



POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Flume and Sqoop

Marco Brambilla

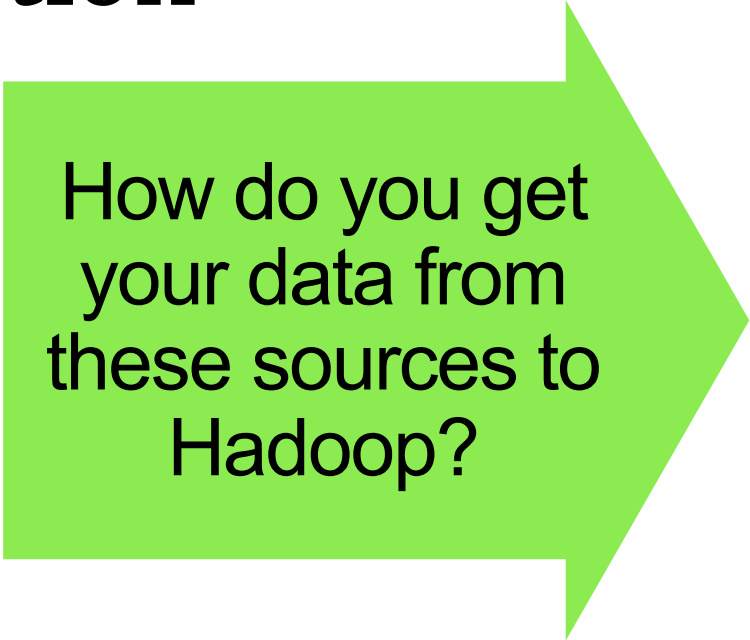
marco.brambilla@polimi.it

 @marcobrambi

Data Ingestion

RDBMS

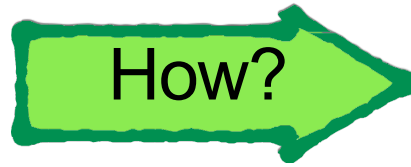
Applications



How do you get
your data from
these sources to
Hadoop?

Hadoop

Application
RDBMS



Hadoop

Normally, Hadoop ecosystem
technologies expose **Java APIs**:
use these APIs to write to
HDFS, HBase etc.

Issues

- Multiple events
- Streaming
- Diverse sources
- Buffering
- ...

Flume and Sqoop can help!

