Kafka

Kafka and its tools

Shivam jaiswal

Topics to Cover

Overview

- Kafka basics
- Kafka Client API
- Kafka connect
- Kafka streams
- Kafka monitoring
- Schema registry and kSQL

What is Kafka?

Kafka is one of the most popular data streaming processing platforms.

Kafka History

Created at LinkedIn and open sourced in 2011

Kafka was designed to handle the data integration problem.

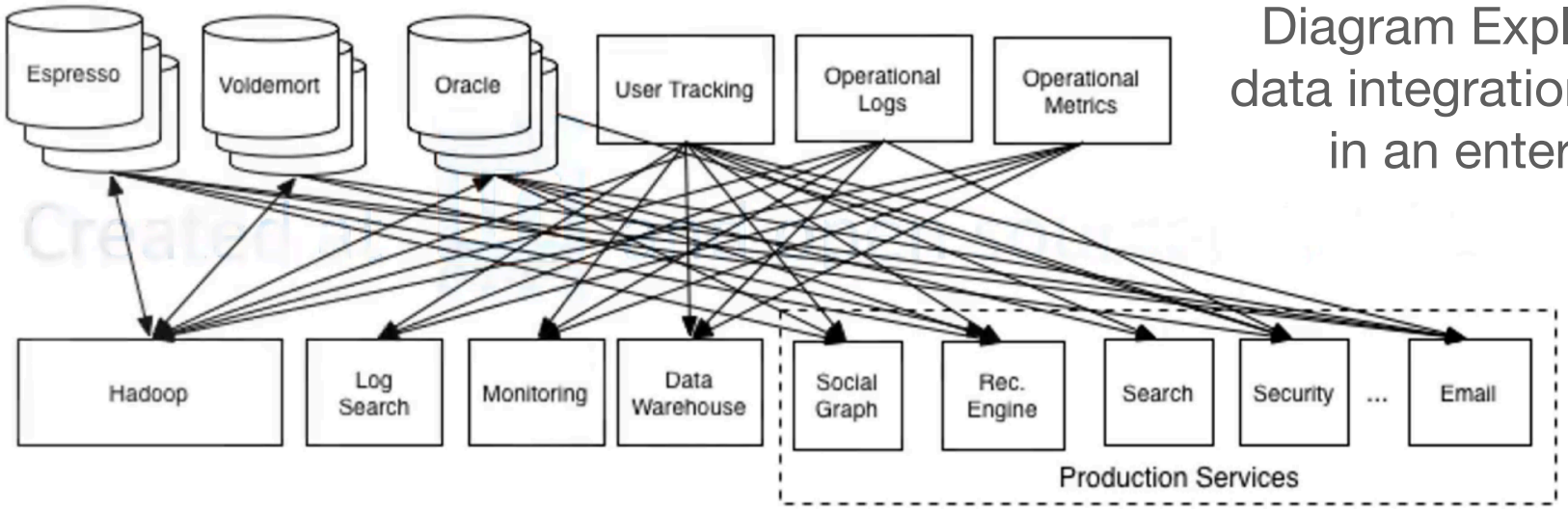
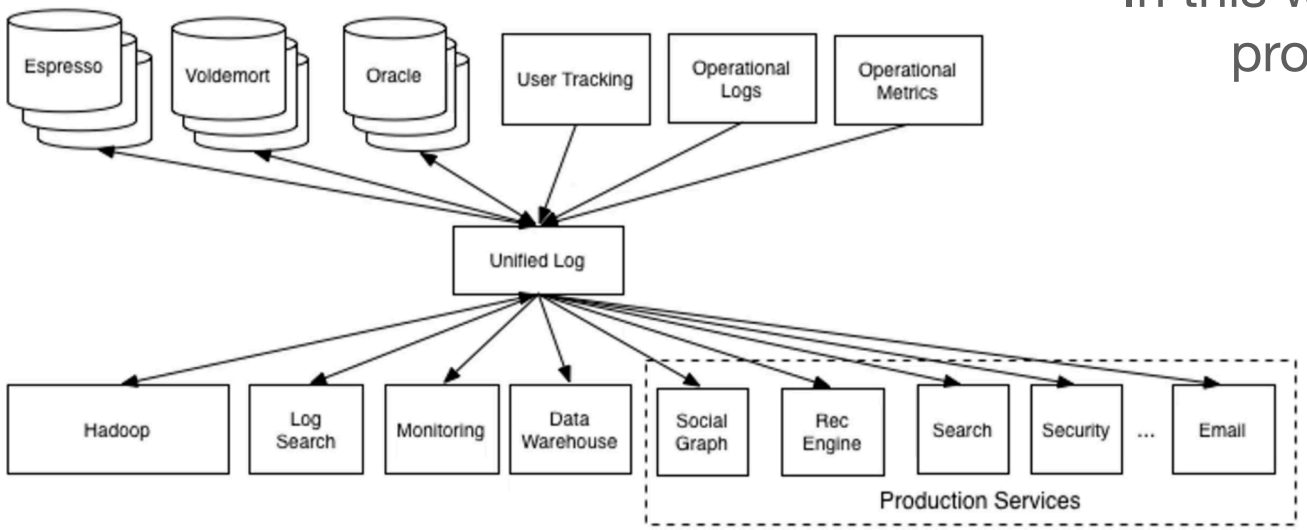
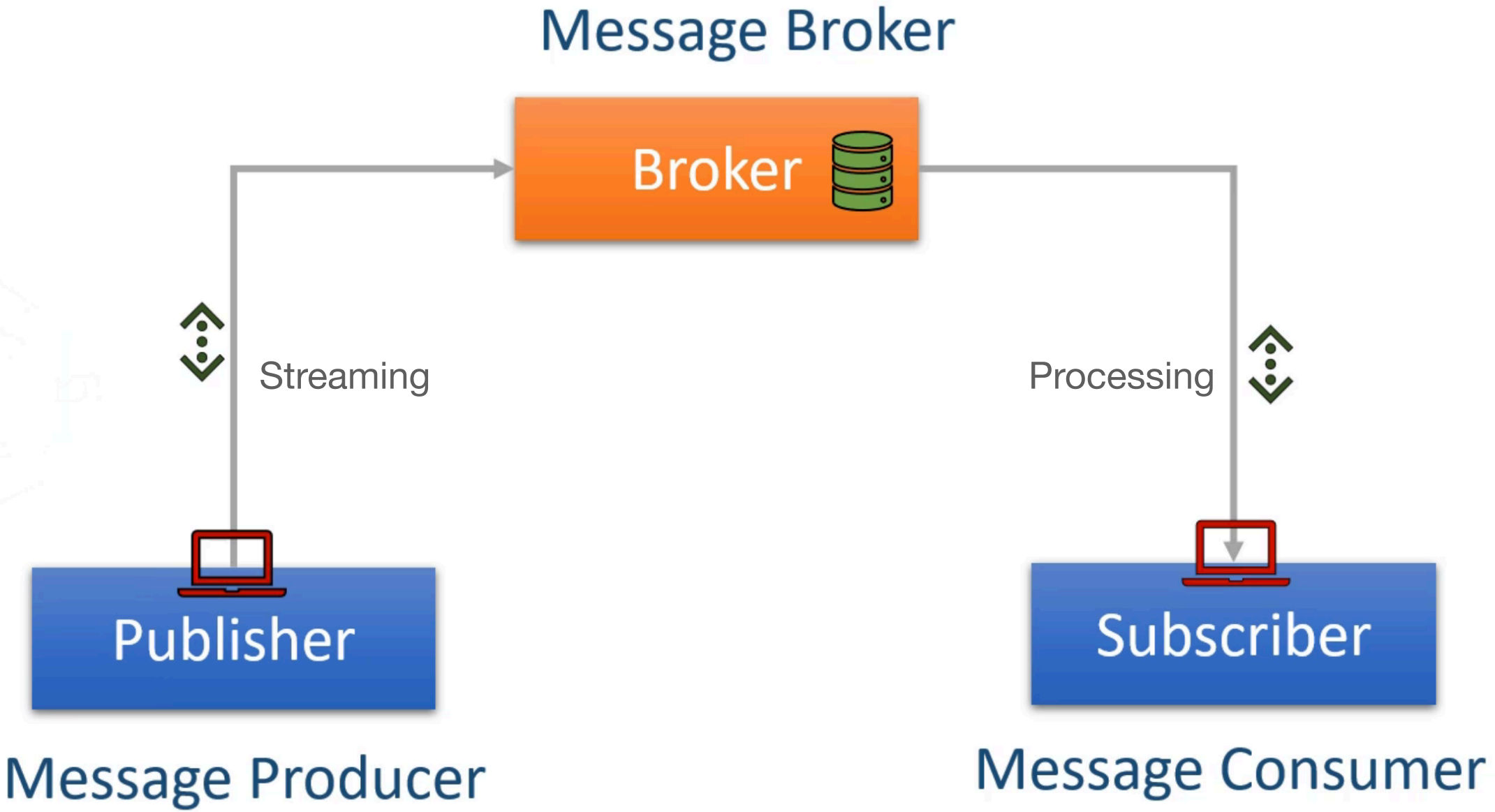


Diagram Explains the data integration problem in an enterprise

Kafka History

Kakfa works as middleware
Data producer and consumers
Connects to a centralised system
In this way data integration
problem resolved.



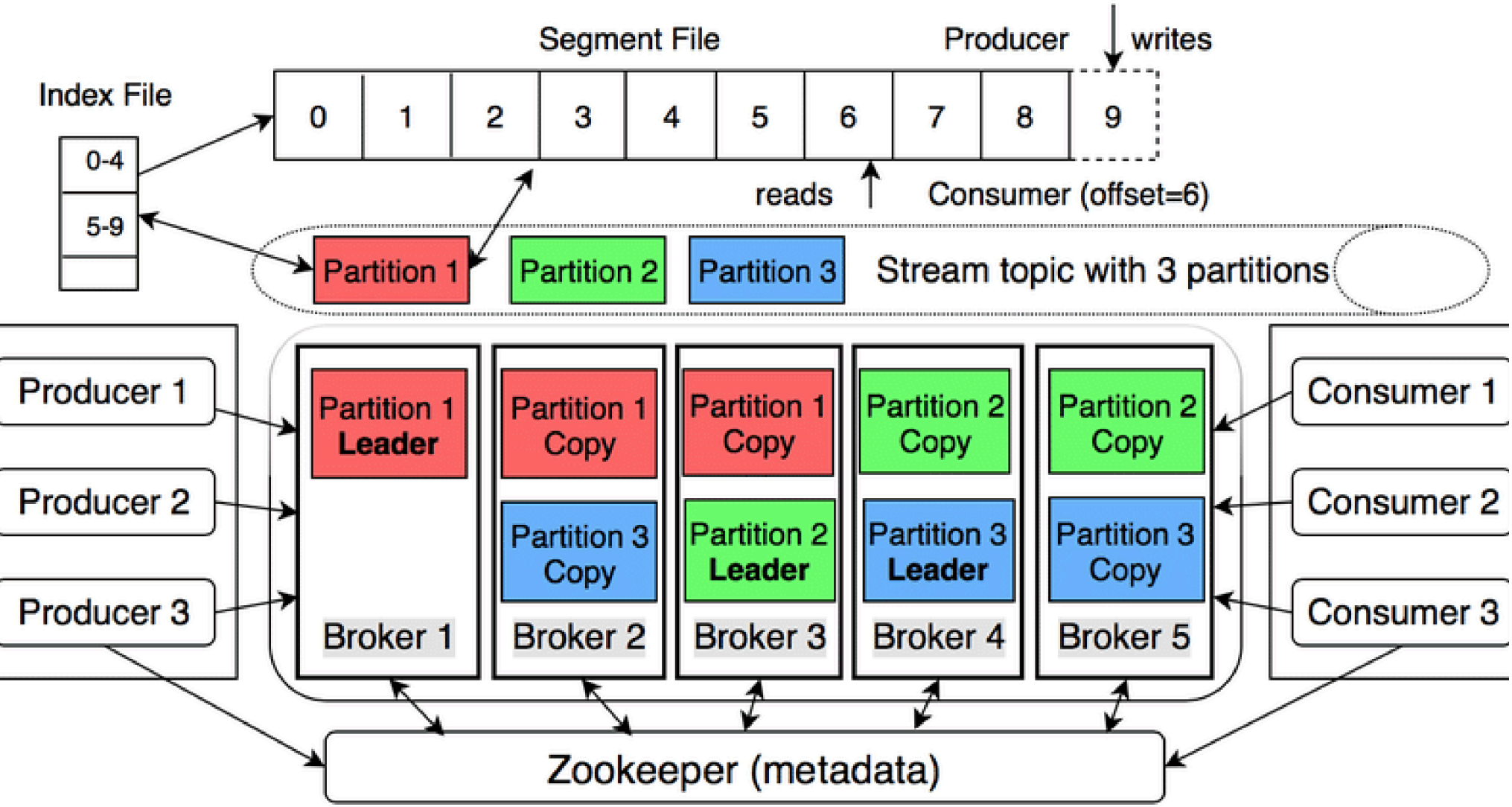

Message Broker
Broker
Streaming
Processing
Publisher
Subscriber
Message Producer
Message Consumer

