# Tecnologías de datos masivos

Doble Grado en Ingeniería en Tecnologías de Telecomunicación y Business Analytics

# SparkStreaming

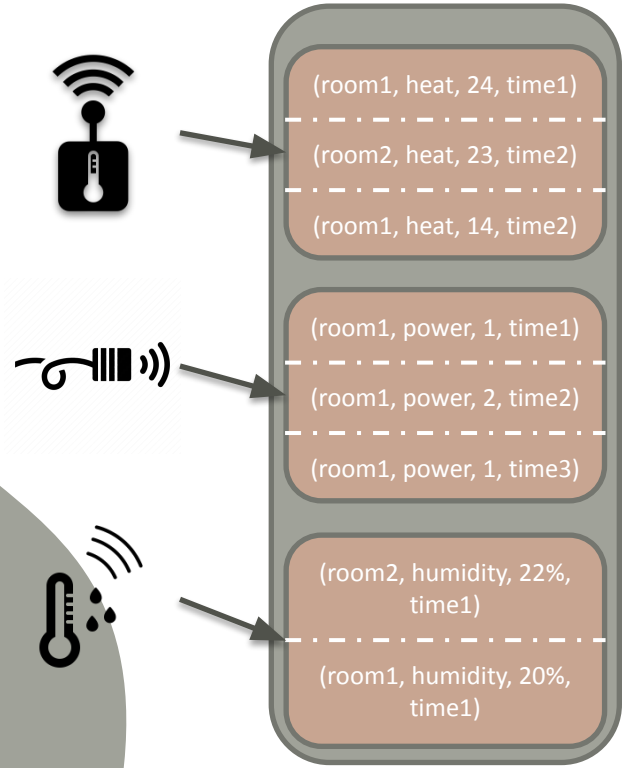
# SparkStreaming - Introducción

- El desarrollo de las redes de telecomunicación modernas hace que nazcan nuevos casos de usos
- Con el IOT todos los dispositivos se pueden conectar y nos pueden dar información útil para analitica
- Se piensan nuevos casos de usos que tienen en cuenta tanto la recepción de los eventos en streaming como la resolución de los mismos en tiempo online
- Las tecnologías existentes de procesamiento se tienen que adaptar las nuevas necesidades online

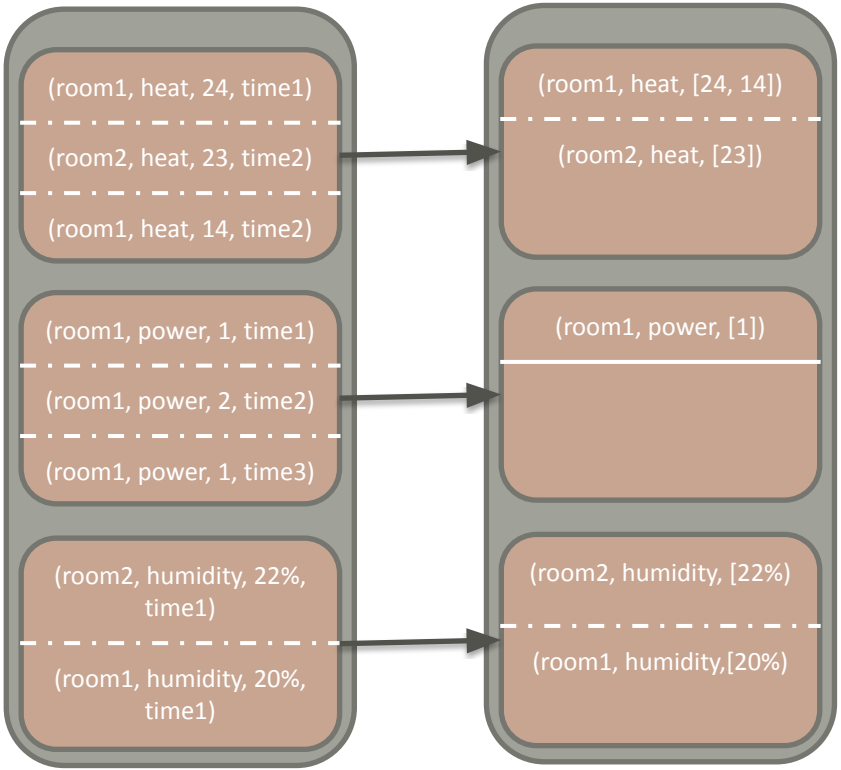
# SparkStreaming - Introducción

- En los primeros años de Spark la tecnología de recepción y emisión de eventos en streaming no estaba muy desarrollada
- Cuando nace Spark no está pensado para realizar tareas en Streaming, sólo en Batch
- La estrategia de Spark para gestionar todas sus casuísticas es adaptarlas a la arquitectura de Spark
- La arquitectura de Spark y su filosofía de tener el mayor tiempo posible la mayor cantidad de datos en memoria y, sobretodo,  su politica ante los errores hacian dificil usarlo para resolver casuistica de streaming