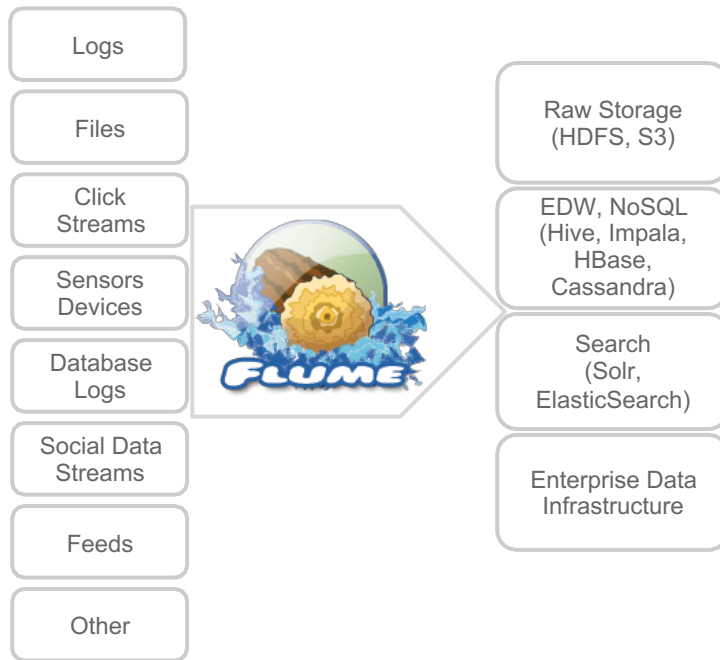
Flume

Apache Flume

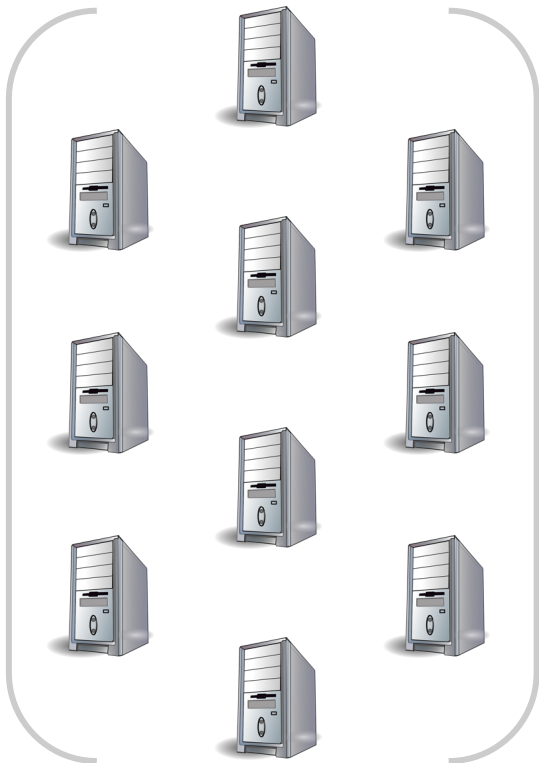
Apache Flume is a **continuous data ingestion system** that is...

- *open-source,*
- *reliable,*
- *scalable,*
- *manageable,*
- *customizable,*

...and designed for **Big Data ecosystem**.



Multiple Sources



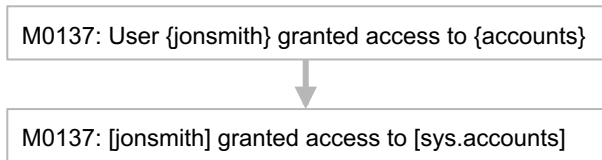
- **Many** physical sources that produce data
- Number of physical sources **changes constantly**
- Sources may exist in **different governance zones**, data centers, continents...

Ever-changing Data

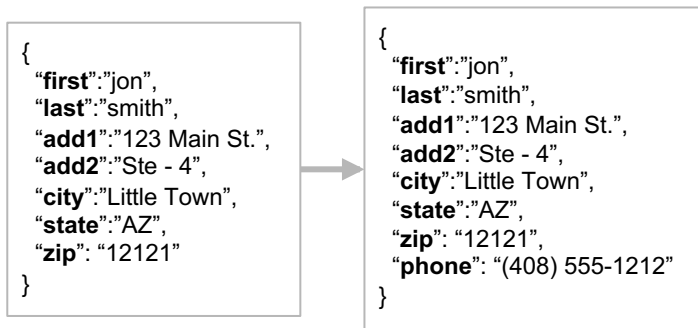
- One of your data centers upgrade to IPv6



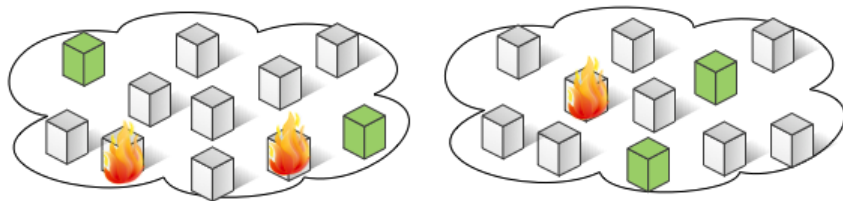
- Application developer changes logs (again)



- JSON data may contain more attributes than expected



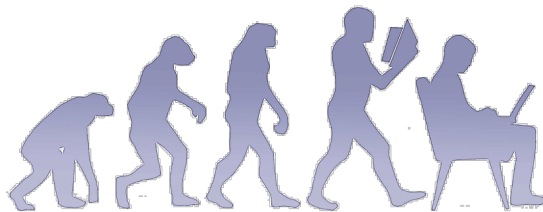
Data Ingestion Perspective



Massive collection of ever changing physical sources...



Never ending data production...



Continuously evolving data structures and semantics...

Flume History

- Originally designed to be a log aggregation system by Cloudera Engineers
- Evolved to handle any type of streaming event data
- Low-cost of installation, operation and maintenance
- Highly customizable and extendable



<http://flume.apache.org/>

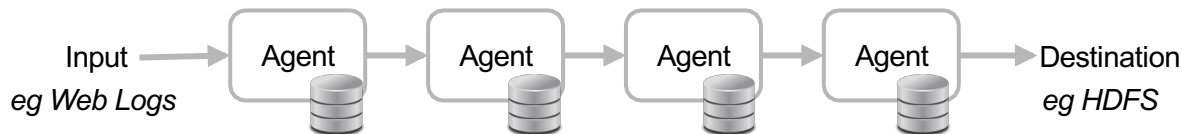
Flume Agent

Simplest unit in Flume

Can connect any number of sources to any number of data stores



The Big Picture



- Distributed Pipeline Architecture
- Optimized for commonly used data sources and destinations
- Built in support for contextual routing
- Fully customizable and extendable

