**https://nagcloudlab.notion.site/Kafka-8cf2e394ef3845d29341371a55259750**

**Introduction**

**What is a Kafka Topic?**

Similar to how databases have tables to organize and segment datasets, Kafka uses the concept of topics to organize related messages.

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2F5mX4ebnraAdPJWgmbLunH%2F6b38d80143a0a47e2bec42381a6947ef%2FApache_Kafka_Cluster_with_4_topics.png&w=3840&q=75>

Kafka topics can contain any kind of message in any format, and the sequence of all these messages is called a data stream.

**Kafka Topics - Warning**

Unlike database tables, Kafka topics are not query-able. Instead, we have to create Kafka producers to send data to the topic and Kafka consumers to read the data from the topic in order.

**What are Kafka Partitions?**

Topics are broken down into a number of partitions. A single topic may have more than one partition, it is common to see topics with 100 partitions.

The number of partitions of a topic is specified at the time of topic creation. Partitions are numbered starting from 0 to N-1, where N is the number of partitions. The figure below shows a topic with three partitions, with messages being appended to the end of each one.

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2F3c2aet87FgzOy28vIeKLOD%2F040c18b3ae4b09e7dc1e545885b74966%2FKafka_Topics_1.png&w=3840&q=75>

*Topic Partitions*

The offset is an integer value that Kafka adds to each message as it is written into a partition. Each message in a given partition has a unique offset.

**Kafka Topics**

Kafka topics are **immutable**: once data is written to a partition, it cannot be changed

**What are Kafka Offsets?**

Apache Kafka offsets represent the position of a message within a Kafka Partition. Offset numbering for every partition starts at 0 and is incremented for each message sent to a specific Kafka partition. This means that Kafka offsets only have a meaning for a specific partition, e.g., offset 3 in partition 0 doesn’t represent the same data as offset 3 in partition 1.

**Kafka Offset Ordering**

If a topic has more than one partition, Kafka guarantees the order of messages within a partition, but messages are not ordered across partitions.

Even though we know that messages in Kafka topics are deleted over time (as seen above), the offsets are not re-used. They are continually incremented in a never-ending sequence.

**Advanced**

**Naming Conventions**

Kafka topic can have any names you want. It is very important to choose a naming convention in your company to maintain some kind of consistency.

If you need ideas, I recommend reading this blog: \*\*[https://cnr.sh/essays/how-paint-bike-shed-kafka-topic-naming-conventions\*\*](https://cnr.sh/essays/how-paint-bike-shed-kafka-topic-naming-conventions**)

Common structures adopted are hierarchical such as <department name>.<team name>.<dataset name>.<data format> (you can define your own).

Suffixing by <data format> can be a good way to indicate in advance how to consume a topic for example .avro , .json, .text, .protobuf, .csv, .log

Valid characters for Kafka topics are the ASCII alphanumerics, ., \_, and - and it is better not to mix . and \_ to avoid metric namespace collisions:

# ****Choosing the Partition Count and Replication Factor****

When creating a topic, we have to provide a partition count and the replication factor. These two are very important to set correctly as they impact the **performance and durability** in the system.

Starting with one set of values and changing them later will have an adverse impact on the system as depicted below.

* If the partitions count increases during a topic lifecycle, you will break your keys ordering guarantees
* If the replication factor increases during a topic lifecycle, you put more pressure on your cluster, which can lead to an unexpected performance decrease due to more network traffic and additional space used on your brokers. Proceed with caution

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2FfKI10LJZGUY0e16UiW88n%2Fa6b33e737a91e2a23cbebf3c0b0f37f3%2FAdv_KT_Choosing_Rep_Factor_1.png&w=3840&q=75>

Adding replicas uses more space and adds more network traffic

**Right First Time**

It is best to get the partition count and replication factor right the first time!

The number of partitions per topic is a million-dollar question and there’s no one answer. In this section, we'll learn a few rules of thumb guidelines that will help us set these parameters correctly.

**How to choose the number of partitions in a Kafka topic**

### **1. Throughput Requirements**

The number of partitions in a topic is a major driver of the topic’s capacity to handle high volumes of data. More partitions mean more parallelism, thus higher throughput. Consider:

* **Producer Throughput**: If you have high-volume producers, increasing the number of partitions allows more producers to write in parallel, increasing overall throughput.
* **Consumer Throughput**: More partitions allow more consumers to read in parallel, which is especially important in a consumer group. Each consumer in a group can read from one or more partitions but a single partition can only be read by one consumer from a group at a time.

### 2. Latency Considerations

While more partitions can increase throughput, they can also lead to higher end-to-end latency due to increased overhead in managing more partitions and possible imbalances in workload across partitions.

### **3. Topic and Data Size**

If your topic is expected to handle a large amount of data, having more partitions can help in managing this data more efficiently. More partitions can also help in faster data compaction and retention operations.

### **4. Future Scaling**

Anticipating future growth is important. While you can increase the number of partitions later, it is a disruptive process and can lead to temporary performance degradation. It's often better to start with a slightly higher number of partitions than currently needed to accommodate future growth.

### **5. Broker and Cluster Capacity**

The number of partitions is also constrained by the capacity of your Kafka brokers and the cluster:

* **Broker Limits**: Each broker can handle a certain number of partitions. While Kafka can handle thousands of partitions per broker, a very high number of partitions can cause issues in terms of memory overhead, file handles, and replication traffic.
* **Cluster Configuration**: The overall size and configuration of your Kafka cluster also play a role. Larger clusters can handle more partitions.

### **6. Replication Factor**

The replication factor multiplies the effect of partitions on the Kafka brokers. For example, a topic with 10 partitions and a replication factor of 3 results in a total of 30 partitions that must be managed across the cluster.

### **7. Practical Rule of Thumb**

A common heuristic used in the industry is:

* Aim for partitions that can handle between 1,000 and 2,000 writes per second.
* Each partition should ideally handle between 10 MB/s and 100 MB/s throughput.

**Example Calculation**

If you expect your system to handle 1 GB of writes per second and aim for each partition to handle 100 MB/s, you might start with approximately 10 partitions.

### **Tools and Metrics**

Utilize Kafka’s built-in monitoring tools and metrics to assess the performance impact of the number of partitions and adjust accordingly based on actual system behavior and performance.

There are several factors to consider when choosing the number of partitions:

* What is the maximum throughput (in MB/s) you expect to achieve when consuming from a single partition? Measure it for your environment.
* If you are sending messages to partitions based on keys, adding partitions later can be very challenging, so calculate throughput based on your expected future usage.
* Having more partitions has advantages. It implies:
  + Better parallelism and better throughput. A partition is a unit of parallelism, so creating more partitions implies more parallelism.
  + It also gives us the ability to run more consumers in a group to scale. For example, if we have 3 partitions, we can have at most 3 consumers active, others will remain inactive.
  + If the cluster contains a high number of brokers, having more partitions will leverage these brokers. For example, if you have a topic with 2 partitions, they can be hosted on two brokers only, and the other brokers will remain idle which is an inefficient usage of resources.
* However, there are downsides to having more partitions (which are slowly disappearing)
  + Each partition will have a partition leader to be elected by Zookeeper. Hence there will be more load on Zookeeper which will increase the time for leader elections.
  + This problem is going to be solved in a Zookeeper-less Kafka, which can you can [**learn about here**](https://www.conduktor.io/kafka/kafka-kraft-mode/) and [**practice while starting Kafka**](https://www.conduktor.io/kafka/starting-kafka/).
  + More files opened by Kafka. There is an OS limit to the number of files that can be opened, although you can and should change it to a high value on your OS settings.

With all this in mind, the following are a few guidelines that will help you choose wisely.

* If you have a **small cluster of fewer than 6 brokers**, create **three times, i.e., 3X,** the number of brokers you have. The reasoning behind it is that if you have more brokers over time, you will have enough partitions to cover that.
* If you have a **big cluster of over 12 brokers**, create **two times i.e.**, **2X,** the number of brokers you have.
* You also want to take into account the **number of consumers** you need to run in a group at the desired peak throughput. If, for example, you need 20 consumers at peak time, you need at least 20 partitions in your topic, regardless of the size of your cluster.
* You also need to consider the **producer throughput**. If you have a high throughput producer or if it is going to increase in the next couple of years, keep the partition count to **3 times the number of brokers**.

It is good to have a high number of partitions per topic, but do not pick an absurdly high number like 1000 unless you know you will need all these partitions.

topic-1 : 10

topic-2 : 20

topic3 : 3

brokers : 3

11 partitions / brokers

throughput

latency

availability

durability

**How to choose the replication factor of a Kafka topic?**

To increase the reliability and fault tolerance, replications of the partitions are necessary. Remember that topic replication does not increase consumer parallelism. The factors to consider while choosing a replication factor are:

* It should be at least 2 and a maximum of 4. The recommended number is 3 as it provides the right balance between performance and fault tolerance. Usually, cloud providers provide 3 data centers/availability zones to deploy to as part of a region.
* The advantage of having a higher replication factor is that it provides a better resilience of your system. If the replication factor is N, up to N-1 broker may fail without impacting availability if acks=0 or acks=1 or N-min.insync.replicas brokers may fail if acks=all
* The disadvantages of having a higher replication factor:
  + Higher latency experienced by the producers, as the data needs to be replicated to all the replica brokers before an ack is returned if acks=all
  + More disk space required on your system

With all this in mind, the following are a few guidelines that will help you choose wisely.

* **Set it to 3** to get started (you must have at least 3 brokers for that)
* If there is a performance issue due to a higher replication factor, you should get a better broker instead of lowering the replication factor.
* Never set it to 1 in production, as it means no fault tolerance. If the partition is lost, you will have data loss.

**Cluster guidelines**

The appropriate size for a Kafka cluster is determined by several factors. \*\*\*\*Following are some general guidelines:

* A Kafka cluster should have a ***maximum of***[***200,000 partitions***](https://blogs.apache.org/kafka/entry/apache-kafka-supports-more-partitions) across all brokers when managed by Zookeeper. The reason is that if brokers go down, Zookeeper needs to perform **a lot of leader elections**. Confluent still recommends up to 4,000 partitions per broker in your cluster.
* This problem should be solved by Kafka in a Zookeeper-less mode ([**Kafka KRaft**](https://www.conduktor.io/kafka/kafka-kraft-mode/))
* If you need more than 200,000 partitions in your cluster, follow the ***Netflix model*** and create more Kafka clusters

**Minimum In-Sync Replicas**

**Kafka Topic Durability & Availability**

For a topic replication factor of 3, topic data durability can withstand 2 brokers' losses. As a general rule, for a replication factor of N, you can permanently lose up to N-1 brokers and still recover your data.

Regarding availability, it is a little bit more complicated... To illustrate, let's consider a replication factor of 3:

* Reads: As long as one partition is up and considered an ISR, the topic will be available for reads.
* Writers:
  + acks=0 & acks=1: as long as one partition is up and considered an ISR, the topic will be available for writing.
  + acks=all:
    - min.insync.replicas=1 (default): the topic must have at least 1 partition up as an ISR (that includes the reader) and so we can tolerate two brokers being down.
    - min.insync.replicas=2: the topic must have at least 2 ISR up, and therefore we can tolerate at most one broker being down (in the case of replication factor of 3), and we have the guarantee that for every write, the data will be at least written twice.
    - min.insync.replicas=3: this wouldn't make much sense for a corresponding replication factor of 3 and we couldn't tolerate any broker going down.
    - in summary, when acks=all with a replication.factor=N and min.insync.replicas=M we can tolerate N-M brokers going down for topic availability purposes.

**Segments and Indexes**

**Kafka Topic Partitions and Segments**

The basic storage unit of Kafka is a partition replica. When you create a topic, Kafka first decides how to allocate the partitions between brokers. It spreads replicas evenly among brokers.

Kafka brokers splits each partition into **segments**. Each segment is stored in a single data file on the disk attached to the broker. By default, each segment contains either 1 GB of data or a week of data, whichever limit is attained first. When the Kafka broker receives data for a partition, as the segment limit is reached, it will close the file and start a new one:

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2F1ZXlvPqblPzO4MZlqD0aQ1%2F5f9b4cece5ac705f92af4904e7fa9d92%2FAdv_Kafka_Topic_Internals_1.png&w=3840&q=75>

*Kafka Topic Partitions & Segments*

Only one segment is ACTIVE at any point in time - the one data is being written to. A segment can only be deleted if it has been closed beforehand. The size of a segment is controlled by two Broker configurations

* **log.segment.bytes:** the max size of a single segment in bytes (default 1 GB)
* [**log.segment.ms**](http://log.segment.ms)**:** the time Kafka will wait before committing the segment if not full (default 1 week)

A Kafka broker keeps an open file handle to every segment in every partition - even inactive segments. This leads to a usually high number of open file handles, and the OS must be tuned accordingly.

**Kafka Topic Segments and Indexes**

Kafka allows consumers to start fetching messages from any available offset. In order to help brokers quickly locate the message for a given offset, Kafka maintains two indexes for each segment:

* An offset to position index - It helps Kafka know what part of a segment to read to find a message
* A timestamp to offset index - It allows Kafka to find messages with a specific timestamp

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2F3rC6g86xmiKinko5Hlvb2l%2Fd739879f317bdb769fcf8eb6fbecdd24%2FAdv_Kafka_Topic_Internals_2.png&w=3840&q=75>

*Kafka Topic Segments and Indexes*

**Considerations for Segment Configurations**

* **log.segment.bytes** As messages are produced to the Kafka broker, they are appended to the current segment for the partition. Once the segment has reached the size specified by the log.segment.bytes parameter, which defaults to 1 GB, the segment is closed and a new one is opened.
  + A smaller segment size means that files must be closed and allocated more often, which reduces the overall efficiency of disk writes.
  + Once a segment has been closed, it can be considered for expiration. Adjusting the size of the segments can be important if topics have a low produce rate. Having a small segment size would mean Kafka has to keep a lot of files open which may lead to **Too many open files** error\*\*.\*\*
* **log.segment.ms** Another way to control when segments are closed is by using the log.segment.ms parameter, which specifies the amount of time after which a segment should be closed. The default is 1 week. Kafka will close a segment either when the size limit is reached or when the time limit is reached, whichever comes first.
  + When using a time-based segment limit, it is important to consider the impact on disk performance when multiple segments are closed simultaneously.

**Log Retention**

The broker configurations to control the log cleaning delete policy are:

* **log.retention.hours**

The most common configuration for how long Kafka will retain messages is **by time**. The default is specified in the configuration file using the log.retention.hours parameter, and it is set to **168 hours, the equivalent of one week**.

Setting it to a higher value will result in more disk space being used on brokers for that particular topic. On the other hand, setting it to a very small value will make data available for less time. Consumers that are not available for a long time may miss the data.

There are two other parameters allowed, log.retention.minutes and log.retention.ms. All three of these specify the same configuration - the amount of time after which messages may be deleted. If more than one is specified, the smaller unit size will take precedence.

Because the Kafka CLI command only allows you to set the ms version of this parameter, so we recommend using that one across all your configurations.

* **log.retention.bytes**

Another way to expire messages is based on the total number of bytes of messages retained. This value is set using the log.retention.bytes parameter, and it is applied **per partition**.

The default is **-1**, meaning that there is no limit and only a time limit is applied. This parameter is useful to set a to positive value if you want to keep the size of a log under a threshold.

One can mix the retention in bytes and in hours to ensure the log is never older than a certain amount of time and never larger than a certain size. This all depends on your use case and storage requirements.

**Broker-level vs Topic-level**

Kafka broker-level topic configurations are prefixed by log. and we can remove it to find the equivalent Kafka topic-level configuration

It is important to note these are minimum guarantees, not hard limits. The active segment does not count toward the Byte limit, and the Time limit can be much greater than expected if the segment is very big (few messages per day in a 1GB segment).

**Configuring Retention by Size and Time**

We learned earlier that new data gets appended into the active segment. Retention by time is performed by examining the last modified time on each log segment file on disk. This is the time that the log segment was closed, and represents the timestamp of the last message in the file.

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2F0fofNoZVx0UOPLFhWTkvX%2Ff0bf1c4c8c0492a9ae89297c886f03d3%2FAdv_Kafka_Topic_Log_Comp_1.png&w=3840&q=75>

*Log Retention by Size and Time*

If you have specified a value for both log.retention.bytes and log.retention.hours, messages may be removed when either criteria is met.

Let us see how we can implement two common use cases with these parameters.

1. **One week of retention**

To specify retention by time, we have to set log.retention.hours to one week. We have also to make sure the data is not expired by size. So, configure the values as shown:

retention.ms = 604800000

retention.bytes = -1

1. **Infinite time retention bounded by 500MB**

We have to set retention.bytes to 500MB. We have also to make sure the data is not expired by time. This can be achieved by setting it to the special -1 value. So, configure the values as shown:

retention.ms = -1

retention.bytes = 524288000

**Log Compaction**

**Kafka Log Cleanup Policies**

Kafka stores messages for a set amount of time and purge messages older than the retention period. This expiration happens due to a policy called **log.cleanup.policy**. There are two cleanup policies:

* **log.cleanup.policy=delete**

This is the default for all the user topics. With this policy configured for a topic, Kafka deletes events older than the configured retention time. The default retention period is a week.

* **log.cleanup.policy=compact**

This policy is the default for Kafka's \_\_consumer\_offsets topic. With this policy on a topic, Kafka only stores the most recent value for each key in the topic. Setting the policy to compact only makes sense on topics for which applications produce events that contain both a key and a value.

**Purpose of Log Cleanup**

Kafka was not initially meant to keep data forever (although now some people are going in that direction with large disks or [**Tiered Storage**](https://cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage)), nor does it wait for all consumers to read a message before deleting it. By configuring a retention policy for each topic, it allows administrators to:

* Control the size of the data on the disk and delete obsolete data
* Limit maintenance work on the Kafka Cluster
* Limit the amount of historical data a consumer may have to consume to catch up on the topic

**Kafka Log Cleanup frequency**

Log cleanup happens on the partition segments. Segments divide partition data into smaller files that can be managed better and cleaned independently.

Cleanup should happen often enough to ensure the files are deleted, but not so often as to affect the broker and disk performance. Smaller log retention sizes might require more frequent checks.

**Kafka Log Compaction Theory**

As we learned in the section on **Kafka Log Cleanup Policies**, Kafka will store messages for a set amount of time and purge messages older than the retention period. However, imagine a case where Kafka is used for storing the salary information about all the employees in a company. In that case, it makes sense to store only the latest salary for each employee rather than the historical data for a limited period of time.

Kafka supports such use cases by allowing the retention policy on a topic to be set to \*\*\*\*compact, **with the property to only retain at least the most recent value for each key in the partition**. It is very useful if we just require a SNAPSHOT instead of full history.

**Kafka Log Compaction Example**

We want to keep the most recent salary for our employees. We created a topic named employee-salary for this purpose. We don't want to know about the old salaries of the employees.

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2FTgidROVeU03NupMlMWSUP%2F3ace538273eedb30bf15bc0b02cb2039%2FAdv_Kafka_Topic_Log_Comp_2.png&w=3840&q=75>

*Kafka Log Compaction Example*

**Kafka Log Compaction Guarantees**

There are some important guarantees that Kafka provides for messages produced on the log-compacted topics.

* Any consumer that is reading from the tail of a log, i.e., the most current data, will still see all the messages sent to the topic. It does not matter whether a topic is log-compacted or not, consumers subscribed to the topic will see messages as they are produced on the topic.
* Ordering of messages at the key level and partition level is kept, log compaction only removes some messages, but does not re-order them. The offsets are kept, only some messages are deleted.
* 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 log.cleaner.delete.retention.ms (default is 24 hours). This gives some heads-up time for the consumers to catch up on the messages that will be deleted.

**Kafka Log Compaction Myth Busting**

Let us look at some of the misconceptions around log compaction and clear them.

* **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 the tail of a log segment as soon as the data arrives. It is not a way to perform de-duplication of messages.

* **It doesn’t prevent you from reading duplicate data from Kafka.**

If a consumer re-starts, it may see duplicate data, based on the at-least-once reading semantics we have seen before

**How Kafka Log Compaction Works**

If compaction is enabled when Kafka starts, each broker will start a compaction manager thread and a number of compaction threads. These are responsible for performing the compaction tasks.

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2F1I3rXzEXxylIEmsUGHFWdl%2Fe924ffd0357e92014f085b80e403f072%2FAdv_Kafka_Topic_Log_Comp_3.png&w=3840&q=75>

*Kafka Log Compaction Cleaner Threads*

* Cleaner threads start with the oldest segment and check their contents. The active segments are left untouched by the cleaner threads.
* If the message it has just read is still the latest for a key, it copies over the message to a replacement segment. Otherwise it omits the message because there is a message with an identical key but a newer value later in the partition.
* Once the cleaner thread has copied over all the messages that still contain the latest value for their key, we swap the replacement segment for the original and move on to the next segment.
* At the end of the process, we are left with one message per key - the one with the latest value.

**Log Compaction Practice**

We want to keep the most recent salary for our employees. We create a topic named employee-salary for this purpose. We don't want to know about the old salaries of the employees.

<https://www.conduktor.io/kafka/_next/image/?url=https%3A%2F%2Fimages.ctfassets.net%2Fo12xgu4mepom%2FTgidROVeU03NupMlMWSUP%2F3ace538273eedb30bf15bc0b02cb2039%2FAdv_Kafka_Topic_Log_Comp_2.png&w=3840&q=75>

*Kafka Log Compaction Example*

Create a log-compacted topic named employee-salary with a single partition and a replication factor of 1. Use Kafka topics CLI and pass appropriate configs as shown below.

* Number of partitions=1. This is to ensure all messages go the same partition.
* cleanup.policy=compact. This enables log compaction for the topic.
* min.cleanable.dirty.ratio=0.001. This is just to ensure log cleanup is triggered always.
* segment.ms=5000. New segment will be created every 5 seconds. Log compaction will happen on closed segments only

kafka-topics.sh --bootstrap-server localhost:9092 --create --topic employee-salary \\

--partitions 1 --replication-factor 1 \\

--config cleanup.policy=compact \\

--config min.cleanable.dirty.ratio=0.001 \\

--config segment.ms=5000

Describe the topic to make sure the configurations have been applied correctly.

kafka-topics.sh --bootstrap-server localhost:9092 --describe \\

--topic employee-salary

Start a Kafka console consumer.

Use the following command to show both the key and value, separated by a comma.

kafka-console-consumer.sh --bootstrap-server localhost:9092 \\

--topic employee-salary \\

--from-beginning \\

--property print.key=true \\

--property key.separator=,

Launch another shell to create a Kafka console producer. We want to send keys for the messages. The separator between the key and the value is a comma.

kafka-console-producer.sh --bootstrap-server localhost:9092 \\

--topic employee-salary \\

--property parse.key=true \\

--property key.separator=,

Produce a few messages with duplicate keys. In the first message shown below, the key is Mark and the value is salary: 1000.

Patrick,salary: 10000

Lucy,salary: 20000

Bob,salary: 20000

Patrick,salary: 25000

Lucy,salary: 30000

Patrick,salary: 30000

Wait a minute, and produce a few more messages (it could be the same messages)

Stephane,salary: 0

Stop the Kafka console consumer and start a new one. We are fetching all the messages from the beginning. **We'll see only the unique keys with their corresponding latest values.**

Bob,salary: 20000

Lucy,salary: 30000

Patrick,salary: 30000

Stephane,salary: 0

Log compaction will take place in the background automatically. We cannot trigger it explicitly. However, we can control how often it is triggered with the log compaction properties.

**Unclean Leader Election**

When the leader for a partition is no longer available, one of the in-sync replicas (ISR) will be chosen as the new leader. This leader election is "clean" in the sense that it guarantees no loss of committed data - by definition, committed data exists on all ISRs.

But what to do when no ISR exists except for the leader that just became unavailable?

• Wait for an ISR to come back online. This is the default behavior, and this makes you run the risk of the topic becoming unavailable.

• Enable unclean.leader.election.enable=true and start producing to non-ISR partitions. We are going to lose all messages that were written to the old leader while that replica was out of sync and also cause some inconsistencies in consumers.

**Trade-off**

**If we allow out-of-sync replicas to become leaders, we will have data loss and data inconsistencies.** If we don't allow them to become leaders, we face lower availability as we must wait for the original leader to become available before the partition is back online.

Overall this is a very dangerous setting and its implication must be understood fully before enabling it. For beginners, it is advisable to leave it to its default value of false.

However, this configuration can be enabled for some use cases including - metrics collection, log collection, and other cases where data loss is somewhat acceptable, at the trade-off of availability.

**Should you enable unclean leader election?**

This in-depth blog can help you answer this question: \*\*[https://www.datadoghq.com/blog/kafka-at-datadog/#unclean-leader-elections-to-enable-or-not-to-enable\*\*](https://www.datadoghq.com/blog/kafka-at-datadog/#unclean-leader-elections-to-enable-or-not-to-enable**)

**How to send Large Messages in Apache Kafka?**

The Kafka max message size is 1MB. In this lesson we will look at two approaches for handling larger messages in Kafka.

Kafka has a default limit of 1MB per message in the topic. This is because very large messages are considered inefficient and an anti-pattern in Apache Kafka.

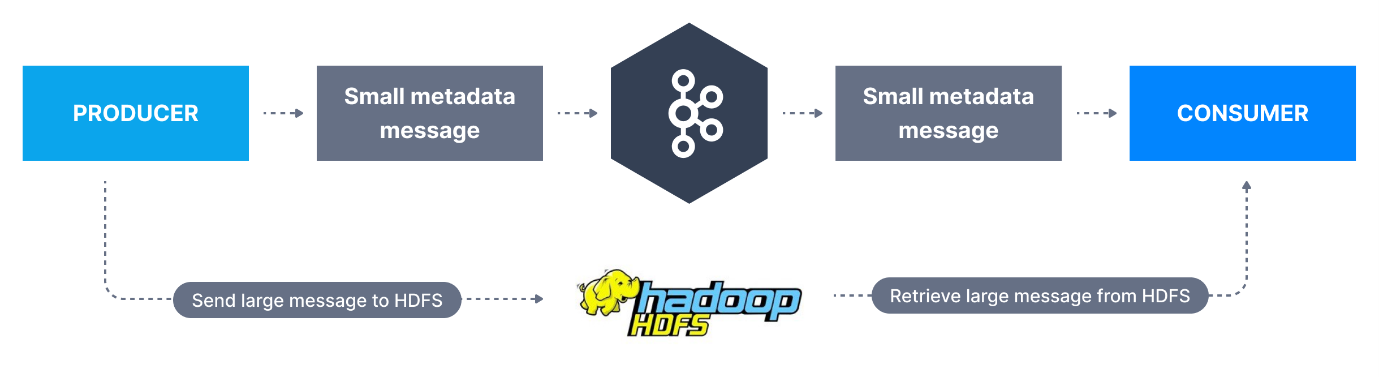
Yet, you may need to send large messages in Apache Kafka.

