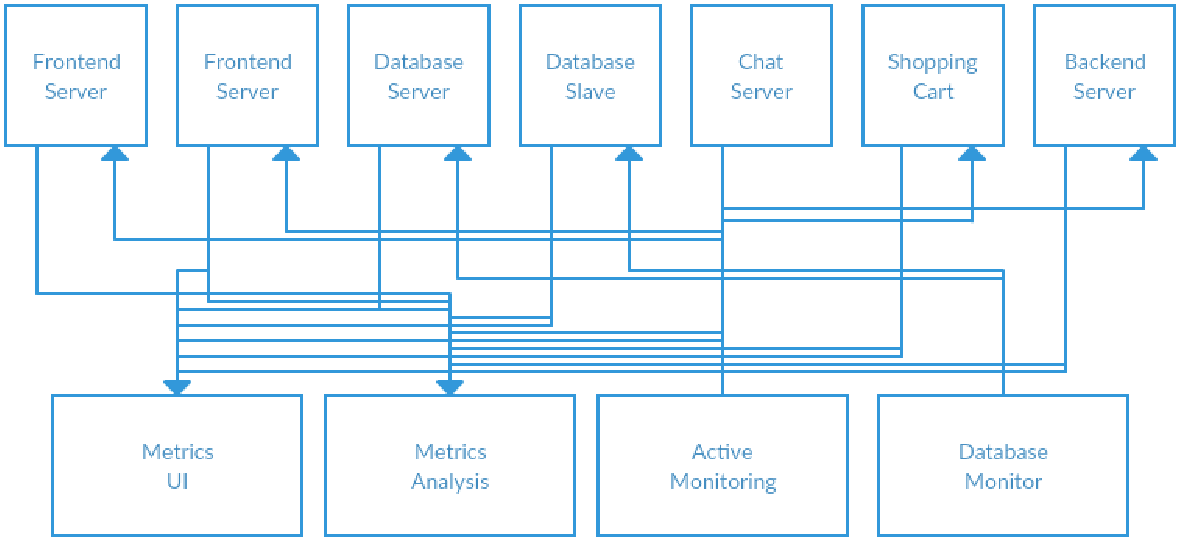
kafka

Kafka Introduction

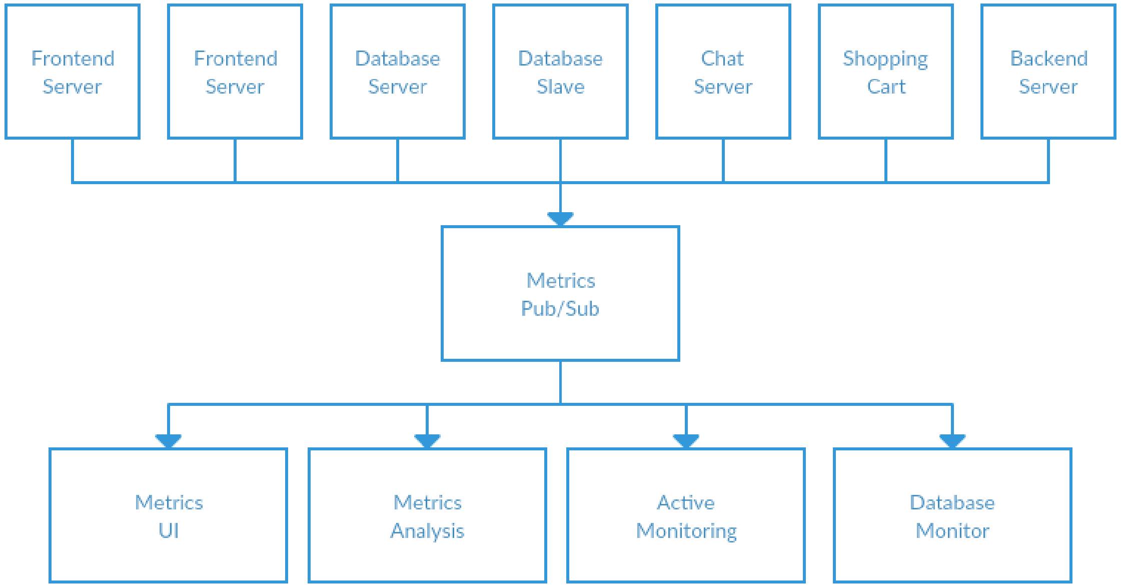
- Kafka is a leading general purpose ***publish-subscribe distributed messaging system***, which offers strong durability, scalability and fault-tolerance support.
- Designed for handling of ***real time activity*** stream data such as logs, metrics collections.
- Written in ***Scala***
- An apache project initially developed at ***LinkedIn*** at the year 2007 then to Apache in 2012 by Jay Kreps in the name of Franz Kafka
- It is **not specifically designed for Hadoop** rather Hadoop ecosystem is just be one of its possible consumers.