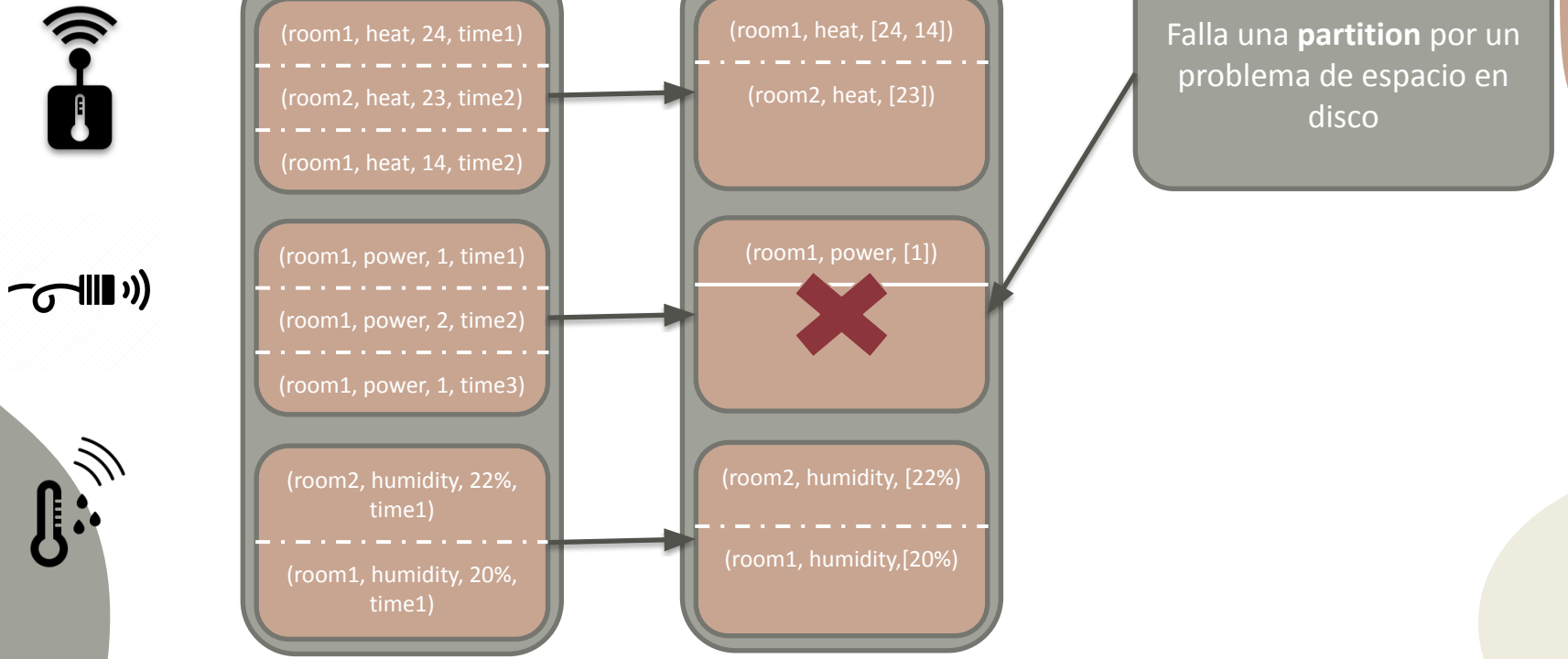
# SparkStreaming - Introducción



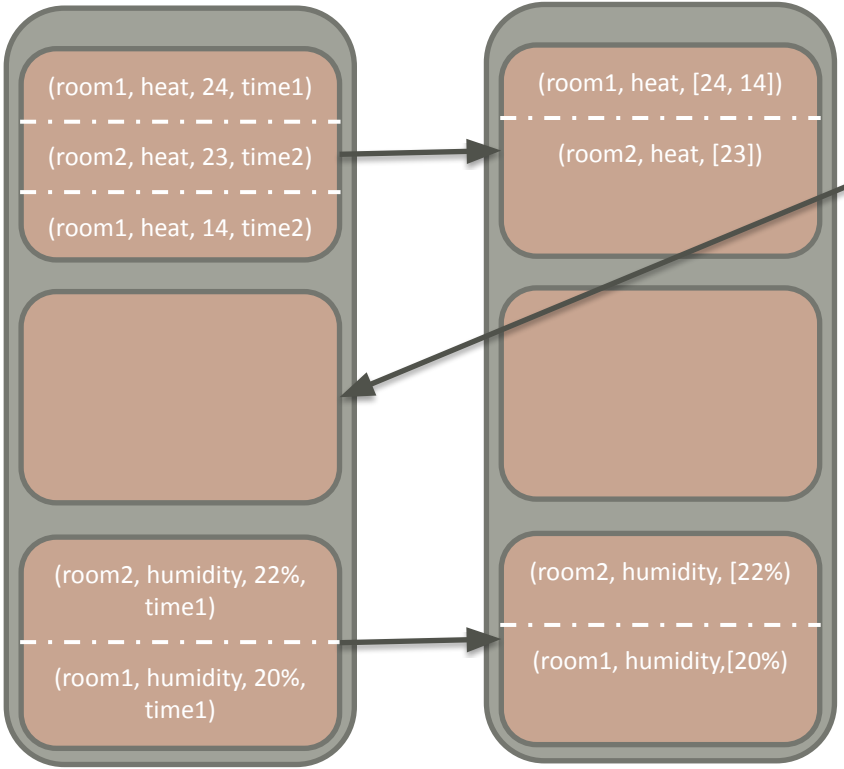
# SparkStreaming - Introducción



# SparkStreaming - Introducción

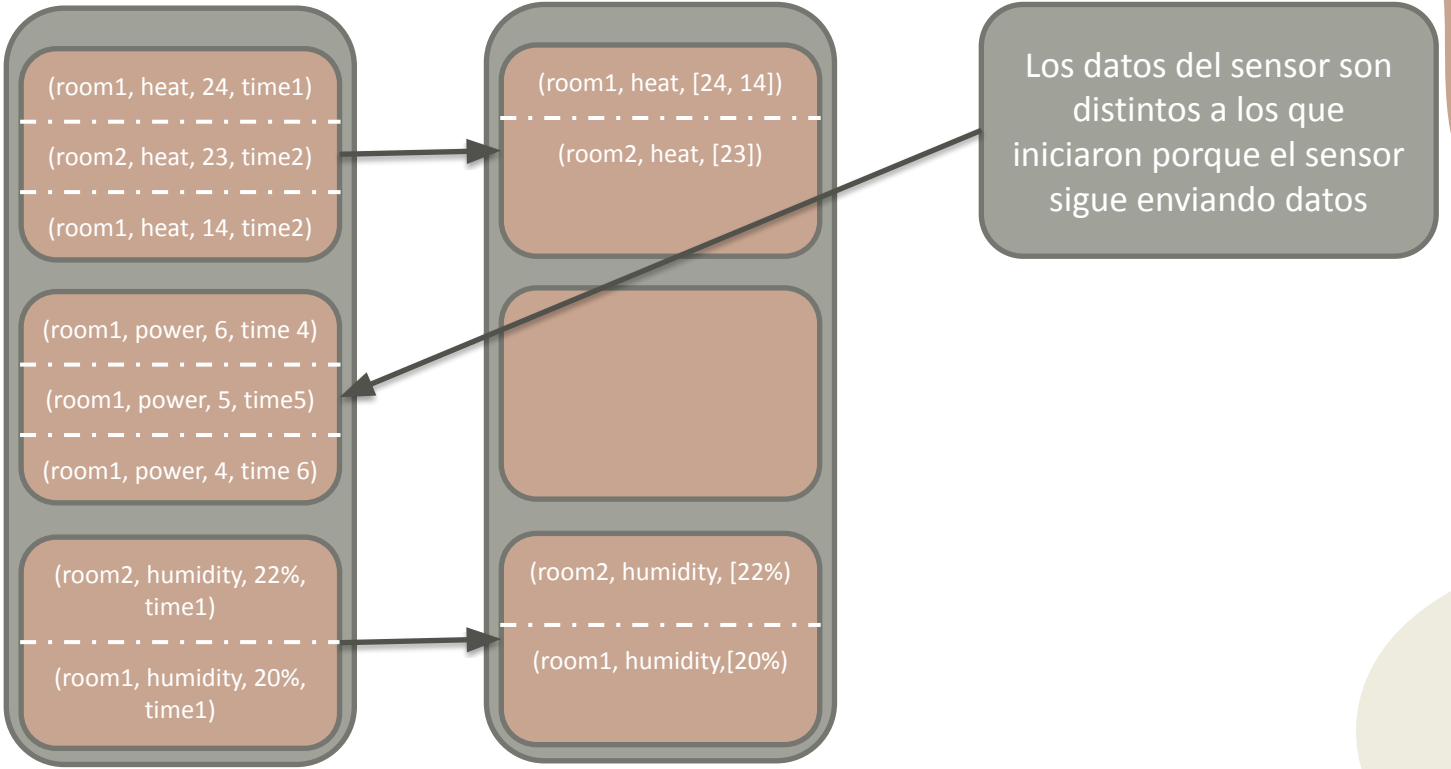


# SparkStreaming - Introducción



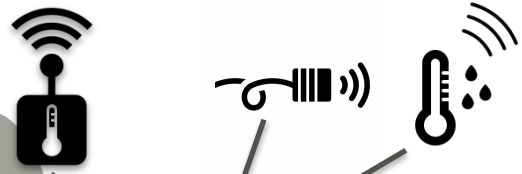

Se relanza dicha particion en otro Executor desde el **BaseRDD**

# SparkStreaming - Introducción

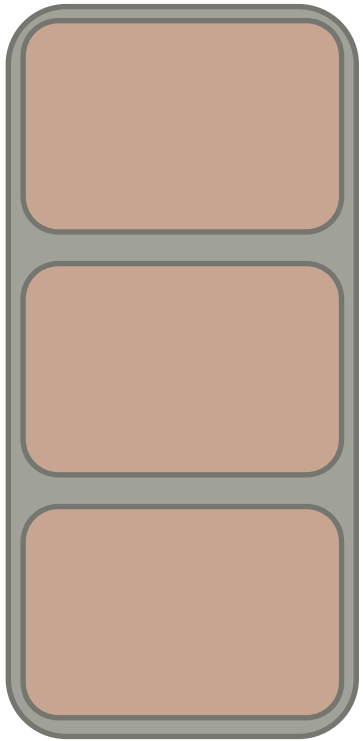


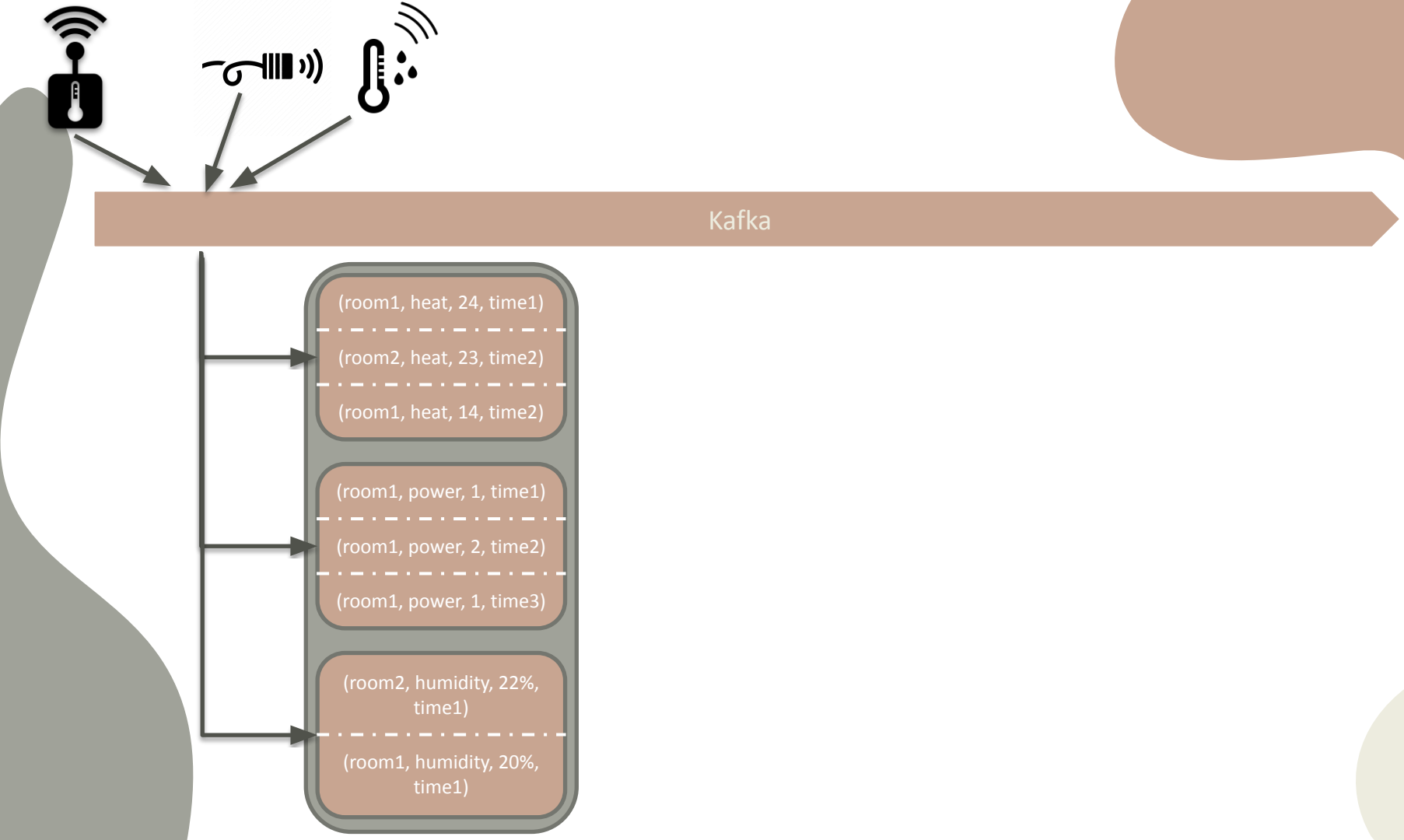
# SparkStreaming - Introducción

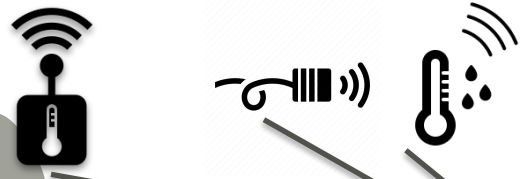
- El caso de Spark no era único y las tecnologías existentes que resolvían casos de uso de batch tenían dificultades para adaptarse al streaming
- Con la llegada de **Apache Kafka** se empiezan a poder resolver varios casos de uso que antes no podrían ser resueltos con **Spark**
- **Apache Kafka** permite persistir los eventos asociados a una temporalidad y leerlos a posteriori
- En un inicio **Apache Kafka** sólo se usaba como almacenamiento intermedio de los datos.
- En la actualidad se ha creado varios proyectos alrededor de **Apache Kafka** que lo lo posibilitan como solución tanto para el almacenamiento intermedio como para el procesamiento de los datos



Kafka




Kafka
(room1, heat, 24, time1)
(room2, heat, 23, time2)
(room1, heat, 14, time2)
(room1, power, 1, time1)
(room1, power, 2, time2)
(room1, power, 1, time3)
(room2, humidity, 22%, time1)
(room1, humidity, 20%, time1)



Los sensores siguen emitiendo señales y se almacenan en una cola de Kafka

Kafka

(room1, heat, 24, time1)

(room2, heat, 23, time2)

(room1, heat, 14, time2)

(room1, power, 1, time1)

(room1, power, 2, time2)

(room1, power, 1, time3)

(room2, humidity, 22%, time1)

(room1, humidity, 20%, time1)

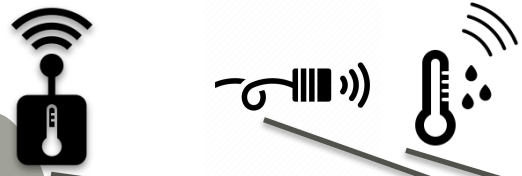
(room1, heat, [24, 14])

(room2, heat, [23])

(room1, power, [1])

(room2, humidity, [22%)

(room1, humidity,[20%)



Los sensores siguen emitiendo señales y se almacenan en una cola de Kafka

Kafka

(room1, heat, 24, time1)

(room2, heat, 23, time2)

(room1, heat, 14, time2)

(room1, power, 1, time1)

(room1, power, 2, time2)

(room1, power, 1, time3)