Kafka Evolution

- Kafka Broker : center server system
- Kafka Clients : Producer and consumer api library
- Kafka Connect : It address the initial data-integration problem.
- Kafka Streams : Another library for creating realtime streaming processing applications.
- KSQL : With this Kafka is aiming to be a realtime database

Kafka Core Concepts

- Producers
- Consumers
- Broker : a server
- Cluster : set of brokers
- Topics : to resolve data segregation problem.
- Partitions : to resolve the storage capacity problems,scalability.
- Offset :
- Consumer Group


Segment File
Producer
writes
Index File
0
1
2
3
4
5
6
7
8
9
0-4
5-9
reads
Consumer (offset=6)
Partition 1
Partition 2
Partition 3
Stream topic with 3 partitions
Producer 1
Producer 2
Producer 3
Partition 1 Leader
Partition 3 Copy
Broker 1
Partition 1 Copy
Partition 3 Copy
Broker 2
Partition 1 Copy
Partition 2 Leader
Broker 3
Partition 2 Copy
Partition 3 Leader
Broker 4
Partition 2 Copy
Partition 3 Copy
Broker 5
Consumer 1
Consumer 2
Consumer 3
Zookeeper (metadata)

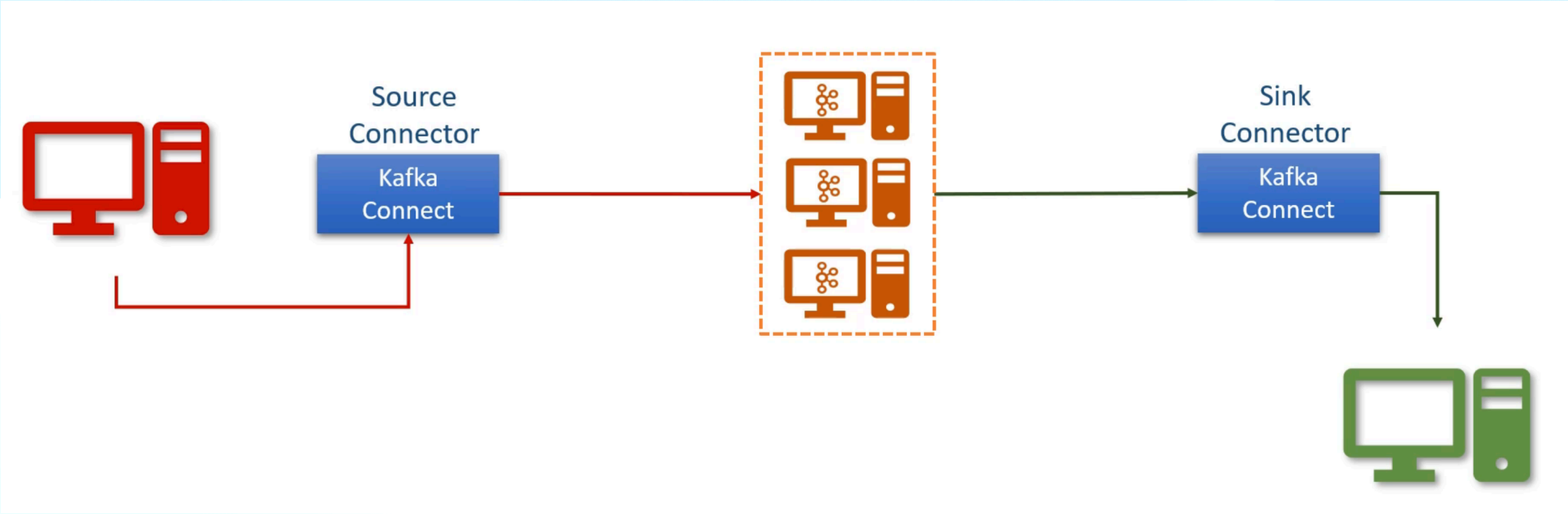
Advanced Kafka Configuration
Important parameters to be aware of

- auto.create.topics.enable=true => set to false in production
- background.threads=10 => increase if you have a good CPU
- delete.topic.enable=false => your choice
- log.flush.interval.messages => don't ever change. Let your OS do it
- log.retention.hours=168 => Set in regards to your requirements
- message.max.bytes=1000012 => increase if you need more than 1MB
- min.insync.replicas=1 => set to 2 if you want to be extra safe
- num.io.threads=8 => ++if your network io is a bottleneck
- num.network.threads=3 => ++if your network is a bottleneck

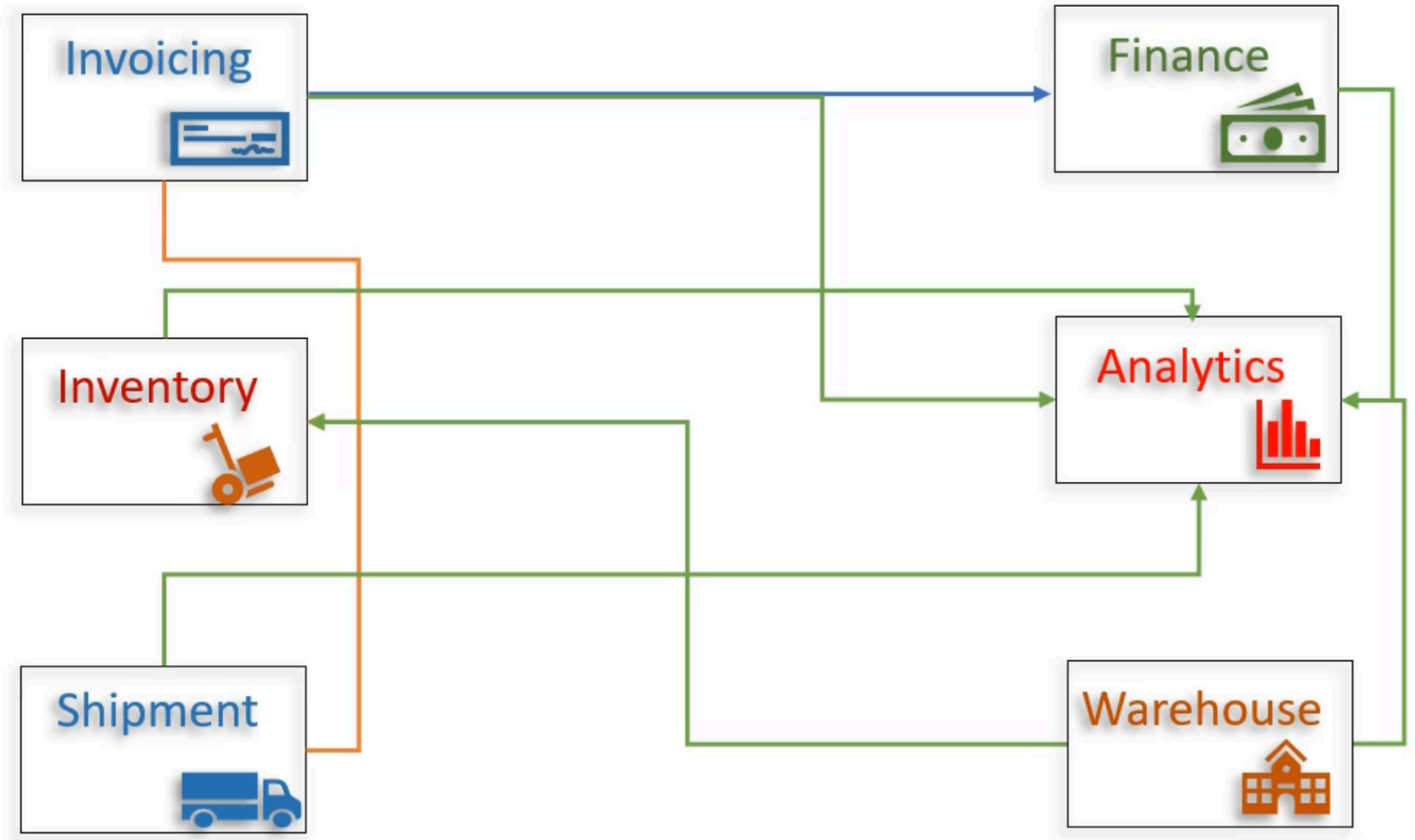
- num.recovery.threads.per.data.dir=1 => set to number of disks
- num.replica.fetchers=1 => increase if your replicas are lagging
- offsets.retention.minutes=1440 => after 24 hours you lose offsets
- unclean.leader.election.enable=true => false if you don't want data loss
- zookeeper.session.timeout.ms=6000 => increase if you timeout often
- broker.rack=null => set your to availability zone in AWS
- default.replication.factor=1 => set to 2 or 3 in a kafka cluster
- num.partitions=1 => set from 3 to 6 in your cluster
- quota.producer.default=10485760 => set quota to 10MBs
- quota.consumer.default=10485760 => set quota to 10MBs