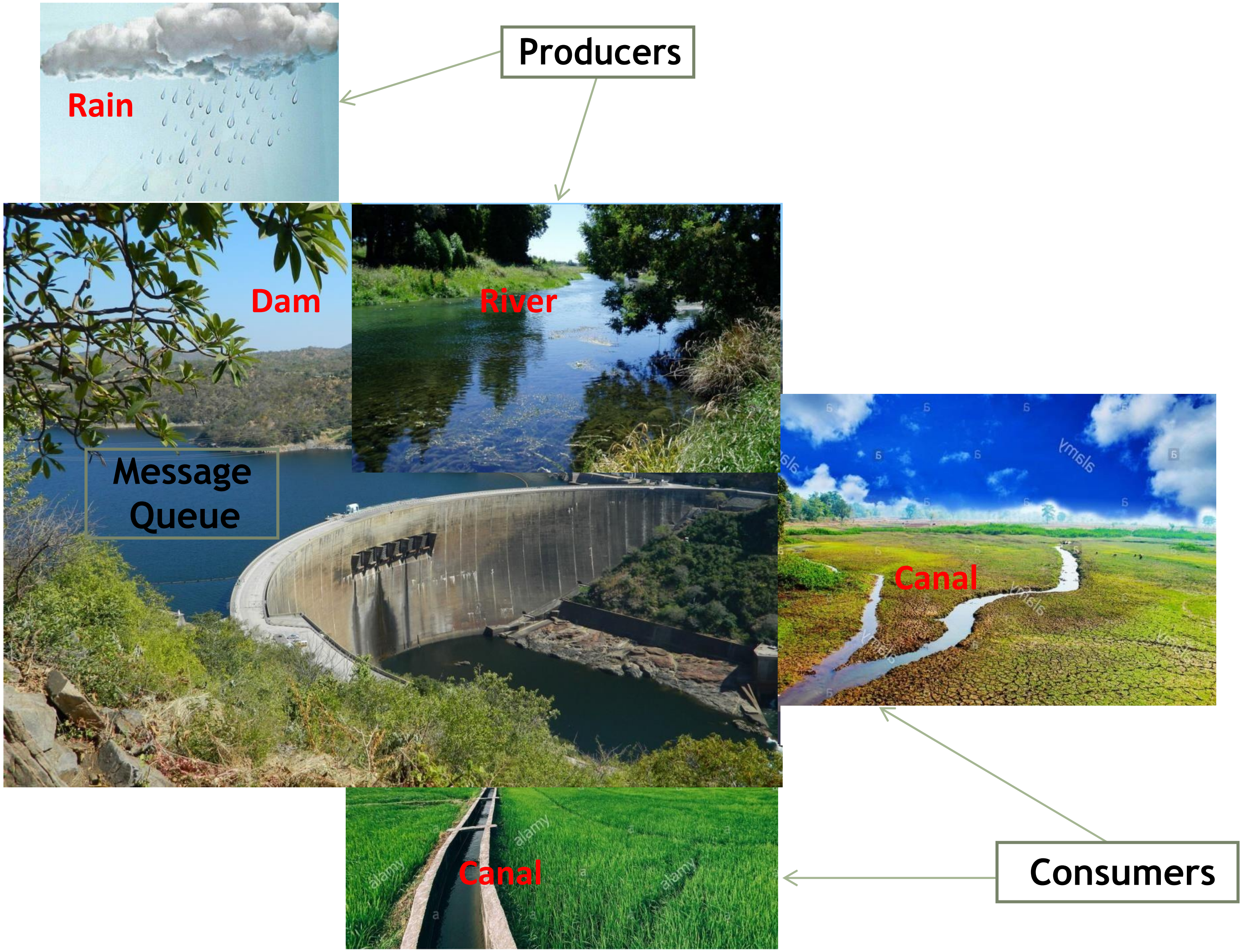
Why we need Kafka

Direct connections



Publish/Subscribe system




Producers
Rain
Dam
River
Message Queue
Canal
Canal
Consumers

Why we need Kafka

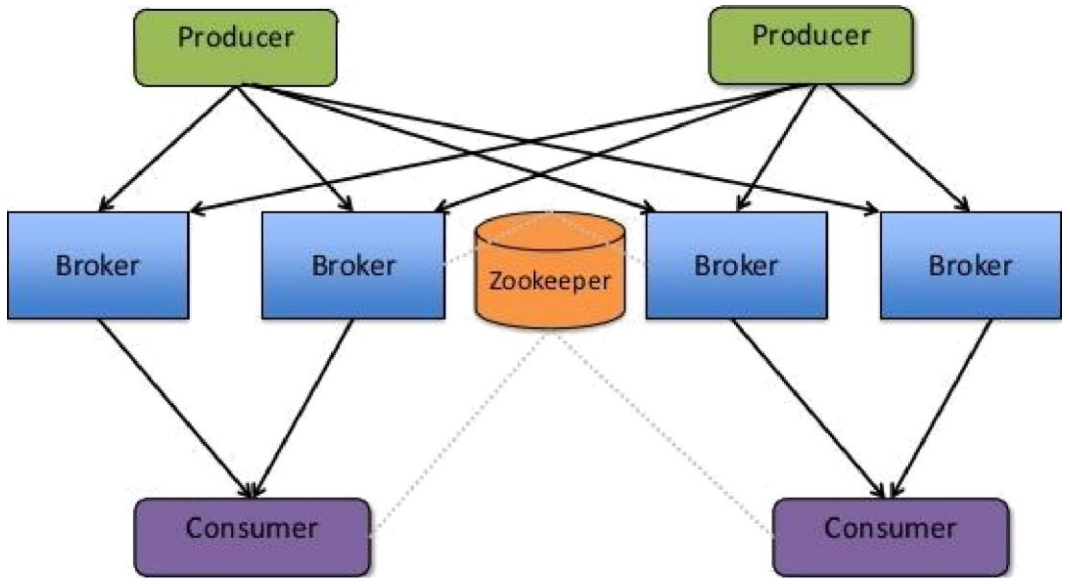
- Kafka is Highly scalable very *easy to add large number of producers and consumers*
- Kafka handle spike
- Kafka also supports different consumption model
- Message durability is high in Kafka – *Persists*
- Integration support for Kafka with other frameworks are high