There are two approaches to sending large messages in Apache Kafka:

**Option 1: using an external store (GB-size messages)**

Modify your client to send large messages (example video files) outside of Kafka and only send to Kafka a reference to these message. This involves extra logic on your end but could prove quite efficient.

The store of your large message could be a cloud store such as Amazon S3, or an on-premise large file storage system such as a network system or HDFS.



ALT

To date there is no library that exists that performs this functionality out of the box, but it shouldn't be too complicated to engineer. Ensure you have written both a custom producer and consumer.

**Option 2: sending larger messages to Kafka (ex less than 10MB but over 1MB)**

Here we need to modify topic, producer and consumer configurations to allow for a bigger message size.

**Broker Side**

It is recommended to leave the max message size default for the Kafka brokers and only override this at the topic level through topic-level configurations.

**Confusing setting**

The broker-side setting is message.max.bytesand the topic-side setting is max.message.bytes

Let's create a topic named large-message

Bash

Copy

kafka-topics.sh --bootstrap-server localhost:9092 --create --topic large-message --partitions 3 --replication-factor 1

​

And add the necessary max.message.bytes configuration for 10MB

Bash

Copy

kafka-configs.sh --bootstrap-server localhost:9092 \

--alter --entity-type topics \

--entity-name configured-topic \

--add-config max.message.bytes=10485880

​

ow our topic is created and configured to receive large messages, but this is not enough/

You must also set the setting replica.fetch.max.bytes=10485880 so that your brokers can replicate the large messages correctly. This setting can only be set in the Kafka config files server.properties and must require a broker restart.

**Consumer Side**

You must also change the max.partition.fetch.bytes configuration on the consumer side and your consumer clients. If this value is smaller than message.max.bytes the consumer will fail to fetch these messages and will get stuck on processing, which is very undesirable.

To set this in your CLI, you can use --consumer-property:

Bash

Copy

kafka-console-consumer.sh --bootstrap-server localhost:9092 \

--topic large-message \

--from-beginning \

--consumer-property max.partition.fetch.bytes=10485880

​

Or in your Java code:

Java

Copy

properties.setProperty(ConsumerConfig.FETCH\_MAX\_BYTES\_CONFIG, "10485880");

​

**Producer Side**

You must change the property max.request.sizeproducer-side to ensure large messages can be sent.

To set this in your CLI, you can use --producer-property:

Bash

Copy

kafka-console-producer.sh --bootstrap-server localhost:9092 \

--topic large-message \

--producer-property max.request.size=10485880

​

Or in your Java code:

Bash

Copy

properties.setProperty(ProducerConfig.MAX\_REQUEST\_SIZE\_CONFIG, "10485880");

**How to change a Kafka Topic Configuration using the CLI?**

Broker configurations impact how topic behave in Kafka. You can find all broker configurations in the [**Kafka documentation**](https://kafka.apache.org/documentation/#brokerconfigs), and you may choose while running Kafka in production to use the defaults or set your own values.

The parameters you will set will impact topic performance and behavior (we discuss these below), and the defaults are usually fine, but some topics may need different values than the defaults, for example for the following values:

* Replication Factor
* Number of Partitions
* Message size
* Compression level
* Log Cleanup Policy
* Min Insync Replicas

The complete list of topic configurations can be found at: \*\*[https://kafka.apache.org/documentation/#topicconfigs\*\*](https://kafka.apache.org/documentation/#topicconfigs**)

kafka-configs.sh --bootstrap-server localhost:9092 --alter --entity-type topics --entity-name configured-topic --delete-config min.insync.replicas

# Quotas

In Apache Kafka, quotas are used to manage and limit the amount of data bandwidth or request rate that clients can utilize, which helps to ensure fair distribution of resources among users or applications and prevent any single client from overwhelming the cluster. Quotas can be set on both producer and consumer bandwidth, as well as on the number of requests to the broker.

### **Types of Quotas in Kafka**

1. **Producer Byte-rate Quotas**: Limit the total bytes sent per second by a producer client.
2. **Consumer Byte-rate Quotas**: Limit the total bytes fetched per second by a consumer client.
3. **Request Rate Quotas**: Limit the number of requests per second a client can make to the broker.

### **Setting up Quotas**

Quotas are defined per user or per client-id or a combination of both, and they are enforced at the broker level. Here’s how you can set these quotas:

### **Step 1: Enable Quota Configuration in Broker**

Make sure your Kafka brokers are configured to use quotas. This is usually enabled by default, but you can check or set these properties in the **server.properties** file:

propertiesCopy code

# Enables quotas for producer and consumer byte rates

quota.producer.default=1048576

quota.consumer.default=1048576

# Enables request rate quotas

quota.window.size.seconds=1

quota.window.num=11

### **Step 2: Setting Quotas Using Kafka Admin Tools**

You can set or change quotas using the Kafka admin tools that come with Kafka distributions. Specifically, you would use the **kafka-configs.sh** script to modify user or client-specific quotas.

### **Setting a Producer Byte-rate Quota**

bashCopy code

# Set a quota of 10 MB/s for a specific client-id

kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'producer\_byte\_rate=10485760' --entity-type clients --entity-name client-id-1

### **Setting a Consumer Byte-rate Quota**

bashCopy code

# Set a quota of 10 MB/s for a specific user

kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'consumer\_byte\_rate=10485760' --entity-type users --entity-name user-1

### **Setting Request Rate Quotas**

bashCopy code

# Set a quota of 100 requests per second for a client-id

kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'request\_percentage=100' --entity-type clients --entity-name client-id-2

### **Step 3: Checking Quota Settings**

To verify the quotas, use the same **kafka-configs.sh** tool:

bashCopy code

# Check quota for a specific client-id

kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type clients --entity-name client-id-1

# Check quota for a specific user

kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type users --entity-name user-1

### **Considerations**

* **Overheads**: Setting too restrictive quotas can lead to throttling, which may unexpectedly slow down your data pipelines.
* **Balance**: It's important to balance the quotas across your use cases and the expected loads. Monitor and adjust as necessary.
* **Dynamic Changes**: Quotas can be changed dynamically without restarting the Kafka brokers, making it easier to adjust to changing load conditions.

By effectively using Kafka's quota management features, you can maintain a stable and predictable environment, especially in multi-tenant scenarios where different clients and applications share the same Kafka infrastructure.

**Benchmarking**

<https://github.com/nagcloudlab/strimzi/blob/main/week-1/play-with-kafka/005-perf-test/01.md>

**Kafka Cluster Config**

<https://github.com/nagcloudlab/strimzi/tree/main/week-1/play-with-kafka/001-kafka-cluster>

# Introduction

# ****The Kafka Ecosystem****

### **Slide 1: Introduction to the Kafka Ecosystem Before Kafka Streams**

* Identified a significant gap in the Kafka ecosystem for supporting stream processing.
* Developers faced challenges in building effective stream processing applications due to the lack of specialized library support.

### **Slide 2: Early Options for Stream Processing with Kafka**

* Option 1: Utilizing Kafka's Consumer and Producer APIs directly to manage data flows.
* Option 2: Integrating with other stream processing frameworks, such as Apache Spark Streaming and Apache Flink, to leverage their capabilities.

### **Slide 3: Limitations of Consumer and Producer APIs**

* Basic and low-level APIs lacking essential stream processing features.
* No built-in support for managing state that is both local and fault-tolerant.
* Sparse availability of operators for effectively transforming data streams.
* Inadequate handling of stream representations and time complexities.

### **Slide 4: Challenges with Other Streaming Frameworks**

* Full-scale streaming platforms introduce unnecessary complexity and management overhead.
* These frameworks often do not integrate seamlessly with Kafka, especially in handling data that intermediates between source and sink topics.

### **Slide 5: Kafka Community’s Response**

* The Kafka community acknowledged the critical need for a native stream processing API within its ecosystem.
* This led to the creation of Kafka Streams, designed to simplify stream processing with direct integration into Kafka, minimizing complexity and enhancing functionality.

# ****Enter Kafka Streams****

### **Slide 1: Introduction to Kafka Streams**

* Kafka Streams released in 2016, marking a significant evolution in the Kafka ecosystem.
* Transition from reliance on hand-rolled features to using standardized, community-developed patterns and abstractions for stream processing.

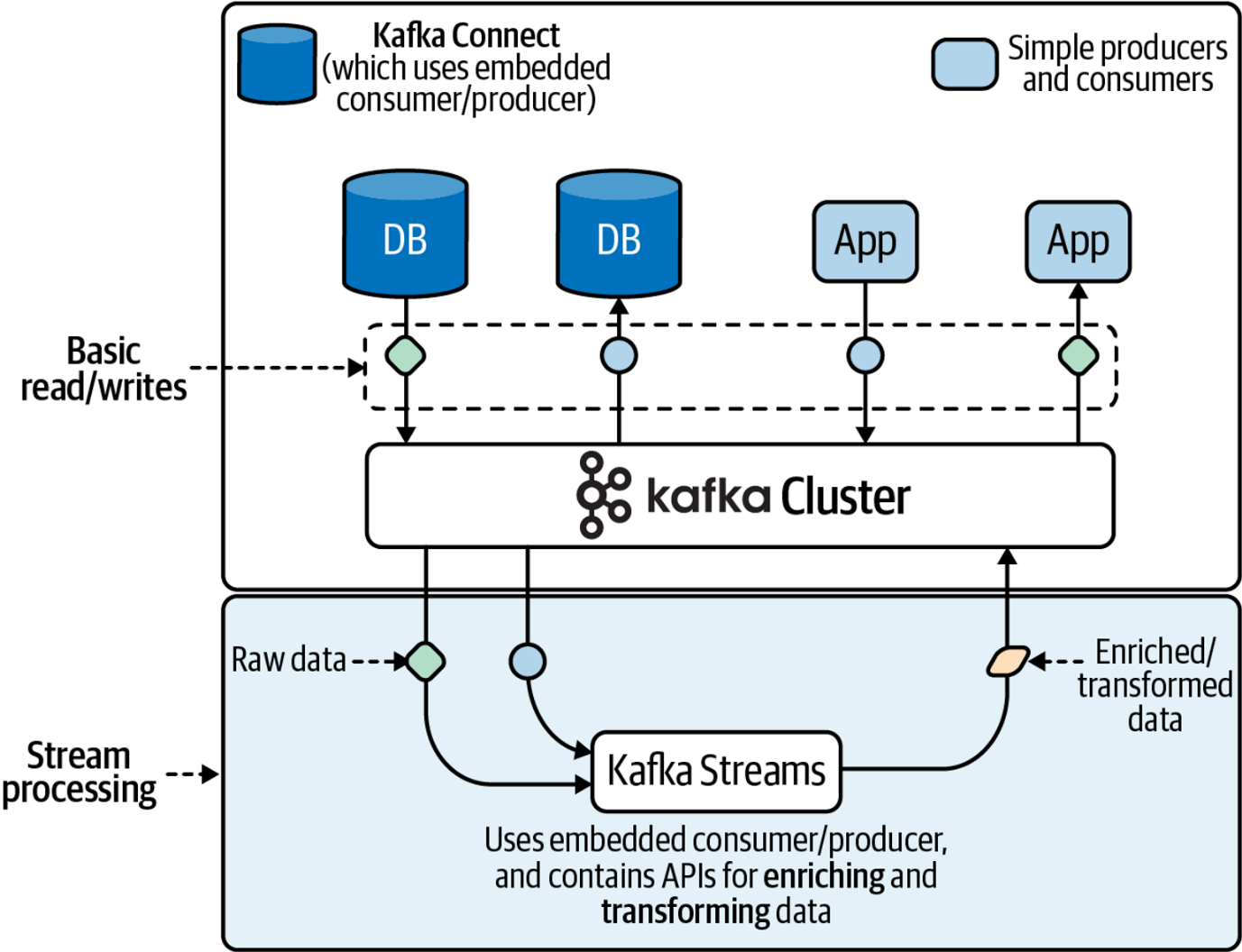
### **Slide 2: Advantages of Kafka Streams Over Previous APIs**

* Unlike basic APIs (Producer, Consumer, Connect), Kafka Streams is specifically designed for processing real-time data streams.
* Facilitates the consumption of real-time event streams and enables complex data transformations and processing directly within Kafka.

### **Slide 3: Kafka Streams Capabilities**

* Provides a robust set of stream processing operators and primitives.
* Allows for the transformation and enrichment of data streams, supporting the creation of new data representations to be sent back to Kafka for downstream consumption.

### **Slide 4: Role of Kafka Streams in the Ecosystem**

* Described as the "brain" of the Kafka ecosystem, integrating and processing data from multiple sources.
* Positioned at a crucial layer in the data pipeline where it enhances data value through sophisticated enrichment and transformation processes. 

### **Slide 5: Impact of Kafka Streams on Stream Processing**

* Simplifies the development of stream processing applications by removing the need for custom-built solutions.
* Reduces the complexity and overhead associated with using alternative stream processing frameworks.

# ****Features at a Glance****

* **High-Level DSL**
  + Mimics Java’s streaming API, offering a fluent and functional approach to data stream processing.
  + Designed to be easy to learn and use.
* **Low-Level Processor API**
  + Provides developers with fine-grained control over their stream processing applications.
  + Ideal for complex, customized processing needs.
* **Data Modeling Abstractions**
  + Allows data to be modeled as either streams or tables, enhancing flexibility.
  + Supports diverse processing strategies and data management approaches.
* **Operators and Utilities**
  + Includes a wide range of operators for building both stateless and stateful stream processing applications.
  + Supports sophisticated data operations and stream manipulations.
* **Time-Based Operations**
  + Features support for time-based operations such as windowing and periodic functions.
  + Enables handling of time-sensitive data and temporal data analysis.
* **Easy Installation**
  + Kafka Streams is just a library, making it easy to integrate into any Java application.
  + Simplifies setup and reduces deployment overhead.
* **Scalability, Reliability, Maintainability**
  + Designed to scale with application needs and ensure reliable data processing.
  + Focuses on maintainability, aiding long-term application stability and performance.

# ****Operational Characteristics****

### **Slide 1: Introduction to Operational Characteristics**

* Focus on three critical goals: Scalability, Reliability, and Maintainability.
* Framework for evaluating Kafka Streams based on these characteristics.

### **Slide 2: Scalability of Kafka Streams**

* Scalable through partition expansion and the use of consumer groups.
* Ability to distribute workload across multiple application instances efficiently.

### **Slide 3: Elasticity of Kafka Streams**

* Kafka Streams supports dynamic scaling by adding or reducing application instances.
* Scalability is limited by the number of tasks available in the application's topology.

### **Slide 4: Reliability Features of Kafka Streams**

* Ensures reliability through automatic failovers and consumer group partition rebalancing.
* Handles faults effectively, minimizing system downtime and maintaining service continuity.

### **Slide 5: Maintainability of Kafka Streams**

* Maintenance involves more than initial development; focus on reducing long-term costs.
* Java-based, simplifying troubleshooting, bug fixes, and ongoing maintenance.
* Streamlined codebase and intuitive API facilitate quick learning and easy upkeep.

# Comparison to Other Systems

### **Slide 1: Comparing Deployment Models**

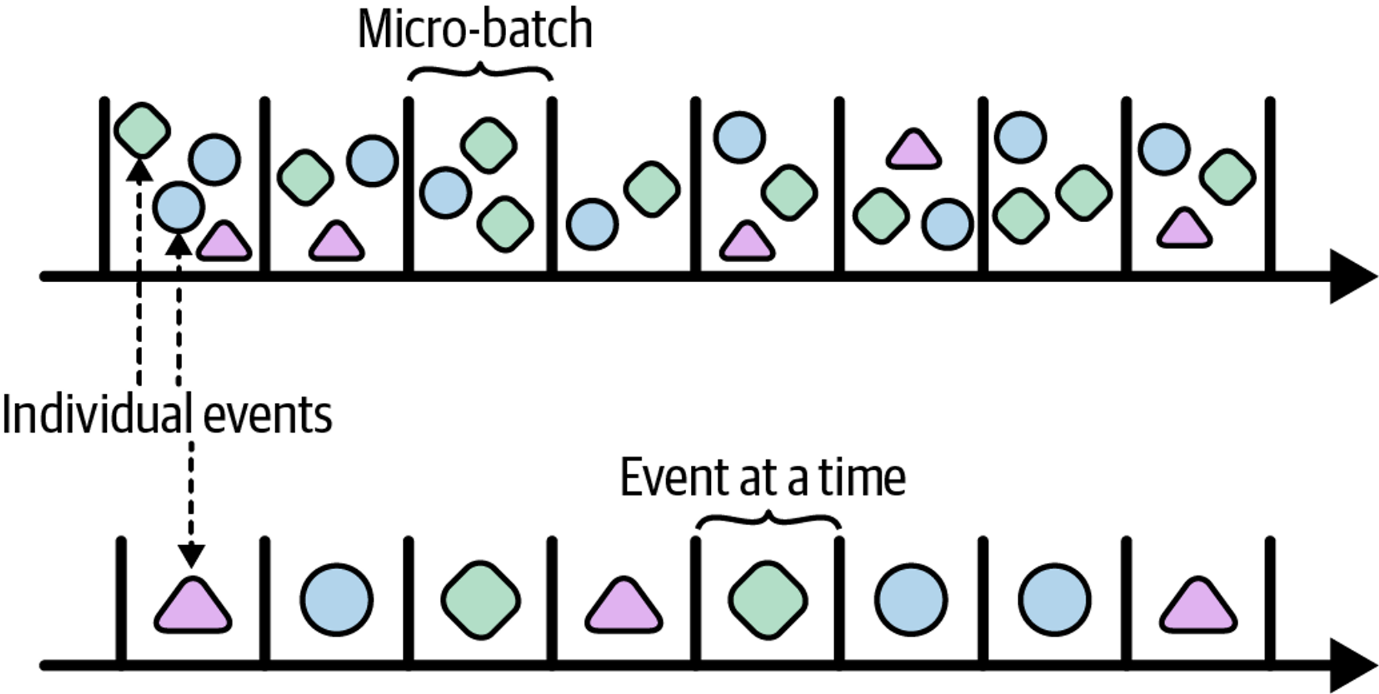
* **Kafka Streams**: Implemented as a Java library, simplifying integration and deployment within existing Java applications.
* **Other Systems (Apache Flink, Spark Streaming)**: Require a dedicated processing cluster, adding complexity and overhead.
* **Real-World Adaptation**: Takes months to adapt to systems like Apache Flink at large companies due to the nuances of cluster management.

### **Slide 2: Deployment Advantages of Kafka Streams**

* **Ease of Setup**: No need for a cluster manager; just add Kafka Streams as a dependency.
* **Operational Flexibility**: Applications are deployed using familiar Java application patterns, enhancing integration with existing tools and systems.

### **Slide 3: Processing Models Compared**

* **Kafka Streams**: Implements true streaming with event-at-a-time processing, handling events immediately as they arrive.
* **Other Systems (Apache Spark Streaming, Trident)**: Use micro-batching, grouping records into batches for processing at fixed intervals.
* **Performance Consideration**: Event-at-a-time processing provides lower latency, while micro-batching may optimize for throughput but with higher latency.



### **Slide 4: Architectural Focus: Kappa vs. Lambda**

* **Kafka Streams (Kappa Architecture)**: Focuses solely on streaming use cases, optimizing for simplicity in stream data operations.
* **Other Frameworks (Apache Flink, Spark)**: Support both batch and stream processing (Lambda Architecture), which can introduce complexity and operational challenges.
* **Hybrid Systems**: Often carry high operational burdens and are limited to the features that intersect between batch and stream processing capabilities.

### **Slide 5: Comparison with Apache Beam**

* **Kafka Streams**: Described as a stream-relational processing platform, integrating streams and tables as first-class entities.
* **Apache Beam**: A stream-only processing platform, relying on execution engines like Flink or Spark for processing.
* **Feature Comparison**: Kafka Streams offers powerful features such as ad-hoc query capabilities for stream states, which are typically lacking in Apache Beam.

### **Slide 6: Advantages of Kafka Streams in Stream Processing**

* Offers a more streamlined and focused approach to stream processing, enhancing development and operational simplicity.
* Ideal for use cases that do not require the integration of batch processing, avoiding the complexity of hybrid systems.

# ****Use Cases****

### **Slide 1: Introduction to Kafka Streams Use Cases**

* Kafka Streams excels in processing unbounded datasets rapidly and efficiently.
* Optimized for low-latency, time-critical applications across various domains.

### **Slide 2: Financial and Trading Applications**

* **Financial Data Processing**: Used by companies like Flipkart for monitoring purchases and detecting fraud.
* **Algorithmic Trading**: Facilitates real-time data analysis for automated trading systems.
* **Market Monitoring**: Critical for real-time tracking of stock and cryptocurrency markets.

### **Slide 3: Retail and Event Management**

* **Inventory Management**: Real-time tracking and replenishment systems, exemplified by Walmart.
* **Event Ticketing**: Manages bookings and seat selections, utilized by Ticketmaster for real-time updates.

### **Slide 4: Media and Communication**

* **Email Tracking**: Monitors email delivery, used by Mailchimp to track performance and issues.
* **Game Telemetry**: Processes gaming data in real time, used by Activision for titles like Call of Duty.
* **Sports Broadcasting**: Supports real-time sports data widgets and broadcasting enhancements (Gracenote).

### **Slide 5: Technology and Innovation**

* **Search and Geospatial Tracking**: Powers real-time search indexing for Yelp and performs complex geospatial calculations.
* **Smart Home/IoT**: Processes data from IoT devices, enhancing smart home systems with real-time analytics.

### **Slide 6: Healthcare and Advertising**

* **Healthcare Monitoring**: Supports predictive healthcare and real-time vitals monitoring at institutions like Children’s Healthcare of Atlanta.
* **Advertising**: Powers real-time ad platforms, improving targeting and performance metrics (Pinterest).

### **Slide 7: Communication and Machine Learning**

* **Communication Infrastructure**: Underpins chat systems and virtual assistants, used by companies like Slack.
* **Machine Learning Pipelines**: Facilitates real-time data processing for machine learning applications, including Twitter and Kafka Graphs.

### **Slide 8: Extending Kafka Streams Capabilities**

* **Microservices**: Ideal for building microservices on real-time event streams.
* **Interactive Queries**: Capable of exposing stream state for dynamic query handling and real-time data aggregation.

Financial data processing (Flipkart), purchase monitoring, fraud detection

Algorithmic trading

Stock market/crypto exchange monitoring

Real-time inventory tracking and replenishment (Walmart)

Event booking, seat selection (Ticketmaster)

Email delivery tracking and monitoring (Mailchimp)

Video game telemetry processing (Activision, the publisher of Call of Duty)

Search indexing (Yelp)

Geospatial tracking/calculations (e.g., distance comparison, arrival projections)

Smart Home/IoT sensor processing (sometimes called AIOT, or the Artificial Intelligence of Things)

Change data capture (Redhat)

Sports broadcasting/real-time widgets (Gracenote)

Real-time ad platforms (Pinterest)

Predictive healthcare, vitals monitoring (Children’s Healthcare of Atlanta)

Chat infrastructure (Slack), chat bots, virtual assistants

Machine learning pipelines (Twitter) and platforms (Kafka Graphs)

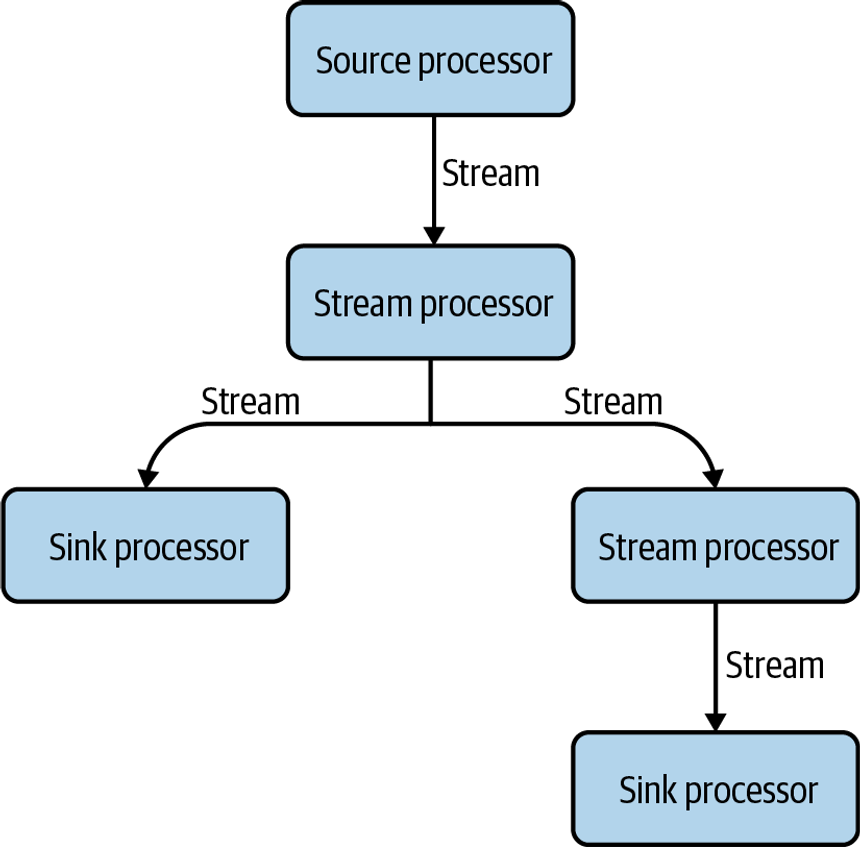
# ****Processor Topologies****

### **Slide 1: Introduction to Processor Topologies**

* Kafka Streams uses dataflow programming (DFP), representing programs as a series of inputs, outputs, and processing stages.
* This model is intuitive and beginner-friendly, structuring logic as a directed acyclic graph (DAG).