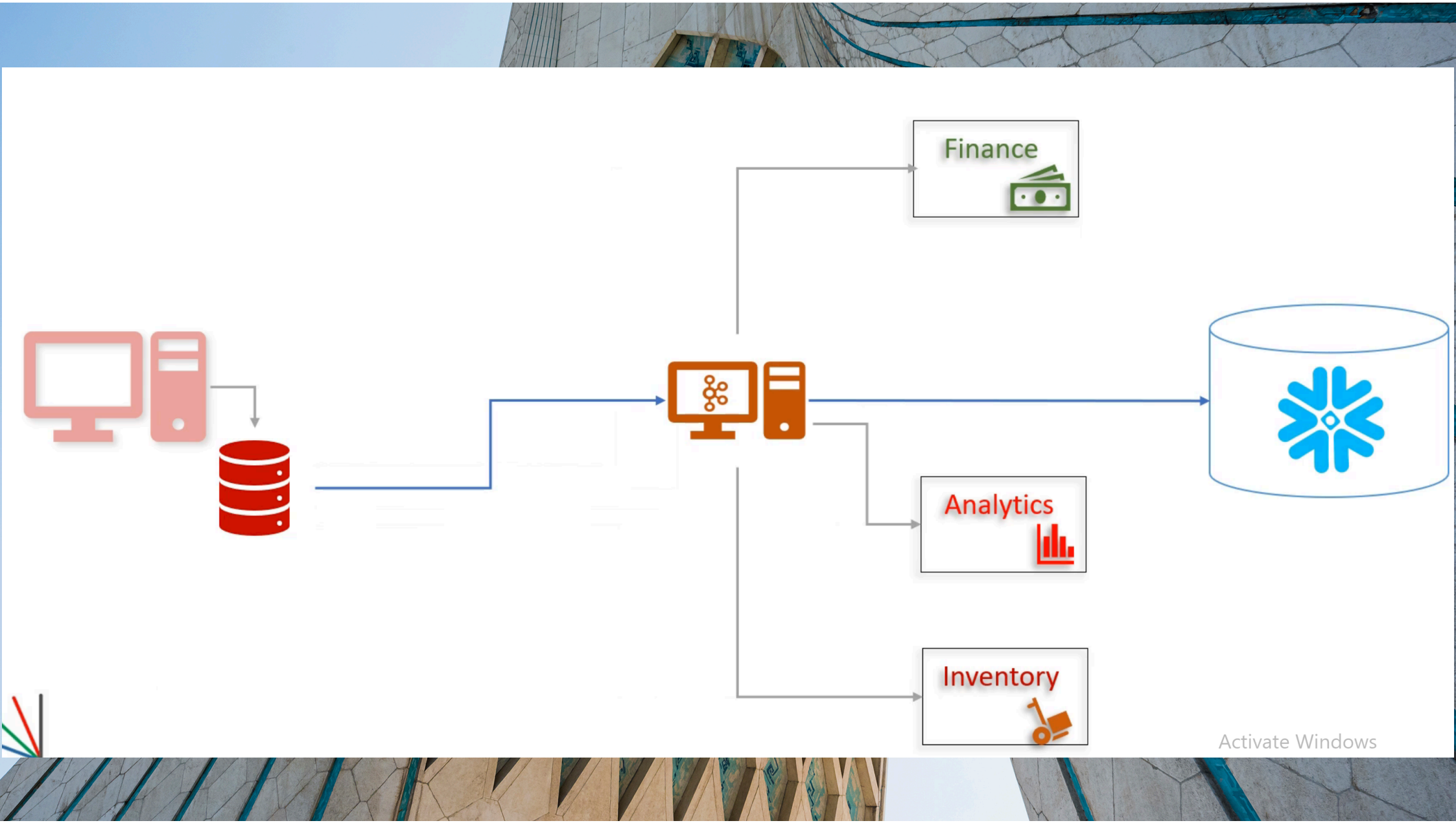
Kafka Connect Core Concepts

- Kafka connect is component of Kafka , for connecting and moving data between Kafka and external systems.

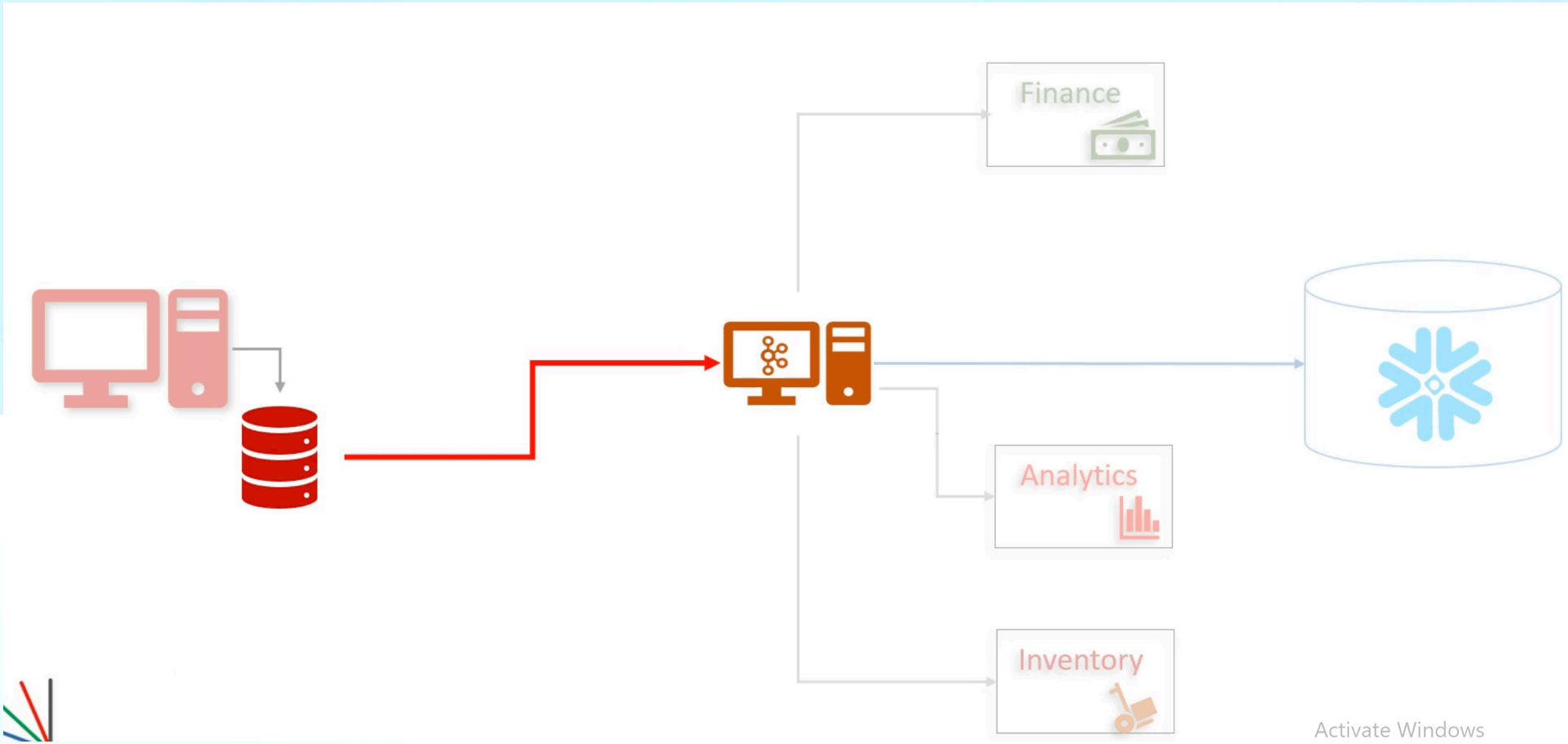


Data Integration Problem





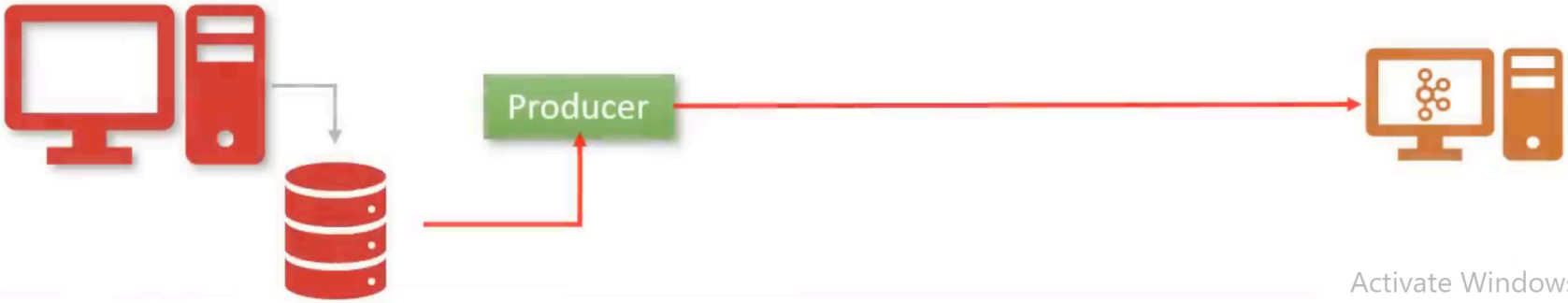
But how we will bring data from service to Kafka cluster.



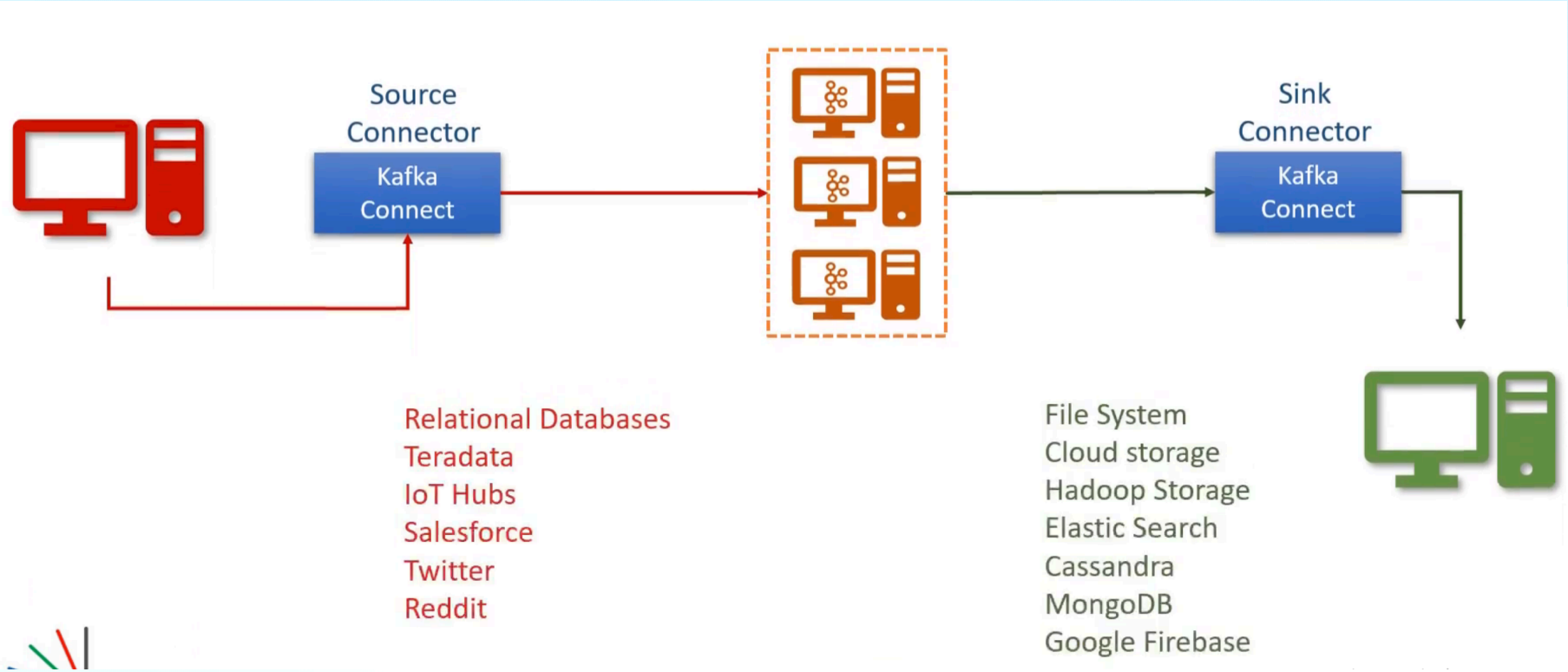
By creating a producer?

If we have Source code we will embedded our producer code in it.

But what if we don't have ??



Then we will be required to create an independent producer. Which will read data from source and send to kafkacluster.



