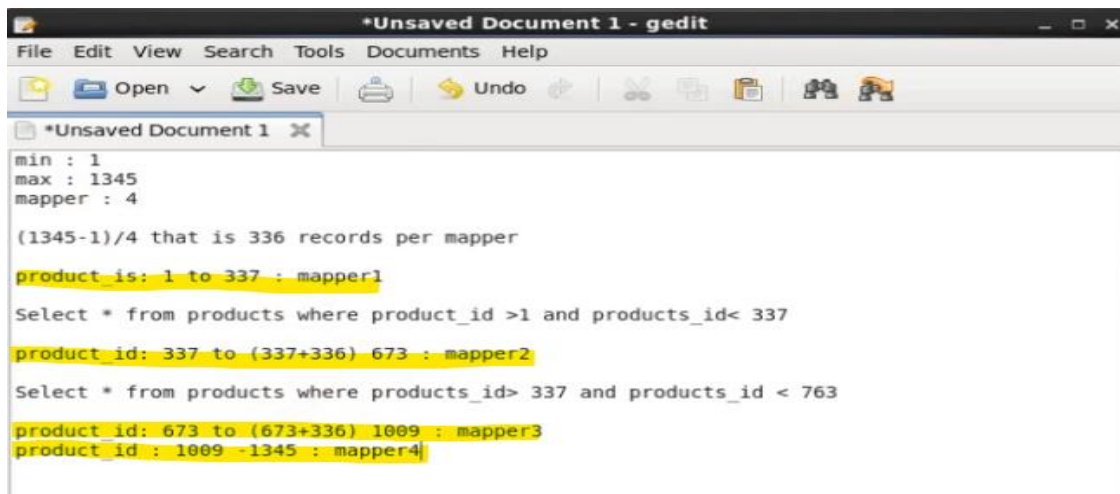
Using query

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--target-dir /user/cloudera/customer-queries \  
--query "Select * from customers where customer_id > 100  
AND $CONDITIONS" \  
--split-by "customer_id"
```

Split-by and Boundary Queries

Split-By – without primary key it does not split the data so it is must to provide primary key

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table product \  
--target-dir /user/cloudera/products_split \  
--split-by "product_id"
```



The screenshot shows a gedit window titled '*Unsaved Document 1 - gedit'. The text inside the editor is as follows:

```
min : 1
max : 1345
mapper : 4

(1345-1)/4 that is 336 records per mapper
product_id: 1 to 337 : mapper1
Select * from products where product_id >1 and products_id< 337
product_id: 337 to (337+336) 673 : mapper2
Select * from products where products_id> 337 and products_id < 763
product_id: 673 to (673+336) 1009 : mapper3
product_id : 1009 -1345 : mapper4
```

Boundary-query

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table products \  
--target-dir /user/cloudera/customer-boundary \  
--boundary-query 'Select min(product_id),max(product_id)  
from products where product_id>100' \  
--split-by product_id
```


Field delimiters

Handling Null

Handling Null

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customer_new \  
--target-dir /user/cloudera/customer-new \  
--null-string "xxx" \  
--null-non-string "yyy"
```

Field Delimiters

Field Delimiters

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--table customers \  
--target-dir /user/cloudera/customer-delimited \  
--columns  
'customer_fname,customer_lname,customer_city' \  
--fields-terminated-by '|'
```

Incremental Appends

Simple Import

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--target-dir /user/cloudera/orders-incremental \  
--table orders
```

Go to target directory and insert

Inserts Data

```
insert into orders (order_id,order_date,order_status)  
values(100004,'2017-11-07 10:02:00','CLOSED');  
insert into orders (order_id,order_date,order_status)  
values(100005,'2017-11-07 10:02:00','CLOSED');  
insert into orders (order_id,order_date,order_status)  
values(100006,'2017-11-07 10:02:00','CLOSED');
```

Incremental Append

```
sqoop import \  
--connect jdbc:mysql://localhost/retail_db \  
--username root --password cloudera \  
--target-dir /user/cloudera/orders-incremental \  
--table orders \  
--incremental append \  
--check-column order_id \  
--last-value 100003
```

Hive Import

Hive Import

```
sqoop import \  
--connect "jdbc:mysql://localhost/retail_db" \  
--username root \  
--password cloudera \  
--table customers \  
--hive-import \  
--create-hive-table \  
--hive-database default \  
--hive-table customer_mysql
```

Hive Import change field delimiter

```
sqoop import \  
--connect "jdbc:mysql://localhost/retail_db" \  
--username root \  
--password cloudera \  
--table customers \  
--fields-terminated-by '|' \  
--hive-import \  
--create-hive-table \  
--hive-database default \  
--hive-table customer_mysql_new
```

To check the Schema

```
sqoop eval \  
> --connect jdbc:mysql://localhost/retail db \  
> --username root \  
> --password cloudera  
> --query "describe customers"
```

PROBLEM 1

Instructions: Connect to MySQL database using sqoop, import all orders that have order_status as COMPLETE

Data Description:

A mysql instance is running on the localhost. In that instance, you will find orders table that contains order's data.

> Installation: localhost

> Database name: retail_db

> Table name: Orders

> Username: root

> Password: cloudera

Output Requirement:

Place the customer's files in HDFS directory "/user/cloudera/problem1/orders/parquetdata"

Use parquet format with tab delimiter and snappy compression.