Flume Events

A Flume Event is the base unit of communication between components



Source pushes events

Sink polls for events

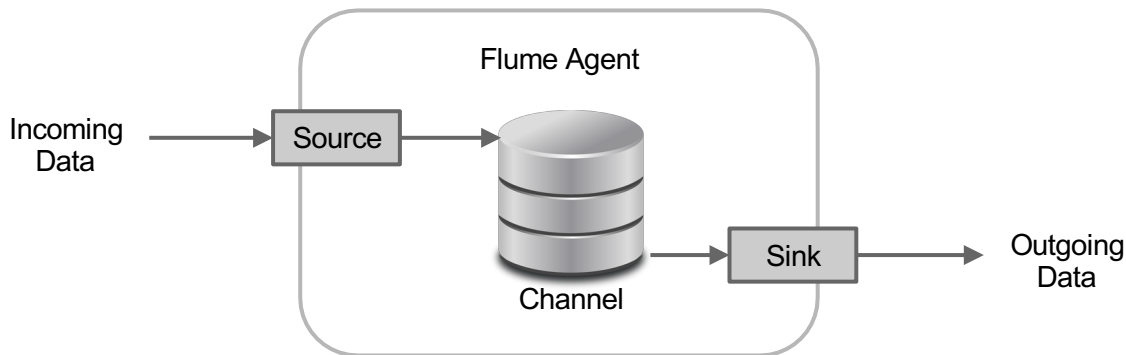
Events

Header: metadata

- Can be used for routing content

Body: actual content

A Flume Agent



Source

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

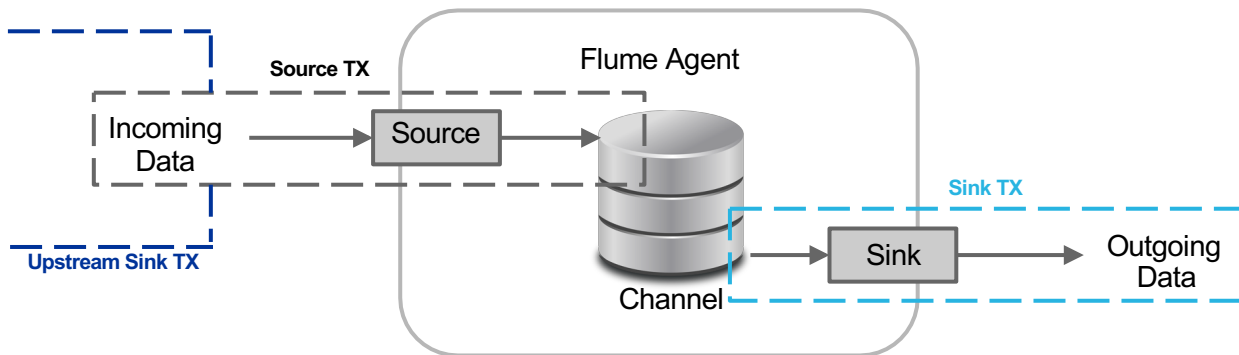
Channel

- Stores data in the order received

Sink

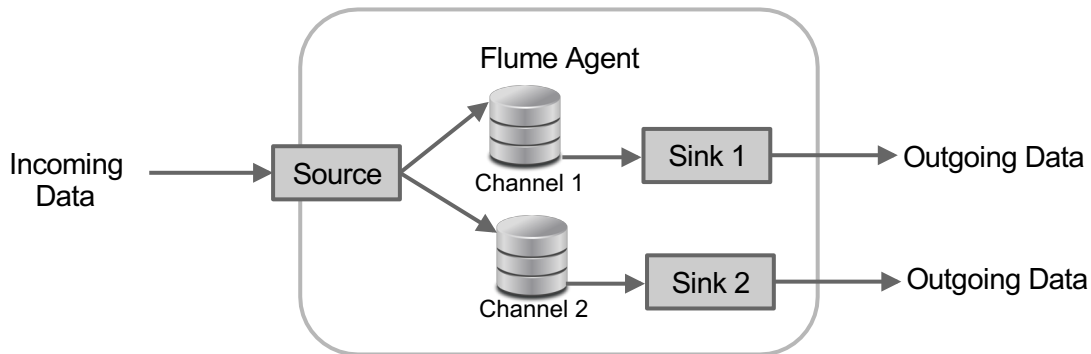
- Removes data from Channel
- Sends data to downstream Agent or Destination