### **Slide 2: Basic Processor Types**

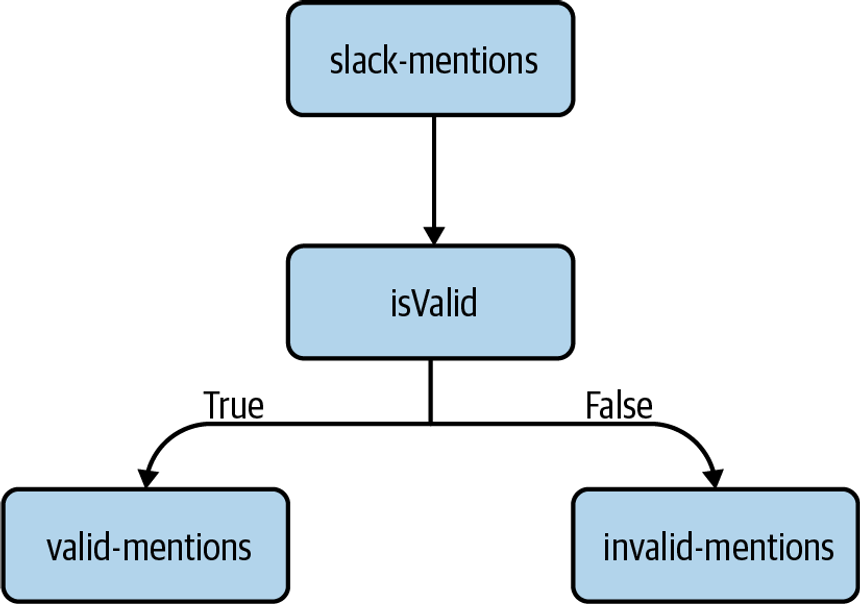
* **Source Processors**: Input points where data flows into the application from Kafka topics.
* **Stream Processors**: Perform transformations and apply logic to incoming data. Common operations include filter, map, and join.
* **Sink Processors**: Output points where processed data is sent back to Kafka or to downstream systems.



**Kafka Streams borrows some of its design from dataflow programming, and structures stream processing programs as a graph of processors through which data flows**

### **Slide 3: Designing a Processor Topology**

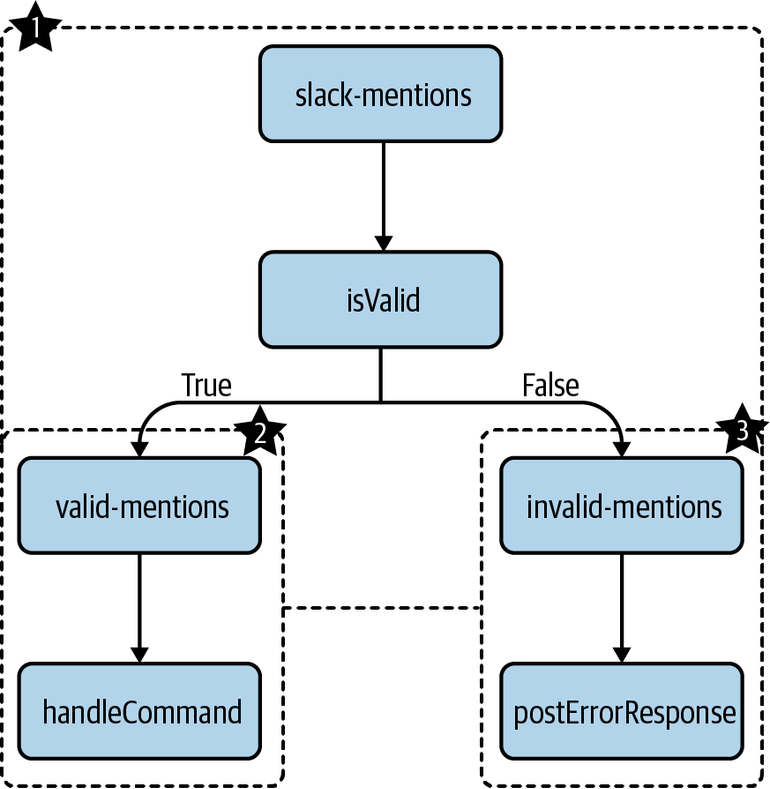
* Example of designing a topology: Creating a chatbot that filters valid and invalid commands from Slack messages.
* Source topic reads messages, stream processor validates them, and sink processors route them to either 'valid-mentions' or 'invalid-mentions'.



**An example processor topology that contains a single source processor for reading Slack messages from Kafka (**slack-mentions**), a single stream processor that checks the validity of each message (**isValid**), and two sink processors that route the message to one of two output topics based on the previous check (**valid-mentions**,** invalid-mentions**)**

### **Slide 4: Understanding Sub-Topologies**

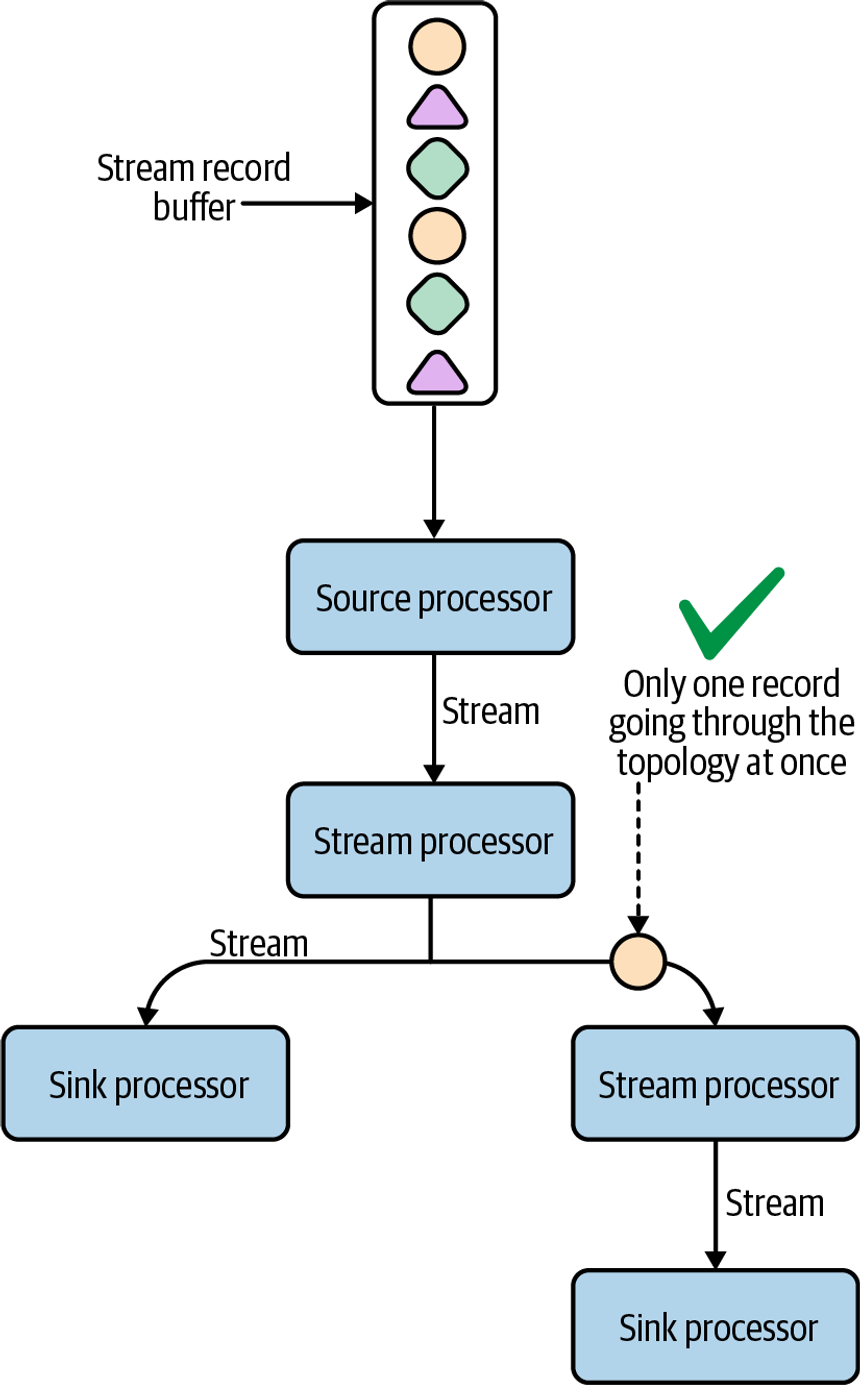
* Sub-topologies allow Kafka Streams to handle multiple source topics and partition workloads more efficiently.
* Each sub-topology can operate independently, increasing the application's scalability and fault tolerance.



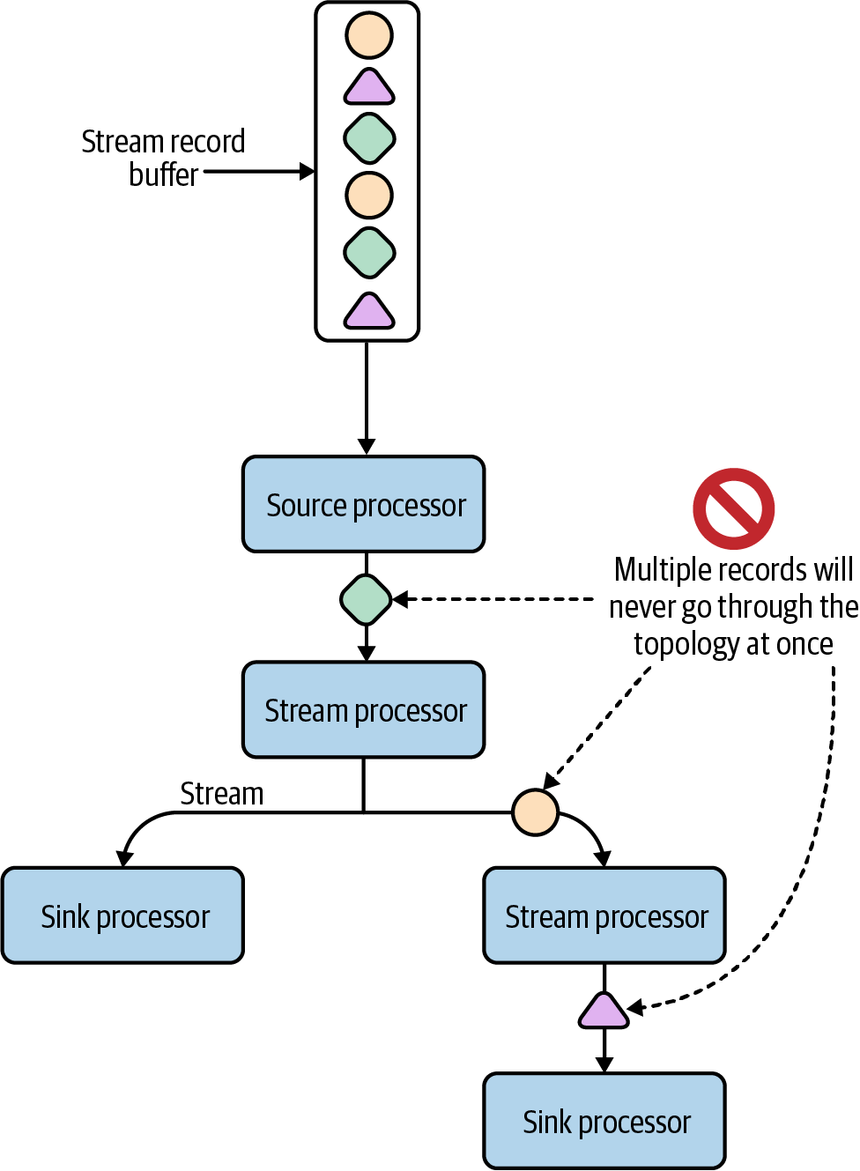
### **Slide 5: Depth-First Processing in Kafka Streams**

* Kafka Streams processes each event completely in one go through the topology before moving to the next (depth-first).
* Ensures straightforward data flow and consistency but can be slowed by intensive operations in the stream.

**In depth-first processing, a single record moves through the entire topology before another record is processed**



**Multiple records will never go through the topology at the same time**



### **Slide 6: Benefits of Dataflow Programming**

* Simplifies understanding of how data moves and is transformed within the application.
* Helps in visualizing and communicating application design to non-technical stakeholders.
* Supports GDPR compliance and data privacy management by clearly defining data pathways.

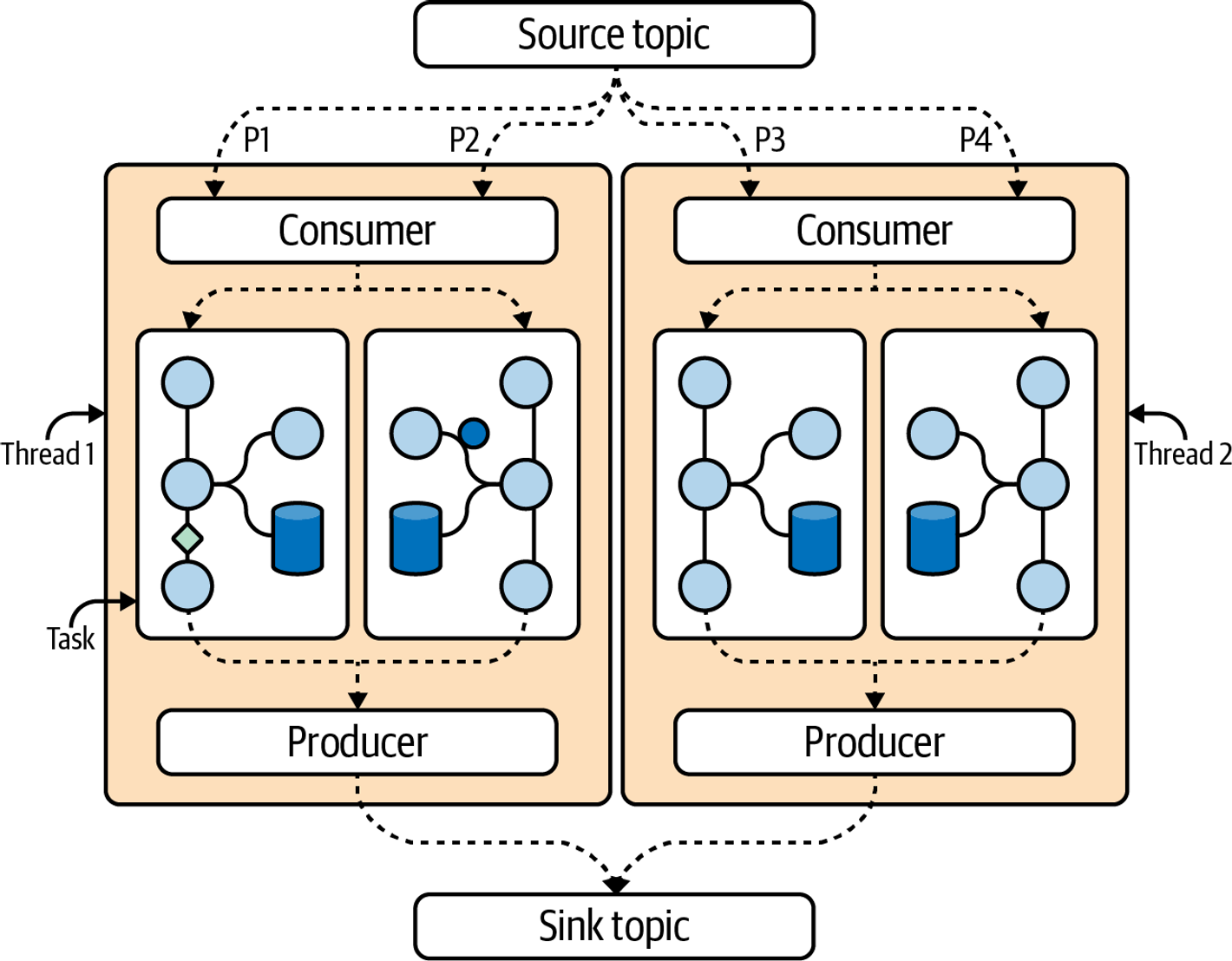
### **Slide 7: Tasks and Stream Threads**

* **Tasks**: Smallest units of parallel execution in Kafka Streams, dependent on the number of partitions in source topics.
* **Threads**: Execute tasks and can be configured to optimize resource utilization and performance.

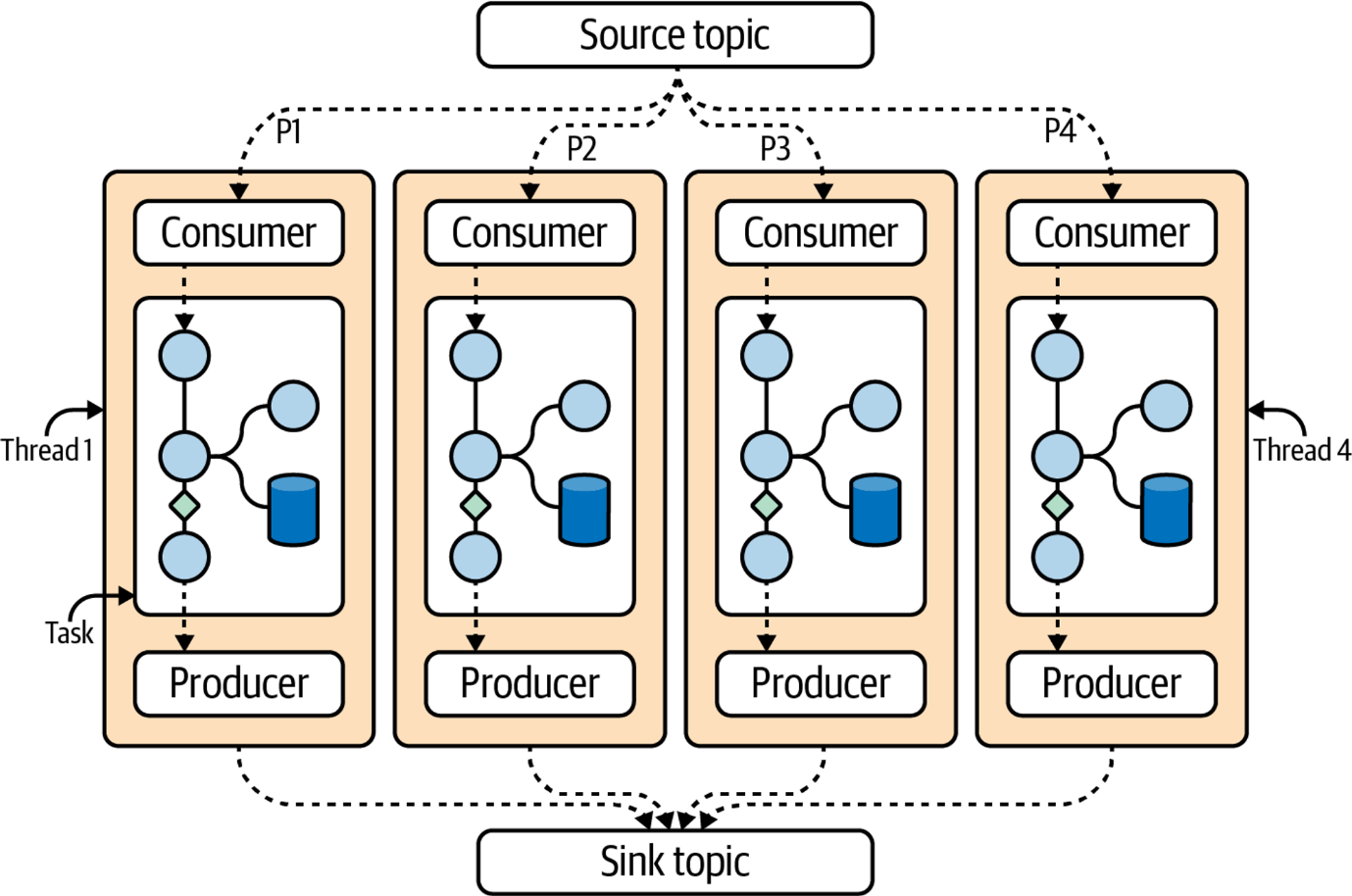
### **Slide 8: Configuring Tasks and Threads**

* Tasks are tied to partitions, determining the maximum parallelism.
* Threads can be adjusted to balance performance and resource usage, with strategies varying based on the application's scale and complexity.

**Four Kafka Streams tasks running in two threads**



**Four Kafka Streams tasks running in four threads**



# ****High-Level DSL Versus Low-Level Processor API****

### **Slide 1: High-Level DSL**

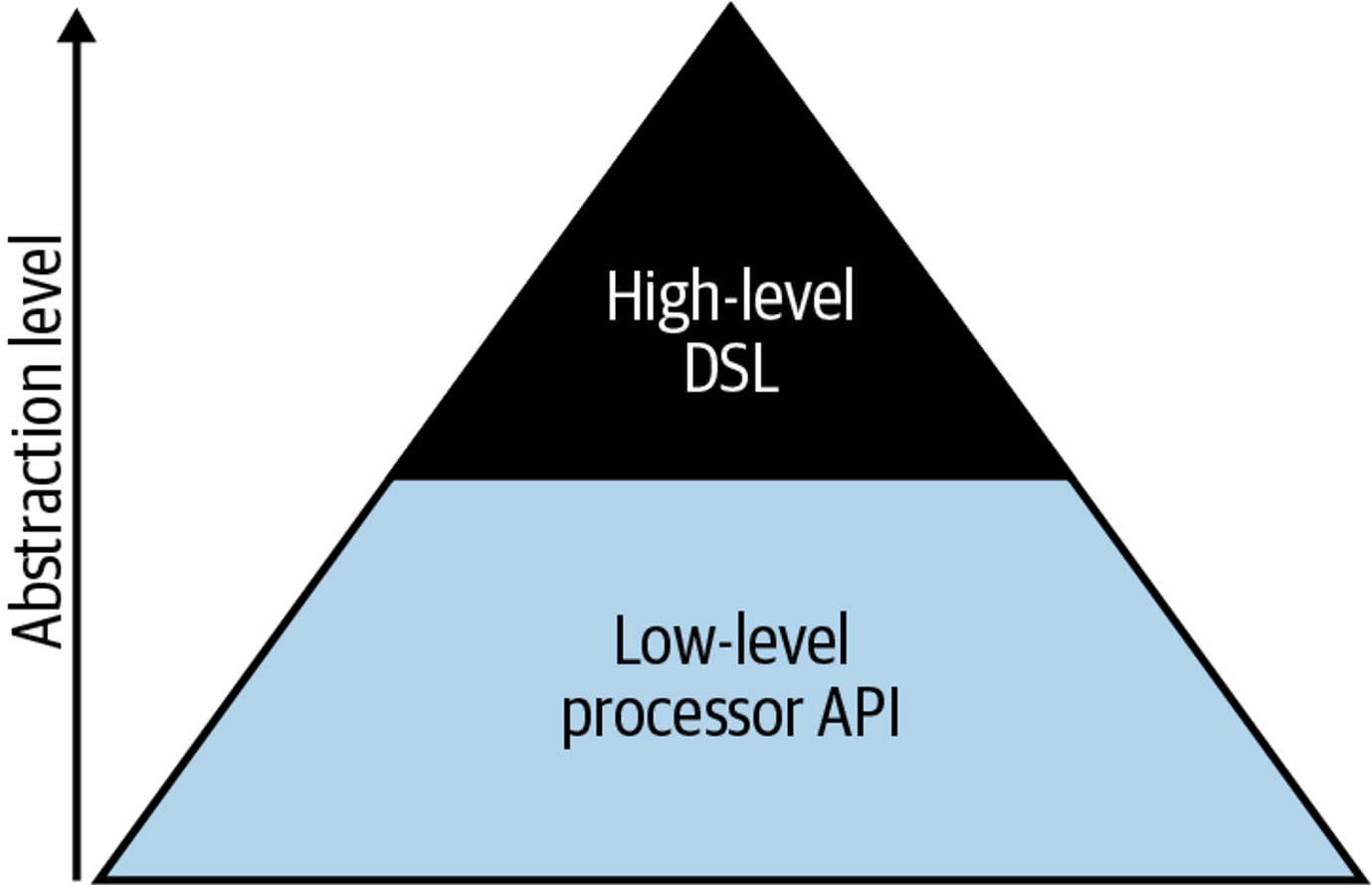
* Built on top of the Processor API, offering a more abstract, user-friendly interface.
* Ideal for developers preferring a functional style of programming and higher-level abstractions like streams and tables.
* Simplifies common stream processing tasks, reducing the need to manage complex details.

### **Slide 2: Low-Level Processor API**

* Provides granular control over stream processing operations.
* Allows access to record metadata, scheduling of periodic functions, and detailed state management.
* Suitable for scenarios requiring fine-grained control over data handling and process timing.

### **Slide 50: Choosing the Right API**

* The choice depends on the project requirements, developer experience, and desired level of control.
* High-Level DSL simplifies development with pre-built operations and constructs.
* Low-Level Processor API offers customization and precise control, suitable for complex, bespoke processing needs.