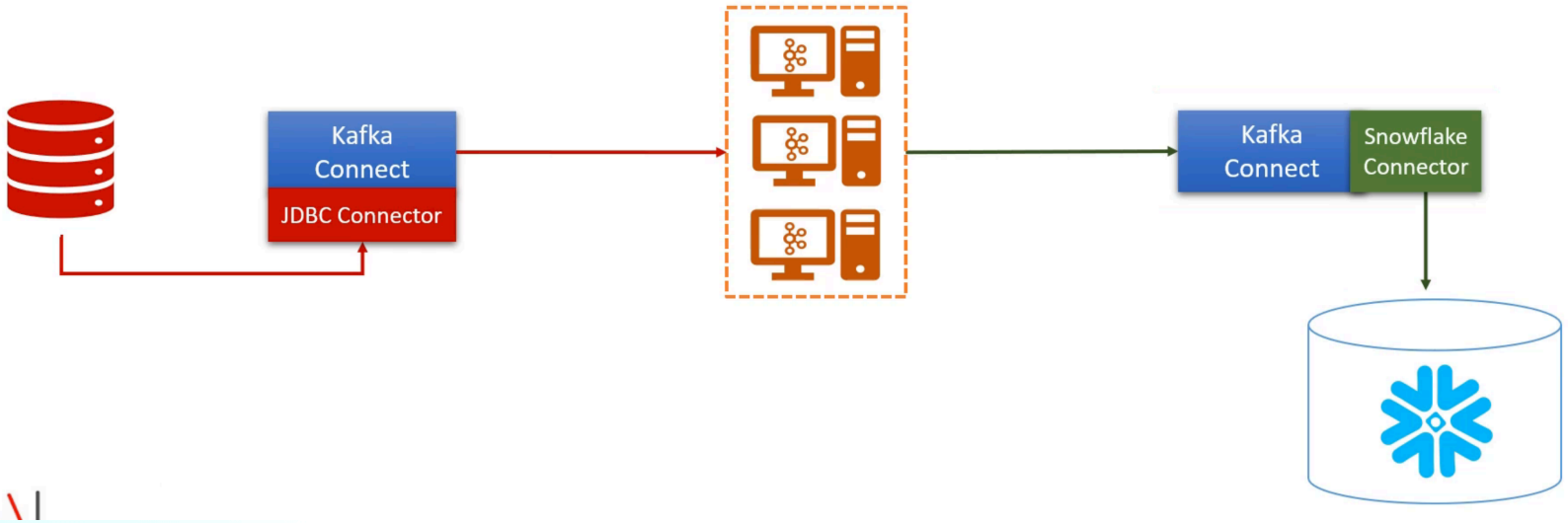
Rather than creating our own independent producer we can use kafka connect and configure it .

We can create our own connector.

Kafka Connect Framework

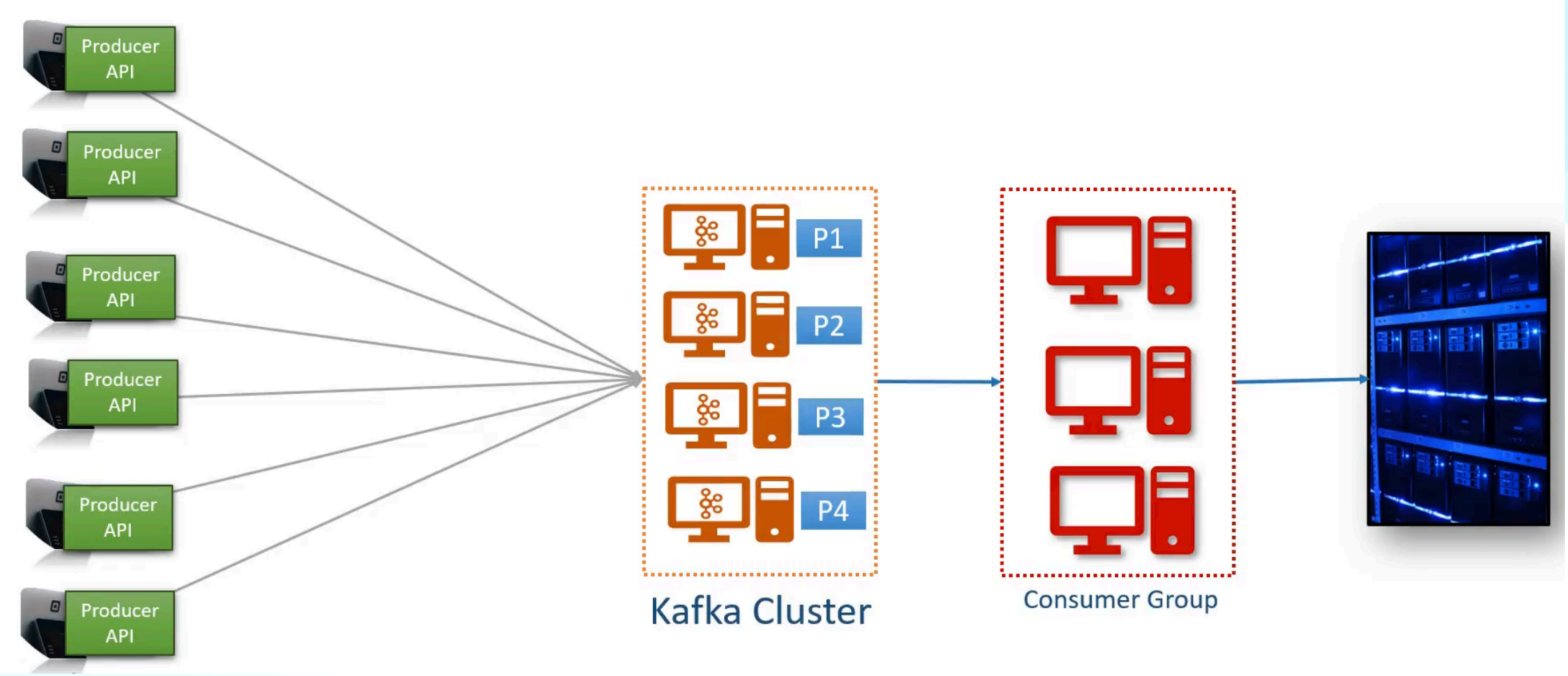
1. Source Connector
   i. SourceConnector
   ii. SourceTask
2. Sink Connector
   i. SinkConnector
   ii. SinkTask

Connect framework takes care of everything I.e.,fault tolerance, error handling ,scalability and all heavy things. We only need to implement the Connector and task classes.



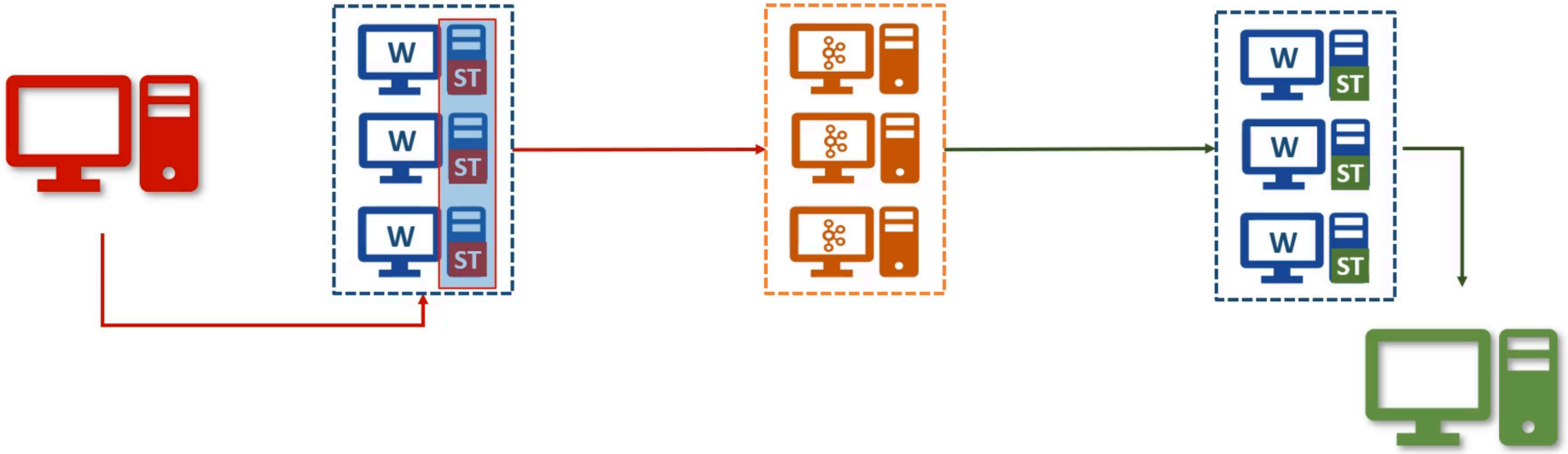
Scalability

We increases producer api, brokers ,consumers etc🤪.



Kafka Connect : Scalability

Kafka connect itself is a cluster.

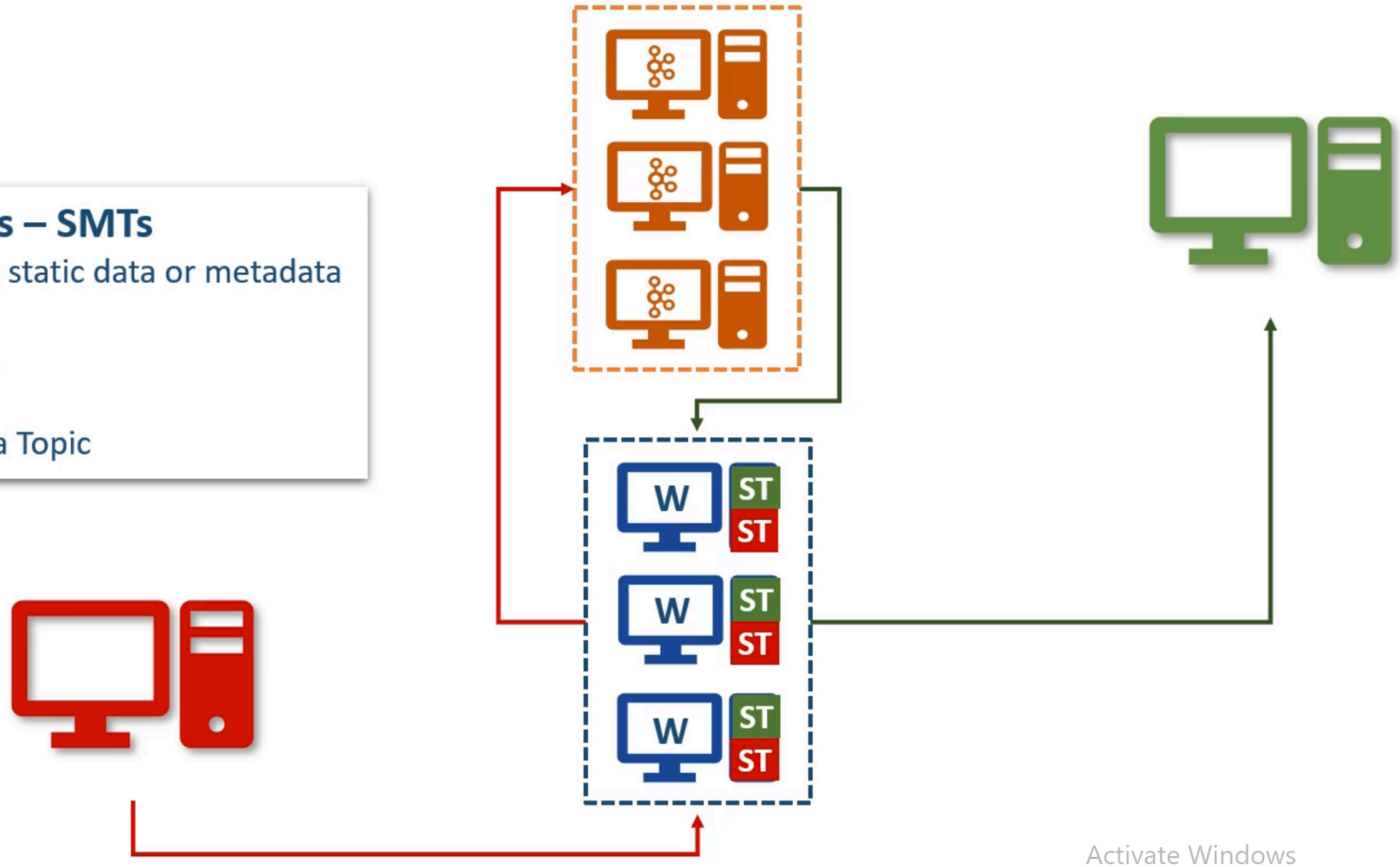


We can add more connectors in a cluster if we have more available space.
If the cluster is fully utilised We can dynamically add workers in kafka connect cluster without stopping any existing connectors.