(room2, humidity, 22%, time1)

(room1, humidity, 20%, time1)

(room1, heat, [24, 14])

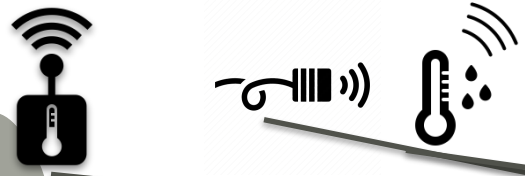
(room2, heat, [23])

(room1, power, [1])

(room2, humidity, [22%)

(room1, humidity,[20%)

Falla una **partition** por un problema de espacio en disco



Los sensores siguen emitiendo señales y se almacenan en una cola de Kafka

Kafka

(room1, heat, 24, time1)

(room2, heat, 23, time2)

(room1, heat, 14, time2)

(room2, humidity, 22%, time1)

(room1, humidity, 20%, time1)

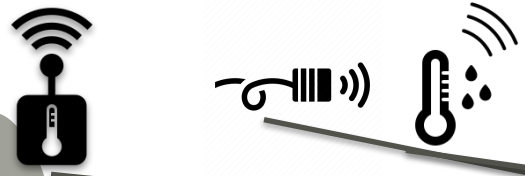
(room1, heat, [24, 14])

(room2, heat, [23])

(room2, humidity, [22%)

(room1, humidity,[20%)

Se relanza dicha particion en otro Executor desde el **BaseRDD**



Los sensores siguen emitiendo señales y se almacenan en una cola de Kafka

Kafka

(room1, heat, 24, time1)

(room2, heat, 23, time2)

(room1, heat, 14, time2)

(room1, power, 1, time1)

(room1, power, 2, time2)

(room1, power, 1, time3)

(room2, humidity, 22%, time1)

(room1, humidity, 20%, time1)

(room1, heat, [24, 14])

(room2, heat, [23])

(room1, power, [1])

(room2, humidity, [22%)

(room1, humidity,[20%)

# SparkStreaming - Introducción

- **Spark** resuelve la problemática de streaming mediante la creación constante de **micro-batches**
- Un **micro-batch** es un corte del tiempo determinado por un tiempo de ventana
- Se creará un RDD con toda la información recibida en cada micro-batch.
- Cada micro-batch se trata por separado y sólo puede juntar la información de varios mediante la operación **updateStateByKey**
- En las primeras versiones de Spark no se puede usar directamente SparkSQL sobre elementos de Streaming.
- En las últimas versiones de **Spark** se ha desarrollado la funcionalidad de usar SparkSQL en streaming llamada **StructuredStreaming**

# StreamingContext

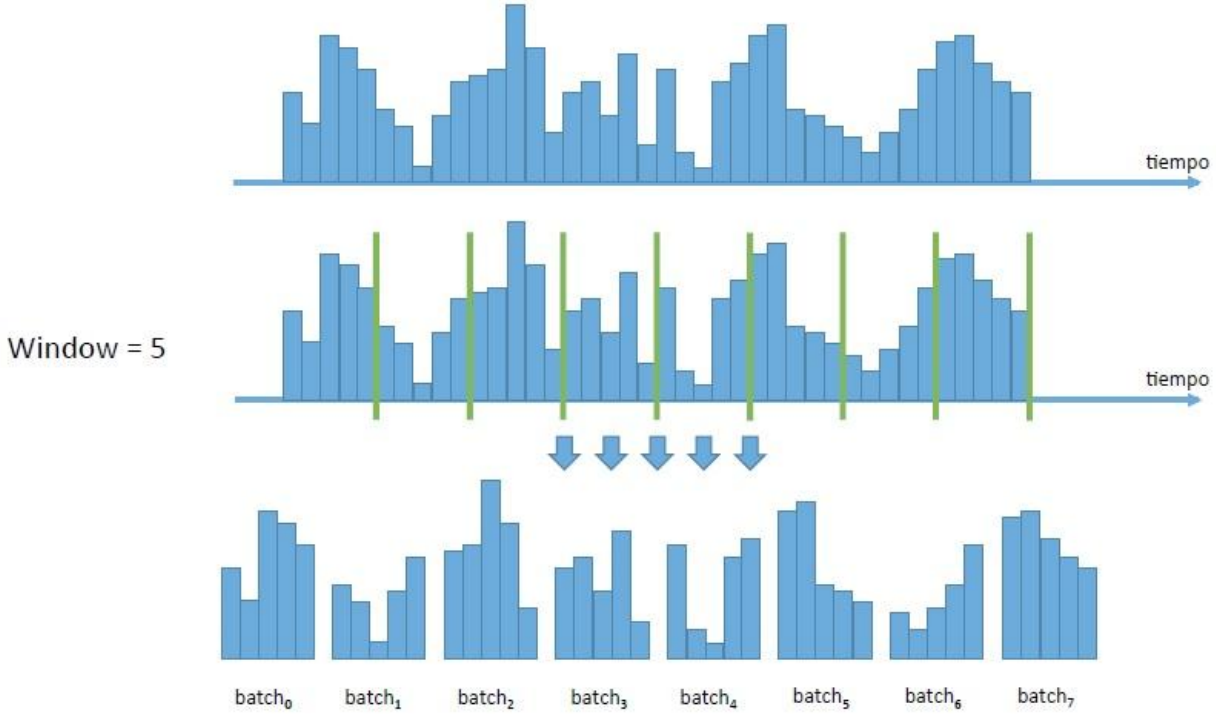
- El problema de Streaming características propias que requieren una serie de propiedades nuevas
- En Spark todo los RDD tienen que crearse mediante el **SparkContext** (*SparSession tiene un SparkContext subyacente)*
- Para la casuistica de streaming **Spark** requiere un nuevo punto de entrada, el **StreamingContext**
- Todo **StreamingContext** tiene un **SparkContext** subyacente y una serie de propiedades propias de Streaming
- Al igual que ocurre en los **SparkSession,** desde el **StreamingContext** se puede acceder al **SparkContext** suyacente

# StreamingContext

- Las propiedades específicas del streaming son:
    - **Tiempo de ventana:** el tiempo de ventana determina cada cuantos segundos se crea una nueva ventana en la que toda su información se transformará en un RDD. Esta propiedad no se puede modificar una vez iniciado el **StreamingContext**
    - **Checkpoint**: En el checkpointing Spark guarda datos de nuestra aplicación en un HDFS una vez por intervalo de tiempo

      Se almacenan dos tipos de datos:
        - Metadatos: Se guarda la información necesaria par poder recuperarse en el punto de fallo sin problemas
        - Datos: Se almacena la información propia de los RDD que forman parte de los distintos DStreams que tenga nuestra **aplication** de **Spark Streaming**

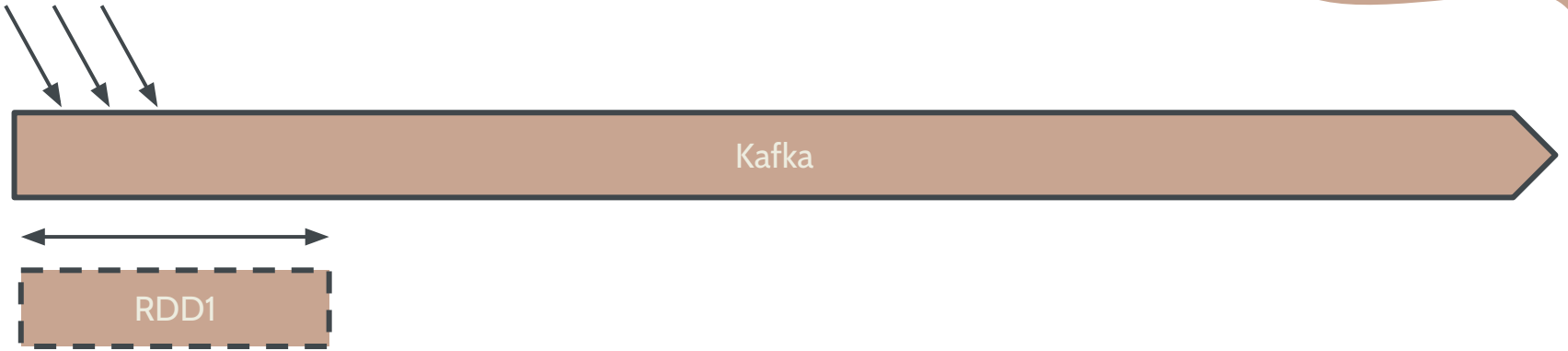
# SparkSQL - Catálogo



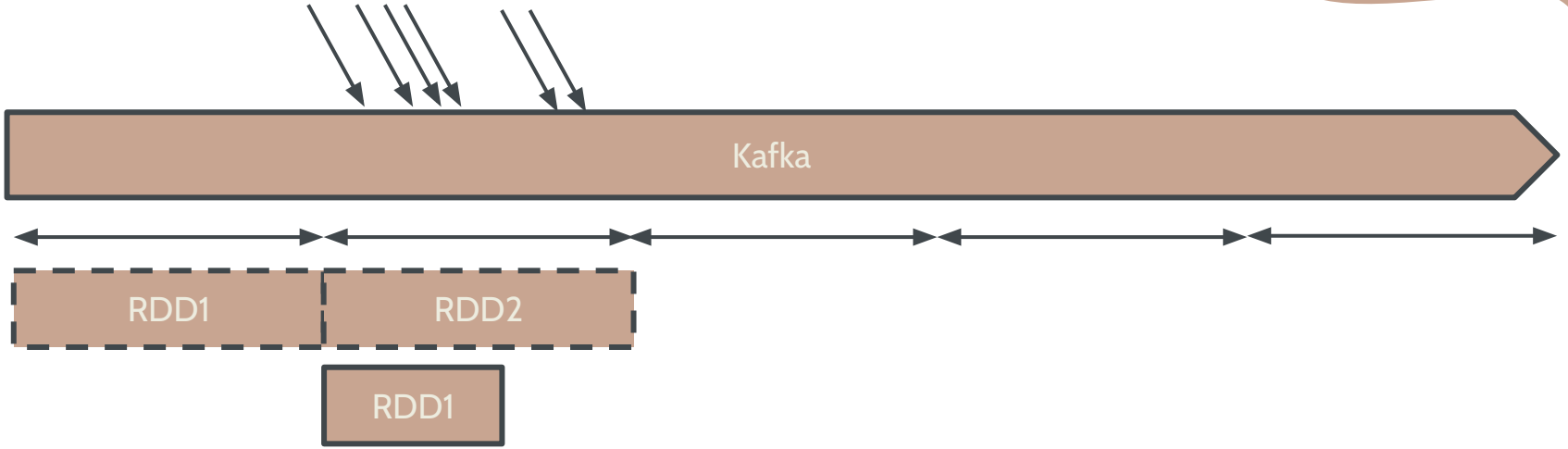
# DStream

- Son la unidad de cómputo de **SparkStreaming**
- Se instancian a partir de un StreamingContext
- Spark siempre trata todas las casuísticas  usando como base los RDD
- Para tratar el Streaming, Spark considera que ahora las colecciones son flujos continuos de datos en los que se puede tratar la información en fragmentos de tiempo
- Con la información que hay en cada uno de esos fragmentos de tiempo Spark crea un **RDD**
- Después de obtener la información Spark la procesa