Null values are represented as -1 for numbers and "NA" for string

Solution1

```
sqoop import \  
--connect "jdbc:mysql://localhost/retail_db" \  
--username root \  
--password cloudera \  
--table orders \  
--compress --compression-codec snappy \  
--target-dir /user/cloudera/problem1/orders/parquetdata \  
--null-non-string -1 --null-string "NA" \  
--fields-terminated-by "\t" \  
--where "order_status='COMPLETE'" \  
--as-parquetfile
```

Problem 2

Instructions:

Connect to mySQL database using sqoop, import all customers that lives in 'CA' state.

Data Description:

A mysql instance is running on the localhost node. In that instance, you will find customers table that contains customer's data.

> Installation: localhost

> Database name: retail_db

> Table name: Customers

> Username: root

> Password: cloudera

Output Requirement:

Place the customers files in HDFS directory

`"/user/cloudera/problem1/customers_selected/avrodata"`

Use avro format and snappy compression.

Load only customer_id,customer_fname,customer_lname,customer_state

Solution2

```
sqoop import \  
--connect "jdbc:mysql://localhost/retail_db" \  
--username root \  
--password cloudera \  
--table customers \  
--compress \  
--compression-codec snappy \  
--target-dir /user/cloudera/problem1/customers_selected/avrodata \  
--where "customer_state='CA'" \  
--columns "customer_id,customer_fname,customer_lname,customer_state" \  
--as-avrodatafile;
```

Problem 3

Instructions: Connect to mySQL database using sqoop, import all customers whose street name contains "Plaza" .

Data Description:

A mysql instance is running on the localhost node. In that instance you will find customers table that contains customers data.

> Installation : localhost

> Database name: retail_db

> Table name: Customers

> Username: root

> Password: cloudera

Output Requirement:

Place the customers files in HDFS directory "/user/cloudera/problem1/customers/textdata"

Save output in text format with fields separated by a '*' and lines should be terminated by pipe

Load only "Customer id, Customer fname, Customer lname and Customer street name"

Sample Output

```
11942*Mary*Bernard*Tawny Fox Plaza|10480*Robert*Smith*Lost Horse
Plaza|.....
```

Solution3

```
sqoop import \
--connect "jdbc:mysql://gateway/retail_db" \
--username root \
--password cloudera \
--table customers \
--target-dir /user/cloudera/problem1/customers/textdata \
--fields-terminated-by '*' \
--lines-terminated-by '|' \
--where "customer_street like '%Plaza%'" \
--columns "customer_id,customer_fname,customer_lname,customer_street"
```

SQOOP EXPORT

Sqoop Export is used to migrate data from HDFS back to relational datastore

Using sqoop export, move data from HDFS to mysql.

HDFS Location: /user/cloudera/export-dir-cust

MySql table: customer_exported

Null String: EMPTY,

Null Non-String: 0.

Mapper: 3

```
sqoop export \  
--connect "jdbc:mysql://localhost/retail_db" \  
--username root \  
--password cloudera \  
--table customer exported \  
--export-dir /user/cloudera/export-dir-cust \  
--input-fields-terminated-by '*' \  
--input-null-string 'EMPTY' \  
--input-null non-string 0 \  
-m 3
```

Export Data from hive to mysql table using sqoop export.

Hive database : default

Hive table: product_hive

MySql database: retail_db

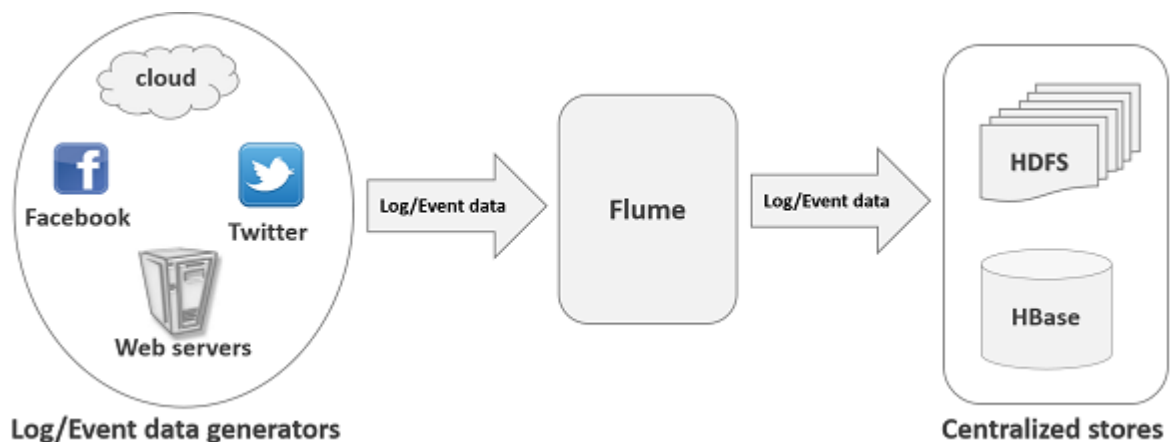
Mysql table: product_exported


```
sqoop export \  
--connect "jdbc:mysql://localhost/retail_db" \  
--username root \  
--password cloudera \  
--table product_exported \  
--hcatalog-table product_hive
```

Apache Flume

Apache Flume is a **data ingestion mechanism for collecting aggregating and transporting large amounts of streaming data such as log files, events (etc...) from various sources to a centralized data store.**

Flume is a highly reliable, distributed, and configurable tool. It is principally **designed to copy streaming data (log data) from various web servers to HDFS.**