Kafka Connect – Transformations?

Single Message Transformations – SMTs

1. Add a new field in your record using static data or metadata
2. Filter or Rename Fields
3. Mask some fields with a Null Value.
4. Change the Record Key
5. Route the record to a different Kafka Topic

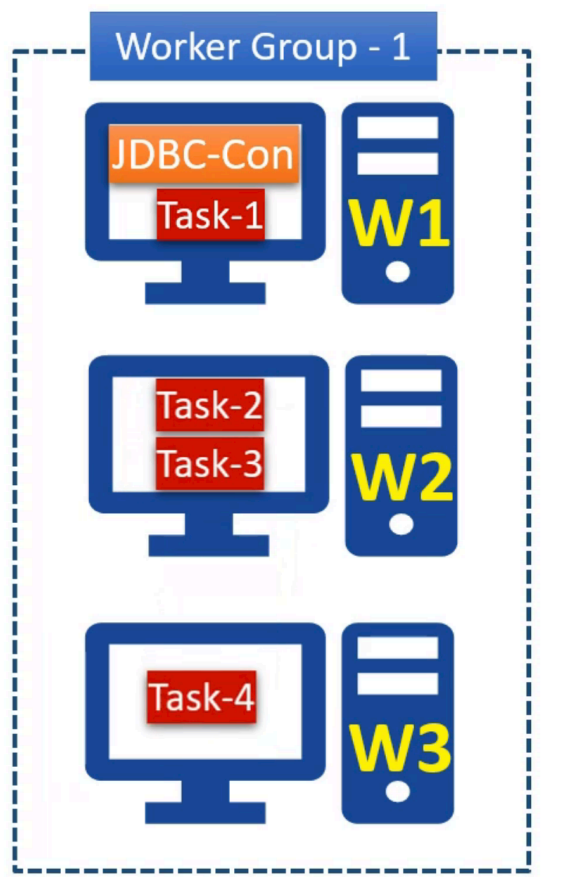


Kafka Connect Architecture

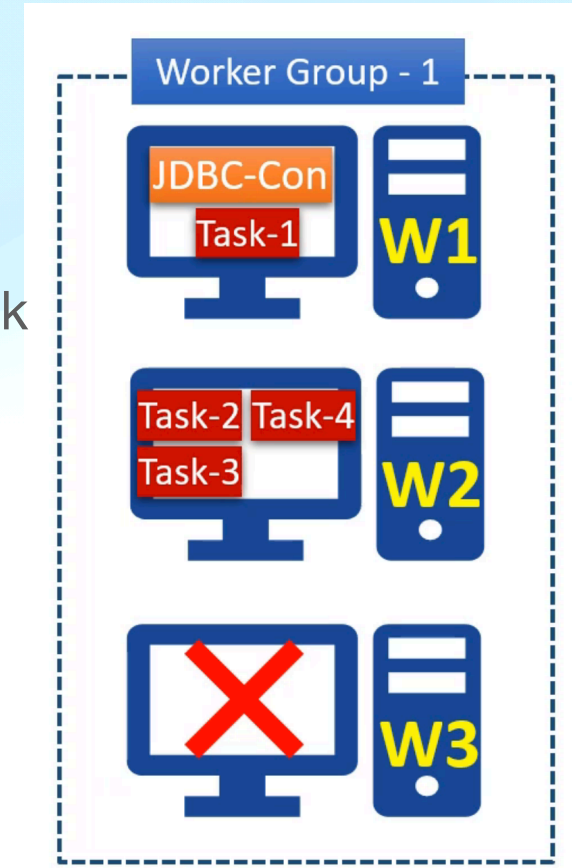
It has 3 components : workers, connectors and tasks.

- Workers are responsible for running connector and tasks.
- Worker group is same as consumer group.
- Worker are fault tolerant and self managed.
- Worker gives us reliability, high availability, scalability and load balancing

Kafka Connect Architecture



If the workers process stops
or crashes, other worker
take care of the task
and as the worker comes back
it reassign some task to it.



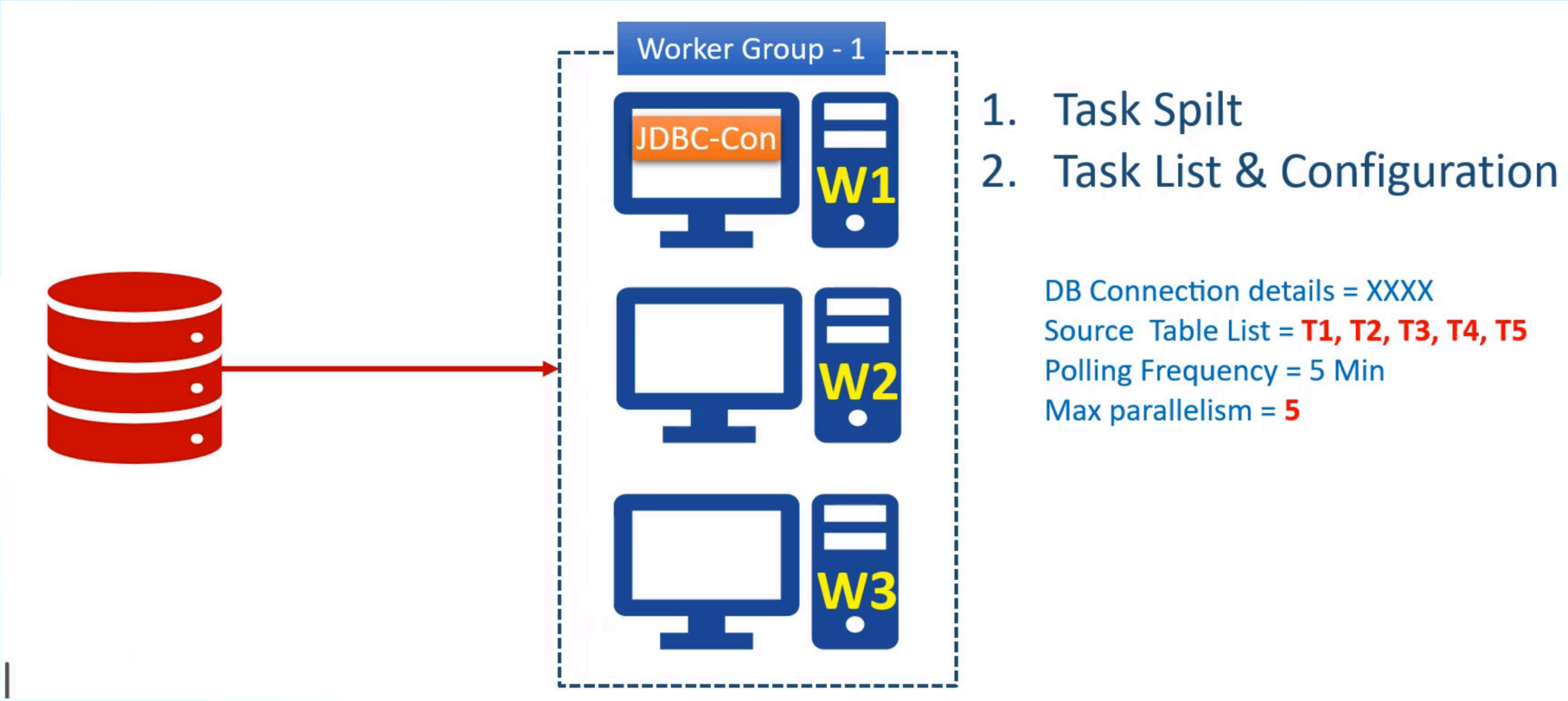
Kafka Connect Architecture

Process

- Install the connector in connect cluster.
- Configure the task using connector.Connectors make sure that each task run as independent by providing it required configs.
- After creating task list and configuring it , connectors give task list to workers.Worker distributes tasks among themselves for load balancing.
- Task are responsible for connecting source system, polling data at regular interval, collecting the records and handling over it to the workers.
- Task is only responsible for interacting with external systems , they do not send record to kafka cluster.
- Worker sends and retrieves data from cluster, task interact with worker and external system, not with kafka cluster.

Kafka Connect Architecture

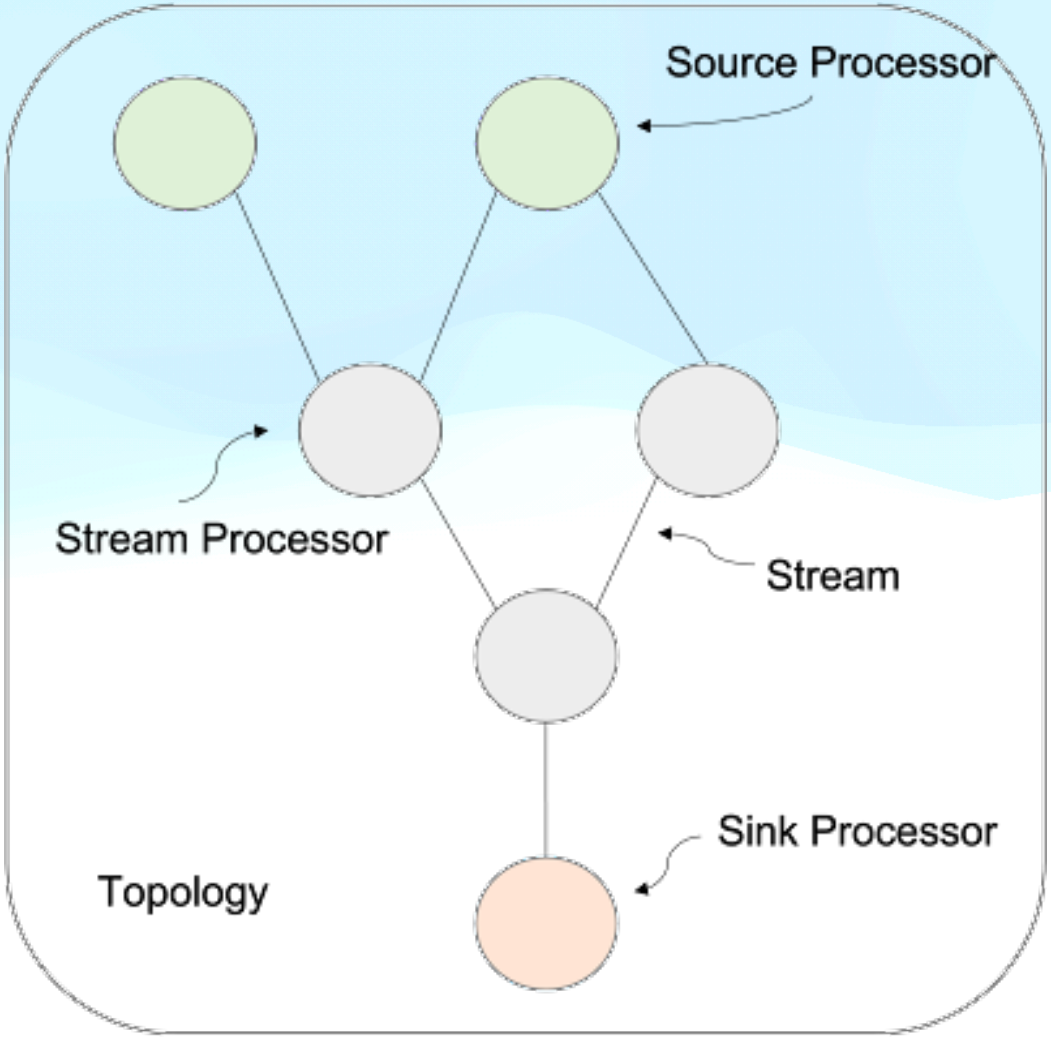
It has 3 components : workers, connectors and tasks.