Transactional Data Exchange



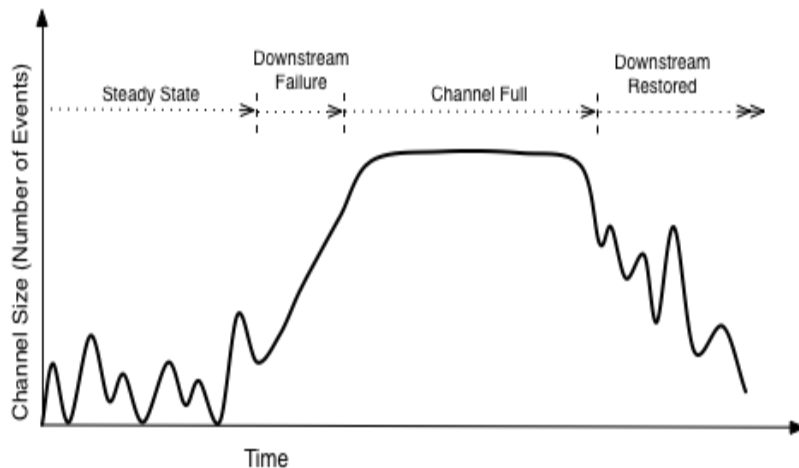
- Source uses transactions to write to the channel
- Sink uses transactions to remove data from the channel
- Sink transaction commits only after successful transfer of data
- This ensures no data loss in Flume pipeline

Routing and Replicating



- Source can **replicate** or **multiplex** data across many channels
- Metadata headers can be used to do **contextual selection** of channels
- Channels can be drained by different sinks to different destinations or pipelines

Why Channels?

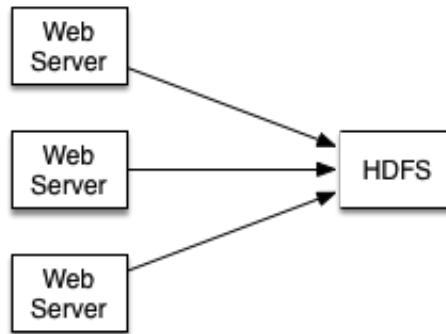


- Buffers data and insulates downstream from load spikes
- Provides persistent store for data in case the process restarts
- Provides flow ordering* and transactional guarantees

Log Aggregation Case

No Flume

- You would like to move your web-server logs to HDFS
- Let's assume there are only 3 web servers at the time of launch
- Ad-hoc solution will likely suffice!



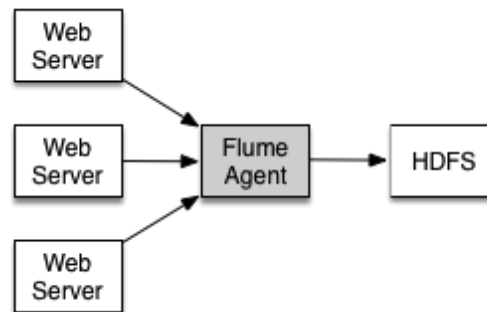
Challenges

- How do you manage your output paths on HDFS?
- How do you maintain your client code in face of changing environment as well as requirements?

1 Flume Agent

Advantages

- Insulation from HDFS downtime
- Quick offload of logs from Web Server machines
- Better Network utilization



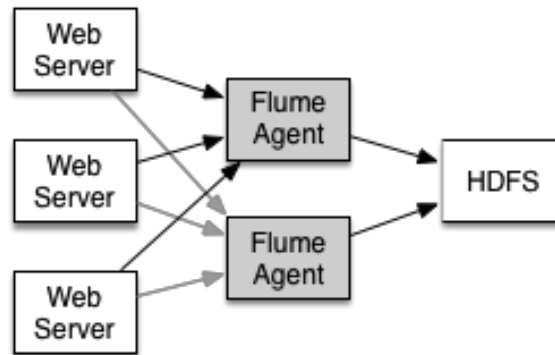
Challenges

- What if the Flume node goes down?
- Can one Flume node accommodate all load from Web Servers?