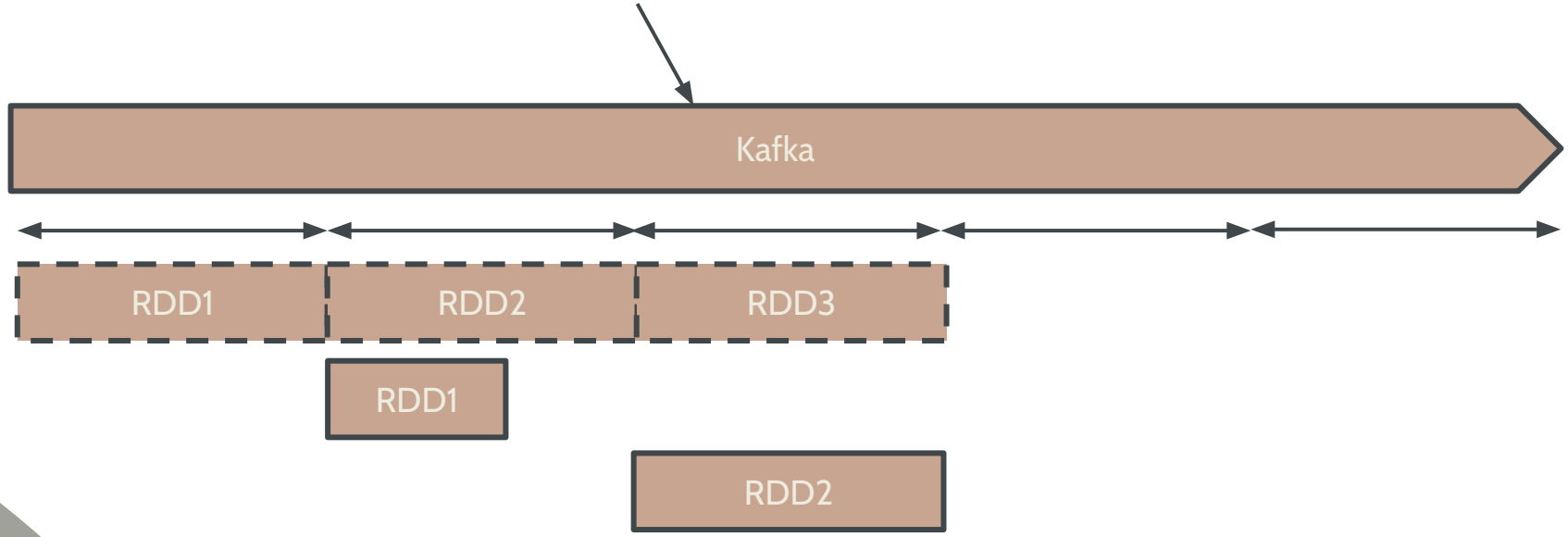
# DStream



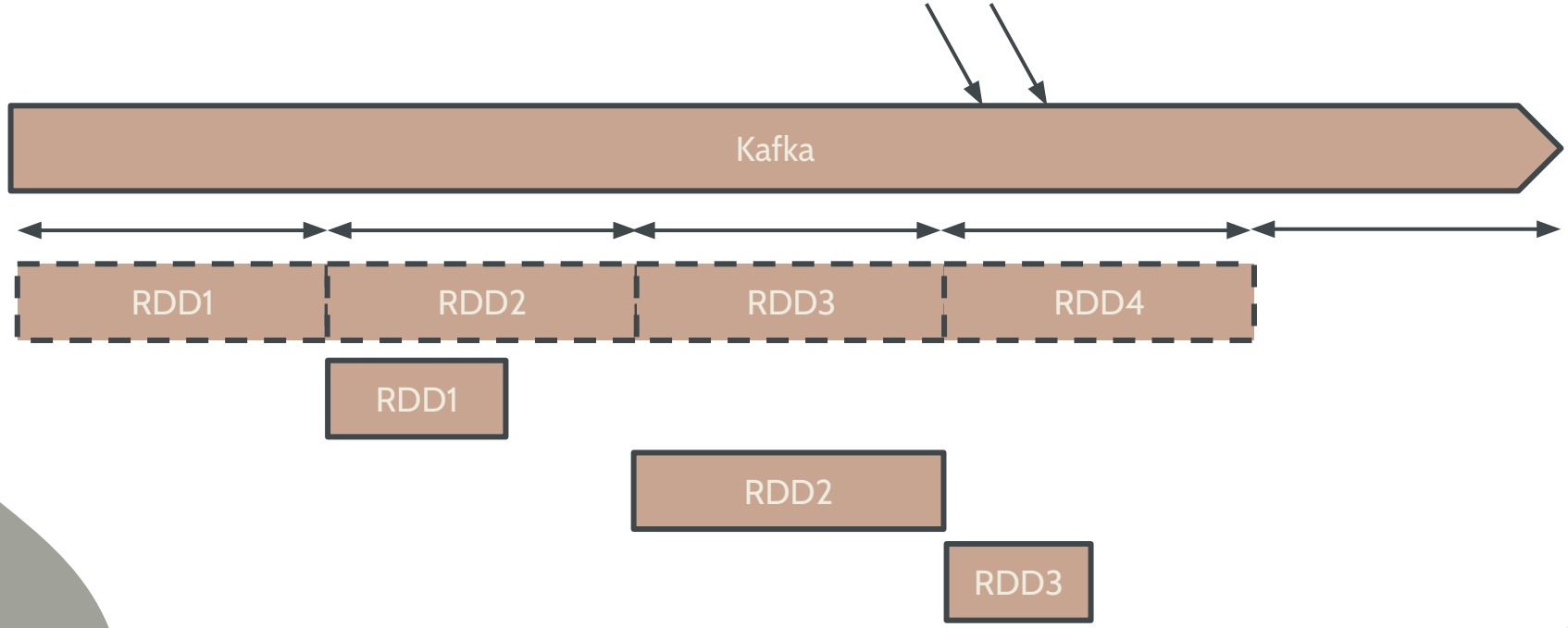
# DStream



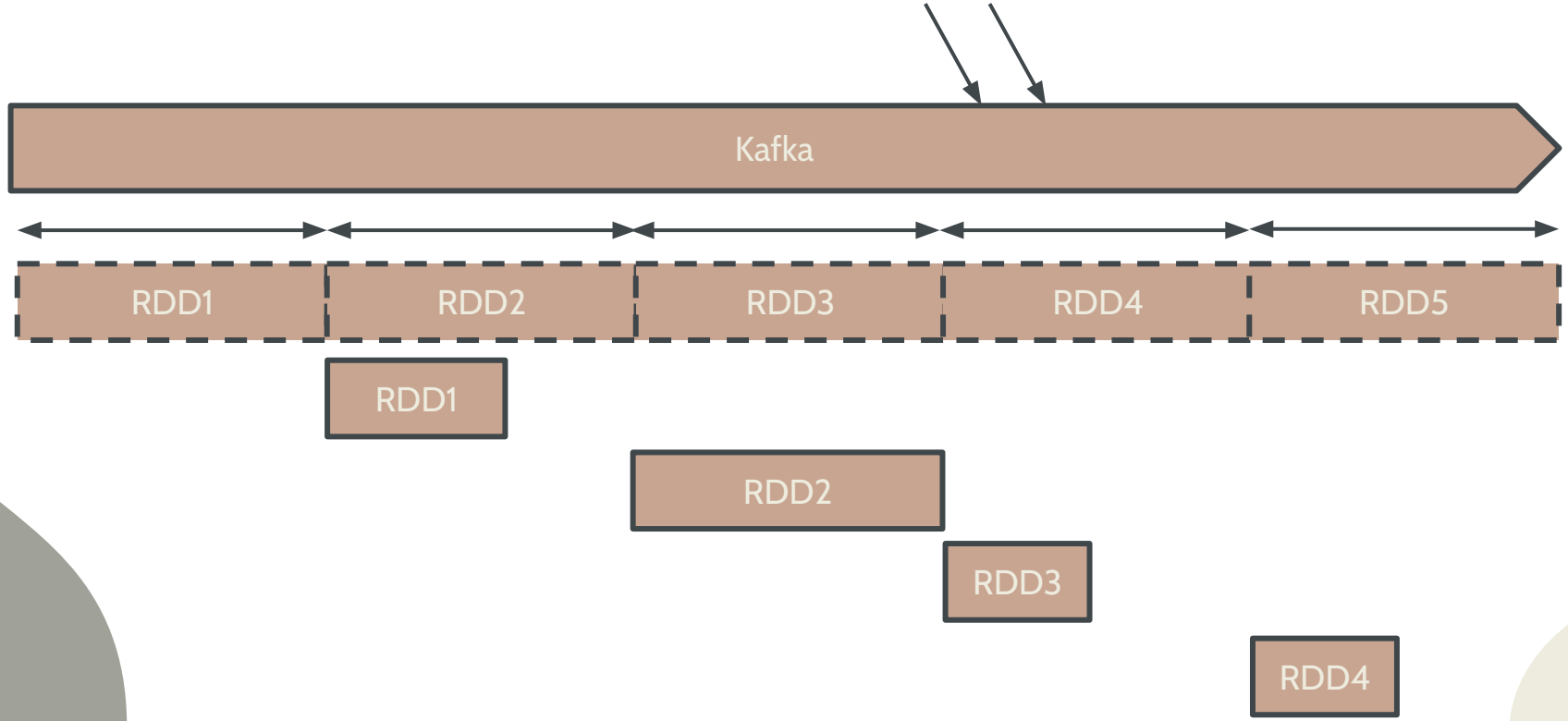
# DStream



# DStream



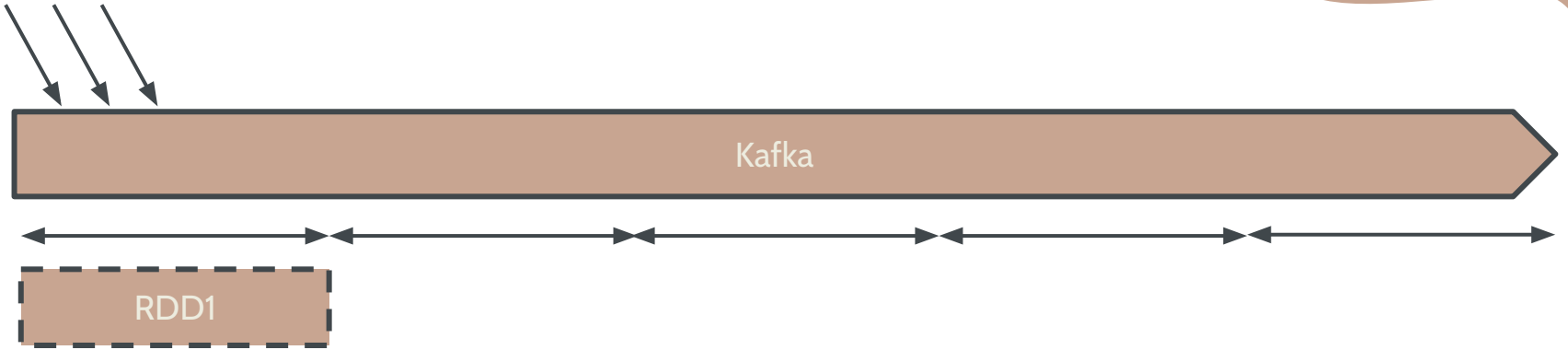
# DStream



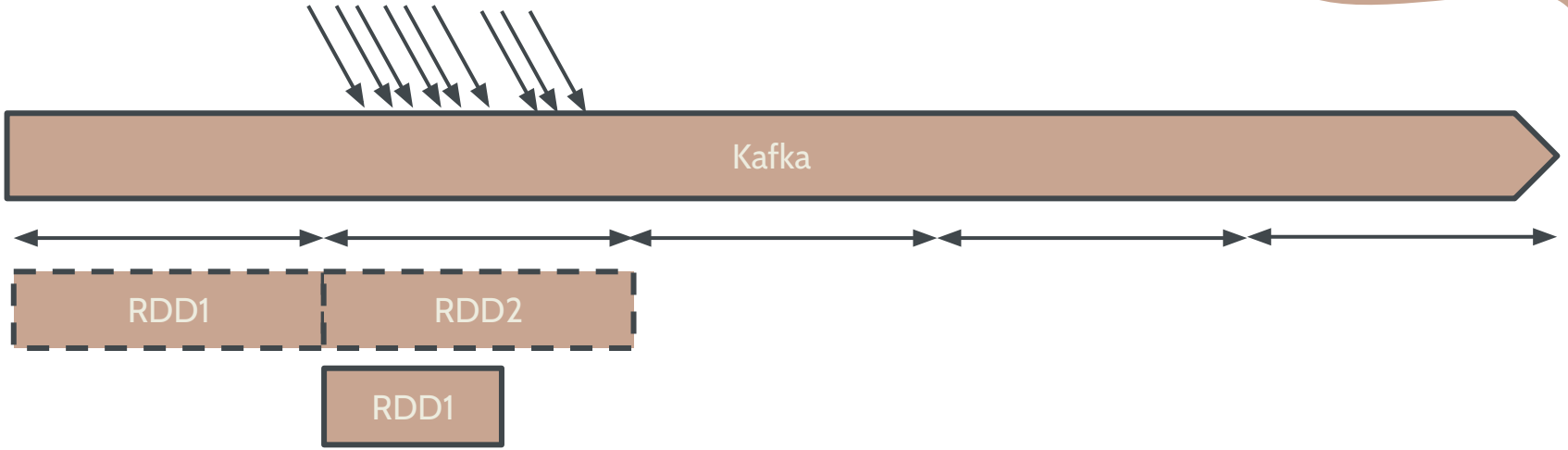
# DStream

- Uno de los mayores problemas de **SparkStreaming** es calcular la ventana
- Siempre hay que tener en cuenta el tamaño máximo(aproximado) de eventos que se pueden recibir
- A partir de esta cantidad se hacen pruebas para calcular el máximo tiempo de ejecución
- Teniendo en cuenta estos dos factores se calcula la ventana de tiempo y se programa el proceso
- Recordar que si se quiere cambiar la ventana de tiempo tenemos que parar el conexto con lo que ello conlleva