Applications of Flume

Apache Flume is a distributed, reliable, and scalable service for efficiently collecting, aggregating, and moving large amounts of streaming data from various sources to a centralized data store or processing system. Here are some advantages of using Apache Flume:

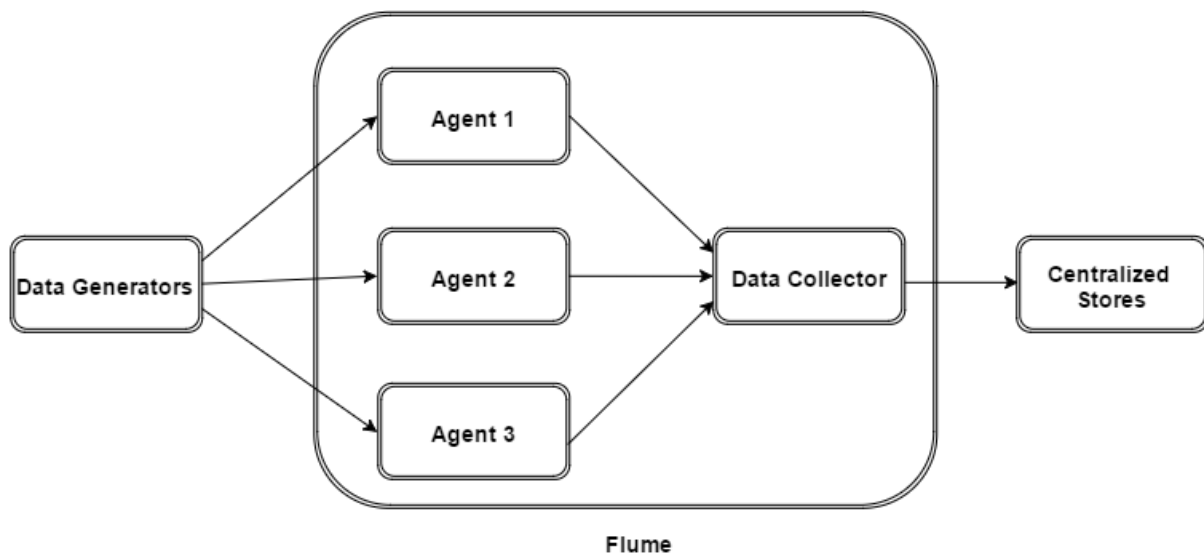
1. **Scalability:** Flume is designed to handle large-scale data ingestion. It supports horizontal scaling, allowing you to add more agents or machines to handle increased data volumes without sacrificing performance.
2. **Reliability:** Flume ensures reliable data delivery by providing fault-tolerant mechanisms. It can handle network failures, node failures, and recover from failures gracefully, ensuring that data is not lost during transit.
3. **Flexibility:** Flume provides a flexible architecture that supports various data sources and sinks. It has built-in support for popular data sources like log files, Twitter, and HTTP, and can integrate with different storage systems such as HDFS, HBase, and Kafka.

4. **Extensibility:** Flume allows you to extend its functionality by implementing custom sources, sinks, and channels. This enables you to integrate Flume with different data systems or develop connectors for specific data sources or sinks.
5. **Data Aggregation:** Flume provides mechanisms for aggregating and filtering data as it flows through the pipeline. It supports complex event processing, allowing you to perform transformations, enrichments, and filtering on the data in-flight.
6. **Fault-tolerant and Transactional:** Flume ensures reliable data delivery through various mechanisms like durable file channels, in-memory channels, and transactional sinks. It guarantees that data is not lost or duplicated during transport.
7. **Monitoring and Management:** Flume provides monitoring capabilities to track the performance and health of the data flows. It offers a web-based monitoring interface and integration with other monitoring tools like Apache Ambari.
8. **Integration with Hadoop Ecosystem:** Flume integrates seamlessly with other components of the Hadoop ecosystem, such as HDFS, HBase, and Apache Kafka. This allows you to build end-to-end data pipelines for data processing, storage, and analysis.
9. **Community and Support:** Apache Flume is an open-source project with an active community. It has a large user base and benefits from ongoing development and improvements. You can find documentation, tutorials, and community support to help you get started and troubleshoot any issues.

Overall, Apache Flume provides a reliable and scalable solution for collecting and transporting streaming data in various data-intensive applications, making it a valuable tool for big data processing and analytics.

Flume Architecture

Data generators (such as Facebook, Twitter) generate data which gets collected by individual Flume agents running on them. Thereafter, a data collector (which is also an agent) collects the data from the agents which is aggregated and pushed into a centralized store such as HDFS or HBase.



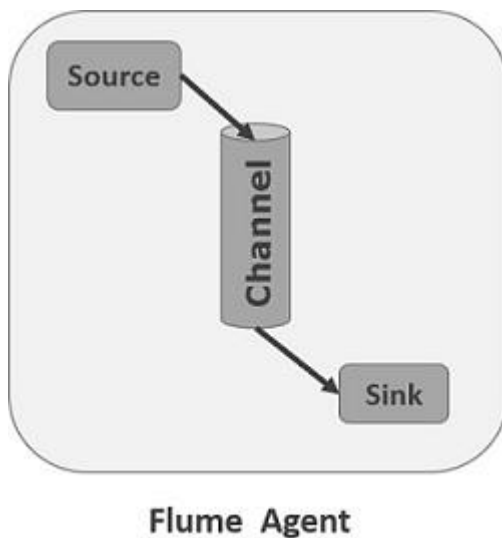
Flume Event

An event is the basic unit of the data transported inside Flume. It contains a payload of byte array that is to be transported from the source to the destination accompanied by optional headers. A typical Flume event would have the following structure –



Flume Agent

An agent is an independent daemon process (JVM) in Flume. It receives the data (events) from clients or other agents and forwards it to its next destination (sink or agent). Flume may have more than one agent. Following diagram represents a Flume Agent



As shown in the diagram a Flume Agent contains three main components namely, source, channel, and sink.