Connector Developer should know

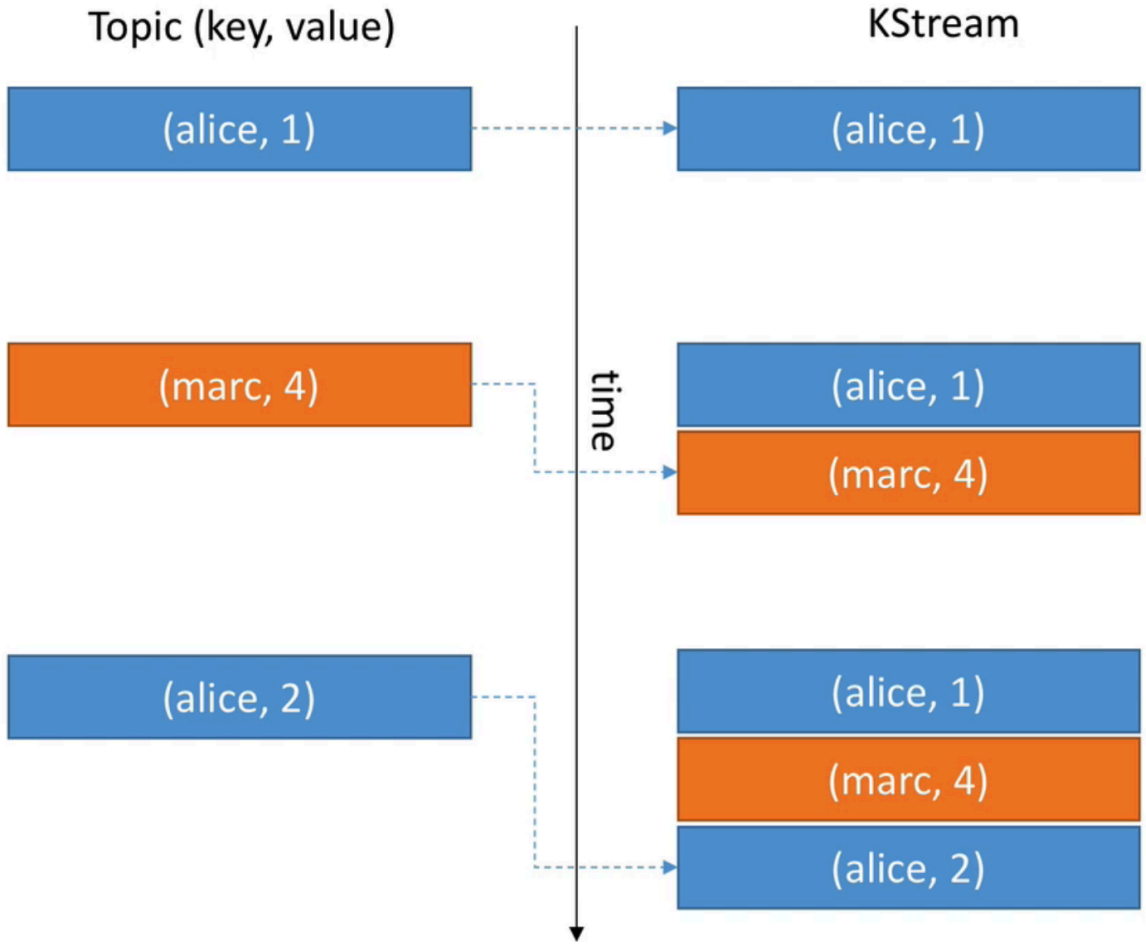
- Reading and writing data to kafka cluster is standard activity , so it is taken care by connect framework(worker).
- Developer needs to take care for the things which are changing for different systems:
  1. How to split the input for parallel processing taken care by connector class.
  2. How to interact with external systems taken care by task class.

Kafka Streams



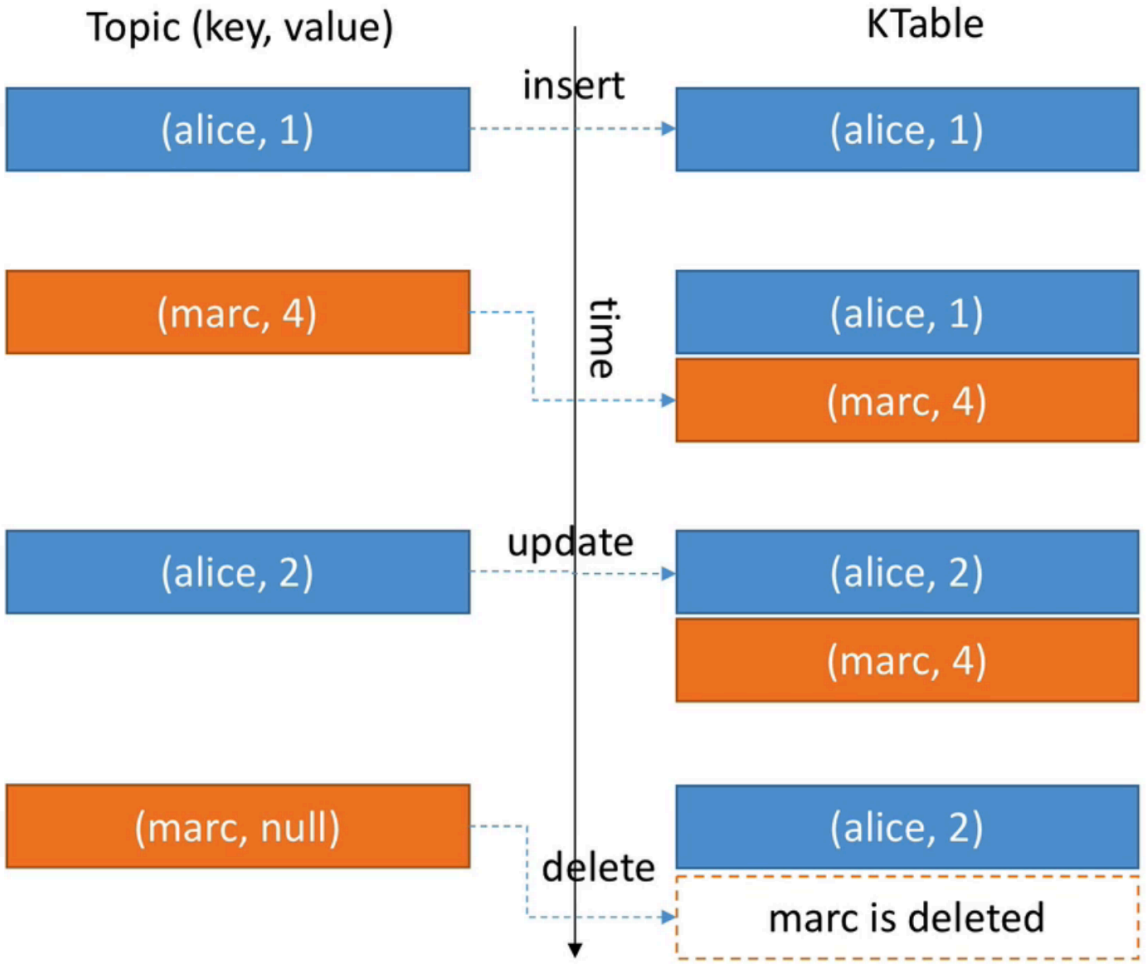
KStreams

- All **inserts**
- Similar to a log
- Infinite
- Unbounded data streams



KTables

- All **upserts** on non null values
- Deletes on null values
- Similar to a table
- Parallel with log compacted topics



When to use KStream vs KTable ?

- KStream reading from a topic that's not compacted
- KTable reading from a topic that's log-compacted (aggregations)

- KStream if new data is partial information / transactional
- KTable more if you need a structure that's like a "database table", where every update is self sufficient
  (think – total bank balance)

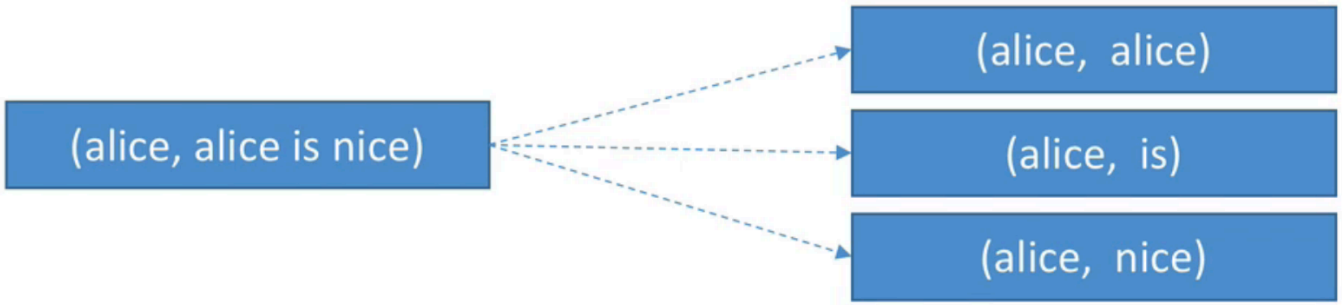
Stateless vs Stateful Operations

- Stateless means that the result of a transformation only depends on the data-point you process
  - Example: a "multiply value by 2" operation is stateless because it doesn't need memory of the past to be achieved.
  - `1 => 2`
  - `300 => 600`
- Stateful means that the result of a transformation also depends on an external information – the state
  - Example: a count operation is stateful because your app needs to know what happened since it started running in order to know the computation result
  - `hello => 1`
  - `hello => 2`

FlatMapValues / FlatMap

- Takes one record and produces zero, one or more record
- FlatMapValues
  - does not change keys
  - == does not trigger a repartition
  - For KStreams only
- FlatMap
  - Changes keys
  - == triggers a repartitions
  - For KStreams only

```
// Split a sentence into words.
words = sentences.flatMapValues(value -> Arrays.asList(value.split("\\s+")));
```



MapValues / Map

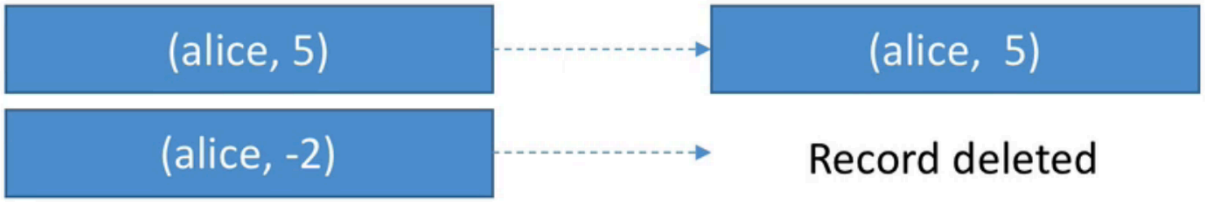
- Takes one record and produces one record
- MapValues
  - Is only affecting values
  - == does not change keys
  - == does not trigger a repartition
  - For KStreams and KTables
- Map
  - Affects both keys and values
  - Triggers a re-partitions
  - For KStreams only

```
// Java 8+ example, using lambda expressions KStream<byte[], String>
uppercased = stream.mapValues(value -> value.toUpperCase());
```

Filter / FilterNot

- Takes one record and produces zero or one record
- Filter
  - does not change keys / values
  - == does not trigger a repartition
  - For KStreams and KTables
- FilterNot
  - Inverse of Filter