2 Flume Agents

Advantages

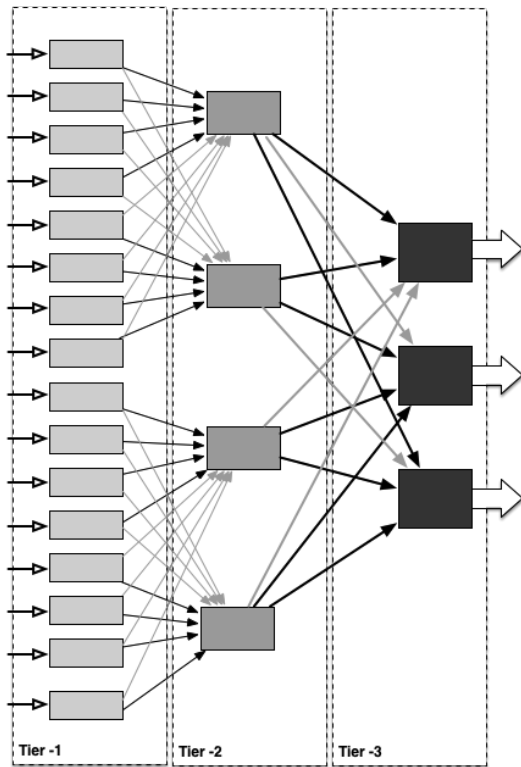
- Redundancy and Availability
- Better handling of downstream failures
- Automatic load balancing and failover



Challenges

- What happens when new Web Servers are added?
- Can two Flume Agents keep up with all the load from more Web Servers?

Multi-Step Flow

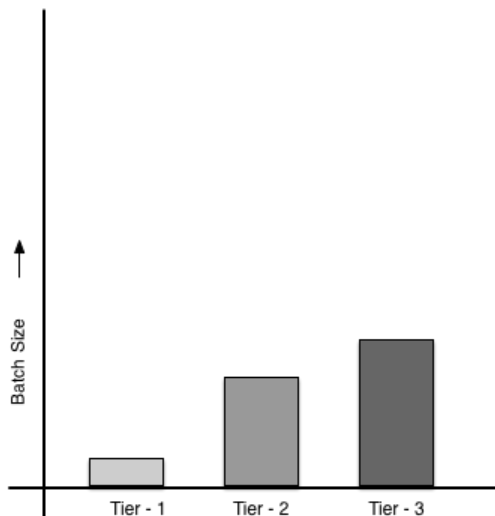


A Converging Flow

- Traffic is aggregated by Tier-2 and Tier-3 before being put into destination system
- Closer a tier is to the destination, larger the batch size it delivers downstream
- Optimized handling of destination systems

Planning and Sizing

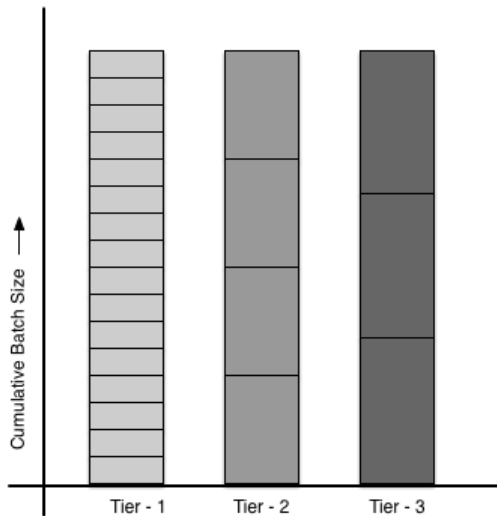
Data Volume Distribution – Agent



Batch Size Variation per Agent

- Event volume is least in the outermost tier
- Event volume increases as the flow converges
- Event volume is highest in the innermost tier

Data Volume Distribution – Tier



Batch Size Variation per Tier

- In steady state, all tiers carry same event volume
- Transient variations in flow are absorbed and ironed out by channels
- Load spikes are handled smoothly without overwhelming the infrastructure

Planning and Sizing

What we know:

- Number of Web Servers
- Log volume per Web Server per unit time
- Destination System and layout (Routing Requirements)
- Worst case downtime for destination system

What we will calculate:

- Number of tiers
- Exit Batch Sizes
- Channel capacity

Rule of Thumb

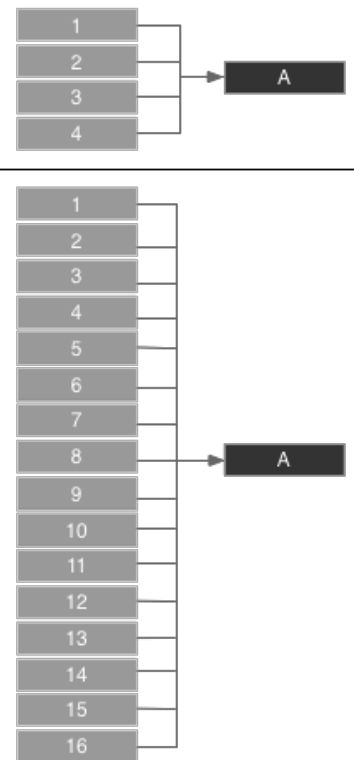
One Aggregating Agent (A) can be used with anywhere from 4 to 16 client Agents

Considerations

- Must handle projected ingest volume
- Resulting number of tiers should provide for routing, load-balancing and failover requirements

Test

Load test to ensure that steady state and peak load are addressed with adequate failover capacity



Exit Size

Rule of Thumb

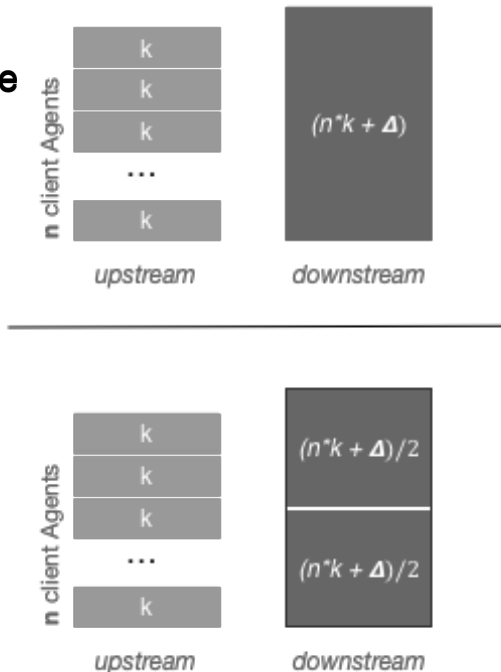
Exit batch size is same as total exit data volume divided by number of Agents in a tier

Considerations

- Having some extra room is good
- Keep contextual routing in mind
- Consider duplication impact when batch sizes are large

Test

Load test fail-over scenario to ensure near steady-state drain



Channel Capacity

Rule of Thumb

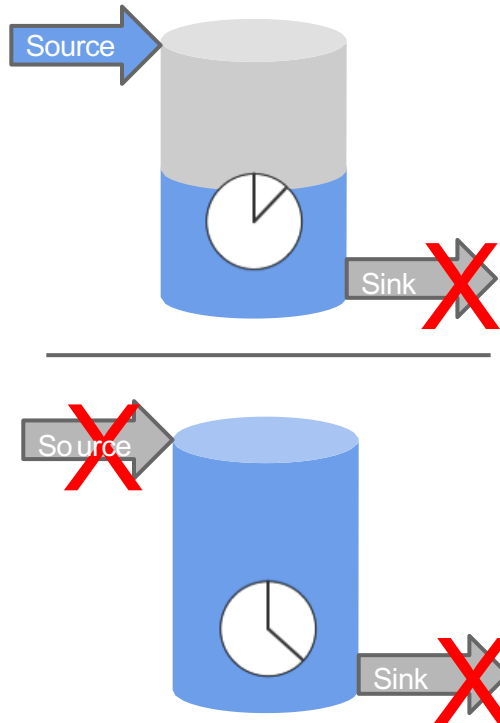
Equal to worst case data ingest rate sustained over the worst case downstream outage interval

Considerations

- Multiple disks will yield better performance
- Channel size impacts the back-pressure buildup in the pipeline

Test

You may need more disk space than the physical footprint of the data size



Summary

Number of Tiers

Calculated with upstream to downstream Agent ratio ranging from **4:1** to **16:1**. Factor in routing, failover, load-balancing requirements...

Exit Batch Size

Calculated for steady state data volume exiting the tier, divided by number of Agents in that tier. Factor in contextual routing and duplication due to transient failure impact...

Channel Capacity

Calculated as worst case ingest rate sustained over the worst case downstream downtime. Factor in number of disks used etc...