Source

A source is the component of an Agent which receives data from the data generators and transfers it to one or more channels in the form of Flume events.

Apache Flume supports several types of sources and each source receives events from a specified data generator.

Example – Avro source, Thrift source, twitter 1% source etc.

Here are some commonly used Flume sources:

1. Avro Source:

- The Avro Source receives Avro-formatted data over the network using the Avro protocol. It acts as a server and can be used with Avro clients to send data to Flume.

2. Thrift Source:

- The Thrift Source enables data ingestion using the Thrift protocol. It listens for Thrift events and deserializes them for further processing.

3. Netcat Source:

- The Netcat Source listens on a specific TCP or UDP port and reads data as a stream. It is useful for ingesting simple text-based data from network sockets.

4. Exec Source:

- The Exec Source executes an external command or program and reads its output as a stream of events. It allows you to ingest data generated by external processes or applications.

5. Spooling Directory Source:

- The Spooling Directory Source continuously monitors a directory for new files. When a new file is added, it reads its content as events. This source is commonly used for log file ingestion.

6. Syslog Source:

- The Syslog Source listens for syslog events on a specific TCP or UDP port. It can handle syslog messages from network devices and systems.

7. HTTP Source:

- The HTTP Source provides an HTTP endpoint that accepts data via HTTP POST requests. It is suitable for ingesting data from webhooks or RESTful APIs.

8. JMS Source:

- The JMS Source consumes messages from Java Message Service (JMS) queues or topics. It can be used to ingest data from enterprise messaging systems.

9. Twitter Source:

- The Twitter Source allows you to collect real-time data from Twitter. It uses the Twitter API to fetch tweets based on specified keywords, user IDs, or other criteria.

These are just a few examples of the sources available in Apache Flume. Flume's extensible architecture also allows you to develop custom sources to ingest data from other types of input streams, based on your specific requirements.

Channel

In Apache Flume, channels play a crucial role in the data flow between sources and sinks. They act as buffers or storage units that temporarily hold the events while they are being processed or transferred.

A channel is a transient store which receives the events from the source and buffers them till they are consumed by sinks. It acts as a bridge between the sources and the sinks.

These channels are fully transactional and they can work with any number of sources and sinks.

Example – JDBC channel, File system channel, Memory channel, etc.

Here are some commonly used Flume channels:

1. Memory Channel:

- The Memory Channel is an in-memory channel that stores events in the JVM heap memory. It provides fast read and write operations, making it suitable for low-latency use cases. However, it has limited capacity and is not fault-tolerant.

2. File Channel:

- The File Channel stores events in disk files. It offers higher capacity compared to the Memory Channel and is more suitable for scenarios with high-volume data streams. It provides durability and can handle larger event backlogs. However, disk I/O operations introduce higher latency.

3. JDBC Channel:

- The JDBC Channel stores events in a relational database using JDBC. It offers persistence and fault-tolerance, allowing you to recover events in case of agent or system failures. It is useful when you need durable storage and recovery capabilities.

4. Kafka Channel:

- The Kafka Channel uses Apache Kafka as the underlying storage for events. It provides scalability, fault-tolerance, and high-throughput data transfer between Flume sources and sinks. It enables integration with the Kafka ecosystem for data processing and analytics.

5. Custom Channels:

- Flume also allows you to develop custom channels based on your specific requirements. You can implement channels that integrate with different storage systems or messaging queues, providing flexibility and compatibility with your existing infrastructure.

Channels act as a bridge between sources and sinks in Flume. When an event is ingested by a source, it is stored in the channel and made available for processing by one or more sinks. Channels ensure the reliable and efficient transfer of data, handling any potential disparities in the processing speed between sources and sinks.

The choice of channel depends on factors such as the volume of data, required durability, fault-tolerance, and integration with external systems. You can configure Flume agents to use specific channels based on your use case and performance requirements.

Sink

In Apache Flume, sinks are components responsible for delivering or storing data received from sources. They define the destination or system where the data will be sent. Flume provides several built-in sinks to handle various use cases.

A sink stores the data into centralized stores like HBase and HDFS. It consumes the data (events) from the channels and delivers it to the destination. The destination of the sink might be another agent or the central stores.

Example – HDFS sink

Here are some commonly used Flume sinks:

1. HDFS Sink:
 - The HDFS Sink writes data to the Hadoop Distributed File System (HDFS). It is commonly used to store data in Hadoop for further processing and analysis.
2. HBase Sink:
 - The HBase Sink writes data to Apache HBase, a distributed and scalable NoSQL database built on top of Hadoop. It is useful for real-time querying and random access to data.
3. Kafka Sink:
 - The Kafka Sink sends data to Apache Kafka, a distributed streaming platform. It allows Flume to integrate with the Kafka ecosystem for real-time processing, analytics, and data streaming.
4. ElasticSearch Sink:
 - The ElasticSearch Sink indexes and stores data in ElasticSearch, a distributed search and analytics engine. It enables efficient searching, indexing, and analysis of the ingested data.