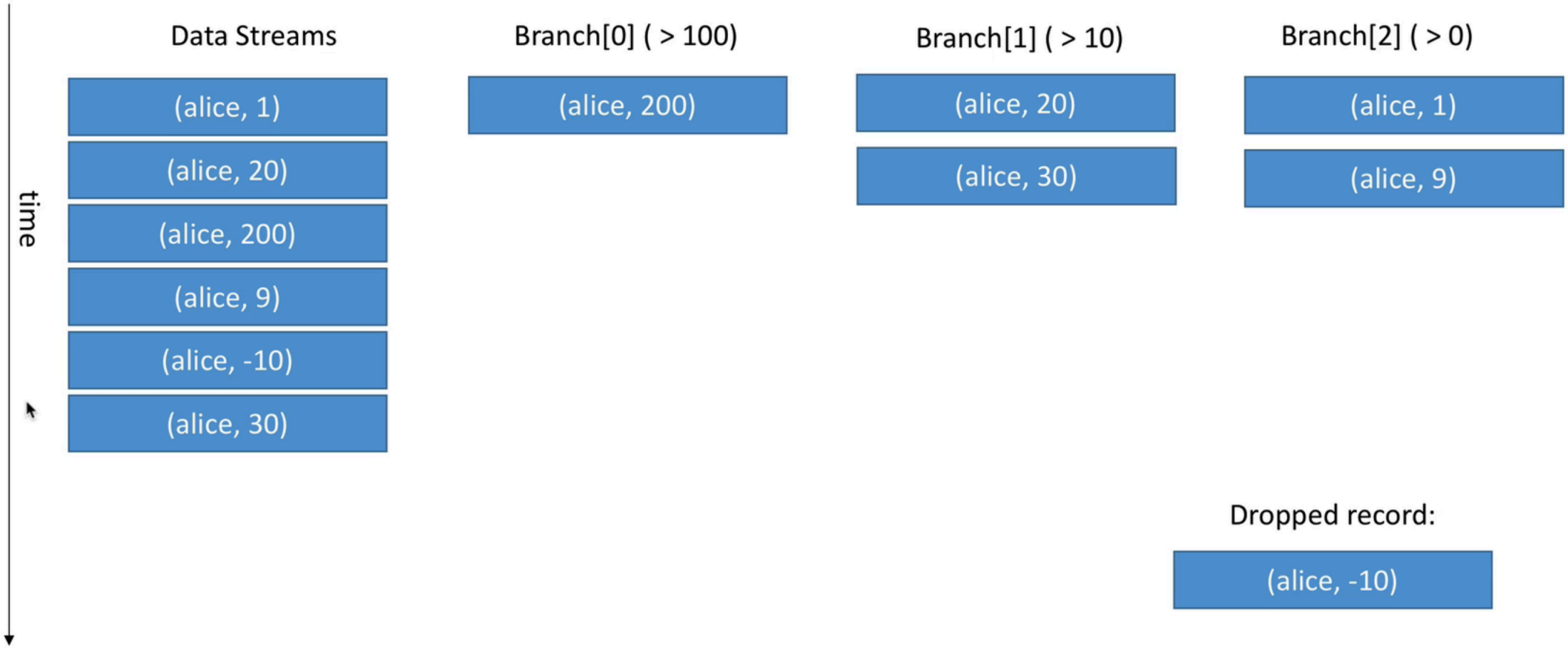
```
// A filter that selects (keeps) only positive numbers
KStream<String, Long> onlyPositives = stream.filter((key, value) -> value > 0);
```



KStream Branch

- Branch (split) a KStream based on one or more predicates
- Predicates are evaluated in order, if no matches, records are dropped
- You get multiple KStreams as a result

```
KStream<String, Long>[] branches = stream.branch(
  (key, value) -> value > 100, /* first predicate */
  (key, value) -> value > 10, /* second predicate */
  (key, value) -> value > 0/* third predicate */
);
```


Data Streams
Branch[0] (> 100)
Branch[1] (> 10)
Branch[2] (> 0)
(alice, 1)
(alice, 20)
(alice, 200)
(alice, 9)
(alice, -10)
(alice, 30)
(alice, 200)
(alice, 20)
(alice, 30)
(alice, 1)
(alice, 9)
time
Dropped record:
(alice, -10)

KStream SelectKey

- Assigns a new Key to the record (from old key and value)
- == marks the data for re-partitioning
- Best practice to isolate that transformation to know exactly where the partitioning happens.

```
// Use the first letter of the key as the new key
rekeyed = stream.selectKey((key, value) -> key.substring(0, 1))
```

Streams marked for re-partition

- As soon as an operation can possibly change the key, the stream will be marked for repartition:
  - Map
  - FlatMap
  - SelectKey
- So only use these APIs if you need to change the key, otherwise use their counterparts:
  - MapValues
  - FlatMapValues
- Repartitioning is done seamlessly behind the scenes but will incur a performance cost (read and write to Kafka)

KTable GroupBy

- GroupBy allows you to perform more aggregations within a KTable
- We have used it in the previous section during our Favourite Colour Example!
- It triggers a repartition because the key changes.

```
// Group the table by a new key and key type
KGroupedTable<String, Integer> groupedTable = table.groupBy(
        (key, value) -> KeyValue.pair(value, value.length()),
        Serdes.String(), /* key (note: type was modified) */
        Serdes.Integer() /* value (note: type was modified) */ );
```

KGroupedStream / KGroupedTable Count

- As a reminder, KGroupedStream are obtained after a `groupBy/groupByKey()` call on a KStream

- `Count` counts the number of record by grouped key.

- If used on KGroupedStream:
  - Null keys or values are ignored
- If used on KGroupedTable:
  - Null keys are ignored
  - Null values are treated as “delete” (= tombstones)

KGroupedStream Aggregate

- You need an initializer (of any type), an adder, a Serde and a State Store name (name of your aggregation)

- Example: Count total string length by key

```
// Aggregating a KGroupedStream (note how the value type changes from String to
Long)
KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
        () -> 0L, /* initializer */
        (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */
        Serdes.Long(), /* serde for aggregate value */
        "aggregated-stream-store" /* state store name */);
```

KGroupedTable Aggregate

- You need an initializer (of any type), an adder, a substractor, a Serde and a State Store name (name of your aggregation)

- Example: Count total string length by key

```
// Aggregating a KGroupedStream (note how the value type changes from String to Long)
KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
        () -> 0L, /* initializer */
        (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */
        (aggKey, oldValue, aggValue) -> aggValue - oldValue.length(), /* subtractor */
        Serdes.Long(), /* serde for aggregate value */
        "aggregated-table-store" /* state store name */);
```

KGroupedStream / KGroupedTable Reduce

- Similar to Aggregate, but the result type has to be the same as an input:

- (Int, Int) => Int (example: a * b)
- (String, String) => String (example concat(a, b))

```
// Reducing a KGroupedStream
KTable<String, Long> aggregatedStream = groupedStream.reduce(
        (aggValue, newValue) -> aggValue + newValue, /* adder */
        "reduced-stream-store" /* state store name */);

// Reducing a KGroupedTable
KTable<String, Long> aggregatedTable = groupedTable.reduce(
        (aggValue, newValue) -> aggValue + newValue, /* adder */
        (aggValue, oldValue) -> aggValue - oldValue, /* subtractor */
        "reduced-table-store" /* state store name */);
```

Reading from Kafka

- You can read a topic as a KStream, a KTable or a GlobalKTable

```
KStream<String, Long> wordCounts = builder.stream(
        Serdes.String(), /* key serde */
        Serdes.Long(), /* value serde */
        "word-counts-input-topic" /* input topic */);
```

```
KTable<String, Long> wordCounts = builder.table(
        Serdes.String(), /* key serde */
        Serdes.Long(), /* value serde */
        "word-counts-input-topic" /* input topic */);
```

```
GlobalKTable<String, Long> wordCounts = builder.globalTable(
        Serdes.String(), /* key serde */
        Serdes.Long(), /* value serde */
        "word-counts-input-topic" /* input topic */);
```

Writing to Kafka

- You can write any KStream or KTable back to Kafka
- If you write a KTable back to Kafka, think about creating a log compacted topic!

- To: Terminal operation – write the records to a topic

```
stream.to("my-stream-output-topic");
table.to("my-table-output-topic");
```

- Through: write to a topic and get a stream / table from the topic

```
KStream<String, Long> newStream = stream.through("user-clicks-topic");
KTable<String, Long> newTable = table.through("my-table-output-topic");
```

KStream Peek

- Peek allows you to apply a side-effect operation to a KStream and get the same KStream as a result.

- A side effect could be:
  - printing the stream to the console
  - Statistics collection

- Warning: It could be executed multiple times as it is side effect (in case of failures)

```
KStream<byte[], String> stream = ...;

// Java 8+ example, using lambda expressions
KStream<byte[], String> unmodifiedStream = stream.peek(
(key, value) -> System.out.println("key=" + key + ", value=" + value));
```

Transforming a KStream to a KTable

- Two ways:
  - Chain a groupByKey() and an aggregation step (count, aggregate, reduce)

```
KTable<String, Long> table = usersAndColours.groupByKey()
                                            .count();
```

  - Write back to Kafka and read as KTable

```
// write to Kafka
stream.to("intermediary-topic");

// read from Kafka as a table
KTable<String, String> table = builder.table("intermediary-topic");
```

Transforming a KTable to a KStream

- It is sometimes helpful to transform a KTable to a Kstream in order to keep a changelog of all the changes to the Ktable (see last lecture on Kstream / Ktable duality)

- This can be easily achieved in one line of code!

```
KTable<byte[], String> table = ...;

// Also, a variant of `toStream` exists that allows you
// to select a new key for the resulting stream.
KStream<byte[], String> stream = table.toStream();
```

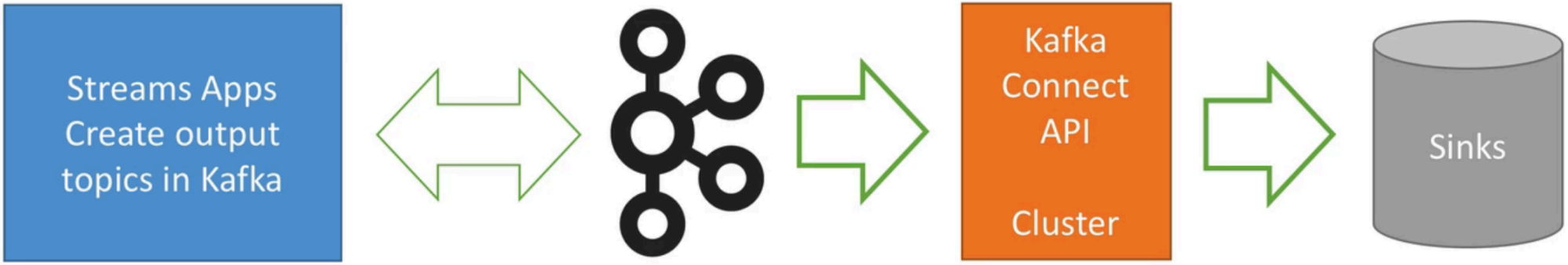
KStream & KTable Duality (from confluent docs)

- **Stream as Table:** A stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table.

- **Table as Stream:** A table can be considered a snapshot, at a point in time, of the latest value for each key in a stream (a stream's data records are key-value pairs).

What if I want to write the result to an external System?