Implementation

Source-Channel-Sink

Source pushes data to channel

Channel stores it until Sink reads and writes it
in output store (polling)

Source – Channel: many-to-many

Channel– Sink: one-to-many

(sink reads only from one channel)

Types of Sources

Spooling Directory

- Text Files, Immutable, Unique name

SysLog

Exec

Custom

Types of Channels

Memory

- In-memory queue
- Fast
- Not persistent
- Limited capacity

Disk

Types of Sinks

HDFS

HBase

Console Log

Local

Directory

Example: Spool to log collector

1. Configure agent(s) in property file
2. Run Flume agent from console

```
$ flume-ng agent  
--conf-file spool-to-logger.properties  
--name agent1
```

Properties File: 1-1-1

spool-to-logger.properties

```
agent1.sources = source1
agent1.sinks = sink1
agent1.channels = channel1

agent1.sources.source1.channels = channel1
agent1.sinks.sink1.channel = channel1

agent1.sources.source1.type = spooldir
agent1.sources.source1.spoolDir = /Users/udi/tmp/spooldir

agent1.sinks.sink1.type = logger
agent1.channels.channel1.type = file
```

HDFS

Sink = `org.apache.flume.sink.hdfs.HDFSEventSink`

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /tmp/flume
agent1.sinks.sink1.hdfs.filePrefix = events
agent1.sinks.sink1.hdfs.fileSuffix = .log
agent1.sinks.sink1.hdfs.inUsePrefix = _
agent1.sinks.sink1.hdfs.fileType = DataStream
```

Transfer

By default, files are rolled over every 30 seconds.

Configurable:

- rollCount
- rollSize

Output file types

SequenceFile = binary format

DataStream = uncompressed files (text)

CompressedStream = compressed files

HTTP to HDFS

```
httpagent.sources = http-source
httpagent.sinks = hdfs-sink
httpagent.channels = ch

httpagent.sources.http-source.channels = ch
httpagent.sinks.hdfs-sink.channel = ch

# Define / Configure Source
#####
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
httpagent.sources.http-source.channels = ch
httpagent.sources.http-source.bind = localhost
httpagent.sources.http-source.port = 8989

# HDFS File Sink
#####
httpagent.sinks.hdfs-sink.type = hdfs
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/hdfs
httpagent.sinks.hdfs-sink.hdfs.filePrefix = events
httpagent.sinks.hdfs-sink.hdfs.fileSuffix = .log
httpagent.sinks.hdfs-sink.hdfs.inUsePrefix = _
httpagent.sinks.hdfs-sink.hdfs.fileType = DataStream

# Channels
#####
httpagent.channels.ch.type = memory
httpagent.channels.ch.capacity = 1000
```

Send and get HTTP Posts

Fixed format: header + body

```
$ curl -X POST \  
-H 'Content-Type: application/json; charset=UTF-8' \  
-d '[ {"headers" : {"eventheader1" : "event1", \  
"eventheader2" : "event2" } , \  
"body" : "This is the \  
body1 }]' \  
http://localhost:8989
```

Bucketing

```
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http
```

Dynamic assignment of files to folders

```
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http/{topic}
```

Files assigned to folder based on the “Topic”
event header value

```
{  
  "headers" : {"topic" : "topic1"} ,  
  "body" : "This is the body1"  
}
```

Time-base bucketing

```
agent1.sinks.sink1.hdfs.useLocalTimeStamp = true
```

```
agent1.sinks.sink1.hdfs.path = /tmp/flume/twitterinterceptor/%Y/%m/%d/%H
```

Spool to HBASE

```
agent1.sinks.hbaseSink.type = org.apache.flume.sink.hbase.AsyncHBaseSink  
agent1.sinks.hbaseSink.table = test_table  
agent1.sinks.hbaseSink.columnFamily = test  
agent1.sinks.hbaseSink.serializer=org.apache.flume.sink.hbase.SimpleAsyncHbaseEventSerializer  
agent1.sinks.hbaseSink.serializer.payloadColumn = pCol
```

Stores in specific Table and Column
They must exist already in HBASE

Many Channels and Sinks

```
httpagent.sources = http-source
httpagent.sinks = hdfs-sink log-sink
httpagent.channels = ch1 ch2

httpagent.sources.http-source.channels = ch1 ch2
httpagent.sinks.hdfs-sink.channel = ch1
httpagent.sinks.log-sink.channel = ch2
# Define / Configure Source
#####
httpagent.sources.http-source.type = org.apache.flume.source.http.HTTPSource
httpagent.sources.http-source.bind = localhost
httpagent.sources.http-source.port = 8989
# HDFS File Sink
#####
httpagent.sinks.hdfs-sink.type = hdfs
httpagent.sinks.hdfs-sink.hdfs.path = /tmp/flume/http
httpagent.sinks.hdfs-sink.hdfs.filePrefix = events
httpagent.sinks.hdfs-sink.hdfs.fileSuffix = .log
httpagent.sinks.hdfs-sink.hdfs.inUsePrefix = _
httpagent.sinks.hdfs-sink.hdfs.fileType = DataStream
# Logger sink
#####
httpagent.sinks.log-sink.type = logger
# Channels
#####
httpagent.channels.ch1.type = memory
httpagent.channels.ch1.capacity = 1000