5. Logger Sink:

- The Logger Sink simply logs the received events to the log files of the Flume agent. It is primarily used for debugging, troubleshooting, and monitoring purposes.

6. Hadoop Metrics2 Sink:

- The Hadoop Metrics2 Sink is used to send Flume metrics to the Hadoop Metrics2 system. It allows you to collect and monitor various metrics related to Flume's performance and behavior.

7. Custom Sinks:

- Flume provides the flexibility to develop custom sinks to integrate with specific systems or applications. Custom sinks allow you to extend Flume's functionality and deliver data to any destination of your choice.

The selection of a sink depends on the specific use case, destination system, and requirements of your data pipeline. You can configure Flume agents to use the appropriate sink based on the desired destination or processing needs.

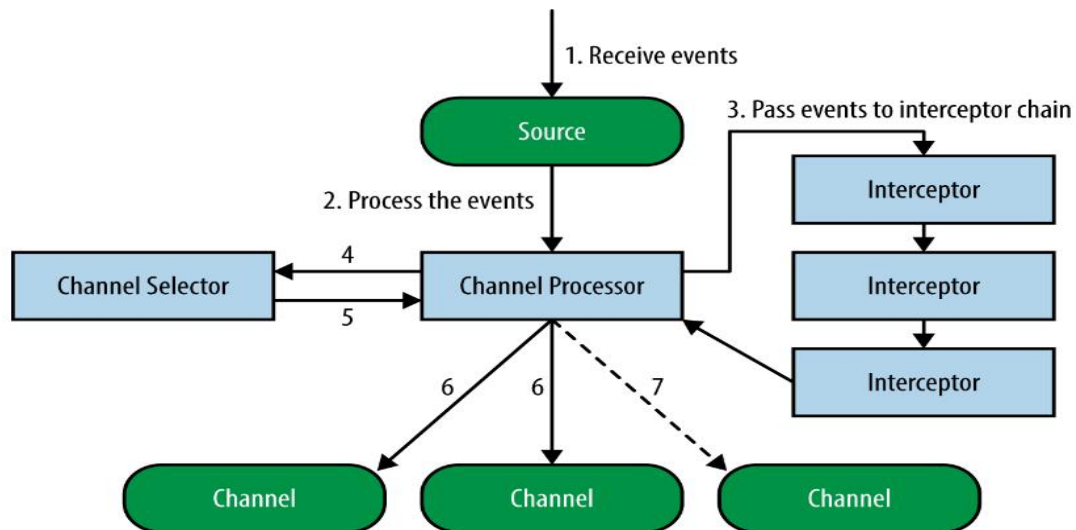
Flume's extensible architecture allows you to build custom sinks, integrating with external systems or implementing specialized data delivery mechanisms based on your unique requirements.

Additional Components of Flume Agent

What we have discussed above are the primitive components of the agent. In addition to this, we have a few more components that play a vital role in transferring the events from the data generator to the centralized stores.

Interceptors

Interceptors are used to alter/inspect flume events which are transferred between source and channel.



In Apache Flume, interceptors are components that allow you to perform transformations, filtering, or enrichment on events as they pass through the Flume pipeline. Interceptors sit between the sources and channels, allowing you to modify the events before they are stored or further processed. Here's an example of using a Flume interceptor:

Let's consider a scenario where you want to add a timestamp to each event before it is stored in HDFS using the Flume HDFS Sink. You can achieve this by creating a custom interceptor.

Implement the Interceptor:

Create a Java class that implements the Interceptor interface provided by Flume. Let's name it `TimestampInterceptor`.

Implement the necessary methods: `initialize()`, `intercept()`, and `close()`.

In the `intercept()` method, modify the event by adding a timestamp attribute or modifying an existing attribute with the current timestamp value.

```
import org.apache.flume.Context;
import org.apache.flume.Event;
import org.apache.flume.interceptor.Interceptor;
import java.nio.charset.StandardCharsets;
import java.util.List;

public class TimestampInterceptor implements Interceptor {

    @Override
    public void initialize() {
        // Initialization logic, if any
    }

    @Override
    public Event intercept(Event event) {
        // Modify the event to add timestamp attribute or modify existing attribute
        String timestamp = String.valueOf(System.currentTimeMillis());
        event.getHeaders().put("timestamp", timestamp);
        return event;
    }

    @Override
    public List<Event> intercept(List<Event> events) {
        for (Event event : events) {
            intercept(event);
        }
        return events;
    }
}
```

```
@Override
public void close() {
    // Cleanup logic, if any
}

public static class Builder implements Interceptor.Builder {

    @Override
    public Interceptor build() {
        return new TimestampInterceptor();
    }

    @Override
    public void configure(Context context) {
        // Configuration logic, if any
    }
}
}
```

Build the Interceptor:

Package the TimestampInterceptor class into a JAR file, along with any required dependencies.

Configure Flume:

In your Flume configuration file (e.g., flume.conf), add or modify the configuration for your source and sink.