Applications

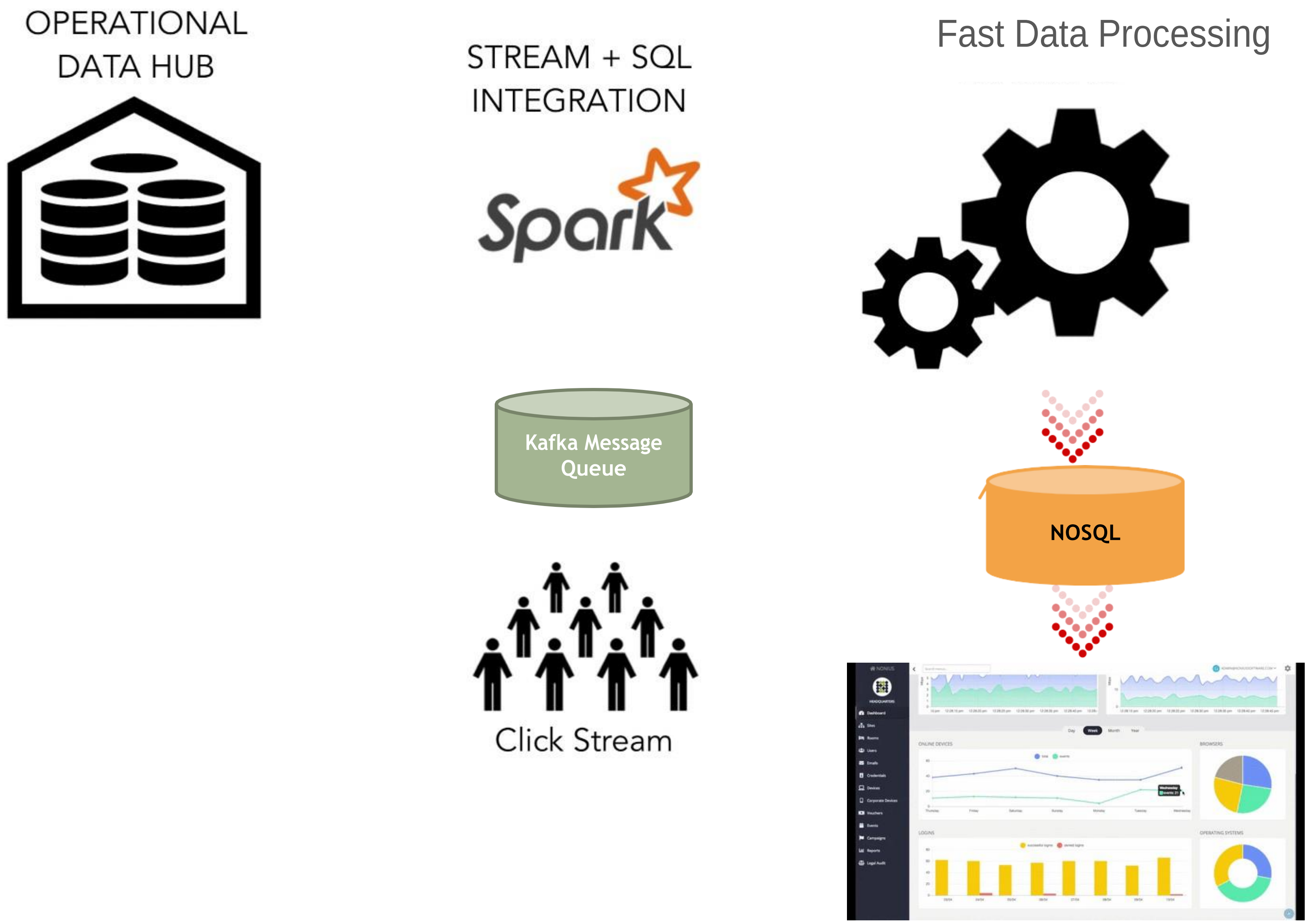
- Clickstream
- Geolocation
- Web Data
- Internet of Things
- Docs, emails
- Server logs