# lab-1

export JAVA\_HOME=$(/usr/libexec/java\_home -v 1.8)

cd /chapter-02/hello-streams

./gradlew build

./gradlew runDSL --info

docker-compose exec kafka bash

kafka-console-producer --bootstrap-server localhost:9092 --topic users

>angie

>guy

>kate

>mark

./gradlew runProcessorAPI --info

# ****Streams and Tables****

Keyed records in a single topic-partition

server-logs topic

| **Key** | **Value** | **Offset** |
| --- | --- | --- |
| mitch | { "action": "login" } | 0 |
| mitch | { "action": "logout" } | 1 |
| elyse | { "action": "login" } | 2 |
| isabelle | { "action": "login" } | 3 |

Streams ( Kstream )

| **Key** | **Value** | **Offset** |
| --- | --- | --- |
| mitch | { "action": "login" } | 0 |
| mitch | { "action": "logout" } | 1 |
| elyse | { "action": "login" } | 2 |
| isabelle | { "action": "login" } | 3 |

Table ( KTable )

| **Key** | **Value** | **Offset** |
| --- | --- | --- |
| mitch | { "action": "logout" } | 1 |
| elyse | { "action": "login" } | 2 |
| isabelle | { "action": "login" } | 3 |

Aggregated Table ( KTable )

| **Key** | **Value** | **Offset** |
| --- | --- | --- |
| mitch | 2 | 1 |
| elyse | 1 | 2 |
| isabelle | 1 | 3 |

Records that are written to Kafka are immutable, so how is it possible to model data as updates, using a table representation of a Kafka topic?

The answer is simple: the table is materialized on the Kafka Streams side using a key-value store which, by default, is implemented using RocksDB

# ****KStream, KTable, GlobalKTable****

### **Slide 1: Introduction to Kafka Streams Abstractions**

* Overview of key abstractions in Kafka Streams' High-Level DSL: KStream, KTable, and GlobalKTable.
* Importance of choosing the right abstraction based on application needs and data handling requirements.

### **Slide 2: KStream Overview**

* **Definition**: KStream represents a partitioned record stream where each data event is independent.
* **Usage**: Ideal for applications that process incoming data records as individual events.
* **Characteristics**: Operates with insert semantics, treating each event as a new and unique entry.

### **Slide 3: KTable Overview**

* **Definition**: KTable is an abstraction of a partitioned table, akin to a changelog stream where updates are tracked by keys.
* **Usage**: Best suited for applications that require maintaining the latest state or value for each key.
* **Characteristics**: Uses update semantics, updating the state of the key with each new event. Each task manages only a part of the full table due to its partitioned nature.

### **Slide 4: GlobalKTable Overview**

* **Definition**: Similar to KTable but differs in that each instance of a GlobalKTable contains a full copy of the dataset, unpartitioned.
* **Usage**: Useful when there is a need to perform join operations with a full dataset accessible across all application instances.
* **Characteristics**: Maintains a complete, consistent view of the data, advantageous for specific join scenarios.

### **Slide 5: Selecting Between KStream, KTable, and GlobalKTable**

* Decision factors include the nature of the data, required processing semantics, and the specific needs of the application.

kStream

KTable

**GlobalKTable**

# ****Stateless Processing****

### **Slide 1: Introduction to Stateless Stream Processing**

* Overview of stateless processing: Each event is consumed, processed independently, and not stored for future reference.
* Emphasizes the simplicity and efficiency of stateless operations within Kafka Streams.

### **Slide 2: Understanding Stateless Operators**

* Definition of stateless operators: Tools within Kafka Streams that operate without needing a memory of previous events.
* Examples include filtering records, transforming records, and enriching data on the fly.

### **Slide 3: Common Stateless Tasks**

* **Filtering Records**: Excluding or including data based on specific criteria.
* **Adding/Removing Fields**: Modifying data structure dynamically during stream processing.
* **Rekeying Records**: Altering the keys associated with data records for grouping or redirection.

### **Slide 4 Advanced Stateless Operations**

* **Branching Streams**: Splitting a stream into multiple streams based on specified conditions.
* **Merging Streams**: Combining multiple streams into a single stream to unify data sources.
* **Transforming Outputs**: Converting data into one or more new formats or representations.

### **Slide 5: Use Case: Streaming Cryptocurrency Data**

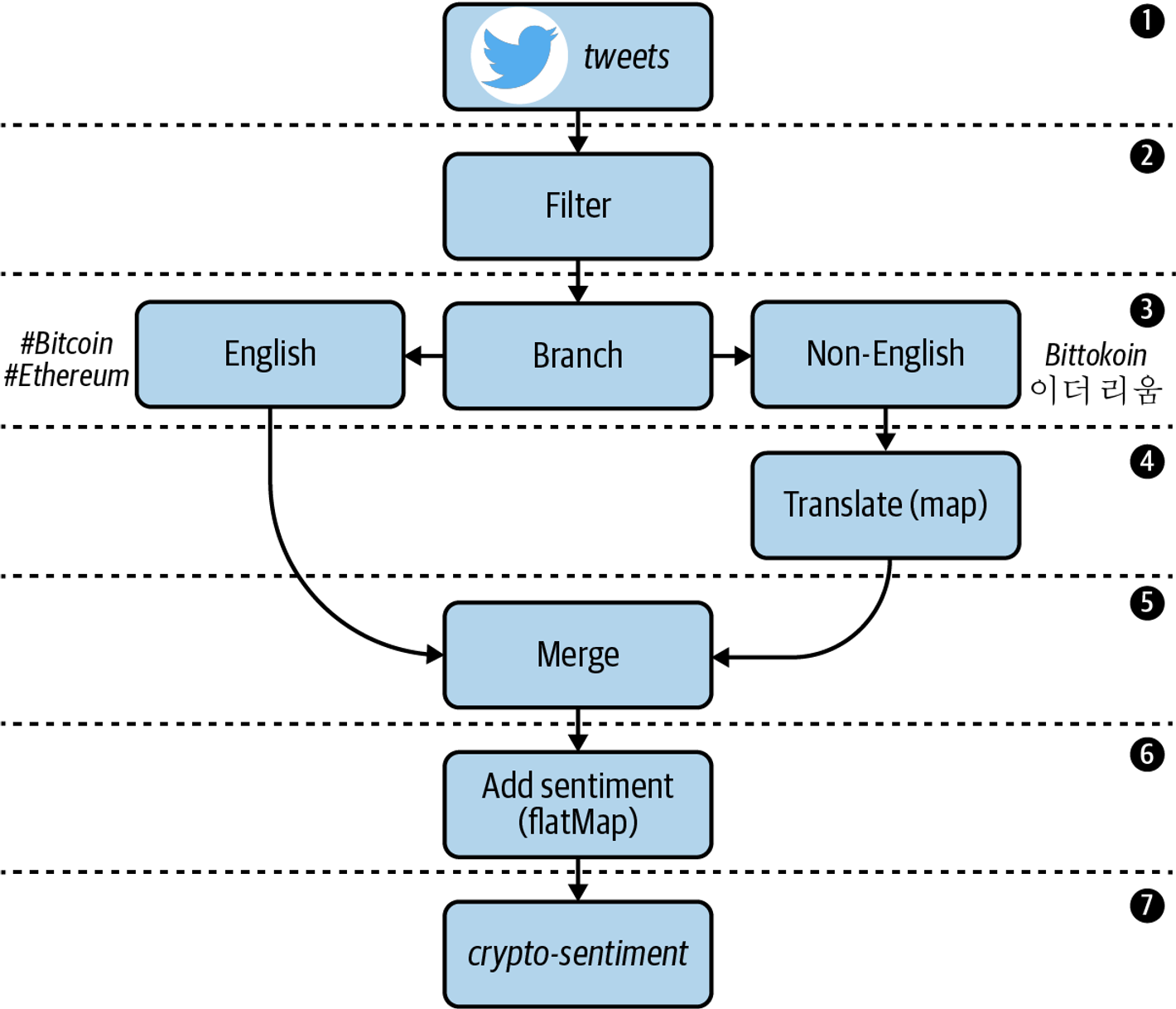
* Example scenario: Streaming data from Twitter on cryptocurrencies.
* Applying stateless operators to transform raw tweets into actionable investment signals.

### **Slide 6: Stateless vs. Stateful Processing**

* Contrast between stateless and stateful processing:
  + **Stateless**: Treats each event independently without maintaining state.
  + **Stateful**: Requires memory of past events for functions like aggregation or windowing.
* Impact on application complexity and operational considerations.

# ****Processing a Twitter Stream****

**The topology that we will be implementing for our tweet enrichment application**



**Adding a KStream Source Processor**

{

"CreatedAt": 1602545767000,

"Id": 1206079394583924736,

"Text": "Anyone else buying the Bitcoin dip?",

"Source": "",

"User": {

"Id": "123",

"Name": "Mitch",

"Description": "",

"ScreenName": "timeflown",

"URL": "<https://twitter.com/timeflown>",

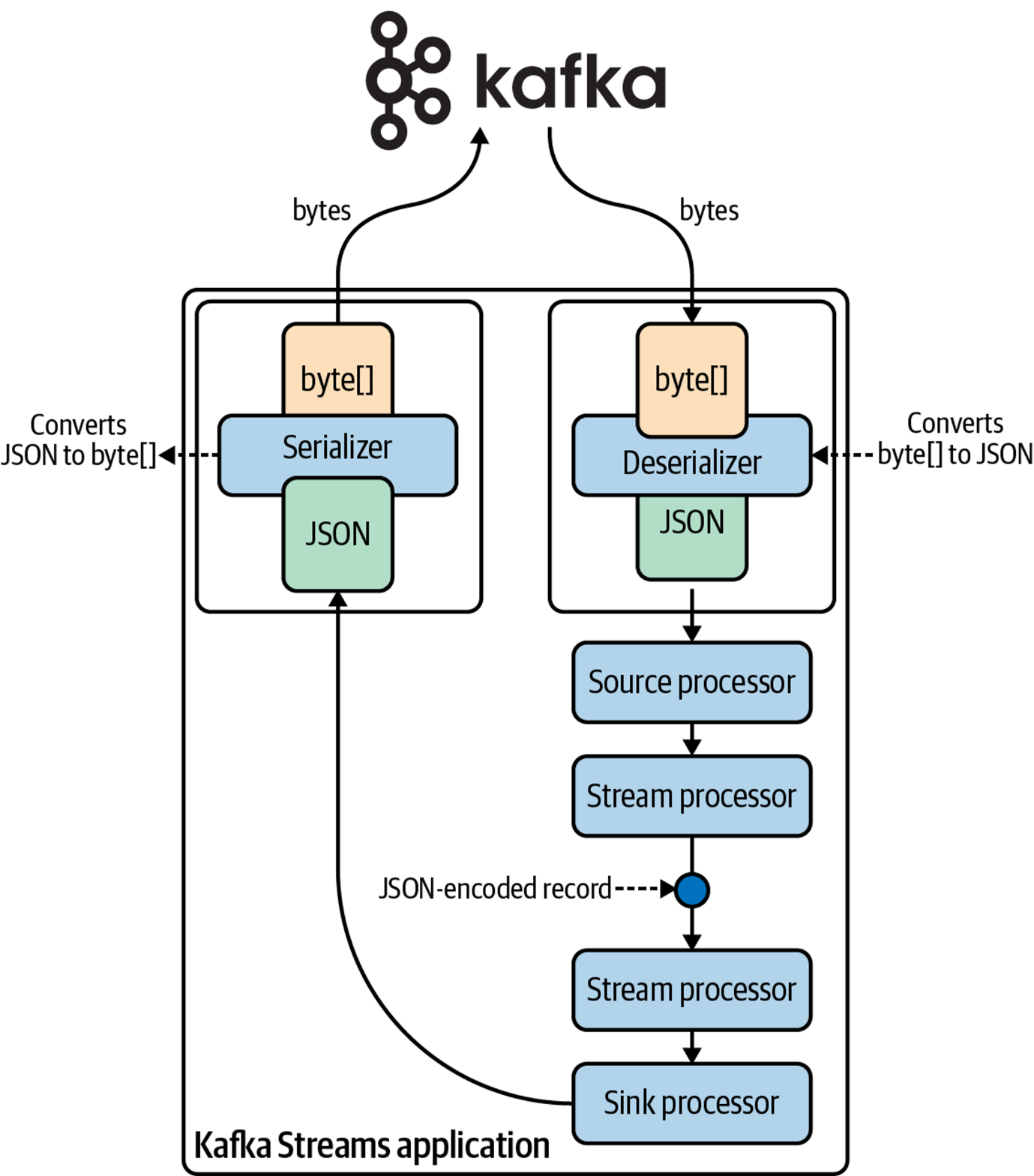
"FollowersCount": "1128",

"FriendsCount": "1128"

}

}

**Serialization/Deserialization**

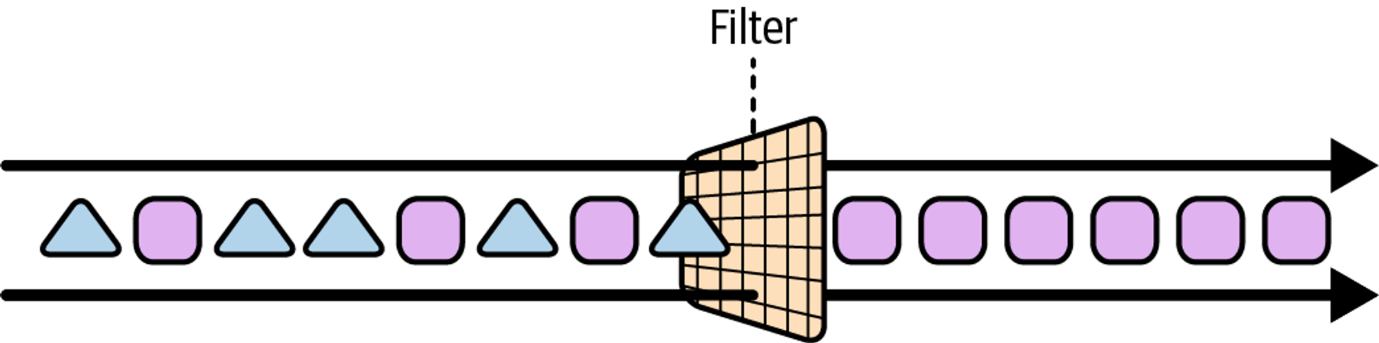


Default Serdes implementations that are available in Kafka Streams

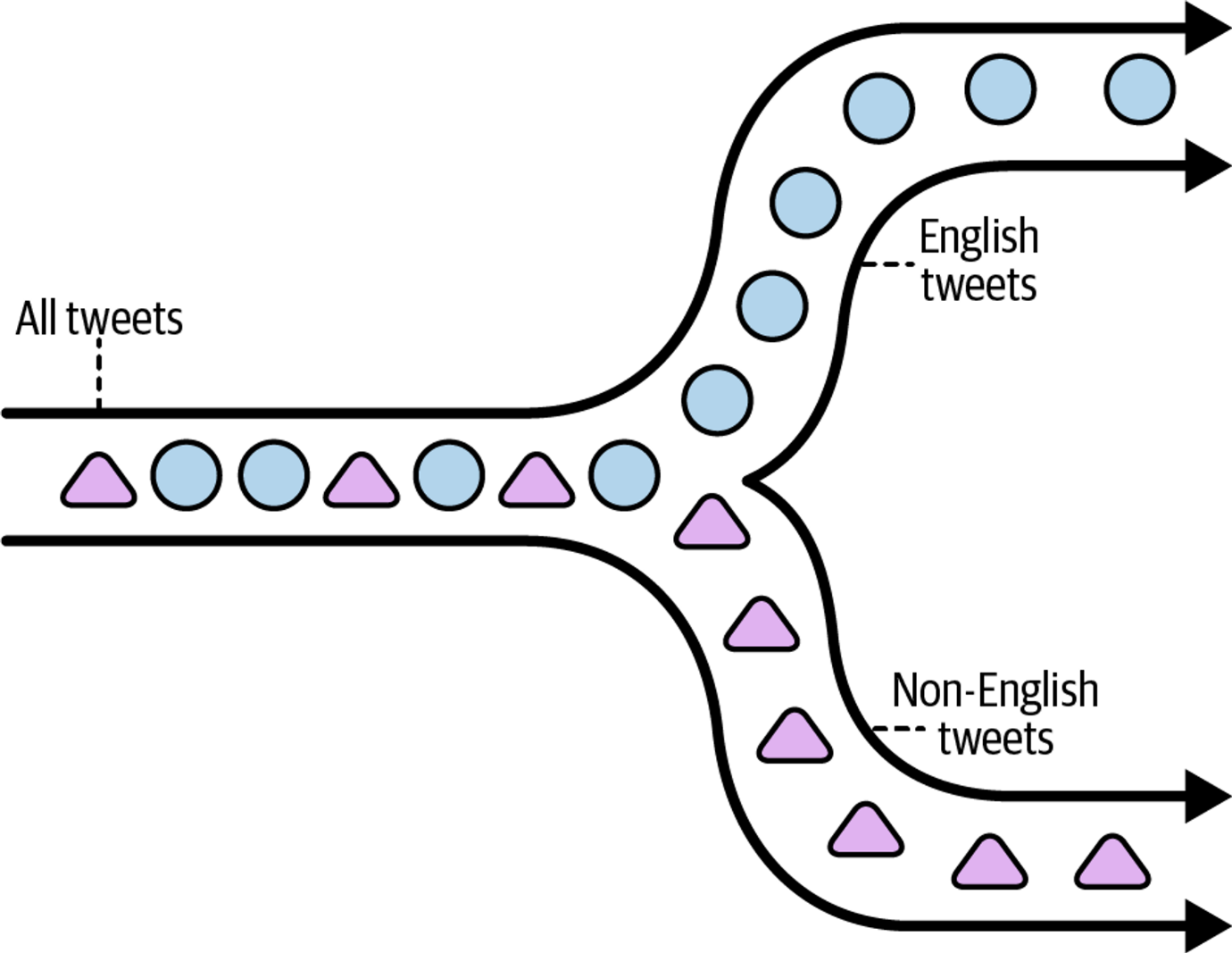
| **Data type** | **Serdes class** |
| --- | --- |
| byte[] | Serdes.ByteArray(), Serdes.Bytes() |
| ByteBuffer | Serdes.ByteBuffer() |
| Double | Serdes.Double() |
| Integer | Serdes.Integer() |
| Long | Serdes.Long() |
| String | Serdes.String() |
| UUID | Serdes.UUID() |
| Void | Serdes.Void() |

**Building a Custom Serdes**

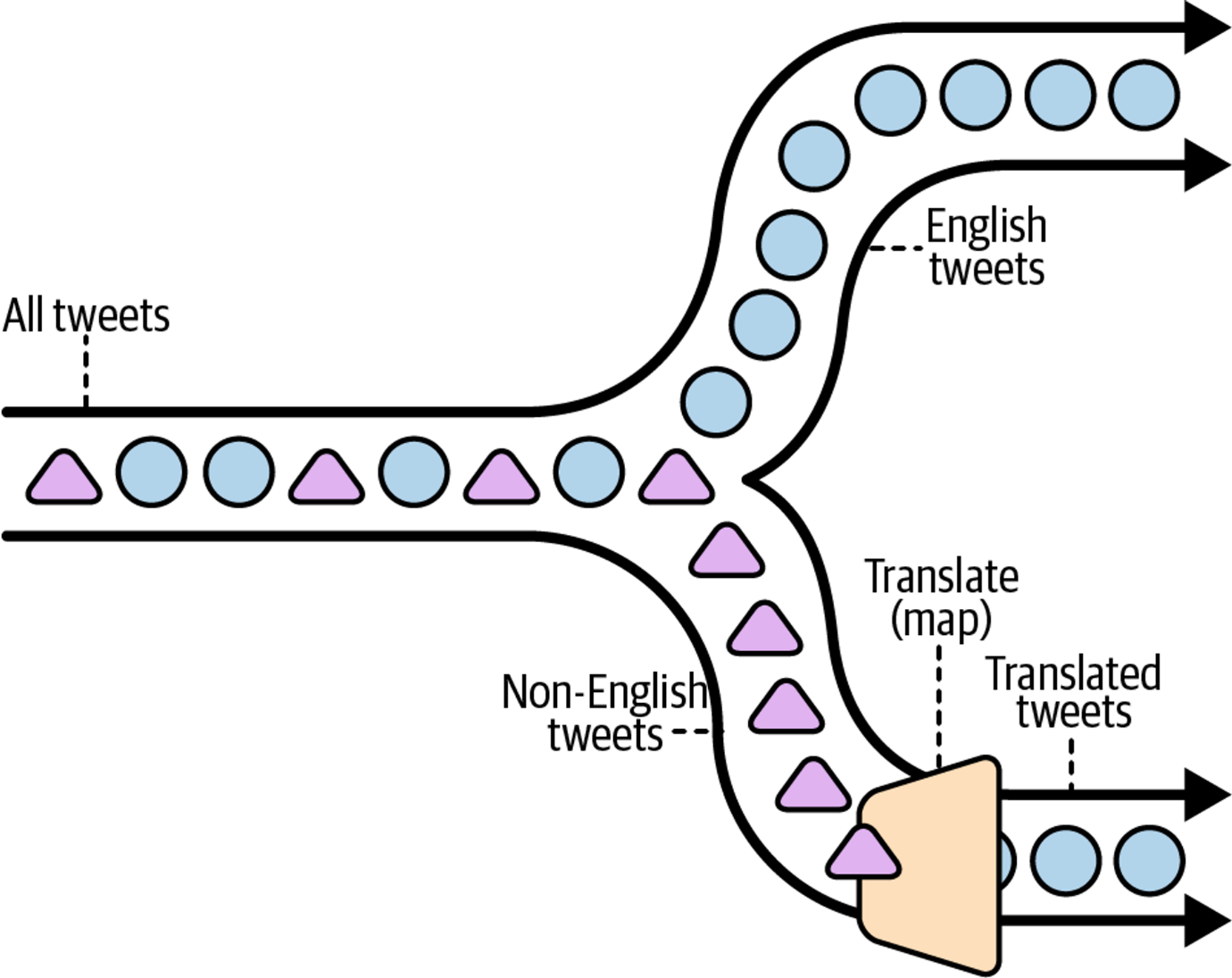
**Filtering Data**



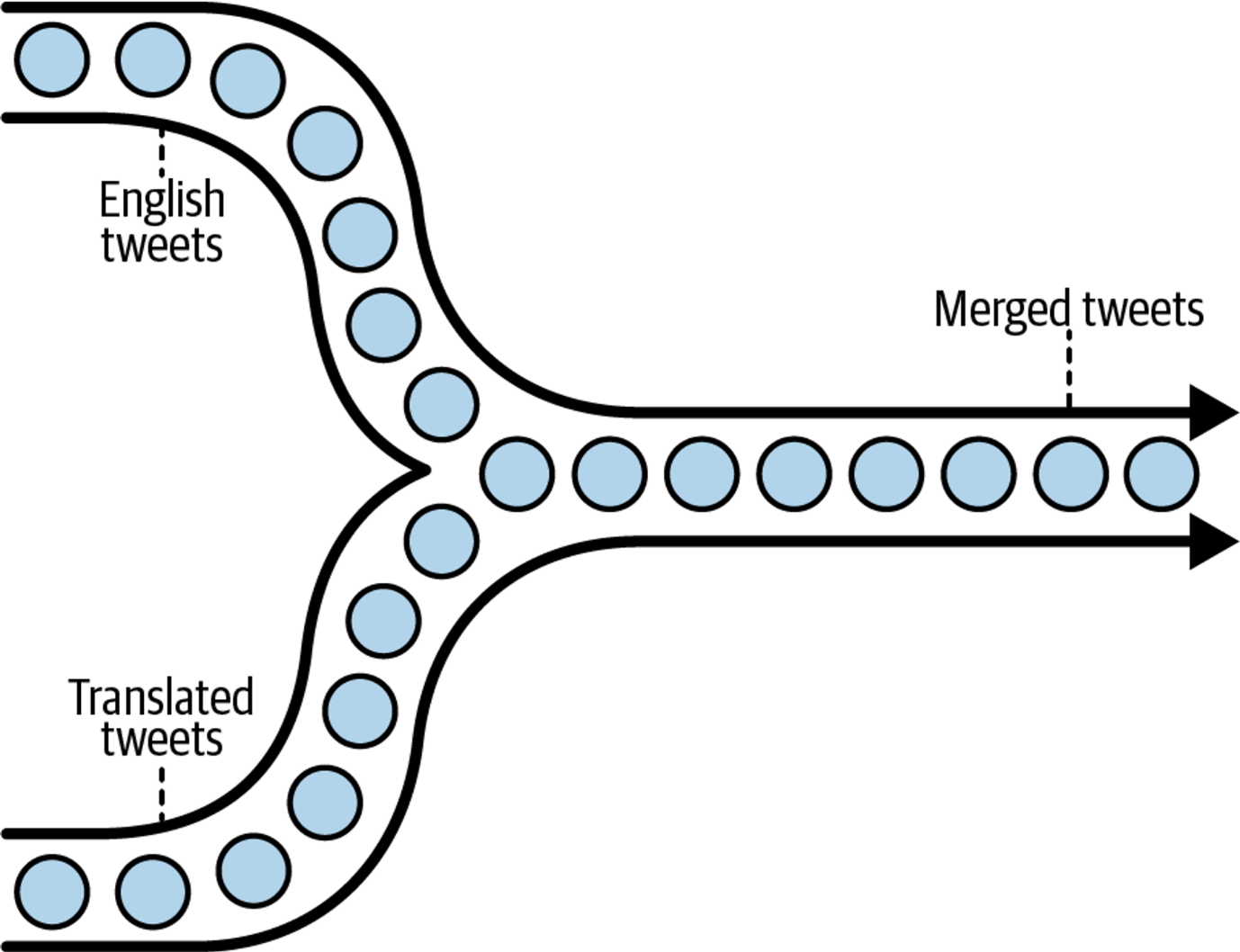
**Branching Data**



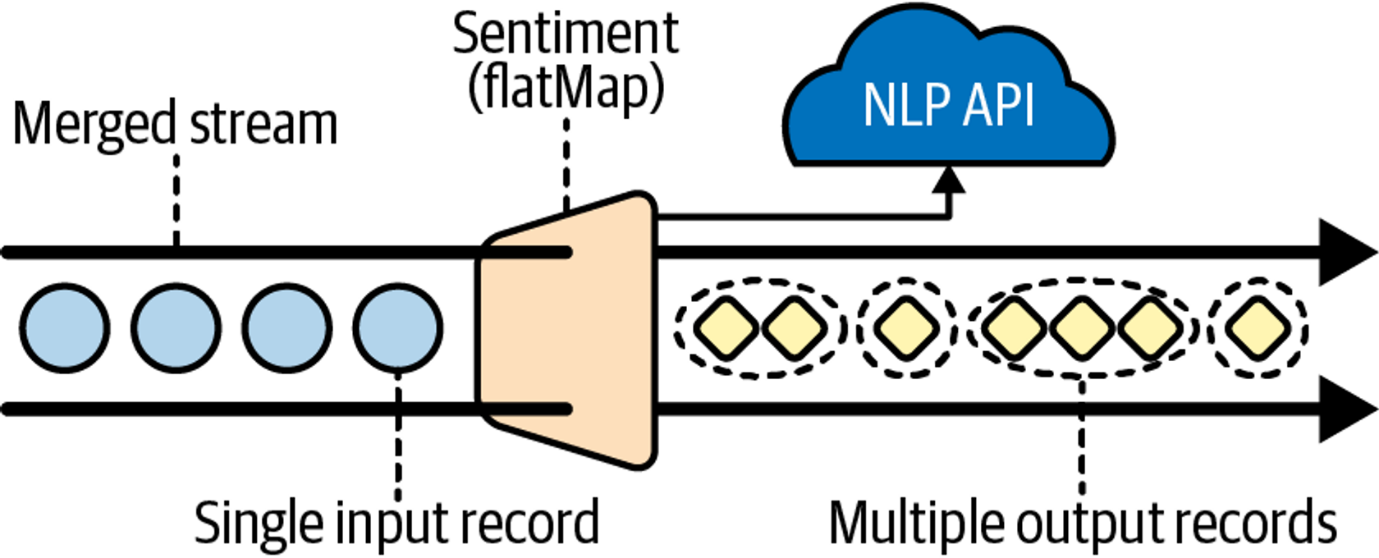
**Translating Tweets**



**Merging Streams**



**Enriching Tweets**



# Stateful Processing

### **Slide 1: Overview of Stateful Stream Processing**

* Transition from stateless to stateful processing captures and uses data over time.
* Enables complex operations like joining, aggregating, and enriching data.