Add the interceptor configuration to the source or sink properties.

```
agent.sources = mySource  
agent.sources.mySource.type = <your-source-type>  
agent.sources.mySource.interceptors = timestampInterceptor  
agent.sources.mySource.interceptors.timestampInterceptor.type =  
com.example.TimestampInterceptor$Builder
```

```
agent.sinks = mySink  
agent.sinks.mySink.type = hdfs  
agent.sinks.mySink.hdfs.path = /path/to/hdfs  
agent.sinks.mySink.hdfs.filePrefix = events  
agent.sinks.mySink.hdfs.rollInterval = 3600  
agent.sinks.mySink.hdfs.rollSize = 0  
agent.sinks.mySink.hdfs.rollCount = 100  
agent.sinks.mySink.hdfs.fileType = DataStream  
agent.sinks.mySink.hdfs.writeFormat = Text  
agent.sinks.mySink.hdfs.batchSize = 100
```

```
agent.sinks.mySink.channel = memoryChannel
```

```
agent.channels = memoryChannel  
agent.channels.memoryChannel.type = memory
```

Start the Flume agent:

Start the Flume agent using the `flume-ng` command, specifying the configuration file.

The events will pass through the interceptor, and the timestamp attribute will be added to each event before being stored in HDFS.

This example demonstrates how to create and configure a custom interceptor in Flume. You can extend this concept to implement various transformations or enrichments on events based on your specific requirements.

Channel Selectors

These are used to determine which channel is to be opted to transfer the data in case of multiple channels. There are two types of channel selectors –

Default channel selectors – These are also known as replicating channel selectors they replicates all the events in each channel.

Multiplexing channel selectors – These decides the channel to send an event based on the address in the header of that event.

Sink Processors

These are used to invoke a particular sink from the selected group of sinks. These are used to create failover paths for your sinks or load balance events across multiple sinks from a channel.



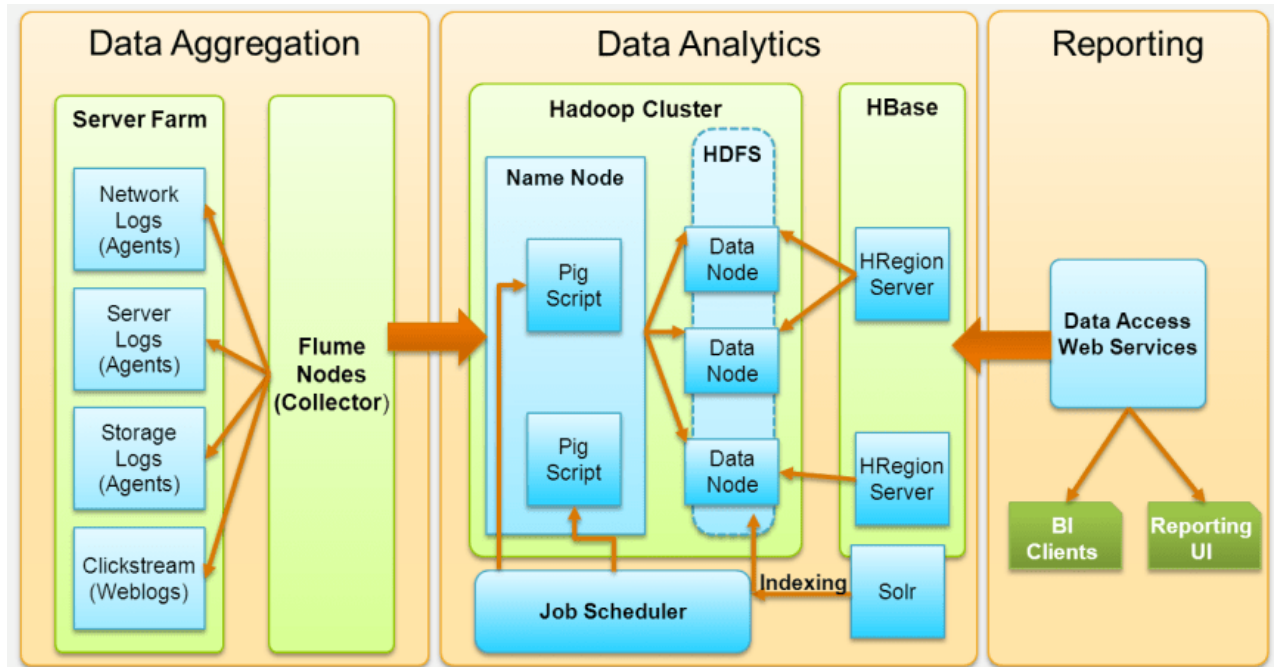
A Flume agent with one flow

A common use case for Flume is loading the weblog data from several sources into HDFS.

Logs would be created in respective Log Servers and logged in local hard discs. This content will then be pushed to HDFS using FLUME framework. FLUME has agents running on Log servers that collect data intermediately using collectors and finally push that data to HDFS and it will be processed by pig or hive and it is stored in structured format into HBase or any other equivalent database. And from there, data will be pulled by Business intelligence tools to generate reports.

Below is the high level overview of the Log analysis flow.

Web server → Flume → HDFS → ETL (Pig/Hive) → Database (HBase) → Reporting



Moving data from Twitter to HDFS

Twitter is a rich source of data in the big data world. Every second around 6000 tweets are posted which contains different types of data. We can fetch those tweets and use the data for various analyses. This tutorial includes installing Apache Flume on Hadoop and retrieving the data from Twitter based on your query.

Prerequisites:

Apache Hadoop must be installed on Ubuntu OS.