Ka&a High levelArchitecture

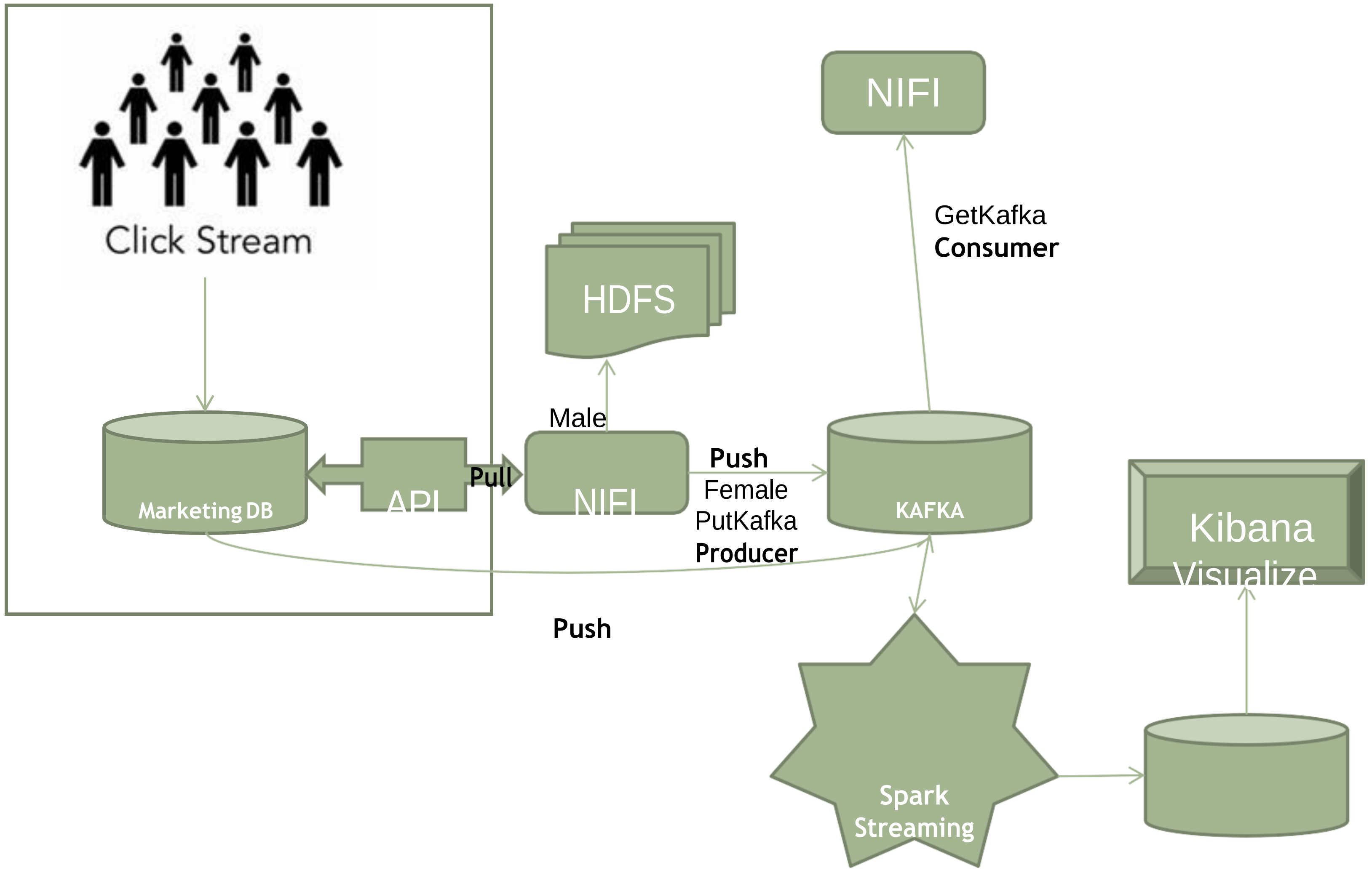
Kafka Architecture



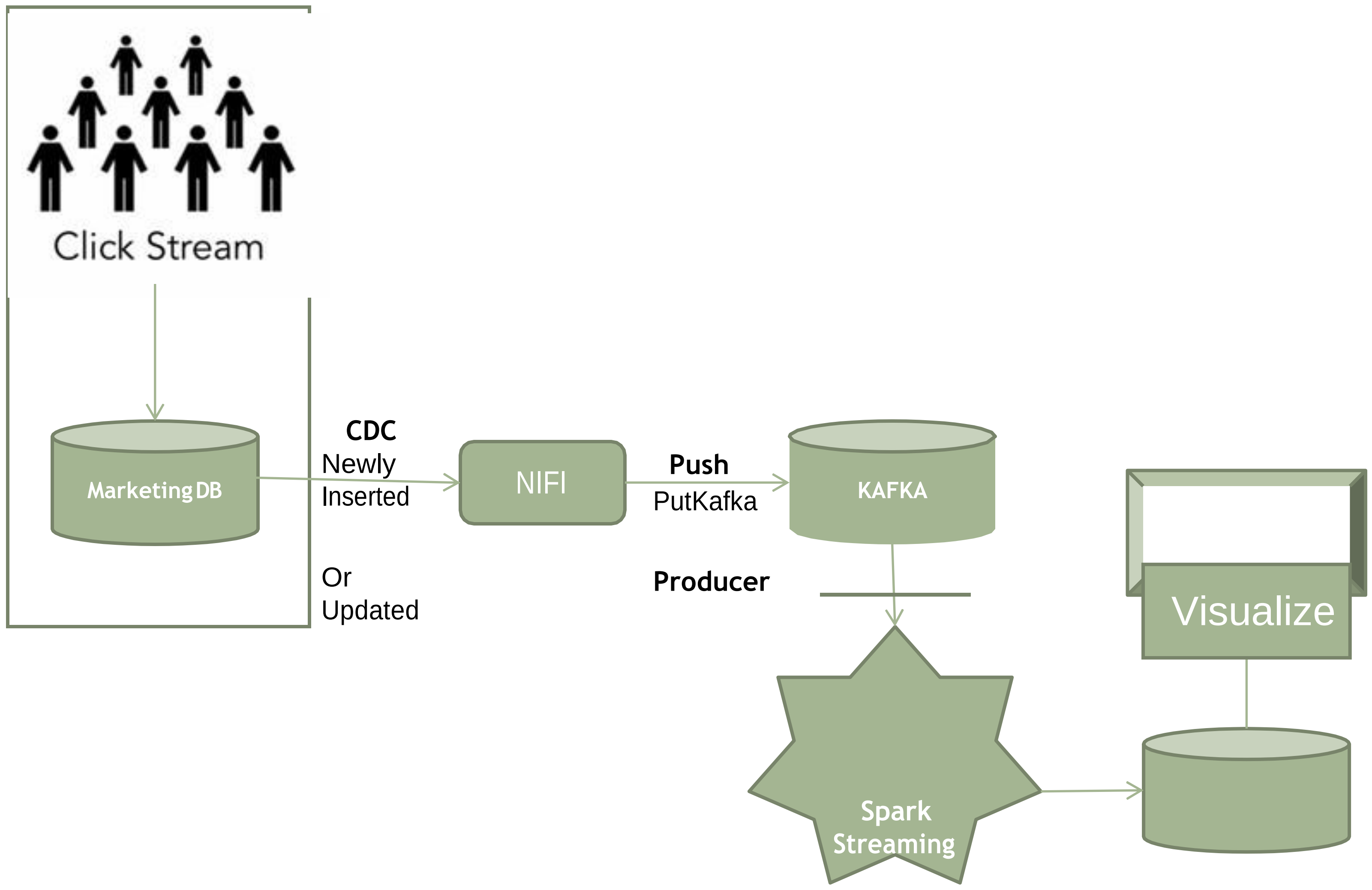
Realtime Data Pipeline



Realtime Data Pipeline – Acquisition, Queue, Processing, Storage, Visualization

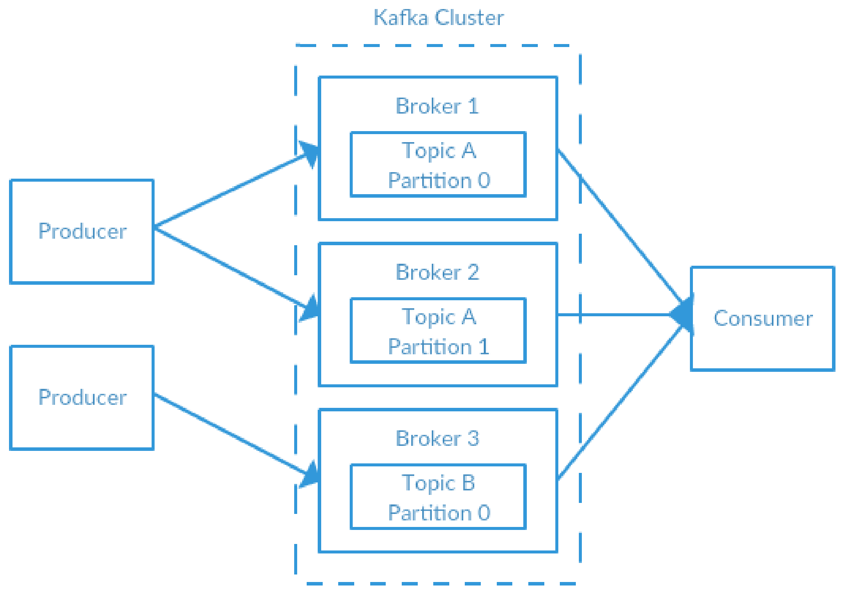


Realtime Data Pipeline



Components

- ➢ Messages
- ➢ Producer
- ➢ Consumer
- ➢ Broker
- ➢ Kafka Cluster
- ➢ Topic
- ➢ Partition
- ➢ Replicas



HDFS	**Kafka**	**SQL**
File System	**Message Queue**	Database
Any Data	Any Data	Only Structured
Distribution & Scalable	Distributed & Scalable	No Distribution & Limited Scalability
File	Topic	Table
Blocks	Partition	Partition
Replica	Replica	No Replica
Fault Tolerance	Fault Tolerant	No Fault Tolerance
Batch	**Message Queue**	Database
Permenant long time data store	**Temporary (staging)**	Only Structured
No Incremental data management using offset	**Offset for incremental data management**	
Large sized file of small in numbers	**Small messages of large in numbers**	
	Used for Messaging System	
Used for Data Lake		