### **Slide 2: Benefits of Stateful Processing**

* Recognizes patterns and behaviors across events.
* Facilitates sophisticated data enrichment and aggregation.
* Creates real-time, point-in-time snapshots (tables) for querying.

### **Slide 3: Stateful Operators and Their Applications**

* Operators handle data with awareness of past events.
* Key operators: join, aggregate, and window for in-depth data analysis.

### **Slide 4: State Management with KTable and GlobalKTable**

* **KTable**: Manages local, partitioned state for efficient processing.
* **GlobalKTable**: Maintains a full copy of the dataset for global access in joins.

### **Slide 5: Practical Applications of Stateful Processing**

* Supports interactive queries for dynamic, real-time data insights.
* Enables the development of low-latency, event-driven microservices.
* Example use case: Analyzing shopping cart abandonment to improve sales strategies.

| **Use case** | **Purpose** | **Operators** |
| --- | --- | --- |
| Joining data | Enrich an event with additional information or context that was captured in a separate stream or table | • join (inner join) |
| • leftJoin |  |  |
| • outerJoin |  |  |
| Aggregating data | Compute a continuously updating mathematical or combinatorial transformation of related events | • aggregate |
| • count |  |  |
| • reduce |  |  |
| Windowing data | Group events that have close temporal proximity | • windowedBy |

**Perhaps the most important place to begin is by looking at how state is stored and queried in Kafka Streams.**

# State Stores

### **Slide 1: Introduction to State Stores**

* **Purpose**: Enable memory of past events for stateful operations like counts and aggregates.
* **Necessity**: Essential for operations that require historical context and continuity in data processing.

### **Slide 2: Characteristics of State Stores**

* **Embedded**: Integrated directly within Kafka Streams tasks, avoiding latency and concurrency issues associated with external stores.
* **Fault Tolerance**: Backed by changelog topics in Kafka to ensure data can be restored after failures.
* **Access Modes**: Supports both read and write operations, with dedicated read-only access for safe external queries.

### **Slide 3: RocksDB and Performance**

* **Implementation**: Utilizes RocksDB, an optimized, embedded key-value store for high performance.
* **Advantages**: Fast read/write operations and flexible byte-stream storage compatible with Kafka’s serialization.

### **Slide 4: Persistent vs. In-Memory State Stores**

* **Persistent Stores**: Data is written to disk asynchronously, allowing recovery of state without full replay of events.
* **In-Memory Stores**: Stores data only in RAM for faster access but slower recovery; used when performance is critical and recovery delays can be mitigated.

### **Slide 5: Configuring and Using State Stores**

* **Configuration**: Can be customized for throughput, recovery times, or simplicity; Kafka Streams often selects sensible defaults.
* **Practical Tip**: Prefer persistent stores for robustness and consider in-memory stores when profiling indicates significant performance benefits.

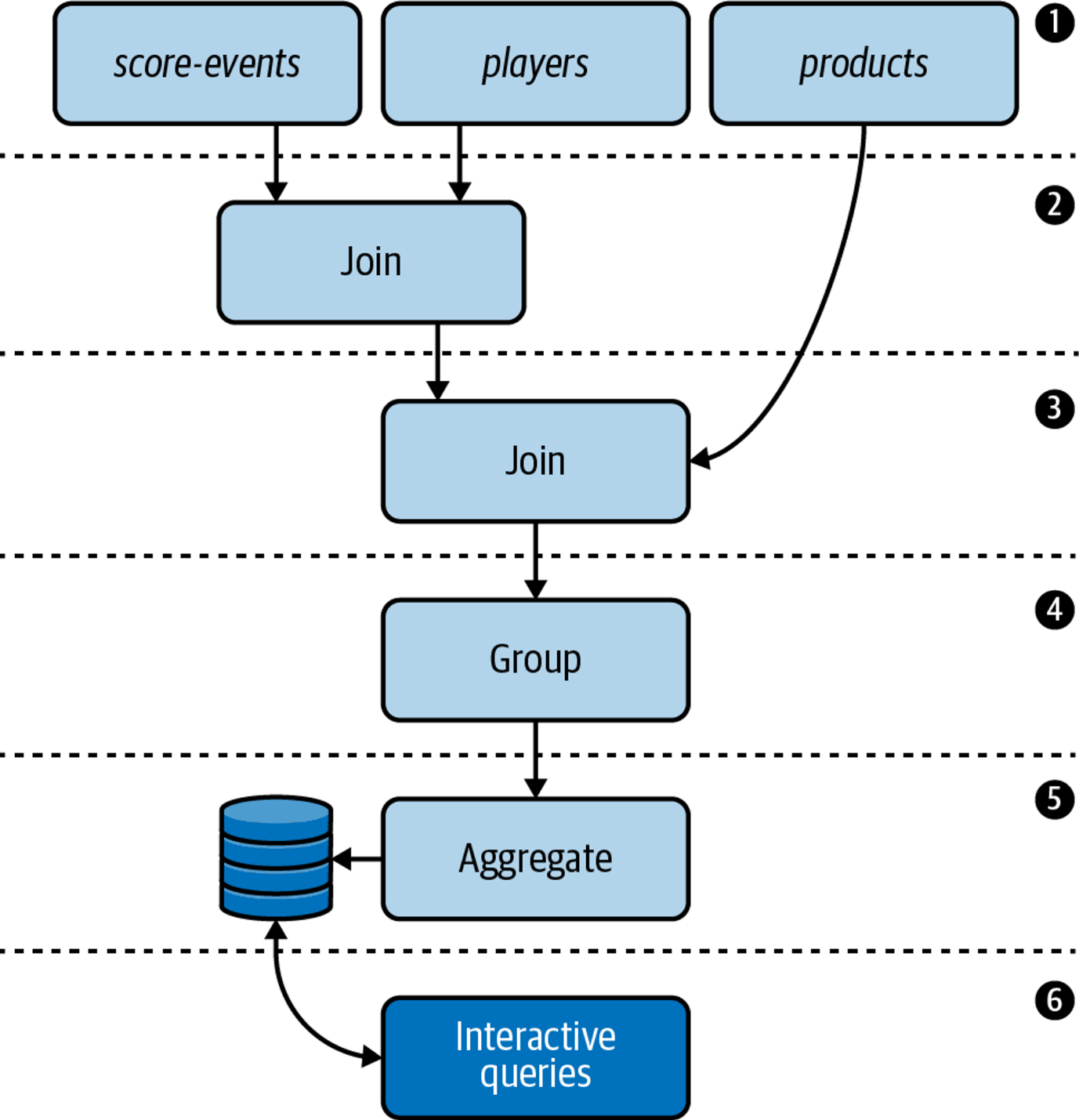
# ****Video Game Leaderboard****

### **Overview**

* **Context**: Using Kafka Streams for real-time, low-latency processing in the video game industry.
* **Application**: Building a leaderboard that updates in real-time, a common feature in competitive gaming platforms like those managed by Activision.

### **Practical Application**

This project showcases how to implement complex stream processing operations such as joins and aggregations in Kafka Streams, addressing advanced business problems like real-time leaderboards in video gaming.



1

Our Kafka cluster contains three topics:

The score-events topic contains game scores. The records are unkeyed and are therefore distributed in a round-robin fashion across the topic’s partitions.

The players topic contains player profiles. Each record is keyed by a player ID.

The products topic contains product information for various video games. Each record is keyed by a product ID.

2 We need to enrich our score events data with detailed player information. We can accomplish this using a join.

3 Once we’ve enriched the score-events data with player data, we need to add detailed product information to the resulting stream. This can also be accomplished using a join.

4 Since grouping data is a prerequisite for aggregating, we need to group the enriched stream.

5 We need to calculate the top three high scores for each game. We can use Kafka Streams’ aggregation operators for this purpose.

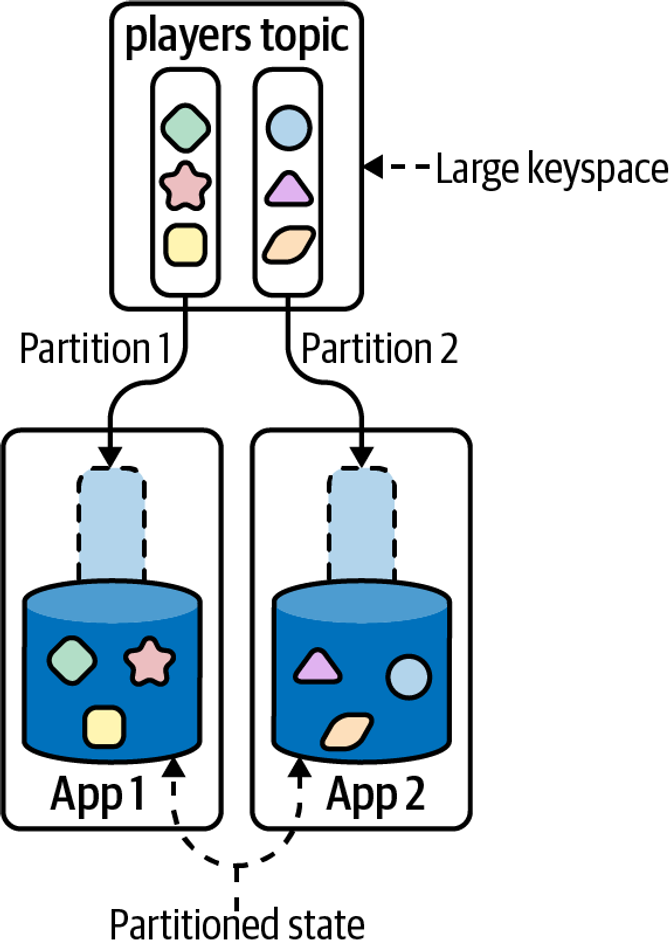
6 Finally, we need to expose the high scores for each game externally. We will accomplish this by building a RESTful microservice using the interactive queries feature in Kafka Streams.

**Adding the Source Processors**

| **Kafka topic** | **Abstraction** |
| --- | --- |
| score-events | ??? |
| players | ??? |
| products | ??? |

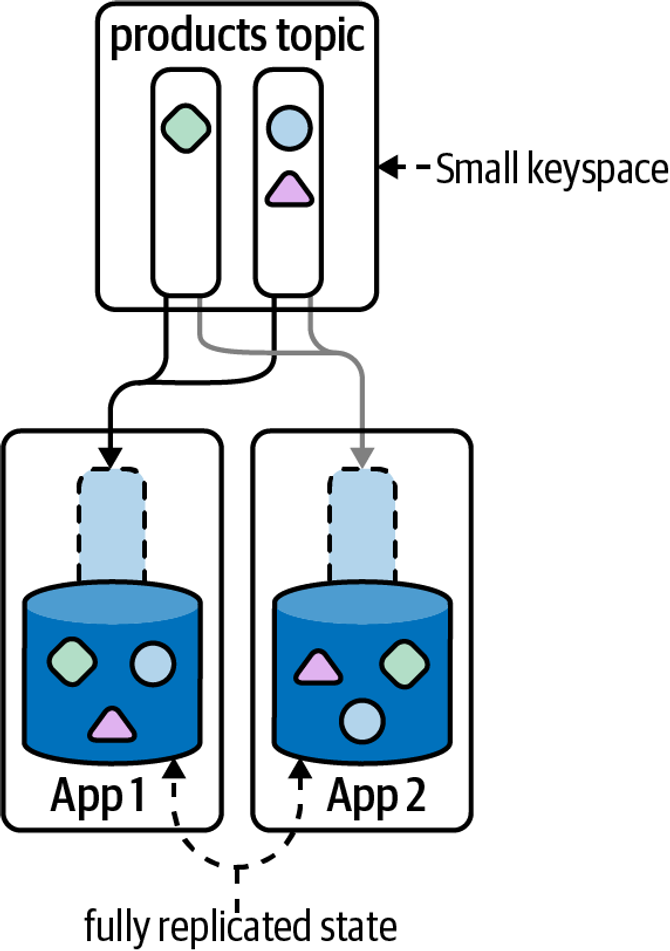
our score-events topic contains raw score events, which are unkeyed (and therefore, distributed in a round-robin fashion) in an uncompacted topic. Since tables are key-based, this is a strong indication that we should be using a KStream for our unkeyed score-events topic

| **Kafka topic** | **Abstraction** |
| --- | --- |
| score-events | KStream |
| players | ??? |
| products | ??? |



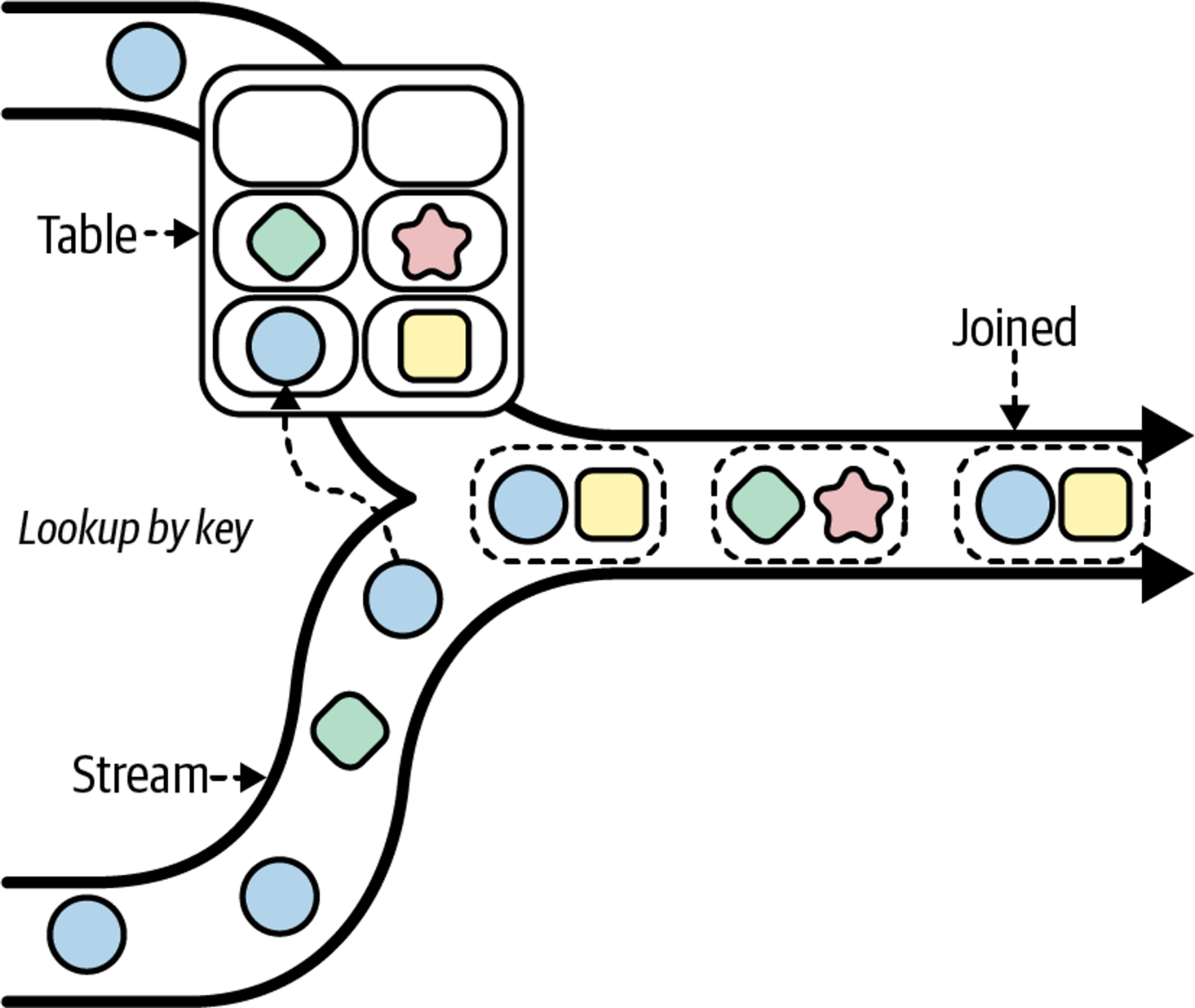
Our updated abstraction table now looks like this:

| **Kafka topic** | **Abstraction** |
| --- | --- |
| score-events | KStream |
| players | KTable |
| products | ??? |



| **Kafka topic** | **Abstraction** |
| --- | --- |
| score-events | KStream |
| players | KTable |
| products | GlobalKTable |

**Joins**



**Join Operators**

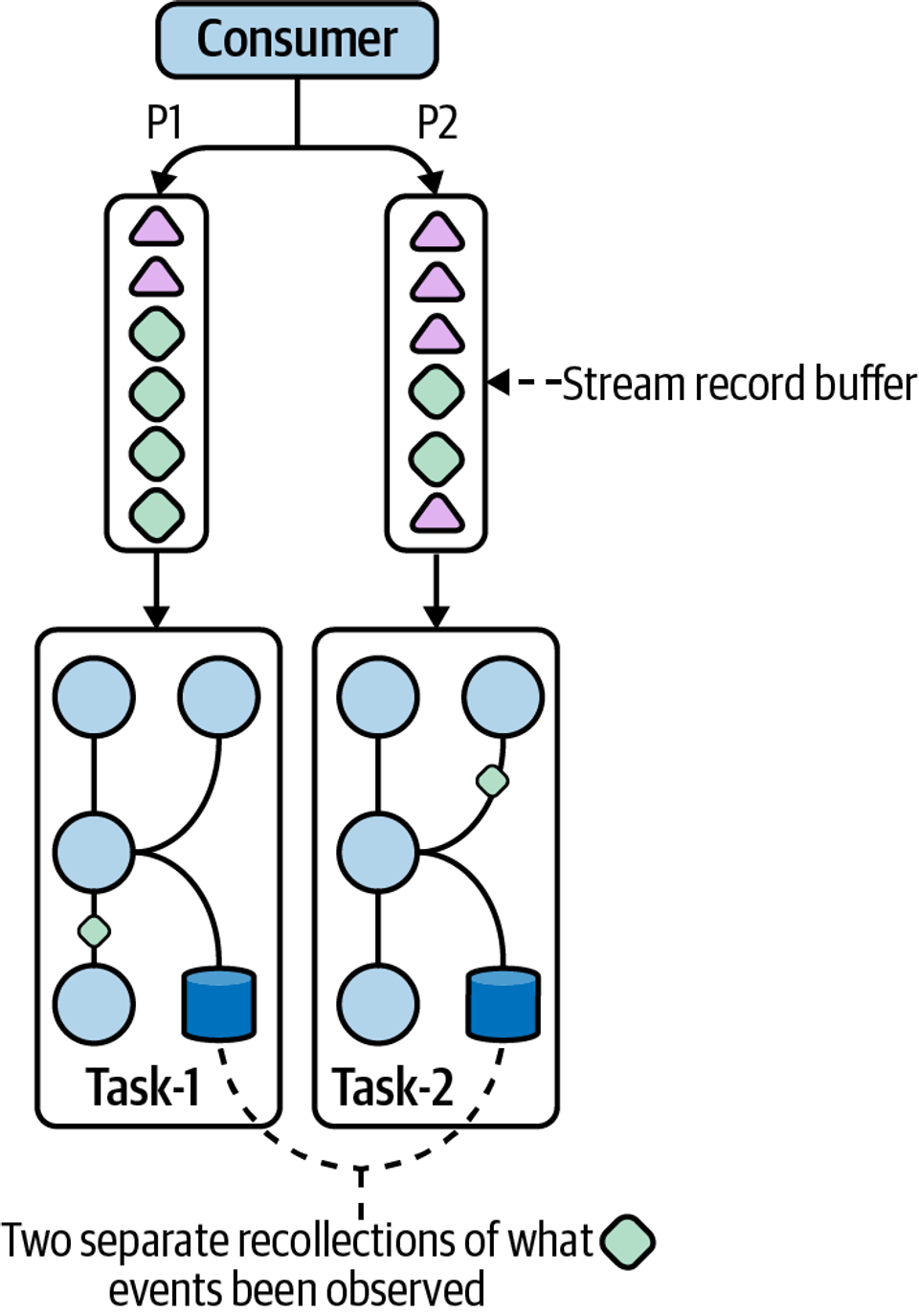
| **Operator** | **Description** |
| --- | --- |
| join | Inner join. The join is triggered when the input records on both sides of the join share the same key. |
| leftJoin | Left join. The join semantics are different depending on the type of join: |
| • For stream-table joins: a join is triggered when a record on the left side of the join is received. If there is no record with the same key on the right side of the join, then the right value is set to null. |  |
| • For stream-stream and table-table joins: same semantics as a stream-stream left join, except an input on the right side of the join can also trigger a lookup. If the right side triggers the join and there is no matching key on the left side, then the join will not produce a result. |  |
| outerJoin | Outer join. The join is triggered when a record on either side of the join is received. If there is no matching record with the same key on the opposite side of the join, then the corresponding value is set to null. |

**Join Types**

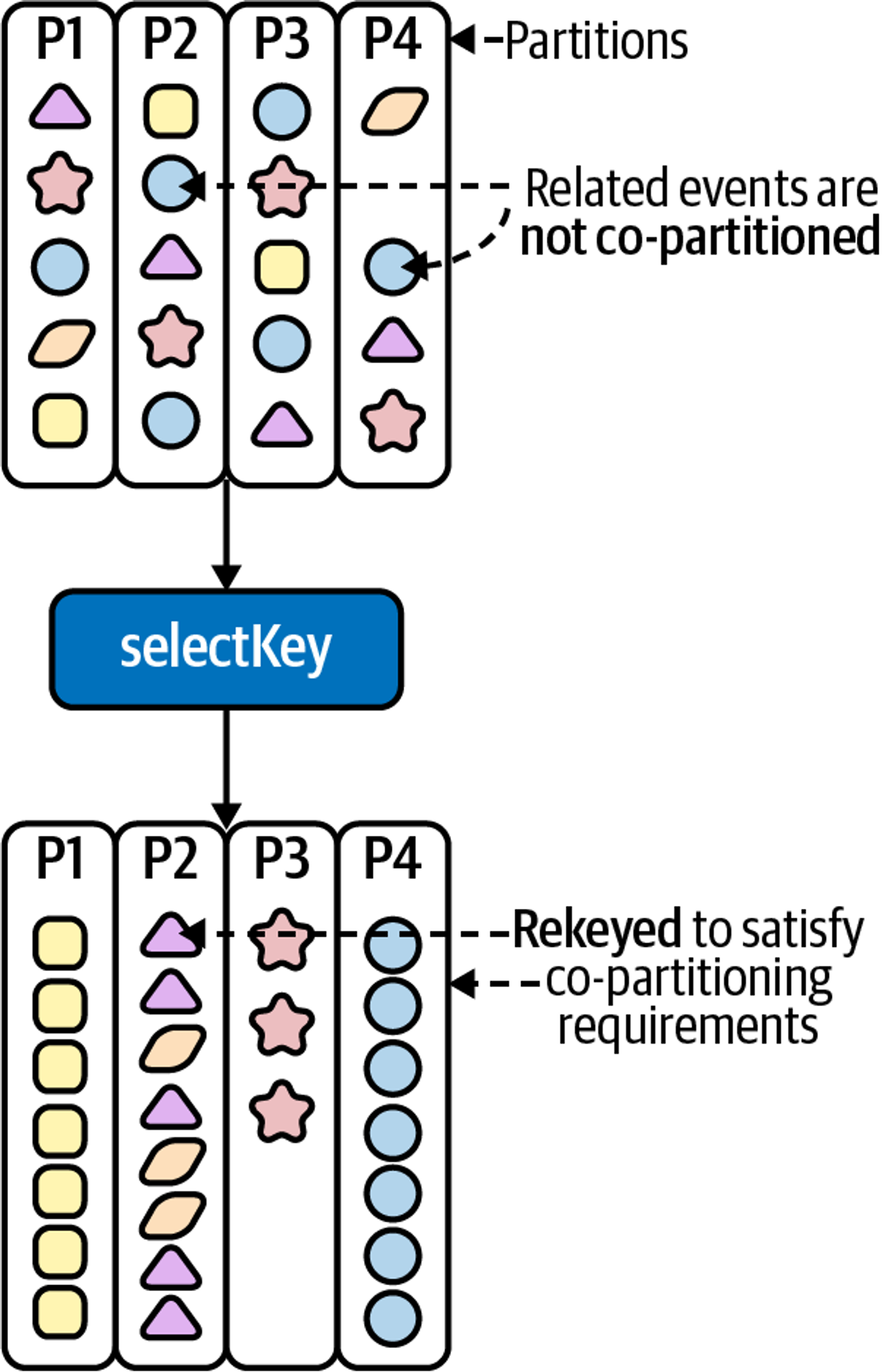
| **Type** | **Windowed** | **Operators** | **Co-partitioning required** |
| --- | --- | --- | --- |
| KStream-KStream | Yes | • join |  |
| • leftJoin |  |  |  |
| • outerJoin | Yes |  |  |
| KTable-KTable | No | • join |  |
| • leftJoin |  |  |  |
| • outerJoin | Yes |  |  |
| KStream-KTable | No | • join |  |
| • leftJoin | Yes |  |  |
| KStream-GlobalKTable | No | • join |  |
| • leftJoin | No |  |  |

**Co-Partitioning**

If we want to join related records but these records aren’t always processed by the same task, then we have an observability problem



Rekeying messages ensures related records appear on the same partition



**Value Joiners**

However, in Kafka Streams, we need to use a ValueJoiner to specify how different records should be combined. A ValueJoiner simply takes each record that is involved in the join, and produces a new, combined record. Looking at the first join, in which we need to join the score-events KStream with the players KTable, the behavior of the value joiner could be expressed using the following pseudocode:

**(scoreEvent, player) -> combine(scoreEvent, player);**

* **KStream to KTable Join (players Join)**
* **KStream to GlobalKTable Join (products Join)**

**Grouping Streams**

There are two operators that can be used for grouping a KStream:

**groupBy groupByKey**

# ****Interactive Queries****

### **Slide 1: Introduction to Stateful Processing with Kafka Streams**

* **Context**: Using Kafka Streams for real-time leaderboard processing in video games.
* **Need for Low Latency**: Critical for gaming applications like those developed by Activision for instant feedback and dynamic updates.