Installation of Apache Flume and Fetching tweets from Twitter-

1. Download and Extract the Apache Flume Binary Files using following commands-

```
$ wget https://downloads.apache.org/flume/1.9.0/apache-flume-1.9.0-bin.tar.gz
```

```
$ tar -xvzf apache-flume-1.9.0-bin.tar.gz
```

2. Download flume-sources-1.0-SNAPSHOT.jar and move it to apache-flume-1.9.0-bin/lib

3. Add the path of downloaded snapshot file in **apache-flume-1.9.0-bin/conf/flume-env.sh** file

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

```
FLUME_CLASSPATH="/home/abhee/apache-flume-1.9.0-bin/lib/flume-sources-1.0-SNAPSHOT.jar"
```

4. Add the following code in the **/.bashrc** file-

```
export FLUME_HOME=/home/abhee/apache-flume-1.9.0-bin
```

```
export FLUME_CONF_DIR=$FLUME_HOME/conf
```

```
export FLUME_CLASSPATH=$FLUME_CONF_DIR
```

```
export PATH="$FLUME_HOME/bin:$PATH"
```


Now Run the following command-

```
$source ~/.bashrc
```

5. Rename these 3 files in lib folder of Flume.

(Just change the extension of these files from .jar to .org)

twitter4j-core-3.0.3.jar twitter4j-media-support-3.0.3.jar twitter4j-stream-3.0.3.jar

to

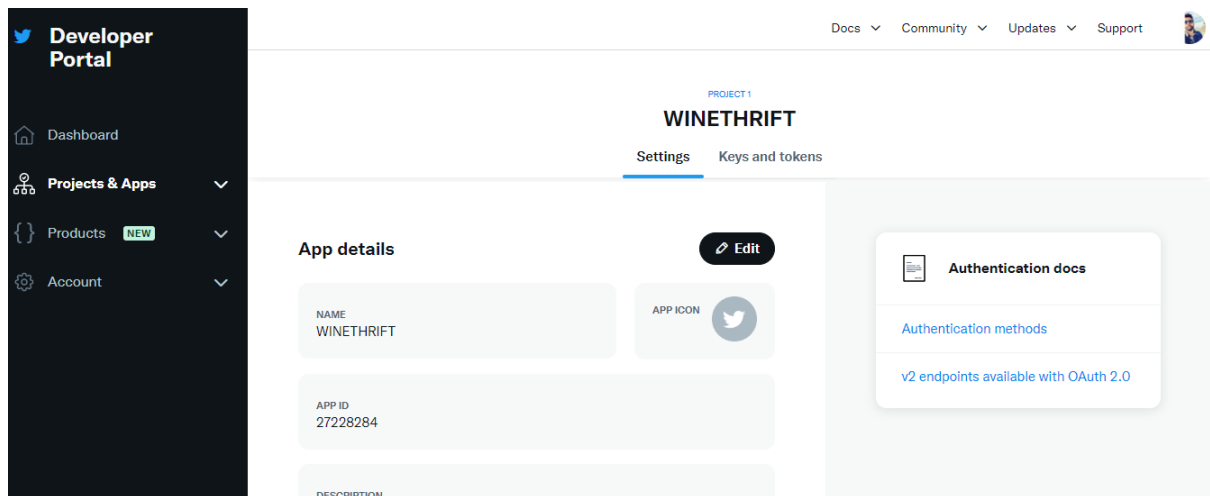
twitter4j-core-3.0.3.org twitter4j-media-support-3.0.3.org twitter4j-stream-3.0.3.org



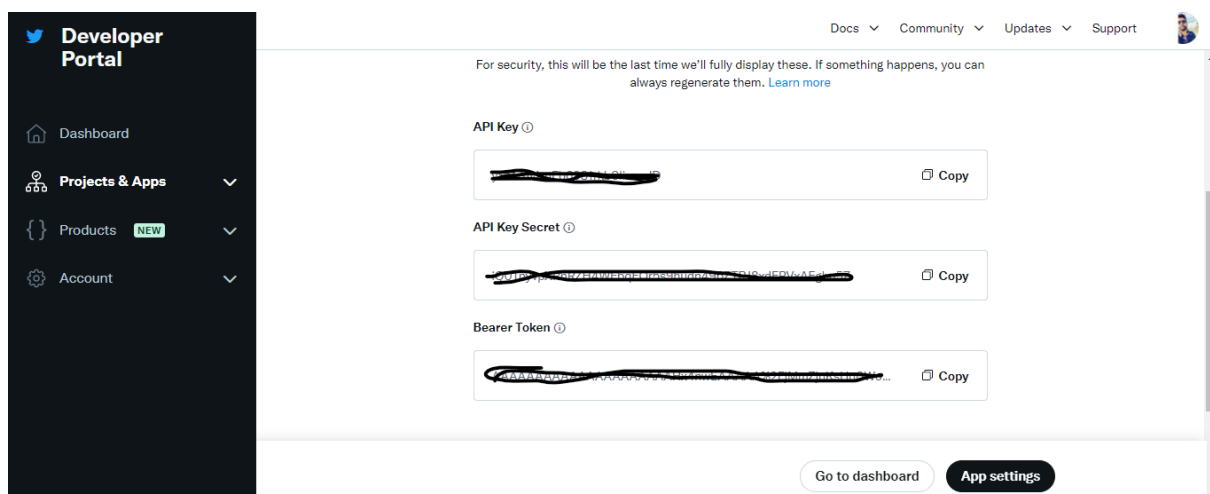
6. Create a Twitter developer account on apps.twitter.com and get the consumerKey, consumerSecret, accessTokenSecret for accessing Tweets.

Apache Sqoop and Apache Flume

Go to create application tab as shown in the below image.