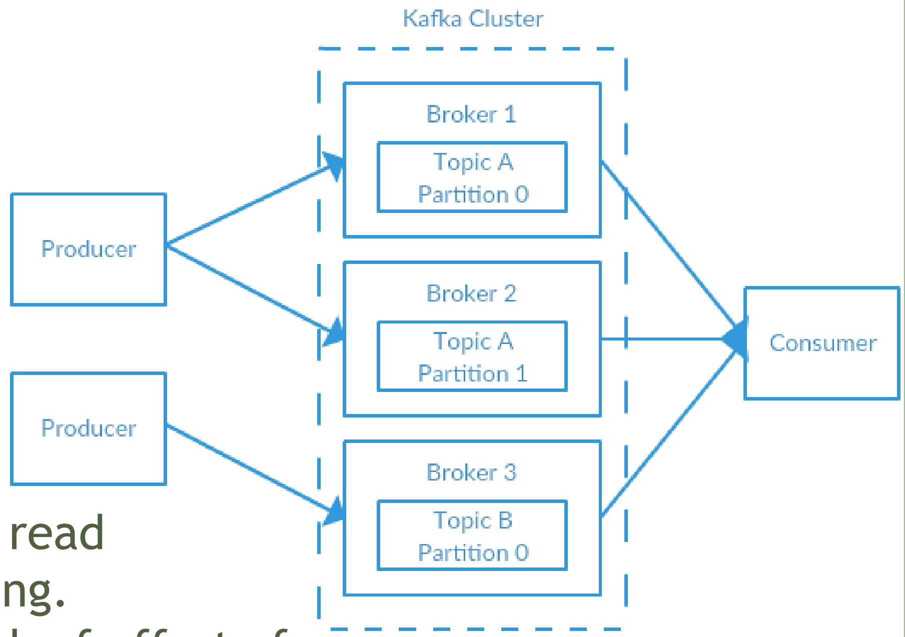
Components (Contd)

- **Messages** : The unit of data within Kafka with optional byte header, eg. Record in a table.

- **Broker – Managed by BigData Platform/Infra Team**
  - A single Kafka server
  - The broker receives messages from producers, assigns offsets to them, and commits on disk.
  - The broker serves consumer responding with the messages based on the partitions.

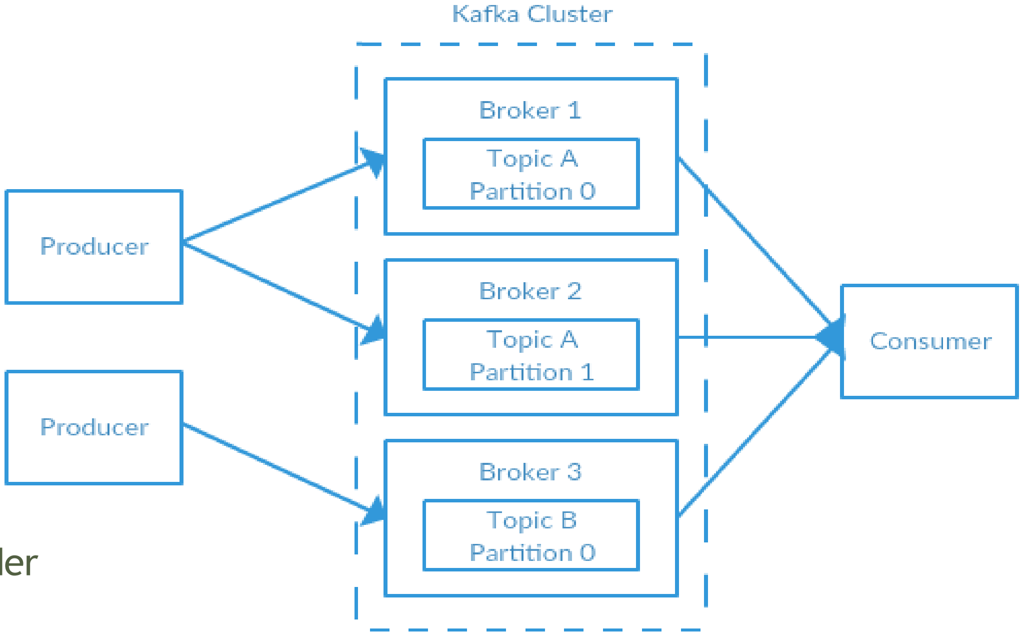
- **Producer**
  - Creates Messages.
  - Maintains messagekey to deliver messages to the particular partitions

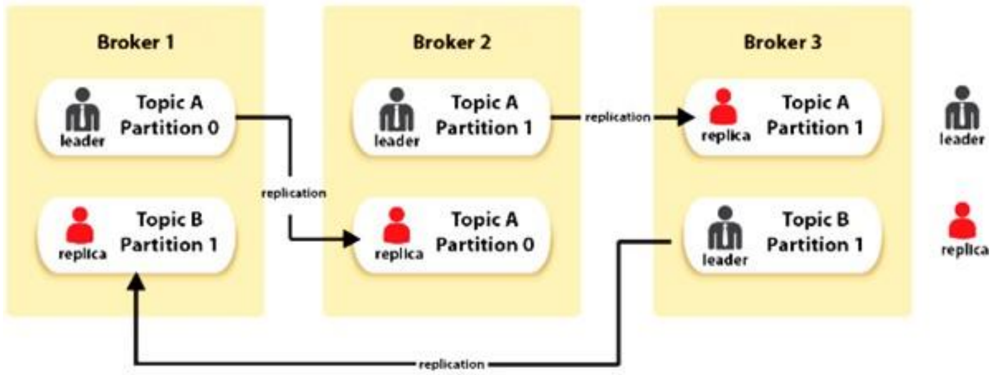
- **Consumer**
  - Consume Messages.
  - Subscribe to one or more topic and read messages for processing or consuming.
  - Broker and Consumer can Keep track of offset of last consumed messages.