httpagent.channels.ch2.type = file
```

Many channels and sinks

Replicating

- Default
- Every event sent everywhere

Multiplexing

- routes events to channels based on their headers

Multiplexing Selector

```
# Selector
#####
httpagent.sources.http-source.selector.type = multiplexing
httpagent.sources.http-source.selector.header = show
httpagent.sources.http-source.selector.mapping.1 = ch1
httpagent.sources.http-source.selector.mapping.2 = ch2
```

Based on header:

- If show = 1, route to ch1
- If show = 2, route ch2

Twitter Source

```
agent1.sources.source1.type = org.apache.flume.source.twitter.TwitterSource
agent1.sources.source1.consumerKey = **
agent1.sources.source1.consumerSecret = **
agent1.sources.source1.accessToken = **
agent1.sources.source1.accessTokenSecret = **
agent1.sources.source1.keywords = @realDonaldTrump, @HillaryClinton
```

Interceptors

Interceptors can process events based on their contents or headers

Simple data processing

Overhead for Flume

To be used with caution!

RegEx Filter Interceptor

```
agent1.sources.source1.interceptors.regexInterceptor.type = regex_filter  
agent1.sources.source1.interceptors.regexInterceptor.regex = .*election.*  
agent1.sources.source1.interceptors.regexInterceptor.excludeEvents = false
```

Flume Summary

- Flume is suitable for large volume data collection, especially when data is being produced in multiple locations
- Once planned and sized appropriately, Flume will practically run itself without any operational intervention
- Flume provides weak ordering guarantee, i.e., in the absence of failures the data will arrive in the order it was received in the Flume pipeline
- Transactional exchange ensures that Flume never loses any data in transit between Agents. Sinks use transactions to ensure data is not lost at point of ingest or terminal destinations.
- Flume has rich out-of-the box features such as contextual routing, and support for popular data sources and destination systems

Sqoop

Sqoop

- Importing from relational DB sources
- PULL-based (no streams)
- Bulk imports
- With one command line action, import data from RDBMS to HDFS/Hive
- Sqoop comes with connectors for many popular RDBMS: MySQL, Oracle, SQL Server, PostgreSQL, etc

Use Cases

Occasional copy for running map-reduce tasks

Keep RDBMS for transactional needs and periodically dump data on HDFS

Command Line Instructions

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1
```

Command Line Instructions

Sqoop uses the primary key to decide how many mappers to use, and for splitting the rows among mappers

`--m 1` fixes 1 mapper

mandatory if the table you are importing doesn't have a primary key, or if you are importing a query

Split-by

explicitly specify the column to use for splitting rows between mappers

Query based importer

```
--query 'select year, month, day from  
tradingDays where year=2016 and $CONDITIONS'
```

- Instead of full table
- \$CONDITIONS mandatory

Import in Hive

```
sqoop import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--hive-import \  
--hive-table=tradingDays \  
--target-dir /mysql/table/tradingDays2 \  
--m 1
```

Sqoop will create an external table and point the table to the directory in HDFS

Incremental Import: Jobs

Periodically archiving on incremental content on HDFS/Hive

Import only the new data

Save an import configuration as a JOB and call it later on repetitively

Job Definition

```
sqoop job \  
--create myjob \  
--import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1
```

Incremental Job Definition

```
sqoop job \  
--create myjob \  
--import \  
--connect jdbc:mysql://localhost:3306/nseProd \  
--username=qt \  
--password=password \  
--table=tradingDays \  
--target-dir /mysql/nseProd \  
--m 1  
--incremental lastmodified  
--check-column ts
```


Behaviour

Only rows added or modified will be imported

Based on the value of `--check-column`

- Ideally a timestamp of last update of the row
- LastValue saves the max value of the column at every import
- Further import will get data $>$ LastValue

Job Execution

```
sqoop job --exec myjob
```



POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

Flume and Sqoop

Marco Brambilla

marco.brambilla@polimi.it

 @marcobrambi