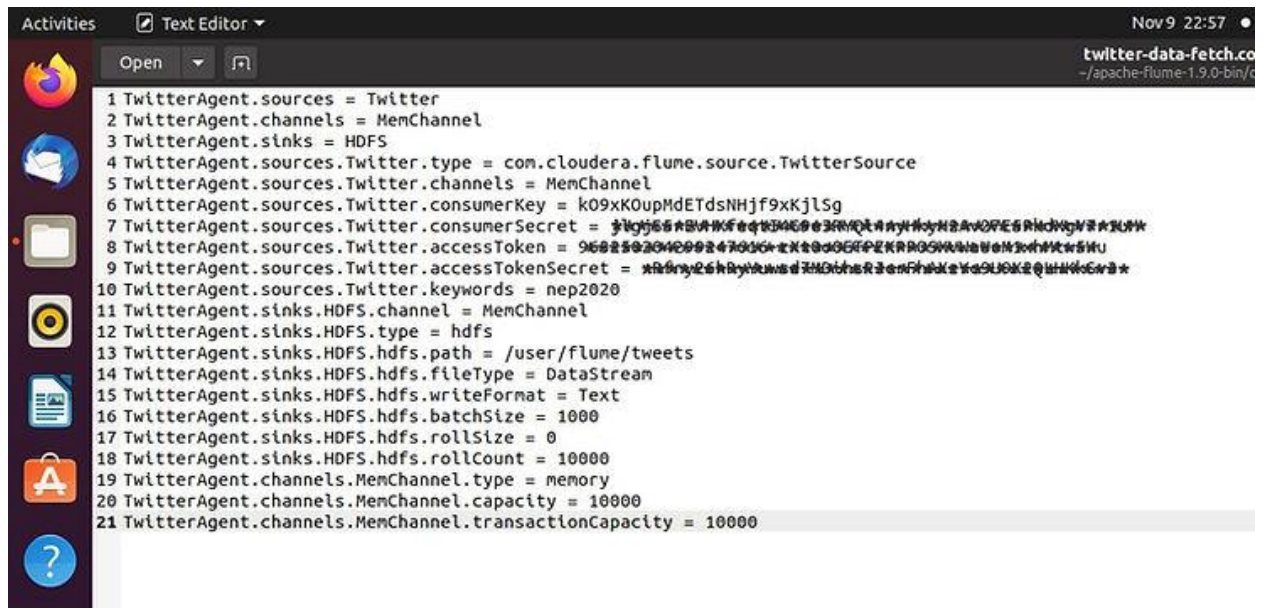
After creating this application, you will find Key & Access token. Copy the key and the access token. We will pass these tokens in our Flume configuration file to connect to this application.



After creating this application, you will find Key & Access token. Copy the key and the access token. We will pass these tokens in our Flume configuration file to connect to this application.

7. Create a file **twitter-data-fetch.conf** in **conf** directory in **apache-flume-1.9.0-bin** and add the following code in the file(Add your Twitter Credentials received from Twitter Developer Account created above)-

```
TwitterAgent.sources = Twitter
TwitterAgent.channels = MemChannel
TwitterAgent.sinks = HDFS
TwitterAgent.sources.Twitter.type = com.cloudera.flume.source.TwitterSource
TwitterAgent.sources.Twitter.channels = MemChannel
TwitterAgent.sources.Twitter.consumerKey = xxxxxxxxxxxxxxxxxxxxxxxxxxxx
TwitterAgent.sources.Twitter.consumerSecret = xxxxxxxxxxxxxxxxxxxxxxxx
TwitterAgent.sources.Twitter.accessToken = xxxxxxxxxxxxxxxxxxxxxxxxxxxx
TwitterAgent.sources.Twitter.accessTokenSecret =
xRfry26hRyYuwsd7MDihsRJcrFhAXzYc9U9X2QLHKkSv3
TwitterAgent.sources.Twitter.keywords = nep2020
TwitterAgent.sinks.HDFS.channel = MemChannel
TwitterAgent.sinks.HDFS.type = hdfs
TwitterAgent.sinks.HDFS.hdfs.path = /user/flume/tweets
TwitterAgent.sinks.HDFS.hdfs.fileType = DataStream
TwitterAgent.sinks.HDFS.hdfs.writeFormat = Text
TwitterAgent.sinks.HDFS.hdfs.batchSize = 1000
TwitterAgent.sinks.HDFS.hdfs.rollSize = 0
TwitterAgent.sinks.HDFS.hdfs.rollCount = 10000
TwitterAgent.channels.MemChannel.type = memory
TwitterAgent.channels.MemChannel.capacity = 10000
TwitterAgent.channels.MemChannel.transactionCapacity = 10000
```



And create a directory `user/flume/tweets` in HDFS to store the tweets fetched from Twitters.

8. In terminal, **go to /home/abhee/apache-flume-1.9.0-bin/** and **run** the following command to start the flume process to fetch the tweets from Twitter-

```
$ bin/flume-ng agent --conf-file conf/twitter-data-fetch.conf --name TwitterAgent -
Dflume.root.logger=INFO,console
```

Now we can see the process will start fetching the data from Twitter and store it into HDFS in /user/flume/tweets.

It will keep on fetching the data, to stop the execution we need to terminate the process manually by pressing Ctrl+Z.

9. Now you can check the fetched data into HDFS using CLI or GUI Web Interface by moving to the location.

The data will be in raw format