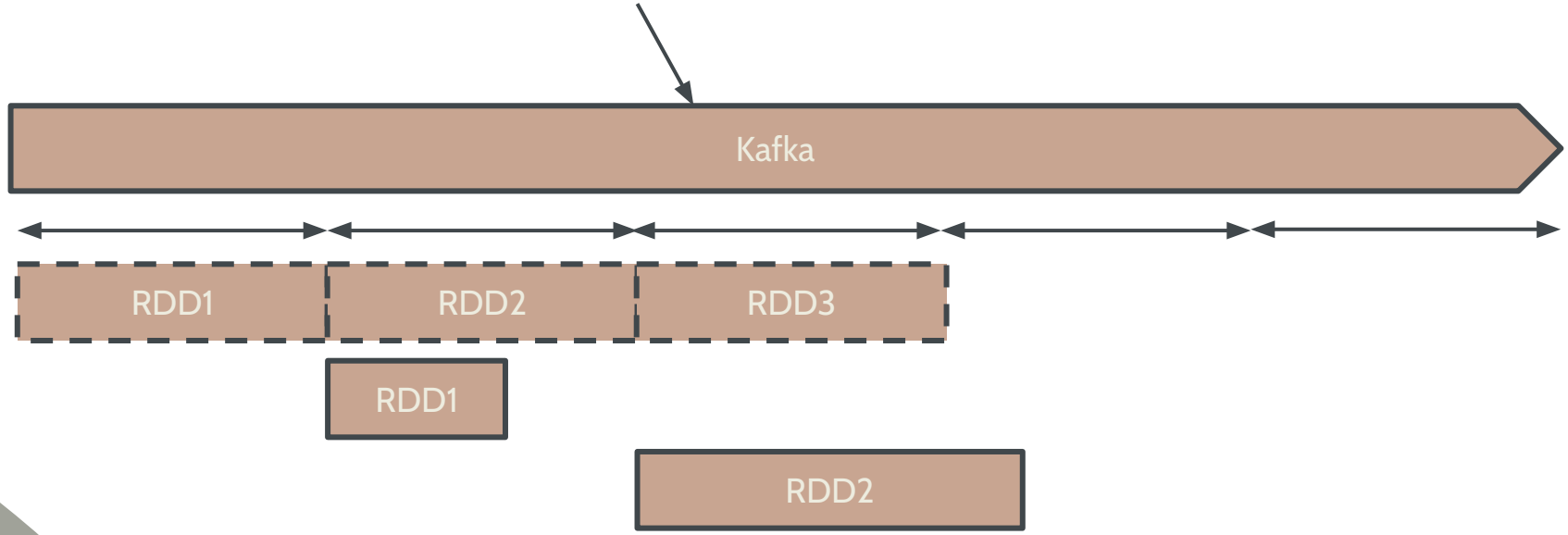
# Problema con el micro-batching



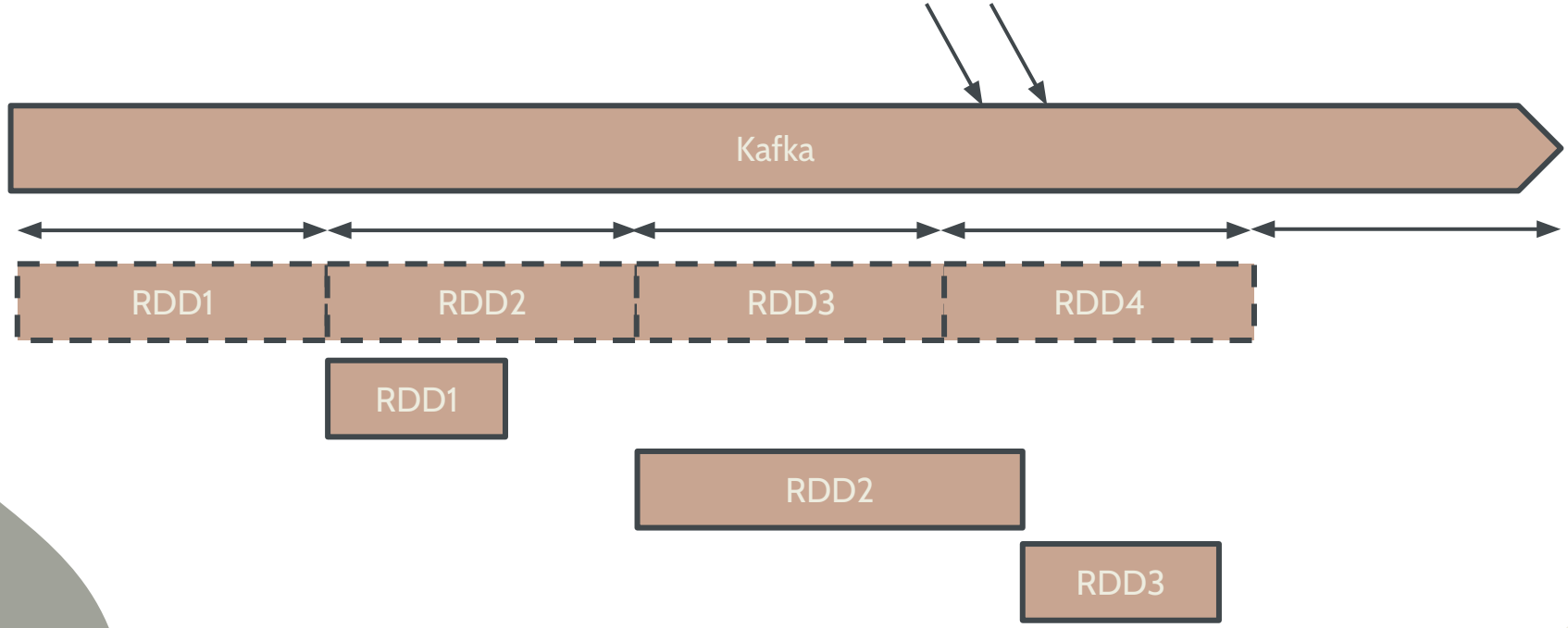
# Problema con el micro-batching



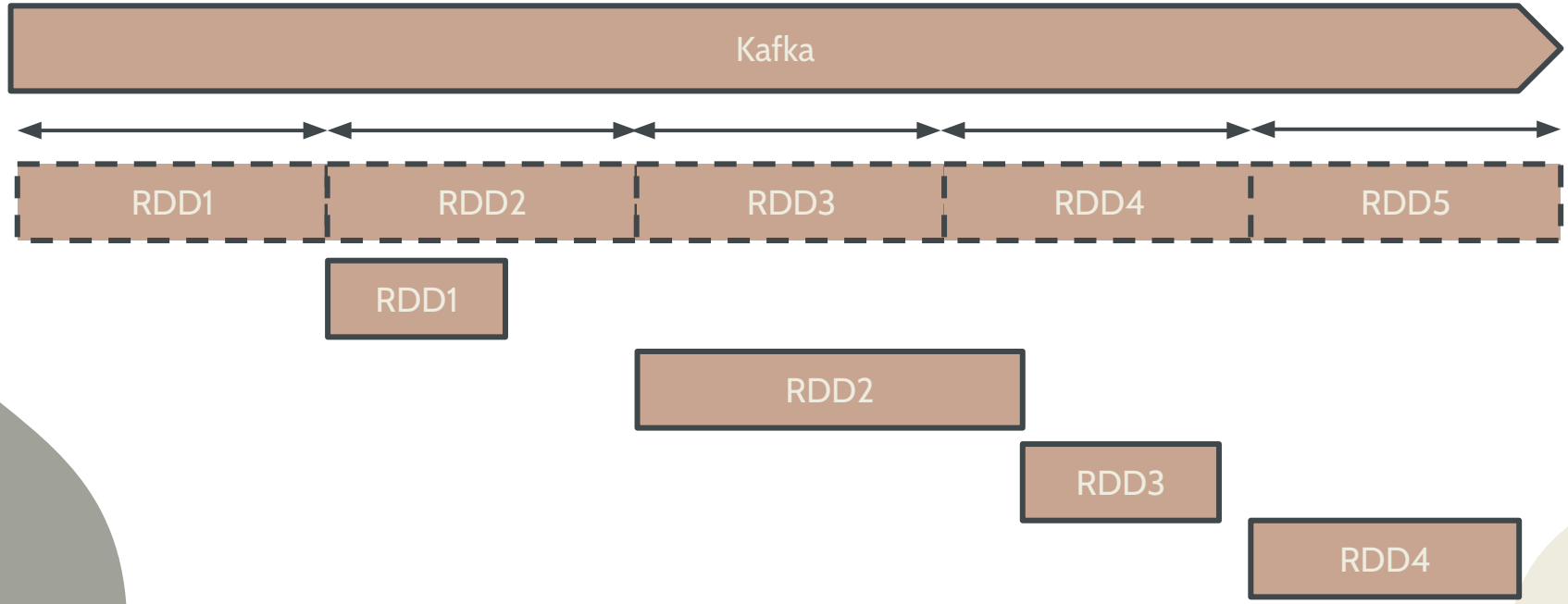
# Problema con el micro-batching



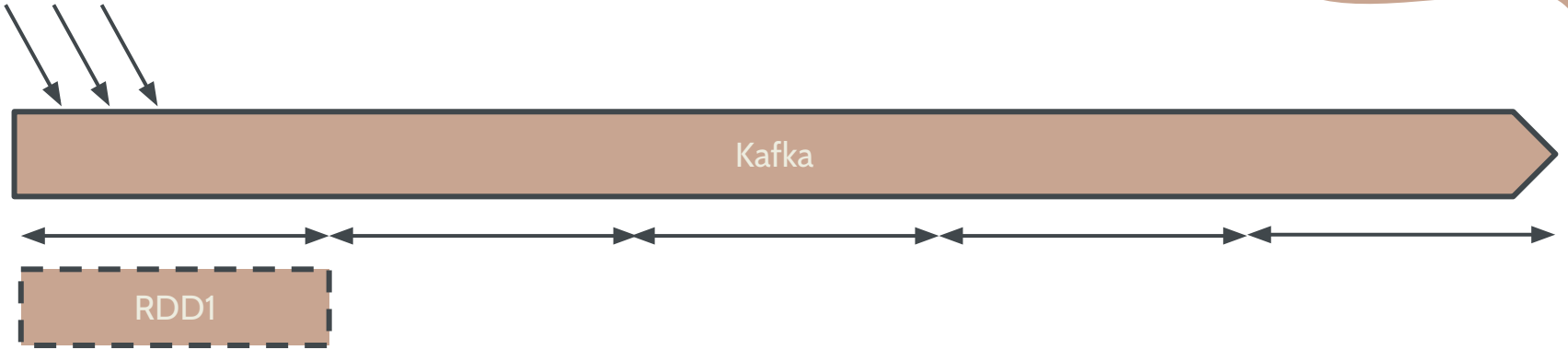
# Problema con el micro-batching



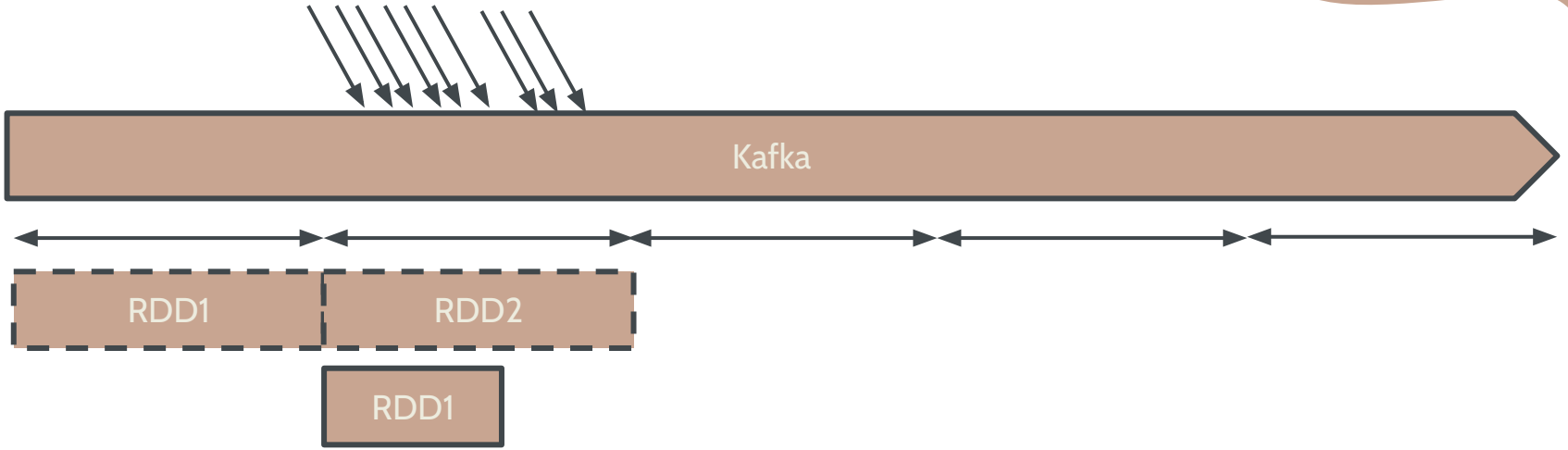
# Problema con el micro-batching



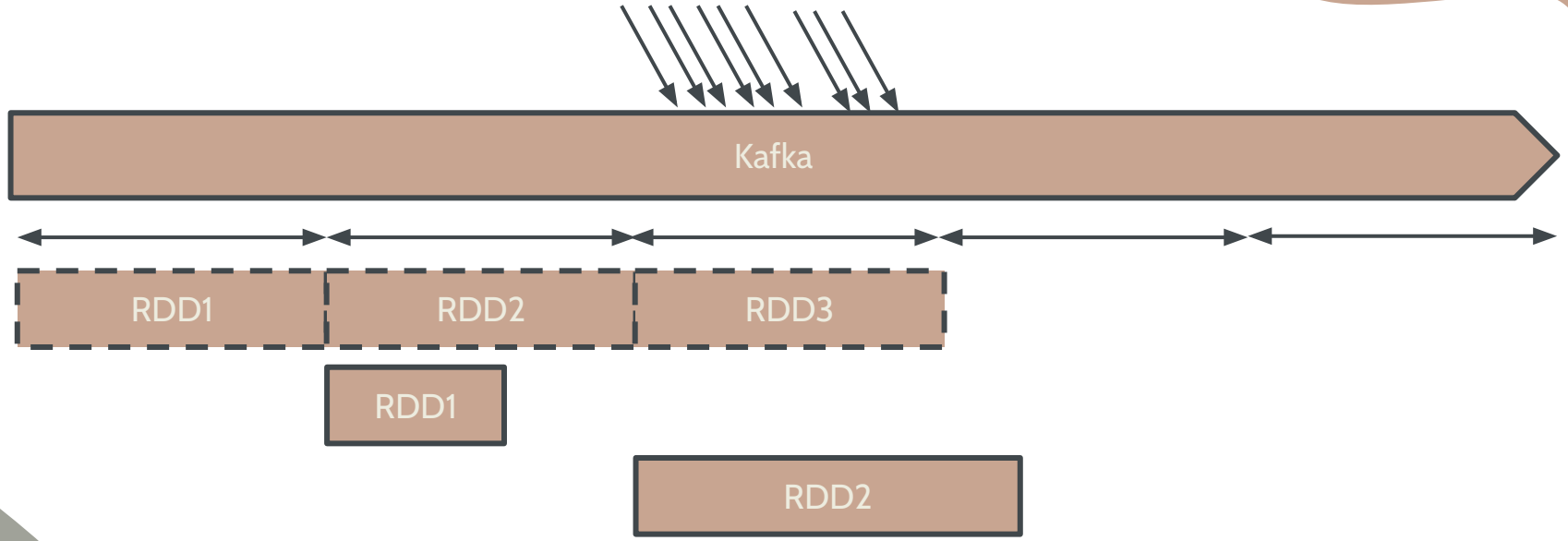
# Problema con el micro-batching



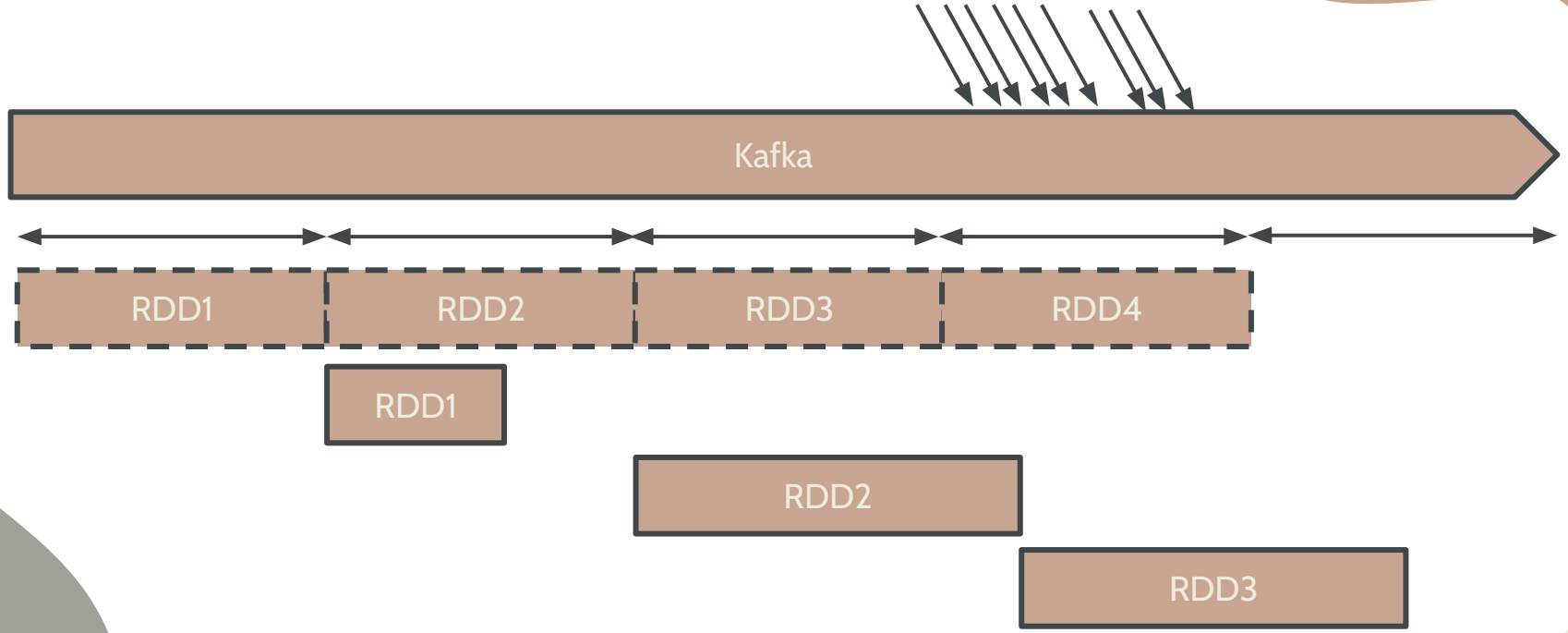
# Problema con el micro-batching



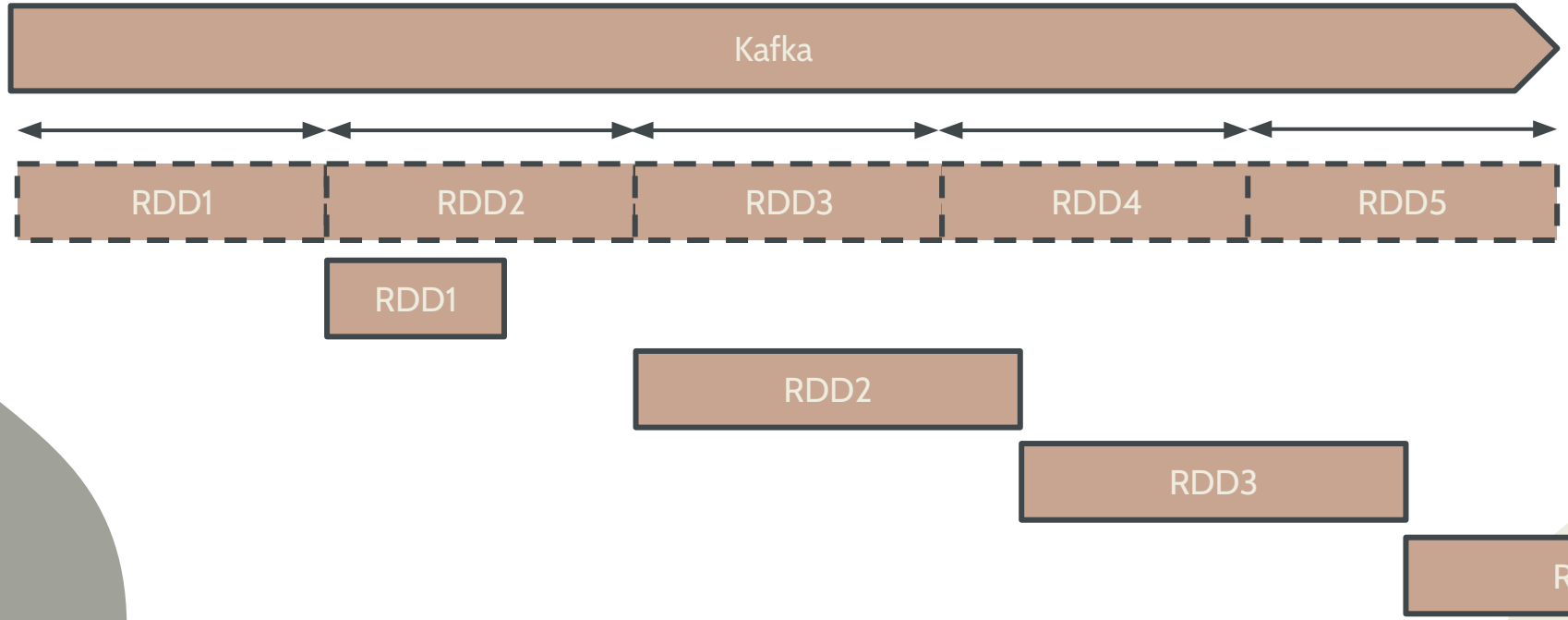
# Problema con el micro-batching



# Problema con el micro-batching



# Problema con el micro-batching



# DStream

- Al igual que los RDD o los Dataframe los DStream se pueden instanciar de varias maneras:
    - Creados desde una fuente externa: Se llaman InputDStream y son los encargados de empezar el flujo de Streaming. Los más básicos son:
        - File: Hace comprobaciones en cada batch de tiempo sobre nuevos ficheros subidos al path indicado y los convierte en DStream

```
>>> ssc = StreamingContext(sc, 1)
>>> ssc.checkpoint(checkpointDirectory)

>>> tweets_strored = ssc.textFileStream(“tweets”)
```

# DStream

- Al igual que los RDD o los Dataframe los DStream se pueden instanciar de varias maneras:
    - Creados desde una fuente externa: Se llaman InputDStream y son los encargados de empezar el flujo de Streaming. Los más básicos son:
        - Desde una cola: Sólo se usa en testing y académicamente, se genera una cola de Arrays y cada posición de la cola representa un micro-batch

```
>>> ssc = StreamingContext(sc, 1)
>>> ssc.checkpoint(checkpointDirectory)

>>> dateVistis = ssc.queueStream([
[(20190101, 'Web', 1), (20190102, 'Web', 3), (20190102, 'Store', 2)],