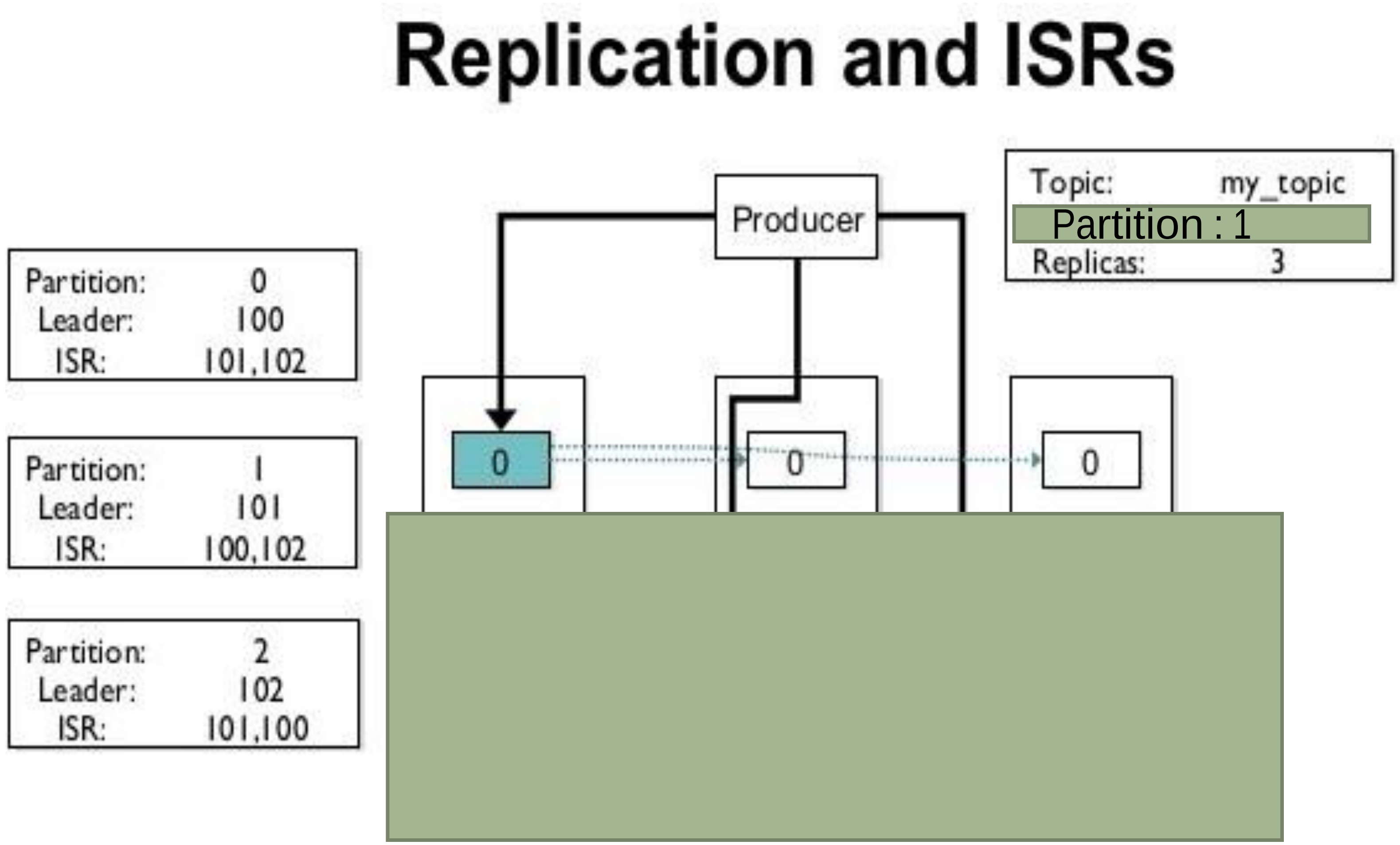
Components (Contd)

- **Kafka Cluster**
  - Group of Brokers
  - One Broker will act as a cluster Controller and assigns leader for the partition.
- **Topic**
  - Category of messages eg. Table/Folder
  - Spread across all Brokers/nodes.
- **Partition**
  - Horizontal division of topics.
  - Scaled across multiple broker nodes.
- **Replicas**
  - Partitions in the Topics are replicated.
  - Producer - ISR maintains – Fire and Forget, Sync and Async.