### **Slide 2: Materializing State for Interactive Queries**

* **Materialized Stores**: Enhance Kafka Streams by allowing state to be persisted and queried.
* **Purpose**: Enable read-only access to state for external queries, crucial for building event-driven microservices.
* **Example**: Aggregating high scores in a leaderboard where each score update adjusts the player's standing.

### **Slide 3: Configuring Materialized State Stores**

* **Using Materialized Class**: Configure state stores to be explicitly named and queryable.
* **Code Example**:
  + Define a KTable with aggregation using **Materialized.as("leader-boards")** to create a persistent key-value store.
  + Configure serialization with **withKeySerde(Serdes.String())** and **withValueSerde(JsonSerdes.HighScores())**.

### **Slide 4: Accessing and Querying State Stores**

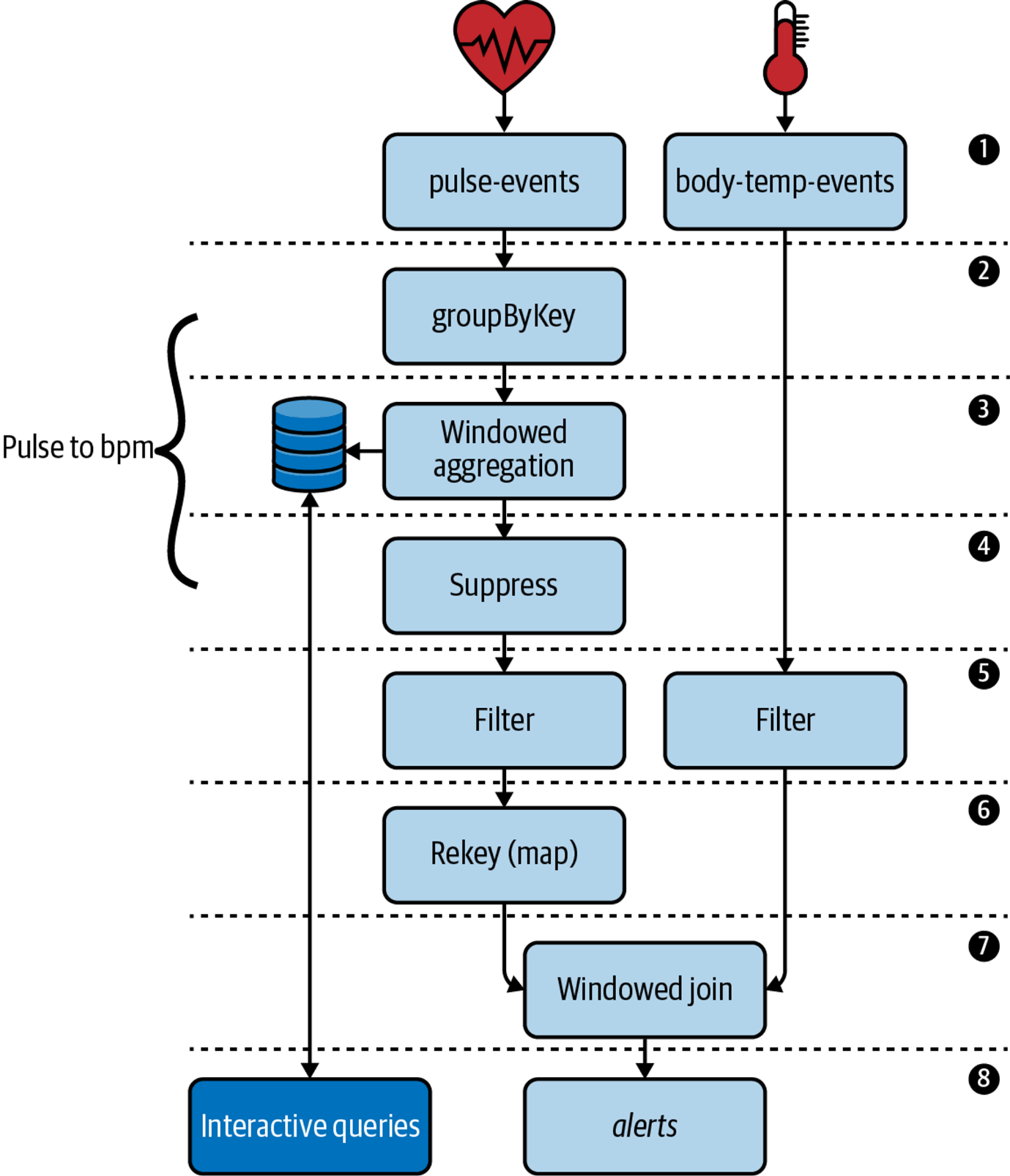
* **Read-Only Access**: Retrieving state in a safe, read-only mode to prevent modifications outside of stream processing.
* **Queryable Types**: Support for various store types including key-value, windowed, and session stores.
* **Retrieving Stores**: Use **streams.store()** with **StoreQueryParameters.fromNameAndType()** to access the materialized state.

### **Slide 5: Interactive Queries in Action**

* **Local vs. Remote Queries**: Local instances query their own state, but remote queries are needed for a complete view across distributed instances.
* **Implementing RESTful Microservices**: Example using Javalin for serving REST requests and OkHttp for making REST calls.
* **Dynamic State Queries**: Demonstrating point lookups (**get()**), range scans (**range()**), and counting entries (**approximateNumEntries()**).

**Windows and Time**

**Patient Monitoring Application**



### **Slide 1: Introduction to Patient Vitals Monitoring**

* **Overview**: Utilizing Kafka Streams to monitor health conditions in real-time by processing data from heartbeat and temperature sensors.
* **Data Sources**:
  + **pulse-events**: Captures heartbeat data, keyed by patient ID.
  + **body-temp-events**: Records body temperature measurements, also keyed by patient ID.

### **Slide 2: Processing Pulse Events**

* **Objective**: Convert raw pulse data into heart rate measured in beats per minute (bpm).
* **Data Grouping**: Grouping records by patient ID to prepare for aggregation.
* **Windowed Aggregation**: Calculate heart rate using a 60-second window to match the bpm measurement interval.

### **Slide 3: Suppressing Intermediate Results**

* **Use of Suppress Operator**: Ensures only the final result of each bpm computation window is emitted.
* **Purpose**: Reduces noise in the data stream and focuses on final, stable bpm measurements for accurate health assessment.

### **Slide 4: Detecting Health Alerts**

* **Thresholds for Alerts**:
  + Heart rate ≥ 100 bpm.
  + Body temperature ≥ 100.4°F.
* **Filtering Strategy**: Apply filters on both vitals streams to identify measurements that indicate potential health risks.

### **Slide 5: Rekeying and Joining Vitals Data**

* **Rekeying for Co-Partitioning**: Adjust heart rate records' keys to ensure they align with temperature records for accurate joins.
* **Windowed Join**: Merge filtered pulse and temperature data to detect conditions indicating possible infections or systemic inflammatory response syndrome (SIRS).

### **Slide 6: Exposing Results and Alerting**

* **Interactive Queries**: Allow real-time querying of the aggregated heart rate data.
* **Alerts Output**: Stream results indicating potential health issues to a dedicated **alerts** topic for immediate response.

### **Slide 7: Conclusion**

* **Impact**: Enhanced real-time health monitoring enables prompt detection and response to critical health changes.
* **Future Steps**: Expand monitoring capabilities and refine detection algorithms based on ongoing data analysis and healthcare provider feedback.

**Time Semantics**

Event time

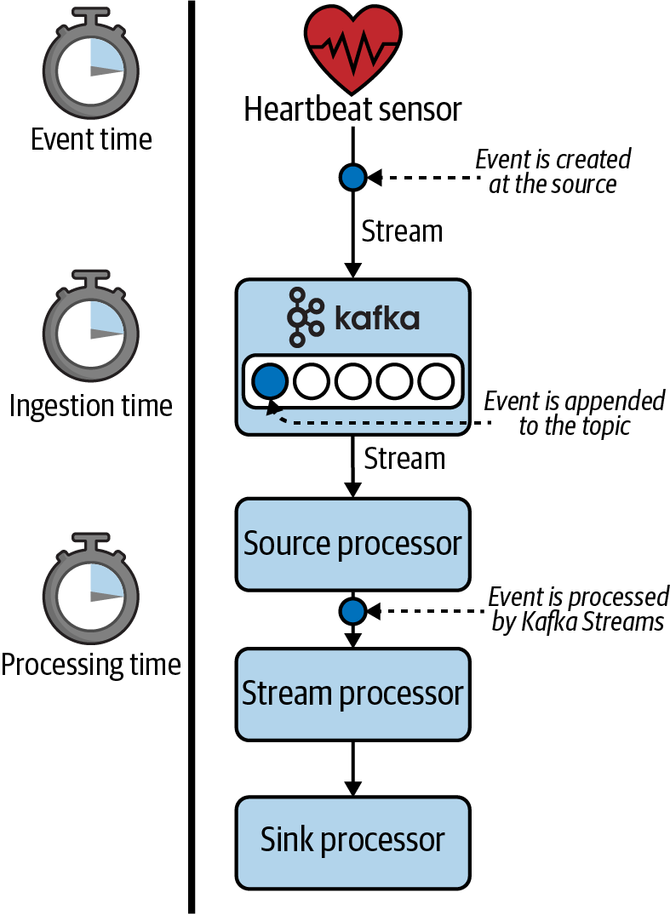
**When an event was created at the source. This timestamp can be embedded in the payload of an event, or set directly using the Kafka producer client as of version 0.10.0.**

Ingestion time

**When the event is appended to a topic on a Kafka broker. This always occurs after event time.**

Processing time

**When the event is processed by your Kafka Streams application. This always occurs after event time and ingestion time. It is less static than event time, and reprocessing the same data (i.e., for bug fixes) will lead to new processing timestamps, and therefore nondeterministic windowing behavior.**



**log.message.timestamp.type (broker level) message.timestamp.type (topic level)**

### **Slide 1: Introduction to Timestamp Extractors**

* **Purpose**: Timestamp extractors associate each Kafka record with a timestamp, crucial for time-dependent operations like windowed joins and aggregations.
* **Functionality**: Extracts timestamps either from the record itself or uses the partition's latest timestamp as a fallback.

### **Slide 2: Default Timestamp Extractors in Kafka Streams**

* **FailOnInvalidTimestamp**: Default extractor that throws an exception if timestamps are invalid, typically set to event creation or ingestion time.
* **LogAndSkipOnInvalidTimestamp**: Similar to the default but logs a warning and skips invalid timestamps, allowing processing to continue.
* **WallclockTimestampExtractor**: Uses system time for timestamps, suitable for processing-time semantics but may not accurately reflect event times.

### **Slide 3: Importance of Accurate Timestamping**

* **Scenario**: Monitoring patient vitals like heart rate and body temperature where accurate timing of data is crucial.
* **Challenge**: Ensuring that windowed operations like calculating heart rate per minute accurately reflect the time events occur, not just when they are processed.

### **Slide 4: Implementing Custom Timestamp Extractors**

* **Custom Extractor Logic**: Extract timestamps embedded in event payloads, essential for precise event-time processing.
* **Example Implementation**:
  + Check if the event has a valid timestamp; if not, fallback to the latest known partition time.
  + Use the **VitalTimestampExtractor** to parse and convert embedded time data from records.

### **Slide 5: Integrating Timestamp Extractors with Kafka Streams**

* **Setup**: Override the default timestamp extractor by configuring **StreamsConfig.DEFAULT\_TIMESTAMP\_EXTRACTOR\_CLASS\_CONFIG**.
* **Application**: Apply the custom timestamp extractor to specific streams to ensure all records are appropriately timestamped for subsequent processing steps.

### **Slide 6: Practical Use Case - Patient Health Monitoring**

* **Data Flow**:
  1. Raw pulse and temperature data arrive at their respective topics, each keyed by patient ID.
  2. Data is timestamped using **VitalTimestampExtractor** to ensure accurate event times.
  3. Timestamped data is used in windowed aggregations to compute metrics like average heart rate.
* **Benefits**: Enables precise monitoring and timely health alerts based on real-time data analysis.

**Window Streaming**

### **Slide 1: Introduction to Windowing in Kafka Streams**

* **Purpose**: Utilize windowing to aggregate and join records based on specific time intervals.
* **Context**: Example of converting raw pulse data into a heart rate measured in beats per minute (bpm).
* **Key Concept**: Windowing allows for grouping records that are close in time, essential for time-dependent aggregations.

### **Slide 2: Types of Windows in Kafka Streams**

* **Tumbling Windows**: Fixed-size, non-overlapping windows that reset at regular intervals. Ideal for cases where each window is an independent time frame.
* **Hopping Windows**: Fixed-size, overlapping windows that advance at specified intervals. Useful for capturing changes over time with some data redundancy.
* **Session Windows**: Dynamic windows that group events by activity sessions. Windows extend with each related event until a timeout period.
* **Sliding Windows**: Used for join operations, where two records fall into the same window if they meet the timing condition.

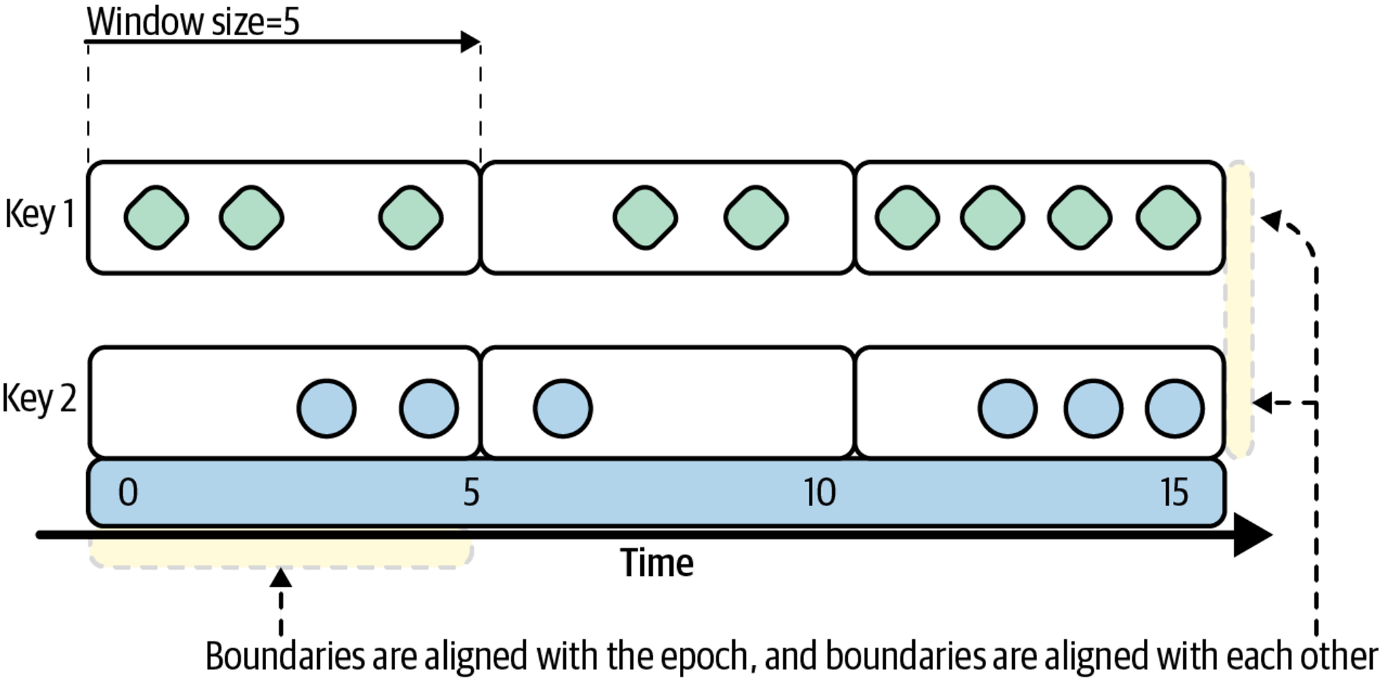
### **Slide 3: Challenges with Windowing**

* **Window Alignment**: Tumbling and hopping windows align with the epoch, affecting how records are grouped by time.
* **Data Sparsity and Overlap**: Hopping windows can lead to redundant processing due to overlapping periods.
* **Session Gaps**: Session windows require careful definition of inactivity gaps to accurately reflect user sessions.

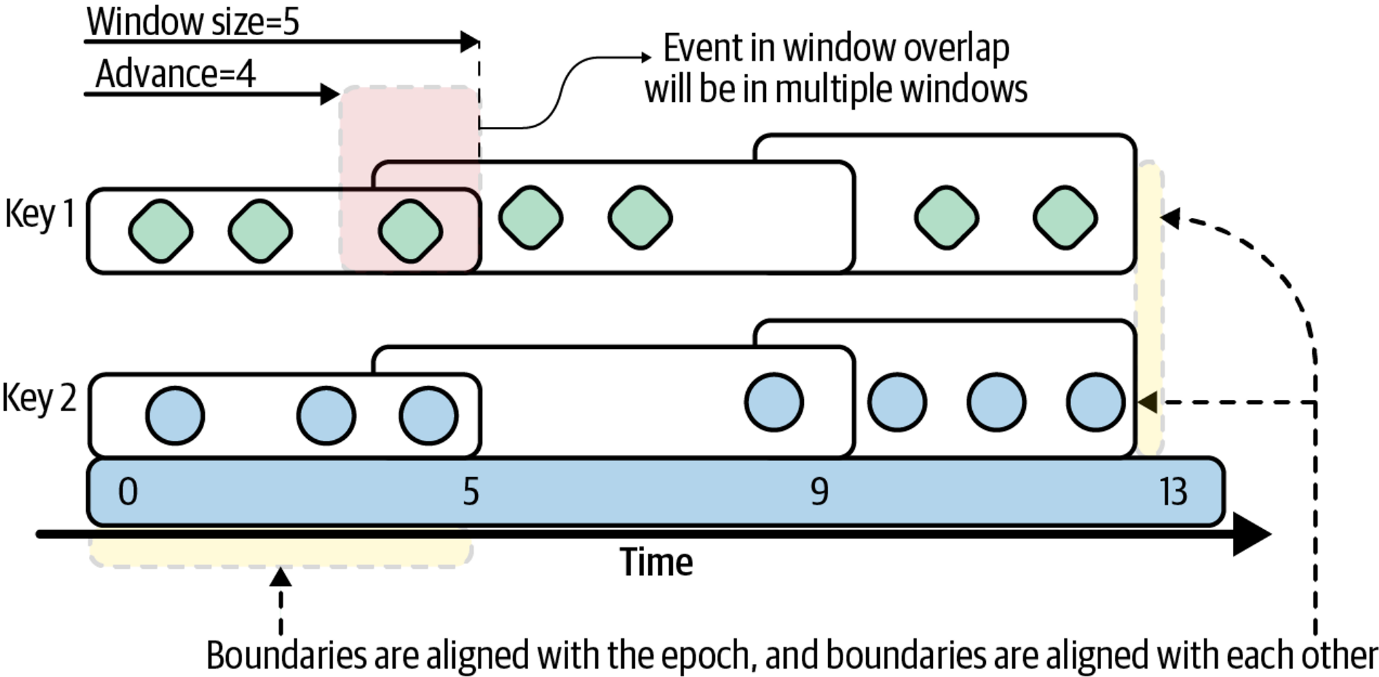
### **Slide 4: Advanced Windowing: Suppressing Intermediate Results**

* **Need**: Avoid intermediate result emissions during window processing, which can mislead in cases like heart rate monitoring.
* **Solution**: Use the **suppress** operator to emit only the final result of each window, ensuring outputs reflect completed time frames.

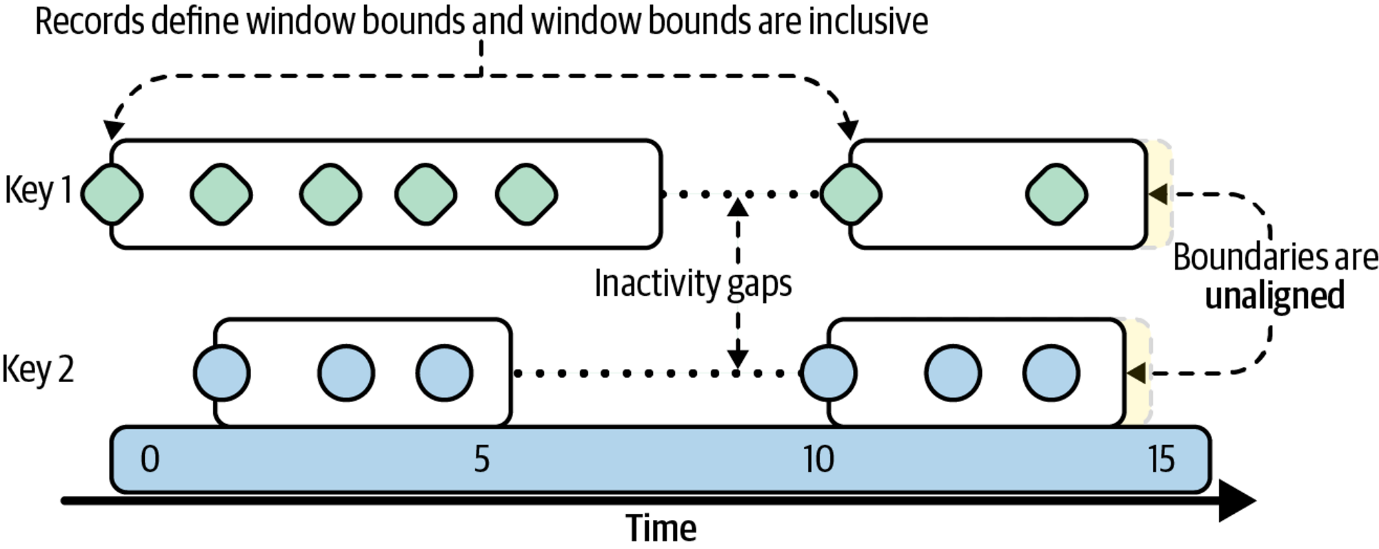
### **Tumbling Window**



**Hopping Wondow**



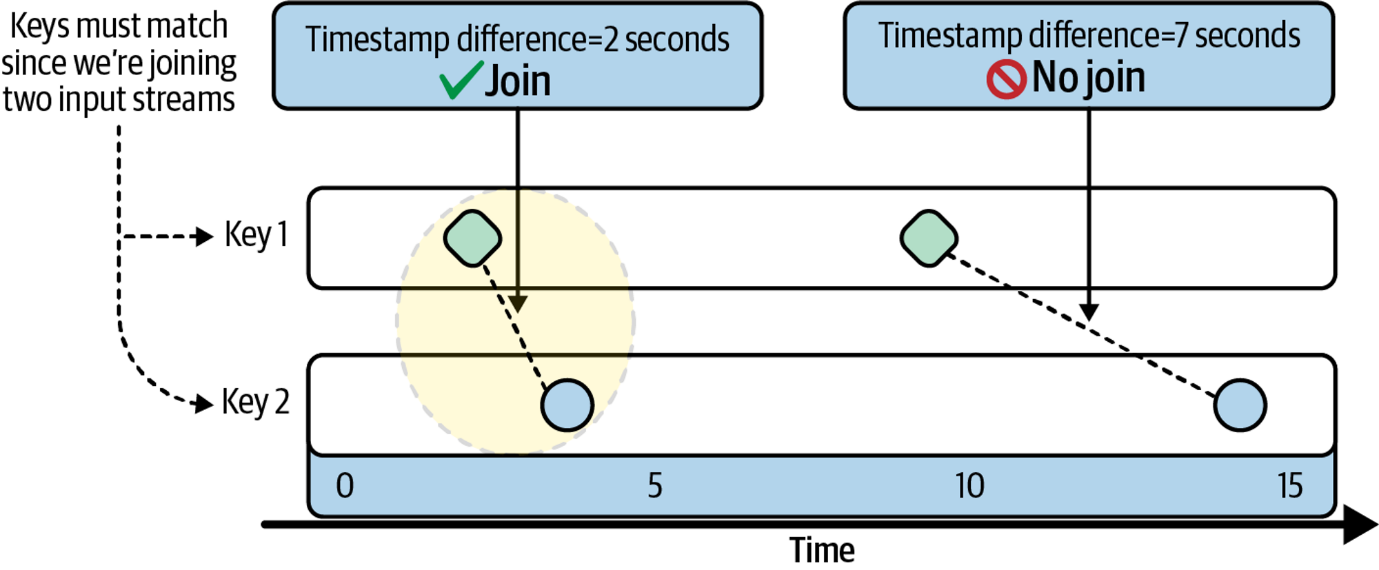
**Session windows**



**Sliding join windows**

**JoinWindows joinWindow = JoinWindows .of(Duration.ofSeconds(5));**

**Timestamps must be less than or equal to five seconds apart to fall within the same window**



# Advanced State Management

### **Slide 1: Introduction to Advanced State Management**

* **Goal**: Enhance reliability and performance of stateful stream processing applications.