[(20190101, 'Store', 1), (20190102, 'Store', 3), (20190102, 'Instagram', 2), (20190202,
'Web', 2), (20190103, 'Web', 7)], [(20190202, 'Store', 2), (20190103, 'Store', 8)],
[(20190402, 'Store', 2), (20190403, 'Store', 8)]])
```

# DStream

- Al igual que los RDD o los Dataframe los DStream se pueden instanciar de varias maneras:
    - Creados desde una fuente externa: Se llaman InputDStream y son los encargados de empezar el flujo de Streaming. Los más básicos son:
        - Desde un socket: Se usa principalmente para testeo, recoge Strings de un puerto configurado como parámetro

```
>>> ssc = StreamingContext(sc, 1)
>>> ssc.checkpoint(checkpointDirectory)

>>> socketLines = ssc.socketTextStream("localhost", 9999)
```

# DStream

- Al igual que los RDD o los Dataframe los DStream se pueden instanciar de varias maneras:
    - Kafka: Apache Kafka es un bus de datos distribuido tolerante a fallos, usando la réplica como protocolo de fallo
    - Kafka divide el flujo de datos en topics en los que los Productores insertan información que es consumida una sóla vez por cada Consumidor, normalmente se consume como json serializado

# DStream

```
# Uso: spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5.py
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.streaming.kafka import KafkaUtils

# Establecer la configuración para la conexión con Kafka
>>> kafka_broker_hostname='{IP_ADDRESS_OF_KAFKA_SERVER}'
>>> kafka_consumer_portno='9092'
>>> kafka_broker=kafka_broker_hostname + ':' + kafka_consumer_portno
>>> kafka_topic_input='tweets'

>>>  tweetsKafkaStream = KafkaUtils.createDirectStream(ssc, [topic],
{"metadata.broker.list": kafka_broker})
```

# DStream

- Al igual que los RDD o los Dataframe los DStream se pueden instanciar de varias maneras:
    - Creados desde otro **DStream**: Al igual que sucedía con los **RDD** cualquier transformación que apliquemos a un **DStream** nos delvolverá otro **DStream** inmutable
    - Al igual que los **RDD** los **DStream** siguen una lazy evaluation por lo que hace falta que haya un **action** para que arranque todo el flujo

```
>>> hash_tags_stored = tweets_strored.flatMap(lambda tweet: [ word for word in tweet.split(" ") if word.startswith("#")])
```

# DStream-transform

- Como hemos dicho casi todas las operaciones que se pueden hacer sobre los RDD se pueden hacer sobre los DStream, por ejemplo no podemos ordenarlos por clave
- Hay un reducido número de operaciones que no se pueden realizar directamente sobre los DStream pero si se pueden hacer sobre los RDD subyacentes a ese DStream
- Para ello podemos usar el método transform de los DStream

transfrom(f: RDD[T]⇒ RDD[U]): DStream[V]

# DStream-transform

```
>>> tweetsStream = inputStream.flatMap(lambda line: line.split(" ").filter(lambda word:
word.startswith("#")).map(lambda hashtag : (hashtag.replace("#", ""), 1)))

>>> sum_hash_tags = tweetsStream.reduceByKey(lambda c1,c2: c1+c2).transform(lambda rdd:
rdd.map(lambda tuple: (tuple[1], tuple[0])).sortByKey(False))
>>> sum_hash_tags.pprint()
>>> ssc.start()
>>> ssc.awaitTermination()

-------------------------------------------
Time: 2021-11-15 00:29:35
-------------------------------------------
(MorataDimision, 2)
(Volcan, 1)
-------------------------------------------
Time: 2021-11-15 00:29:36
-------------------------------------------
(MorataDeMiVida, 1)
(Qatar2022, 1)
```

# DStream-transform

- Otra, por no decir la mayor, funcionalidad del método transform es fusionar los mundos de Batch y Streaming

- Además no sólo podemos juntar Batch y Streaming sino que también podemos juntar SQL en batch y Streaming**.