- Although it is theoretically doable to do it using Kafka Streams library, it is NOT recommended
- The recommend way of doing so is Kafka Streams to transform the data and then using **Kafka Connect API** (see my other course)



Refresher on Log Compaction

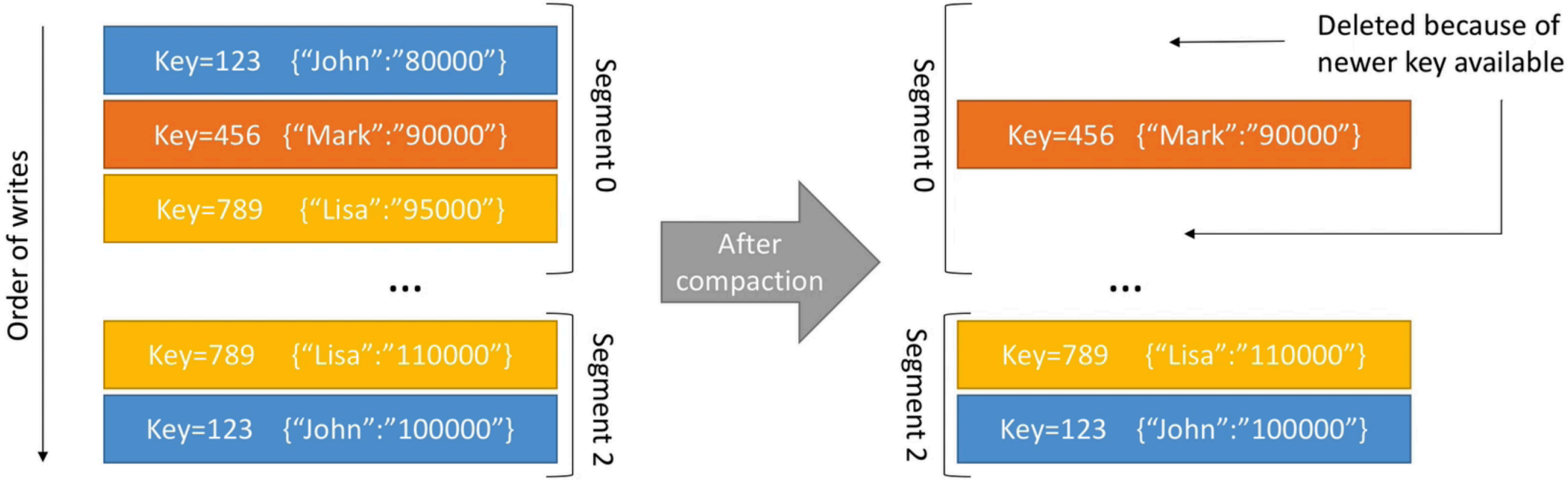
- Log Compaction can be a huge improvement in performance when dealing with KTables because eventually records get discarded
- This means less reads to get to the final state (less time to recover)
- Log Compaction has to be enabled by you on the topics that get created (source or sink topics)

Log Cleanup Policy: Compact

- Log compaction ensures that your log contains at least the last known value for a specific key within a partition
- Very useful if we just require a SNAPSHOT instead of full history (such as for a data table in a database)
- The idea is that we only keep the latest "update" for a key in our log

Log Compaction: Example

- Our topic is: employee-salary
- We want to keep the most recent salary for our employees



Log Compaction Guarantees

- **Any consumer that is reading from the head of a log will still see all the messages sent to the topic**
- Ordering of messages it kept, log compaction only removes some messages, but does not re-order them
- The offset of a message is immutable (it never changes). Offsets are just skipped if a message is missing
- Deleted records can still be seen by consumers for a period of delete.retention.ms (default is 24 hours).

Log Compaction Myth Busting

- It doesn't prevent you from pushing duplicate data to Kafka
  - De-duplication is done after a segment is committed
  - Your consumers will still read from head as soon as the data arrives
- It doesn't prevent you from reading duplicate data from Kafka
  - Same points as above
- Log Compaction can fail from time to time
  - It is an optimization and it the compaction thread might crash
  - Make sure you assign enough memory to it and that it gets triggered

Kafka

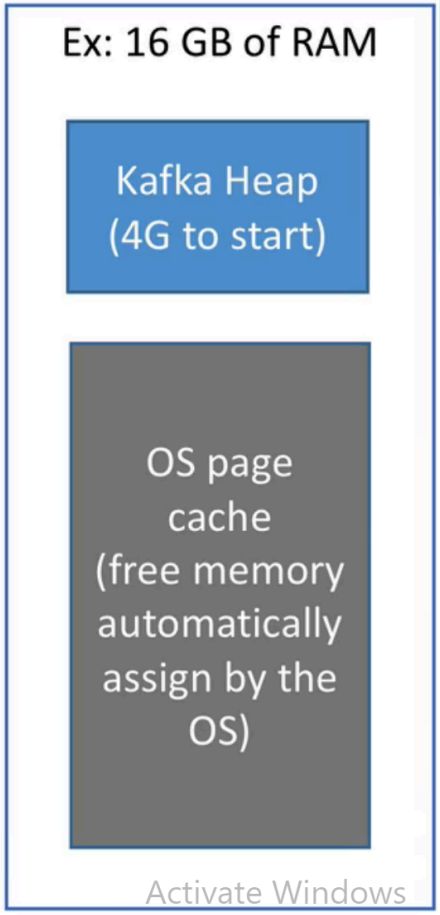
Administration

Running Kafka on AWS in Production

- Separate your instances between different availability zones
- Use st1 EBS volumes for the best price / performance ratio
- Mount multiple EBS volumes to the same broker if you need to scale
- Use *r4.xlarge* or *m4.2xlarge* if you're using EBS (these instances are EBS optimized). You can use something smaller but performance may degrade
- Setup DNS names for your brokers / fixed IPs so that your clients aren't affected if you recycle your EC2 instances

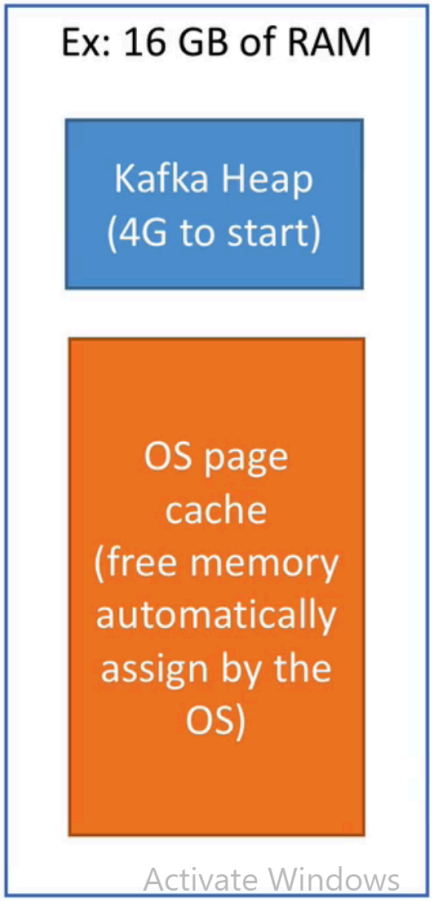
Factors impacting Kafka performance
RAM – Java Heap

- When you launch Kafka, you specify Kafka Heap Options (KAFKA_HEAP_OPTS environment variable)
- I recommend to assign a MAX amount (-Xmx) of 4GB to get started to the kafka heap:
- `export KAFKA_HEAP_OPTS="-Xmx4g"`
- Don't set –Xms (starting heap size):
  - Let heap grow over time
  - Monitor the heap over time to see if you need to increase Xmx
- Kafka should keep a low heap usage over time, and heap should increase only if you have more partitions in your broker



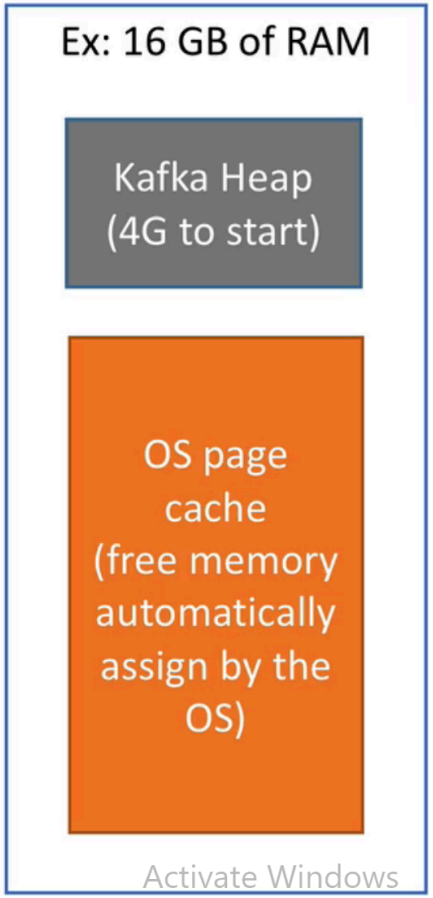
Factors impacting Kafka performance RAM

- Kafka has amazing performance thanks to the page cache which utilizes your RAM
- Understanding RAM in Kafka means understanding two parts:
  - The Java HEAP for the Kafka process
  - The rest of the RAM used by the OS page cache
- Let's understand how both of those should be sized
- Overall, your Kafka production machines should have at least 8GB of RAM to them (the more the better – it's common to have 16GB or 32GB per broker)



Factors impacting Kafka performance
RAM – OS Page Cache

- The remaining RAM will be used automatically for the Linux **OS Page Cache**.
- This is used to buffer data to the disk and this is what gives Kafka an amazing performance
- You don't have to specify anything!
- Any un-used memory will automatically be leveraged by the Linux Operating System and assign memory to the page cache
- Note: Make sure swapping is disabled for Kafka entirely **vm.swappiness=0 or vm.swappiness=1 (default is 60 on Linux)**



Factors impacting Kafka performance CPU

- CPU is usually not a performance bottle neck in Kafka because Kafka does not parse any messages , but can become one in some situations
- If you have SSL enabled, Kafka has to encrypt and decrypt every payload, which adds load on the CPU
- Compression can be CPU bound if you force Kafka to do it. Instead, if you send compressed data, make sure your producer and consumers are the ones doing the compression work (that's the default setting anyway)
- Make sure you monitor Garbage Collection over time to ensure the pauses are not too long

Factors impacting Kafka performance
Operating System (OS)

- Use Linux or Solaris, running production Kafka clusters on Windows is not recommended.
- Increase the file descriptor limits (at least 100,000 as a starting point) https://unix.stackexchange.com/a/8949/170887
- Make sure only Kafka is running on your Operating System. Anything else will just slow the machine down.

Factors impacting Kafka performance Network

- Latency is key in Kafka
  - Ensure your Kafka instances are your Zookeeper instances are geographically close!!!
  - Do not put one broker in Europe and the other broker in the US
  - Having two brokers live on the same rack is good for performance, but a big risk if the rack goes down.
- Network bandwidth is key in Kafka
  - Network will be your bottleneck.
  - Make sure you have enough bandwidth to handle many connections, and TCP requests.
  - Make sure your network is high performance
- Monitor network usage to understand when it becomes a bottleneck

Factors impacting Kafka performance Disks: I/O

- Reads are done sequentially (as in not randomly), therefore make sure you should a disk type that corresponds well to the requirements
- Format your drives as XFS (easiest, no tuning required)
- If read / write throughput is your bottleneck
  - it is possible to mount multiple disks in parallel for Kafka
  - The config is log.dirs=/disk1/kafka-logs,/disk2/kafka-logs,/disk3/kafka-logs…
- Kafka performance is constant with regards to the amount of data stored in Kafka.
  - Make sure you expire data fast enough (default is one week)
  - Make sure you monitor disk performance

Factors impacting Kafka performance Other

- Make sure you have enough file handles opened on your servers, as Kafka opens 3 file descriptor for each topic-partition-segment that lives on the Broker.
- Make sure you use Java 8
- You may want to tune the GC implementation: (see in the resources)
- Set Kafka quotas in order to prevent unexpected spikes in usage

Thank You