### **Slide 2: Persistent Store Disk Layout**

* **Default Storage**: Located at **/tmp/kafka-streams**, configurable via **StreamsConfig.STATE\_DIR\_CONFIG**.
* **Inspection Benefits**: Allows for direct observation and troubleshooting from disk file structure.

### **Slide 3: Fault Tolerance Mechanisms**

* **Changelog Topics**: Essential for state recovery; capture every state update.
* **Standby Replicas**: Reduce downtime by keeping backup copies of state, facilitating faster recovery.

### **Slide 4: Rebalancing and Its Impact**

* **Trigger**: Occurs due to consumer group membership changes.
* **Strategies**:
  + Prevent unnecessary state migrations.
  + Minimize state recovery time during rebalancing.

# Emitting Window Results

### **Slide 1: Introduction to Window Emission Complexity**

* **Key Challenges**: Handling unbounded event streams that may arrive out of order or be delayed.
* **Event-Time Semantics**: Events aren't always processed in the order they occur, complicating window-based computations.

### **Slide 2: Issues with Window Result Emission**

* **Out-of-Order Data**: Events may arrive out of timestamp order due to network issues or multiple producers.
* **Delayed Events**: Late-arriving data can disrupt the accuracy of window computations.
* **Example Scenario**: Patient vitals monitoring where intermittent networking issues cause delayed data entries.

### **Slide 3: Trade-offs in Window Emission**

* **Completeness vs. Latency**: Deciding whether to wait for all data (optimizing for completeness) or emit results as they come (reducing latency).
* **Impact of Choices**: Affects the timeliness and accuracy of alerts for critical patient conditions.

### **Slide 4: Managing Delayed and Out-of-Order Data**

* **Grace Periods**: Configuring windows to remain open longer to accommodate late data.

TimeWindows.of(Duration.ofSeconds(60)).grace(Duration.ofSeconds(5))

* **Use Case**: Allows heart rate calculations to include data that arrives up to five seconds late.

### **Slide 5: Suppressing Intermediate Window Results**

* **Continuous Refinement**: By default, Kafka Streams emits updates with each new event, leading to intermediate results.
* **Suppression Techniques**: Utilizing the **suppress** operator to emit only the final result of a window.

.suppress(Suppressed.untilWindowCloses(BufferConfig.unbounded().shutDownWhenFull()))

### **Slide 6: Implementing Suppression in Kafka Streams**

* **Configurations**:
  + **Buffer Configs**: How suppressed events are stored in memory.
  + **Buffer Full Strategies**: Handling scenarios when the buffer reaches capacity.
* **Strategic Choices**:
  + Use **Suppressed.untilWindowCloses** for emitting final results only.
  + Configure **BufferConfig.unbounded()** to ensure all events are captured.

### **Slide 7: Practical Application in Patient Monitoring**

* **Final Implementation**: Suppressing intermediate results to ensure only accurate, complete heart rate calculations are reported.
* **System Behavior**:
  + If the buffer is full, the system shuts down to prevent inaccurate data emission.
  + Ensures patient safety by providing reliable data for critical health decisions.

### **Slide 5: Advanced Configurations and Techniques**

* **Customizing State Stores**: Modifying changelog topics, disabling/enabling logging.
* **Operational Optimizations**:
  + Aggressive topic compaction to reduce state size.
  + Using tombstones to clean up old state data.
  + Configuring Kafka Streams to enhance performance and reduce latency.

### **Persistent Stores**

* **Function**: Persistent stores in Kafka Streams are used to hold the state of an application locally on disk. This is critical for stateful operations such as aggregations, joins, and windowing.
* **Location**: By default, these are stored in the **/tmp/kafka-streams** directory, although this can be configured to a different path to ensure data persistence across reboots.
* **Fault Tolerance**: While persistent stores provide a local cache of state, they are inherently limited to the machine they reside on, meaning that any local failures require state recovery.
* **Types**: Kafka Streams supports both in-memory and disk-based (persistent) state stores. Persistent stores are preferred for large states or states that must survive application restarts.

### **Changelog Topics**

* **Function**: Changelog topics serve as a Kafka topic that logs every change made to the state store. They are crucial for restoring state after a failure.
* **Fault Tolerance**: Changelog topics leverage Kafka's native replication to ensure that state changes are preserved even if the application instance fails, allowing full state recovery.
* **Recovery**: In the event of an application restart or failure, Kafka Streams uses the changelog topic to replay and restore the state to its most recent condition.
* **Performance**: Writing to a changelog topic introduces some overhead, as each state modification must be sent to Kafka. However, this is critical for ensuring the durability and fault tolerance of the state.

### **Interaction and Use Case**

* **Persistent Store as Local Cache**: Acts as the immediate storage for state operations, providing fast read and write capabilities required for stream processing tasks.
* **Changelog Topic as Backup**: Ensures that all changes to the persistent store are recorded externally in Kafka, which is essential for recovery scenarios. It's effectively a backup mechanism.
* **Complementary Roles**: In operation, persistent stores and changelog topics complement each other. While the persistent store provides quick access to current state data, the changelog ensures that this state can be rebuilt after a failure.

### **Rebalancing: Enemy of the State (Store)**

### **Understanding Failure and Rebalancing in Kafka Streams**

* **Changelog Topics**: Used for reconstructing state post-failure. Every change to the state store is recorded, allowing Kafka Streams to restore state from these logs.
* **Standby Replicas**: Reduce recovery time by maintaining ready-to-go copies of state stores on other instances. They help ensure quick state recovery when the primary instance fails.

### **Impact of State Store Loss**

* **Operational Disruption**: Losing access to a state store, even temporarily, can disrupt operations significantly, especially in applications that are heavily dependent on state.
* **Replay Time**: Replaying messages from a large changelog topic to restore state can be time-consuming, taking from minutes to hours depending on the topic size.

### **Rebalancing: Key Challenges and Terminology**

* **Rebalancing**: Occurs when Kafka redistributes work across consumer group members due to changes like membership alterations.
* **Group Coordinator**: A broker responsible for managing consumer group membership and initiating rebalances.
* **Group Leader**: A consumer that assigns partitions within the consumer group during a rebalance.

### **Strategies to Minimize Rebalancing Impact**

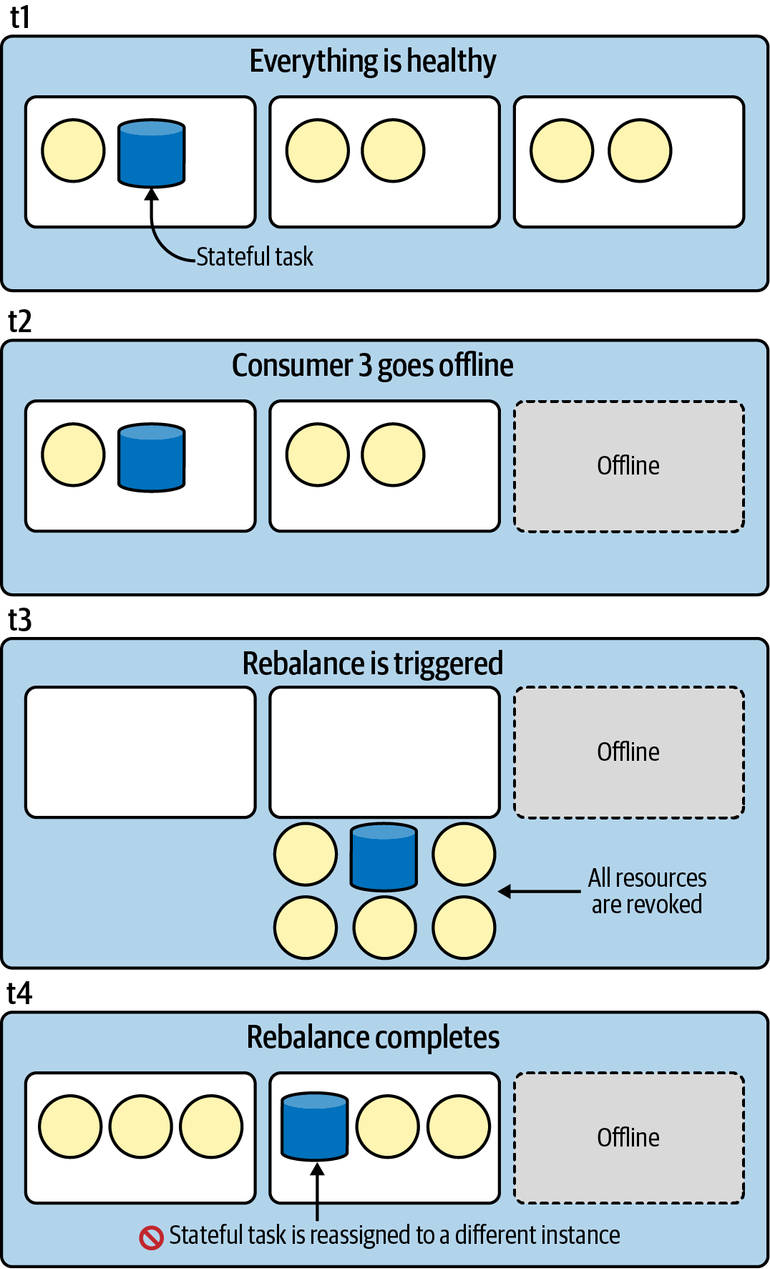
1. **Prevent State Migration**:
   * **Objective**: Avoid moving stateful tasks as much as possible to prevent the need for state reinitialization.
   * **Sticky Assignments**: Kafka Streams attempts to assign stateful tasks to the same instance across rebalances to minimize state transfer.
2. **Speed Up Recovery**:
   * **Quick Recovery Techniques**: Use standby replicas to quickly restore state without needing to replay the entire changelog.
   * **Configuration Settings**: Adjust settings like replication factor and checkpoint intervals to balance between recovery speed and resource usage.

### **Actions to Minimize Rebalancing Effects**

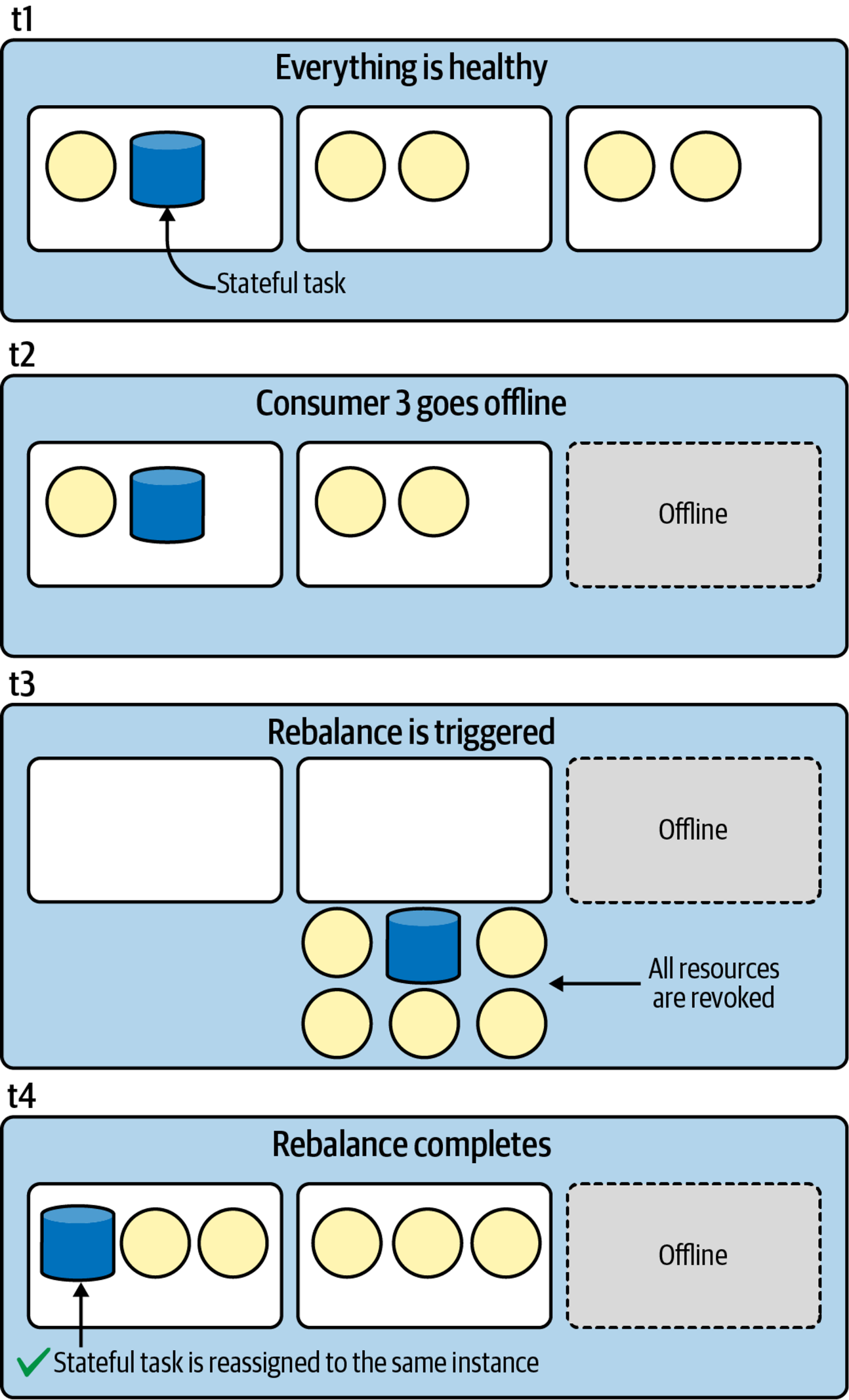
* **Proactive Measures**: Kafka Streams employs techniques like sticky task assignment to reduce the likelihood of state migration during rebalances.
* **Reactive Measures**: When rebalancing is unavoidable, Kafka Streams leverages standby replicas and optimized recovery processes to minimize downtime and performance impacts.

**Preventing State Migration**

**Nonsticky partition assignment**



**Sticky partition assignment using Kafka Streams’ built-in partition assignor**



# ****Processor API****

### **When to Use the Processor API**

* **Access to Record Metadata**: Useful for accessing specific data like topic, partition, and offset.
* **Periodic Functions**: Ability to schedule functions that perform operations at regular intervals.
* **Control Over Record Forwarding**: More precise control over how and when records are forwarded to downstream processors.
* **Granular State Store Access**: Enhanced ability to interact directly with state stores.
* **Overcoming DSL Limitations**: Ideal for scenarios where the DSL might be restrictive.

### **Adding Processors Using the Processor API**

* **Source Processors**: Attach to source topics to read incoming data streams.
* **Stream Processors**: Perform transformations or computations on the stream.
* **Sink Processors**: Send processed data to output topics or external systems.

### **Scheduling Periodic Functions**

* Utilize punctuators to execute code at a set interval, helping with tasks like cleaning up state stores or emitting periodic updates.

### **Mixing Processor API with DSL**

* The flexibility of Kafka Streams allows for using both Processor API and DSL within the same application, enabling a balanced approach that leverages the simplicity of the DSL and the power of the Processor API for complex processing needs.

### **Difference Between Processors and Transformers**

* **Processors**: Operate within the Processor API, allowing full access to the processing context and the ability to maintain state.
* **Transformers**: More focused on transforming data streams without the need to interact directly with the state or the entire Kafka Streams context.

# KsqlDB

### **What is ksqlDB?**

* **Open Source Database**: Launched by Confluent in 2017, designed specifically for event streaming.
* **Integration of Kafka Components**: Combines Kafka Connect and Kafka Streams, simplifying stream processing under a unified SQL interface.

### **Key Features of ksqlDB**

* **Data Modeling**: Supports defining data as streams or tables via SQL.
* **SQL Operations**: Enables SQL-based operations like join, aggregate, transform, filter, and window without needing Java coding.
* **Push Queries**: Continuously running queries that emit results to clients as new data arrives, useful for event-driven applications.
* **Materialized Views**: Allows creation and querying of real-time views from streams or tables using pull queries for on-demand data retrieval.
* **Connectors**: Facilitates integration with external data systems, enhancing capabilities for streaming ETL workflows.

### **Advantages of Using ksqlDB**

* **Ease of Use**: High-level SQL interface makes stream processing accessible to non-programmers.
* **Efficiency**: Integrates core Kafka functionalities into a single platform, reducing complexity.
* **Flexibility**: Suitable for a wide array of applications from microservices to complex ETL pipelines.

### **When to Use ksqlDB**

### **Simplified Stream Processing**

* **High-Level Abstraction**: Easier to use than lower-level APIs with a SQL interface.
* **Interactive Workflows**: Managed runtime with CLI and REST services for dynamic query submissions.

### **Developer Productivity and Maintainability**

* **Less Code**: Stream topologies are expressed in SQL, reducing code complexity and maintenance.
* **Faster Onboarding**: Familiar SQL interface eases learning curve for new developers.
* **Interactive Development**: Quick feedback loops with a built-in test harness simplify development and testing.

### **Architectural Simplicity**

* **Unified System**: Combines management of data connectors and stream transformations.
* **Single JVM Option**: Can run Kafka Connect within the same JVM for streamlined operations.

### **Deployment and Operational Efficiency**

* **Turnkey Deployments**: Supports Docker and cloud-based solutions for easy setup.
* **Cross-Project Consistency**: Declarative SQL syntax promotes uniformity across projects.

### **Use Case Suitability**

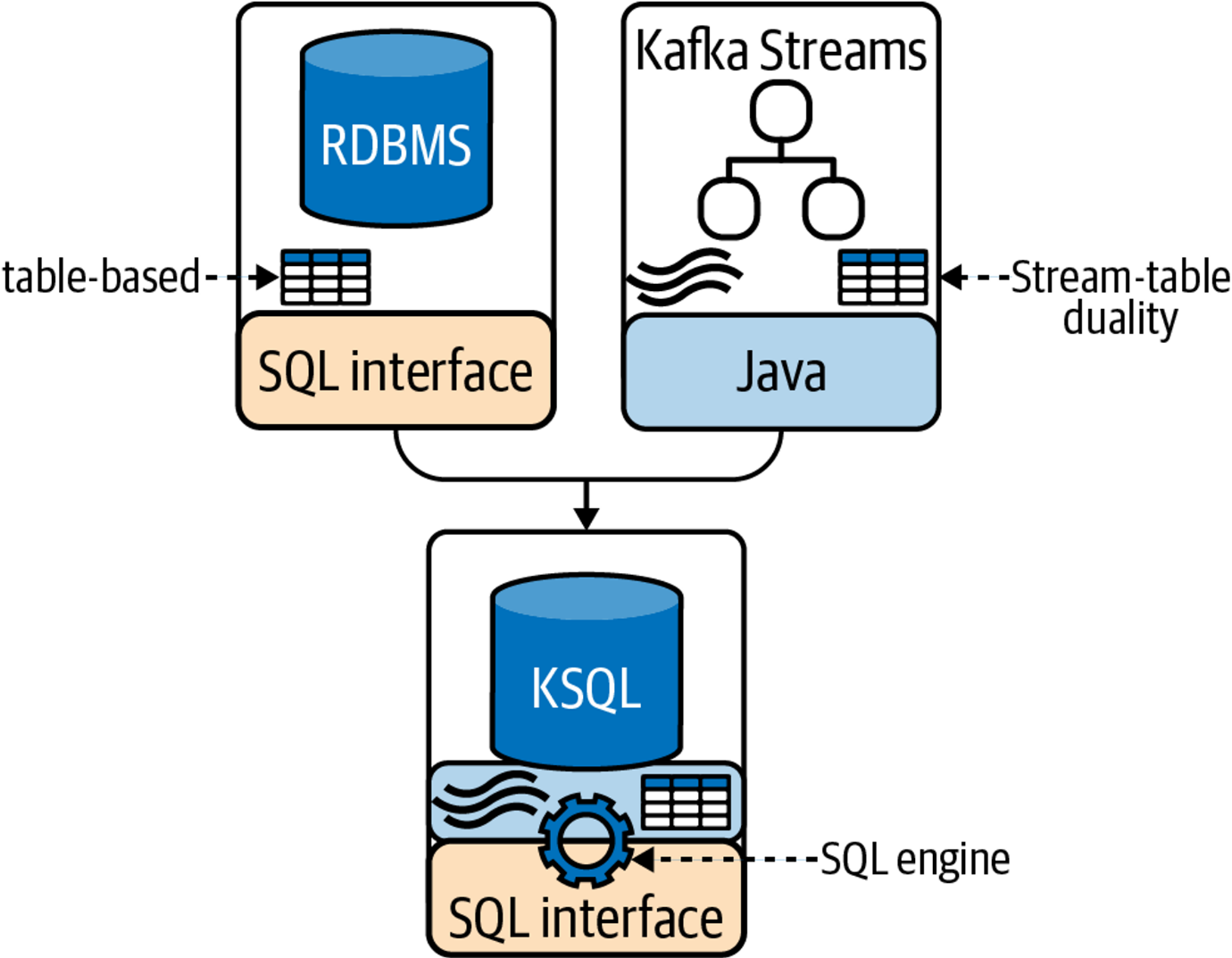
* **Data Exploration**: Enhanced support for topic inspection and querying materialized views.
* **Custom Functions**: Extensible with custom Java functions, though frequent JVM-level customizations may suggest evaluating the abstraction level.

### **When to Prefer ksqlDB**

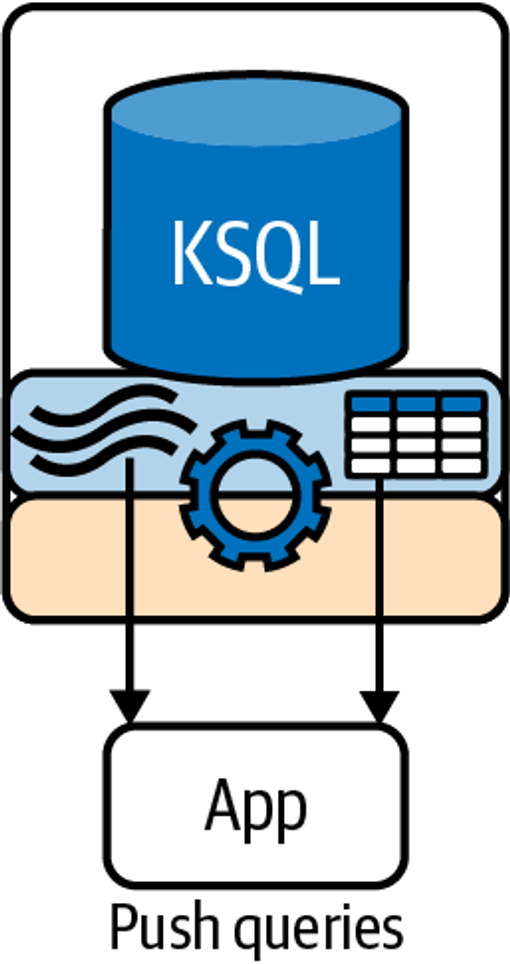
* **Streamlined Projects**: When projects benefit from simplified setup, rapid development, and SQL capabilities.
* **Complex Stream Processing**: Use Kafka Streams if needing lower-level control, custom data processing, or advanced monitoring.

# Evolution of a New Kind of Database

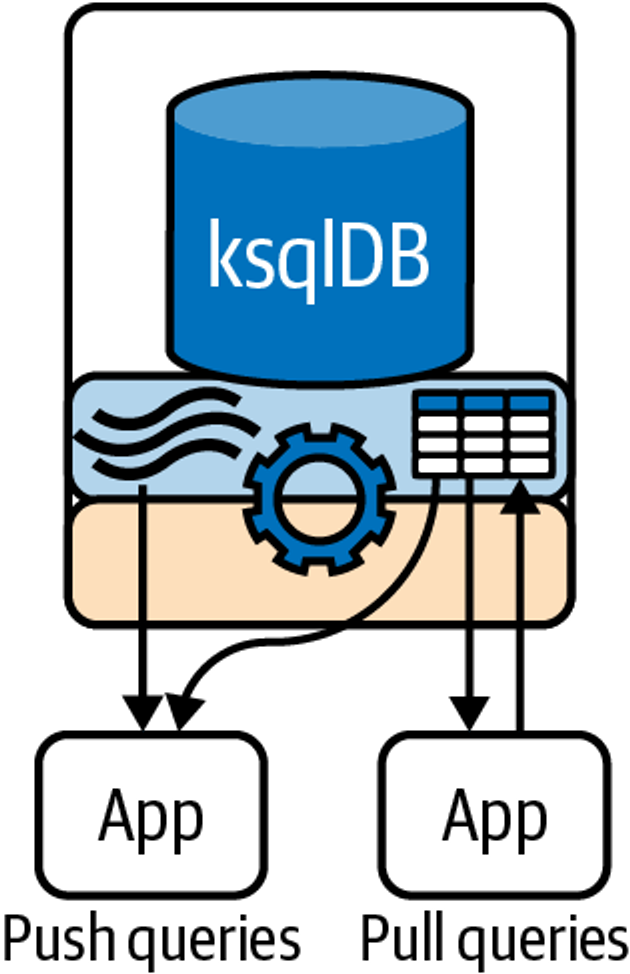
**The first phase of ksqlDB’s evolution combined Kafka Streams with features of traditional SQL databases, including a SQL interface**



Push queries automatically emit results whenever new data becomes available; this allows applications/clients to listen for new data



**ksqlDB supports both push and pull queries**



ksqlDB evolved into a system that supports both data transformation and integration



1. **Early Development**:
   * Initially known as KSQL.
   * Integrated Kafka Streams for stream processing.
   * Provided a SQL interface to build applications without Java/Scala.
2. **SQL Interface Adoption**:
   * Adopted from RDBMS to ease usage.
   * Enabled users familiar with SQL to easily transition to streaming data applications.
3. **Transition from KSQL to ksqlDB**:
   * Renamed to ksqlDB to reflect enhanced capabilities.
   * Introduced pull queries alongside existing push queries for more database-like interactions.
4. **Kafka Streams Integration**:
   * Core feature allowing continuous query execution.
   * Push queries emit results in real-time as data arrives.
5. **Kafka Connect Integration**:
   * Added in the ksqlDB evolution to handle ETL processes.
   * Allows direct definition and management of connectors via SQL, simplifying data integration.
6. **Enhanced SQL for Streaming**:
   * Extends standard SQL to handle streams and tables.
   * Supports complex operations like joins, aggregations, and windowing on streaming data.
7. **Dual Query Capabilities**:
   * **Push Queries**: Continuous and emit results as data flows.
   * **Pull Queries**: On-demand queries similar to traditional database lookups.
8. **Significance of Evolution**:
   * ksqlDB's development showcases its unique position as a streaming database.
   * Enhancements have focused on making stream processing more accessible and powerful.