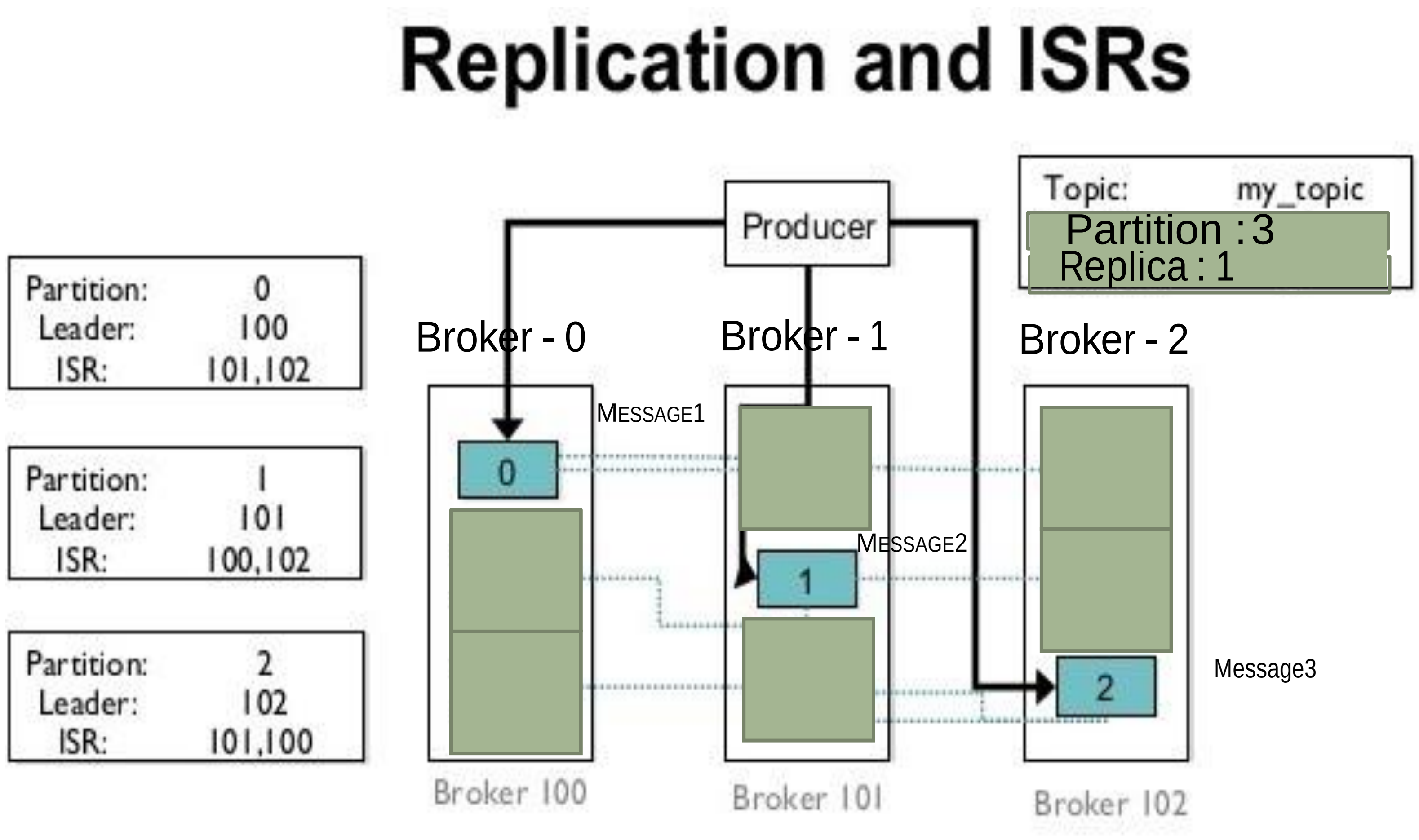




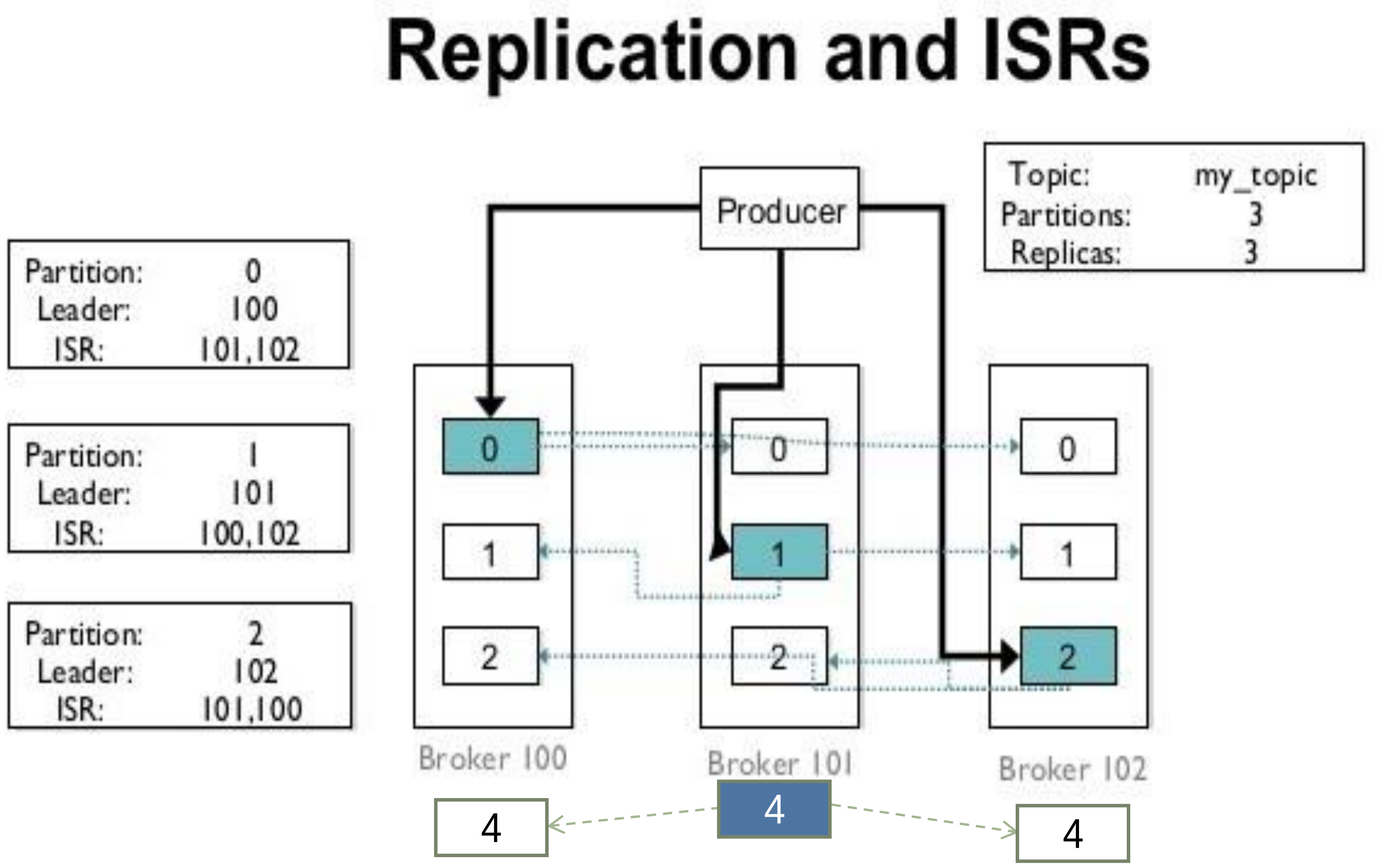
Replicas & ISRs

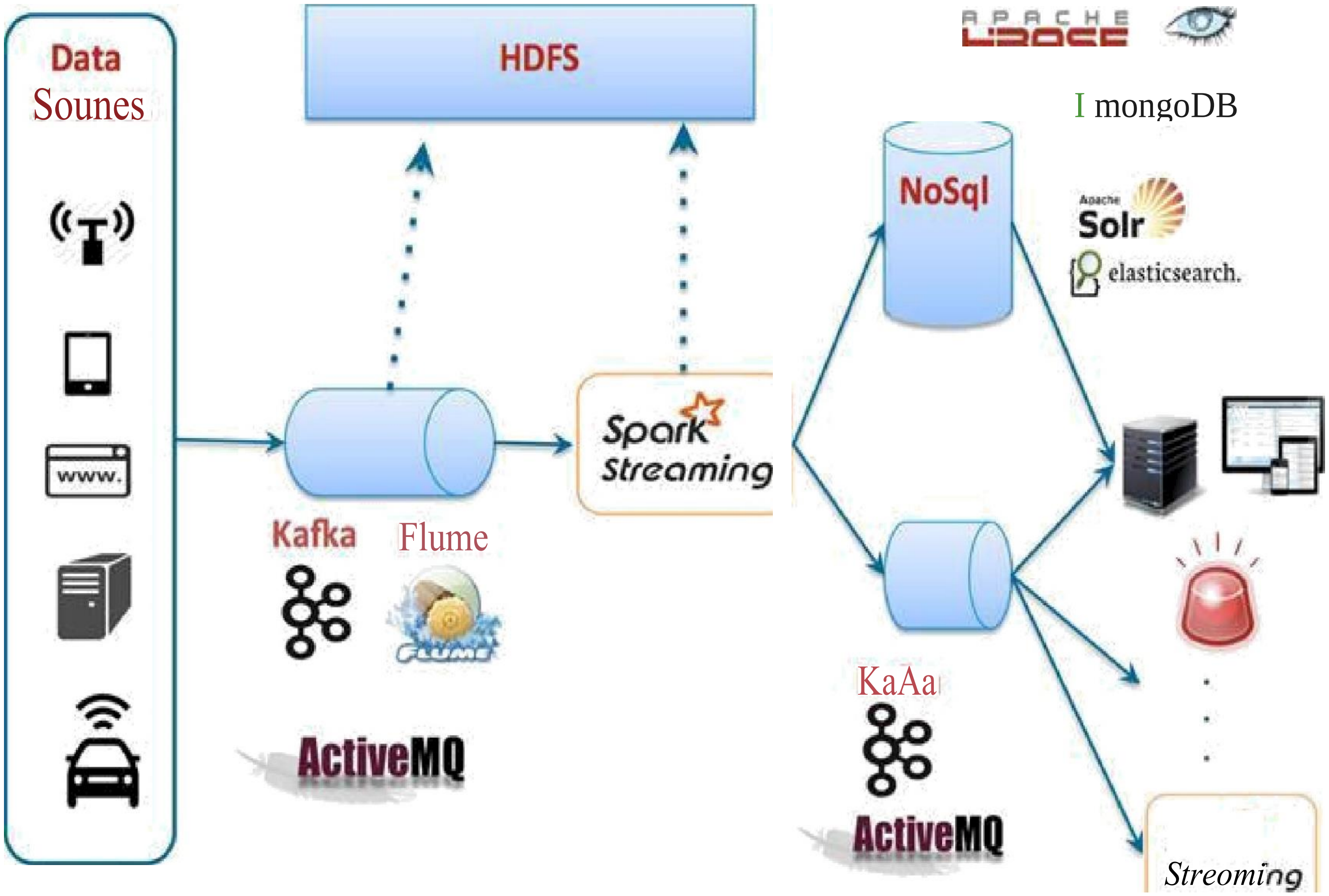


Replicas & ISRs



Replicas & ISRs




Data
Sounes
www.
HDFS
Kafka
Flume
ActiveMQ
Spark Streaming
NoSql
APACHE
I mongoDB
Apache Solr
elasticsearch.
KaAa
ActiveMQ
Streoming

KAFKAWorkouts

1) Start the Zookeeper Coordination service:

```
zookeeper-server-start.sh -daemon /usr/local/kafka/config/zookeeper.properties
```

2) Start the Kafka server:

```
kafka-server-start.sh -daemon /usr/local/kafka/config/server.properties
```

Workouts:

Create a topic (with one replica and one partition)

```
kafka-topics.sh --create –-bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --
topic cts_topic
```

We can now see that topic if we run the list topic command:

```
kafka-topics.sh --list --bootstrap-server localhost:9092
```

Produce messages using Producer:

Kafka comes with a command line client that will take input from a file or from standard input and send it out as messages to the Kafka cluster. By default each line will be sent as a separate message. Run the producer and then type a few messages into the console to send to the server

```
kafka-console-producer.sh --broker-list localhost:9092 --topic cts_topic
```

Start a Consumer

Kafka also has a command line consumer that will dump out messages to standard output.

```
kafka-console-consumer.sh –-bootstrap-server localhost:9092 --topic cts_topic --from-beginning
```

Only look at the incremental logs

```
kafka-console-consumer.sh --zookeeper localhost:9092 --topic cts_topic
```

If you have each of the above commands running in a different terminal then you should now be able to type messages into the producer terminal and see them appear in the consumer terminal.

Setting up a multi-broker cluster

So far we have been running against a single broker. For Kafka, a single broker is just a cluster of size one, so nothing much changes other than starting a few more broker instances. But just to get feel for it, let's expand our cluster to three nodes (still all on our local machine).

First we make a config file for each of the brokers:

```
cp config/server.properties config/server-1.properties
cp config/server.properties config/server-2.properties
```

Now edit these new files and set the following properties:

The broker.id property is the unique and permanent name of each node in the cluster. We have to override the port and log directory only because we are running these all on the same machine and we want to keep the brokers from all trying to register on the same port or overwrite each others data.

```
config/server-1.properties
===========================

listeners=PLAINTEXT://:9093
broker.id=1 port=9093
log.dir=/tmp/kafka-logs-1

config/server-2.properties
============================

listeners=PLAINTEXT://:9094
broker.id=2 port=9094
log.dir=/tmp/kafka-logs-2
```

need to start the two new nodes:

```
kafka-server-start.sh -daemon $KAFKA_HOME/config/server-1.properties

kafka-server-start.sh -daemon $KAFKA_HOME/config/server-2.properties
```

Now create a new topic with a replication factor of three with a single partition:

```
kafka-topics.sh --create –-bootstrap-server localhost:9092 --replication-factor 3 --partitions 1 --topic clickstream
```

```
kafka-topics.sh --list –-bootstrap-server localhost:9092
```

Want to alter partitions use --alter

```
kafka-topics.sh –-bootstrap-server localhost:9092 --alter --topic clickstream --partitions 4
```

How can we know which broker is doing what? To see that run the "describe topics" command:

```
kafka-topics.sh --describe –-bootstrap-server localhost:9092 --topic clickstream
```

We can run the same command on the original topic we created to see where it is:

```
kafka-topics.sh --describe –-bootstrap-server localhost:9092 --topic cts_topic
```

Let's publish a few messages to our new topic clickstream:

```
kafka-console-producer.sh --broker-list localhost:9092 --topic clickstream
```

Now let's consume these messages:

```
kafka-console-consumer.sh –-bootstrap-server localhost:9092 --topic clickstream --from-beginning
```

Leadership has switched to one of the slaves and node 1 is no longer in the in-sync replica set:

```
 kafka-topics.sh --describe –-bootstrap-server localhost:9092 --topic clickstream
```

But the messages are still be available for consumption even though the leader that took the writes originally is down:

```
kafka-console-consumer.sh –-bootstrap-server localhost:9092 --from-beginning --topic clickstream
```

If you want to delete the topic (mark for deletion, at the time of flush will be deleted else use delete.topic.enable=true)

```
kafka-topics.sh --delete –-bootstrap-server localhost:9092 --topic cts_topic
```