**Architecture**

**1. ksqlDB Servers:**

* **Role**: Execute stream processing applications as a set of SQL queries.
* **Composition**: Each server acts like a Kafka Streams application instance.
* **Scalability**: Servers are scaled by adding more with the same **ksql.service.id** to distribute workload.
* **Cluster Formation**: Multiple servers with the same service ID form a ksqlDB cluster, allowing workload isolation and scalability.
* **Components**:
  + **SQL Engine**: Parses SQL statements into Kafka Streams topologies and executes them.
  + **REST Service**: Facilitates query submission and cluster management through HTTP requests.

**2. ksqlDB Clients:**

* **Interface**: Allows interaction with ksqlDB servers via a RESTful API.
* **Tools**: Includes the ksqlDB CLI and UI for user interactions, such as query submission and management. **Access Methods**:
  + **CLI (Command Line Interface)**: Interactive command tool for direct server communication.
  + **UI (User Interface)**: Web-based platform for managing and visualizing ksqlDB functionalities.

**3. Integration Components:**

* **Kafka Streams**: Core component for stream processing, underlying the SQL engine to run queries and manage state.
* **Kafka Connect**: Integrated for managing connectors directly through ksqlDB, facilitating ETL processes and data integration.

**4. Operational Modes:**

* **Interactive Mode**: Uses the REST API for dynamic query execution and system interaction.
* **Headless Mode**: Runs predefined queries from a file without interactive input, suitable for production environments.

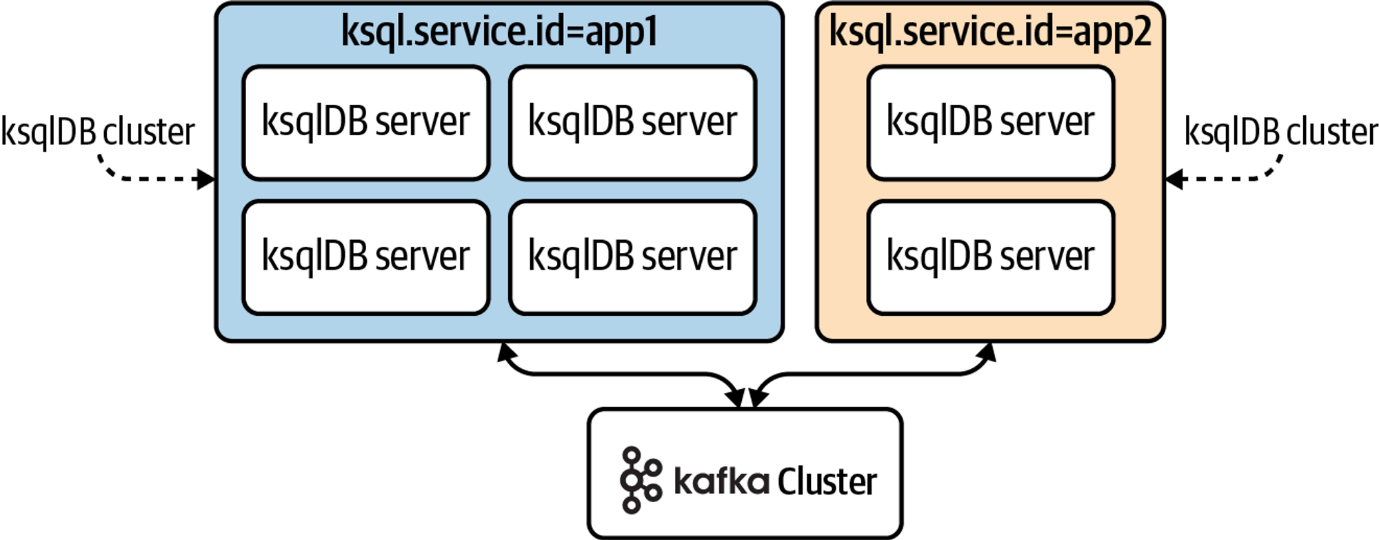
**5. Configuration and Management:**

* **Listeners Configuration**: Defines network settings for server communication.
* **SSL Configuration**: Ensures secure communication between clients and servers.
* **Scaling Operations**: Adding or removing servers dynamically adjusts the cluster's capacity and load distribution.

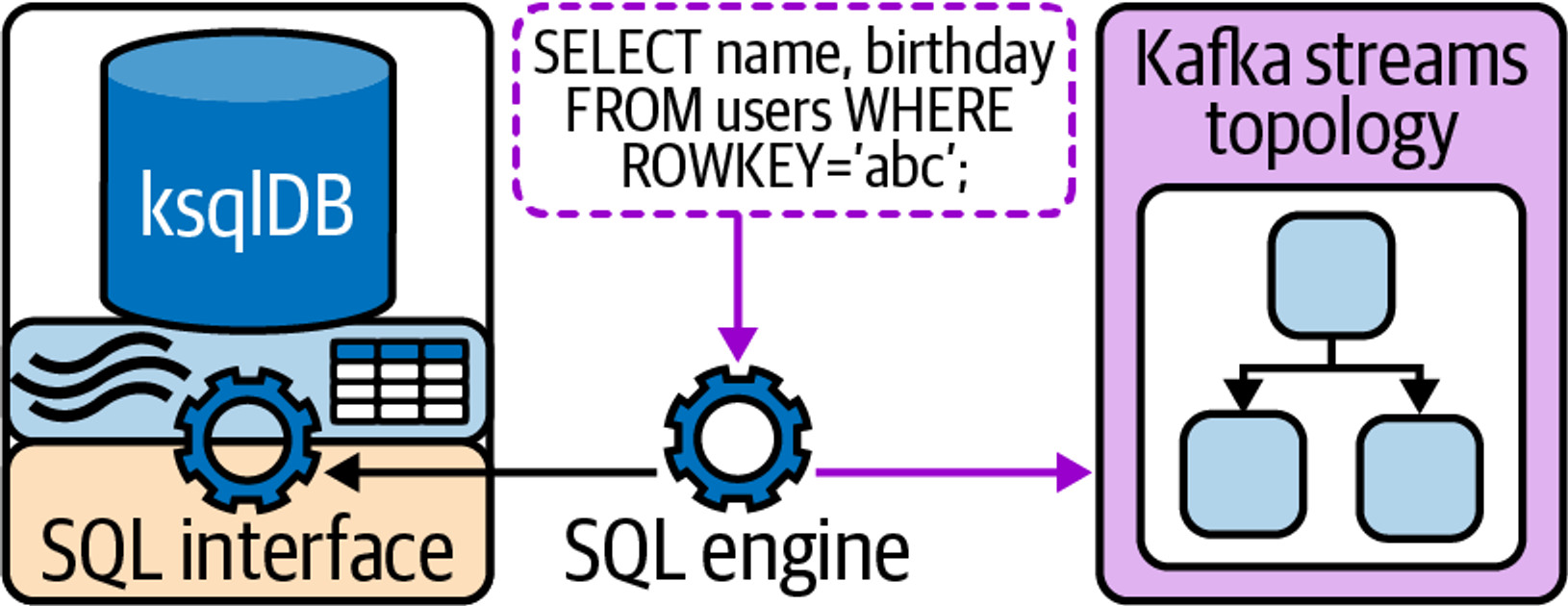
**6. Key Features:**

* **Query Execution**: Both push and pull queries are supported, enabling real-time data streaming and on-demand data retrieval.
* **Data Handling**: SQL engine processes and converts SQL commands into actionable Kafka Streams operations.
* **High Availability and Fault Tolerance**: Inherent from Kafka's robust ecosystem, ensuring reliable data processing and system stability.

Two ksqlDB clusters processing data from Kafka independently.



**The SQL engine converts SQL statements into Kafka Streams topologies.**



**Deployment Modes**

**1. Interactive Mode**

* **Description**: Allows real-time interaction with ksqlDB servers via the REST API.
* **Capabilities**:
  + Dynamically submit, modify, and terminate queries, streams, tables, and connectors.
  + Utilizes the command topic for replicating SQL commands across all nodes in the cluster, ensuring consistent state and operation across the cluster.
* **Characteristics**:
  + Default mode of operation for ksqlDB, enabling flexibility and ease of use.
  + Ideal for development and testing environments where frequent modifications are necessary.
* **Deployment**:
  + No special configuration needed beyond default setup.
  + Depicted in Figure 8-8, illustrating active interaction through various clients like CLI, UI, or custom applications.

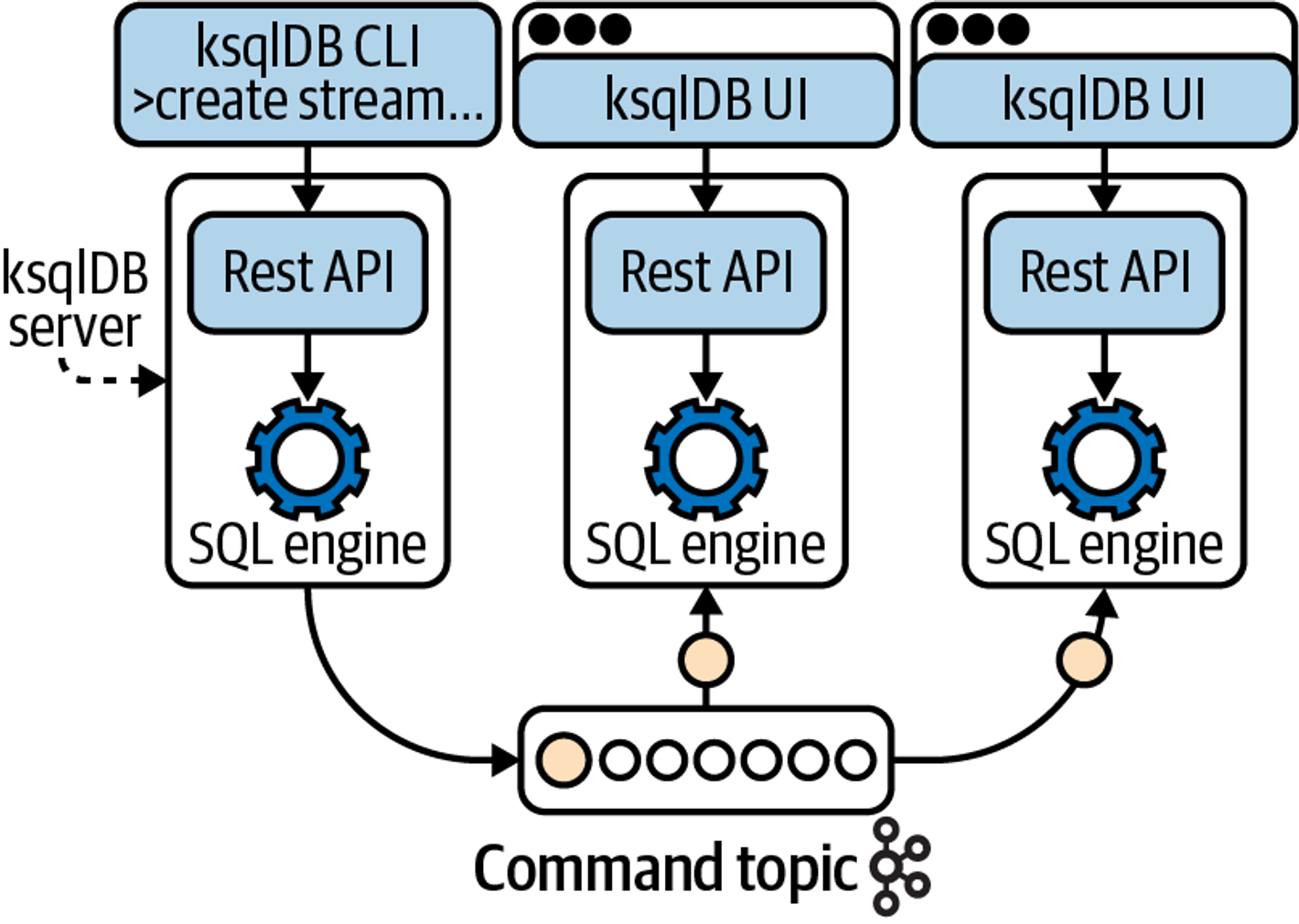
**2. Headless Mode**

* **Description**: Runs ksqlDB servers without REST API interactivity, preventing any runtime modifications to the running queries.
* **Capabilities**:
  + Executes a predefined set of queries loaded from a configuration file, providing stability and security for production environments.
  + Does not use the command topic for replication, relying instead on internal configurations and possibly a config topic for essential metadata.
* **Characteristics**:
  + Ensures a locked-down environment, suitable for production deployments where changes to the query logic are not allowed during runtime.
  + Requires specifying the path to a query file during server startup to set the operational queries.
* **Deployment**:
  + Requires configuration via **queries.file** property to indicate the location of the SQL file containing persistent queries.

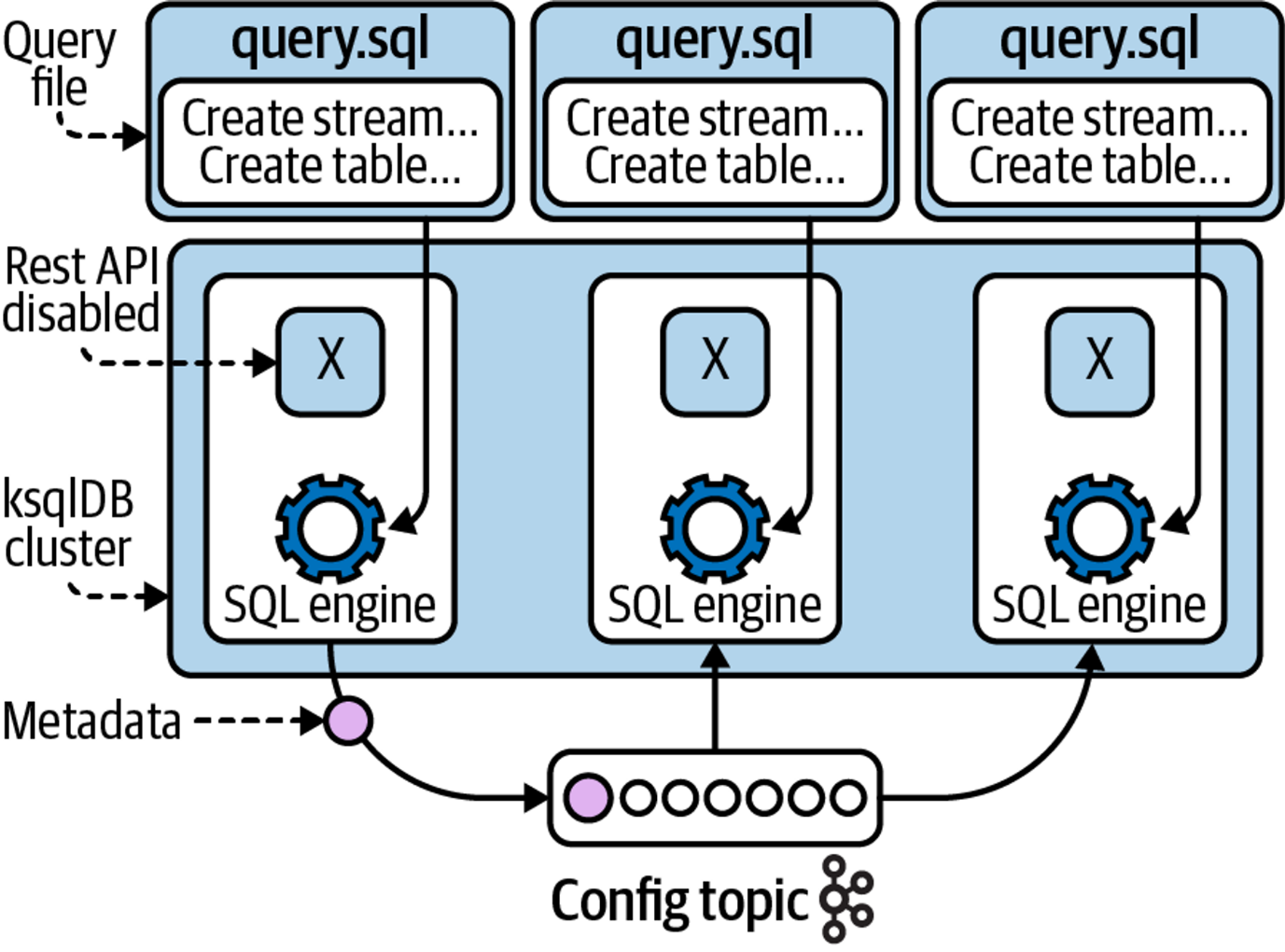
**Choosing the Right Mode**:

* **Interactive Mode**: Opt for this mode if you need a dynamic and flexible environment where you can experiment and modify queries on the fly. Ideal for development, testing, or environments where changes are frequent.
* **Headless Mode**: Best suited for production environments where stability is critical, and changes to queries should be controlled and limited to deployment cycles.

ksqlDB running in interactive mode, allowing clients (e.g., the ksqlDB CLI, UI, the Java client, or even curl) to submit queries via the REST API.



ksqlDB running in headless mode.



# Schema Registry

<https://www.notion.so/nagcloudlab/Kafka-Schema-Registry-__-v2-3156398b741145e1ba06ac2c72aa8775?pvs=4>

# Kafka REST Proxy

The Kafka REST Proxy is a component of the Confluent platform that provides a RESTful interface to interact with Kafka clusters.

It allows you to produce and consume messages, view topic and partition information, and perform other Kafka-related tasks using HTTP requests.

This can be particularly useful for integrating Kafka with systems that do not have native Kafka clients.

### **Key Features**

1. **Produce Messages**: Send messages to Kafka topics using HTTP POST requests.
2. **Consume Messages**: Retrieve messages from Kafka topics using HTTP GET requests.
3. **Admin Operations**: Perform administrative tasks like retrieving metadata about topics and clusters.
4. **Schema Registry Integration**: Integrate with Confluent Schema Registry to manage Avro schemas.

### **Setting Up Kafka REST Proxy**

1. **Download and Install**: Kafka REST Proxy is part of the Confluent Platform. You can download it from the Confluent website.
2. **Configuration**: The main configuration file is **kafka-rest.properties**. Key configuration parameters include:
   * **bootstrap.servers**: Kafka brokers to connect to.
   * **schema.registry.url**: URL of the Confluent Schema Registry.
   * **listeners**: HTTP(S) port and interface on which the REST Proxy will listen.

Example **kafka-rest.properties** file:

bootstrap.servers=localhost:9092

schema.registry.url=http://localhost:8081

listeners=http://localhost:8082

1. **Starting the REST Proxy**: Use the following command to start the REST Proxy:

./bin/kafka-rest-start ./etc/kafka-rest/kafka-rest.properties

#### Basic Operations

#### Producing Messages

To produce messages to a Kafka topic, you can use an HTTP POST request. For example, to send a message to the topic **my-topic**:

Plain Text

Copy

curl -X POST -H "Content-Type: application/vnd.kafka.json.v2+json" \ --data '{"records":[{"value":{"foo":"bar"}}]}' \ http://localhost:8082/topics/my-topic

#### Consuming Messages

To consume messages from a Kafka topic, you can use an HTTP GET request. For example, to consume messages from the topic **my-topic**:

**Create a consumer instance**:

Plain Text

Copy

curl -X POST -H "Content-Type: application/vnd.kafka.v2+json" \ --data '{"name": "my\_consumer\_instance", "format": "json", "auto.offset.reset": "earliest"}' \ http://localhost:8082/consumers/my\_consumer\_group

​

**Subscribe to the topic**:

Plain Text

Copy

curl -X POST -H "Content-Type: application/vnd.kafka.v2+json" \ --data '{"topics":["my-topic"]}' \ http://localhost:8082/consumers/my\_consumer\_group/instances/my\_consumer\_instance/subscription

**Consume messages**:

Plain Text

Copy

curl -X GET -H "Accept: application/vnd.kafka.json.v2+json" \ http://localhost:8082/consumers/my\_consumer\_group/instances/my\_consumer\_instance/records

#### Admin Operations

You can perform various administrative operations using the Kafka REST Proxy, such as retrieving metadata about topics and partitions.

#### List Topics

To list all topics in the Kafka cluster:

Plain Text

Copy

curl -X GET http://localhost:8082/topics

​

#### Get Topic Details

To get details about a specific topic:

Plain Text

Copy

curl -X GET http://localhost:8082/topics/my-topic

### **Integrating with Schema Registry**

If you're using Avro-encoded messages, you can integrate the REST Proxy with Confluent Schema Registry to manage schemas. Ensure the **schema.registry.url** is configured in your **kafka-rest.properties**.

### **Example Use Case**

Imagine you have a web application that needs to send events to a Kafka topic whenever a user performs a specific action. Instead of using a Kafka client library, you can use Kafka REST Proxy to produce messages directly from your web application's backend:

1. **User performs action on the web application.**
2. **Backend service sends an HTTP POST request to Kafka REST Proxy:**
3. curl -X POST -H "Content-Type: application/vnd.kafka.json.v2+json" \\
4. --data '{"records":[{"value":{"user\_action":"clicked\_button"}}]}' \\
5. <http://localhost:8082/topics/user-actions>

This setup simplifies integration and allows you to leverage Kafka without needing to manage Kafka client libraries in your application.

# Kafka MirrorMaker2

Kafka MirrorMaker 2 (MM2) is an advanced tool for replicating data across Kafka clusters.

It is part of the Kafka Connect framework and provides more robust and flexible replication capabilities compared to the original MirrorMaker.

MM2 supports active-active replication, making it suitable for more complex replication scenarios.

### **Key Features**

1. **Active-Active Replication**: Support for bi-directional data replication.
2. **Topic Renaming**: Ability to rename topics during replication.
3. **Offset Syncing**: Synchronizes consumer offsets across clusters.
4. **Configuration via Kafka Connect**: Leverages Kafka Connect for configuration and management.

### **Features of Kafka MirrorMaker 2**

1. **Multi-cluster Replication**:
   * MM2 allows for replication between multiple Kafka clusters, enabling data synchronization across different data centers or cloud regions.
2. **Active-Active and Active-Passive Replication**:
   * Supports both active-active and active-passive replication modes. In active-active, data is replicated bidirectionally, while in active-passive, data is replicated in one direction only.
3. **Topic Filtering**:
   * Enables selective replication of topics using regular expressions, allowing you to replicate only the necessary data.
4. **Offset Synchronization**:
   * Synchronizes consumer offsets between clusters, ensuring seamless failover and consumer group continuity.
5. **Transformation Capabilities**:
   * Provides the ability to transform messages during replication, such as modifying topic names or message content.
6. **Monitoring and Metrics**:
   * Integrates with Kafka’s metrics system, providing visibility into replication performance and status.
7. **Automatic Failover**:
   * Supports automatic failover configurations, enhancing high availability and disaster recovery capabilities.
8. **Security Features**:
   * Supports SSL/TLS encryption and SASL authentication, ensuring secure data replication.

### **Use Cases for Kafka MirrorMaker 2**

1. **Disaster Recovery**:
   * Set up a secondary Kafka cluster in a different data center or cloud region to ensure business continuity in case of a failure in the primary cluster.
2. **Multi-Datacenter Deployments**:
   * Enable data replication across multiple data centers to support geographically distributed applications.
3. **Hybrid Cloud Architectures**:
   * Facilitate data replication between on-premises Kafka clusters and cloud-based Kafka services for hybrid cloud architectures.
4. **Data Sovereignty**:
   * Ensure compliance with data sovereignty regulations by replicating data to clusters located in specific geographical regions.
5. **Global Data Distribution**:
   * Distribute data globally to bring it closer to end-users, reducing latency and improving performance for applications with a global user base.
6. **Backup and Archival**:
   * Use MM2 to create backups of Kafka topics in a separate cluster for long-term storage and archival purposes.
7. **Seamless Migration**:
   * Migrate Kafka workloads between clusters with minimal downtime, enabling seamless transitions during infrastructure changes.
8. **Cross-Cluster Analytics**:
   * Replicate data to a dedicated analytics cluster, allowing for isolated analytics workloads without impacting the production cluster’s performance.

# Kafka Security

<https://docs.google.com/presentation/d/1i9wCIX4lXlBQeqDHBoGRckuCA_VDtnQr/edit?usp=sharing&ouid=116008769779639876320&rtpof=true&sd=true>

<https://medium.com/@innovationchef/kafka-security-setup-98e07de6bd63>

<https://medium.com/jinternals/kafka-ssl-setup-with-self-signed-certificate-part-1-c2679a57e16c>

[**https://nagcloudlab.notion.site/Strimzi-4480045fc2284902a51202ade37d29d3**](https://nagcloudlab.notion.site/Strimzi-4480045fc2284902a51202ade37d29d3)

**Kindly do other workarounds for Rs 90 as well for reversal.**

**This is the second time I am facing this irrelevant charges from ICICI Bank. Similar kind of issue I have faced in my savings account even though that was my salary account.**

**I have visited many times in the branch to solve in savings bank account issue that is not resolved so I have closed savings account.**

**The same circumstances again came with a Credit Card.**