- Con esta opción podemos juntar datos históricos con datos en tiempo real para hacer más potente aún a Spark Streaming

# DStream-transform

```
>>> tweetsStream = inputStream.flatMap(lambda line: line.split(" ").filter(word =>
word.startsWith("#")).map(hashtag => (hashtag.replace("#", ""), 1)))

>>> sum_hash_tags = tweetsStream.reduceByKey(lambda c1,c2: c1+c2).transform(lambda rdd:
rdd.map(lambda tuple: (tuple[1], tuple[0])).sortByKey(False))

>>> authors = spark.read.parquet("authors")

def calculate_treding_topic_by_author(rdd, authors):
   dfTweet =  rdd.toDF(“author”, “hashtag”, “counter”)
   author_condition = col(“author”) == col(“tweet_author”)
   return dfTweet.join(authors, author_condition).orderBy(‘counter.desc).limit(10)

>>> trending_topic_by_author = sum_hash_tags.transform(lambda rdd:
calculate_treding_topic_by_author(rdd, authors))
```

# DStream-acciones

- El número de acciones asociadas a un DStream disminuye mucho con respecto a un RDD
- Muchas de las acciones de un RDD se convierten en transformaciones porque la información no puede ser devuelta al Driver
- De las opciones que existen para DStream una en concreto ejecuta parte de su código en el Driver

# DStream-acciones

- **pprint**: imprime los 10 primeros registros por micro  batch de nuestros registros. así como el timestamp de cuando se ejecución la acción

- **saveAsHadoopFiles**:  admite dos parámetros que determina dónde guardamos nuestros datos, la estructura de las carpetas será la siguiente:prefix-TIME_IN_MS[.suffix] siendo los parámetros prefix y suffix

- **saveAsTextFiles**:  admite dos parámetros que determina dónde guardamos nuestros datos en modo fichero de texto, la estructura de las carpetas será la siguiente:prefix-TIME_IN_MS[.suffix] siendo los parámetros prefix y suffix

- **foreachRDD**: La acción más común en Spark Streaming, admite una función que no devuelve nada la cual se ejecuta en el Driver por lo que se puede usar cualquier clase. Esta acción suele llamar a acciones de los RDD y volcar los datos en cualquier sistema de almacenamiento, a un socket, a un broker de Kafka, etc

# DStream- operaciones con estado

- Hasta ahora sólo se ha hablado de cómo **Spark Streaming** trata la información mirco-batch a micro-batch
- Algunos casos de uso requieren mezclar la información de distintos micro-batch para dar el resultado
- Hay varias formas de añadir información entre distintos **micro-batchs**
    - **Operaciones en Ventana**: se mezcla la información de **N micro-batchs** fijos
    - **Operaciones con estado**: Se actualiza el valor de una clave con la información de distintos **micro-batchs**

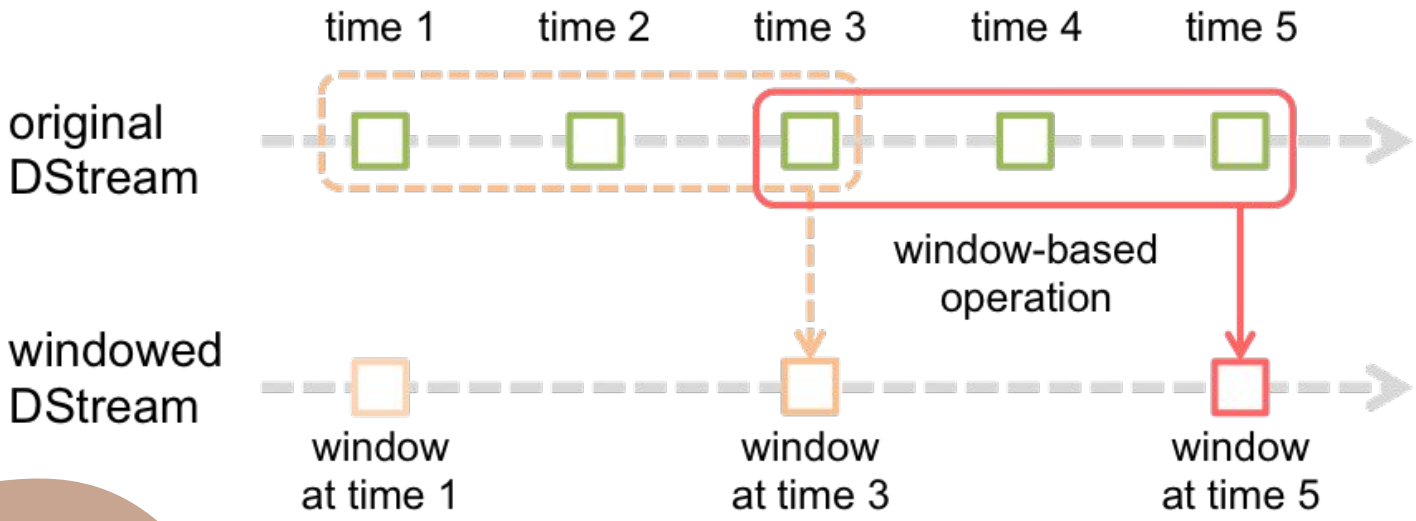
# DStream- operaciones con ventana

- Este nuevo DStream no tratará los datos por cada microbatch si no que se computará los correspondientes a los microbatchs que entren en su ventana de tiempo

- La duración de la ventana tiene que ser múltiplo del tiempo de ventana definido en el **StreamingContext**

**window**(window: Duration) :DStream[U]

# DStream- operaciones con ventana



# DStream- operaciones con ventana

- Se ejecutará la función pasada por parámetro a todos los valores por clave que se encuentre dentro de la duración de la ventana

- La duración de la ventana tiene que ser múltiplo del tiempo de ventana definido en el **StreamingContext**

**reduceByKeyAndWindow**(f: (V, V) ⇒ V, windowDuration: Duration) :DStream[U]

# DStream- operaciones con ventana

```
>>> ssc = StreamingContext(sc, 1)
>>> dateVistis = ssc.queueStream([
[('Web', 1), ('Web', 3), ('Store', 2), ('social_media', 1), ('Store', 2)],
[('Store', 1), ('Web', 3), ('Store', 2), ('Web', 2), ('social_media', 3)],
[('Web', 2), ('Store', 1), ('social_media', 1)]])
>>> visit_aggregated = dateVistis.reduceByKeyAndWindow(lambda x, y: x + y, 3)
>>> visit_aggregated.pprint()
-------------------------------------------
Time: 2021-11-14 10:00:54
-------------------------------------------
('Store', 4)
('social_media', 1)
('Web', 4)
-------------------------------------------
Time: 2021-11-14 10:00:55
-------------------------------------------
('Store', 7)
('social_media', 4)
('Web', 9)
```

# DStream- operaciones con ventana

```
-------------------------------------------
Time: 2021-11-14 10:00:56
-------------------------------------------
('Store', 4)
('social_media', 4)
('Web', 7)

-------------------------------------------
Time: 2021-11-14 10:00::57
-------------------------------------------
('Store', 1)
('social_media', 1)
('Web', 2)

-------------------------------------------
Time: 2021-11-14 10:00:58
-------------------------------------------
```

# DStream- operaciones con ventana

- A las funciones de ventana se les puede suministrar un parámetro que determine el periodo de *sliding*
- Este parametro determina el periodo que espera **Spark Streaming** hasta lanzar el siguiente cálculo de ventana
- Este parámetro tambien tiene que ser múltiplo del tiempo de ventana

**reduceByKeyAndWindow**(f: (V, V) ⇒ V, windowDuration: Duration, slideDuration=None ):DStream[U]

# DStream- operaciones con ventana

```
>>> ssc = StreamingContext(sc, 1)
>>> dateVistis = ssc.queueStream([
[('Web', 1), ('Web', 3), ('Store', 2), ('social_media', 1), ('Store', 2)],
[('Store', 1), ('Web', 3), ('Store', 2), ('Web', 2), ('social_media', 3)],
[('Web', 2), ('Store', 1), ('social_media', 1)]])
>>> visit_aggregated = dateVistis.reduceByKeyAndWindow(lambda x, y: x + y, 3, 2)
>>> visit_aggregated.pprint()

-------------------------------------------
Time: 2021-11-14 10:00:54
-------------------------------------------
('Store', 4)
('social_media', 1)
('Web', 4)
-------------------------------------------
Time: 2021-11-14 10:00:56
-------------------------------------------
('Store', 4)
('social_media', 4)
('Web', 7)
```

# DStream- operaciones con ventana

```
-------------------------------------------
Time: 2021-11-14 10:00:58
-------------------------------------------
('Store', 1)
('social_media', 1)
('Web', 2)
```

# DStream- operaciones con estado

- Algunas casuísticas requieren que Spark agregue información de todo el flujo de datos de Streaming
- Para estas casuisticas **Spark Streaming** nos permite un tipo de operaciones que mantienen un estado para los datos en cada **micro-batch**
- Este tipo de operaciones pueden hacer que la memoria global de nuestro cluster se llene con todos los objetos de la historia del flujo de datos
- En las versiones actuales de **Spark Streaming** permite limpiar parte del **DStream** con la información agregada para evitar que se llene la memoria del cluster

# DStream- operaciones con estado

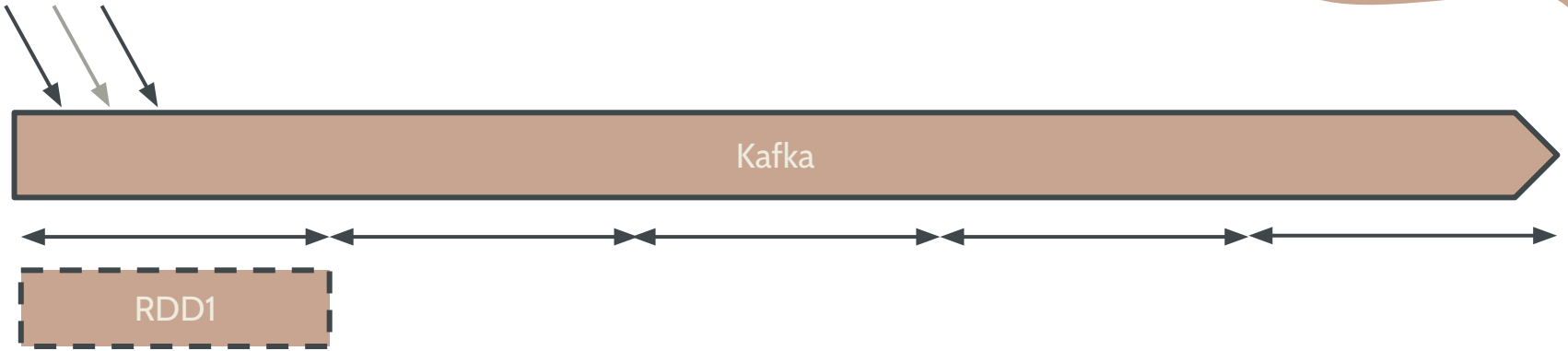
```
>>> def store_min_max_temperatures (newValues, oldValues):
      max_in_batch = max([field[1] for field in new_values])
      min_in_batch = min([field[2] for field in new_values])
      time_stored = max([field[0] for field in new_values])
      if oldValues is None :
          max = max_in_batch
          min = min_in_batch
          time_stored = time_stored
      else:
          max = max([max_in_batch, oldValues[1]])
          min =  min([min_in_batch, oldValues[2]])
          last_time_stored =  oldValues[0]
      if time_stored - last_time_stored < 200:
          return [time_stored, max, min]
      else:
          return None
```

# DStream- operaciones con estado

```
>>> ssc = StreamingContext(sc, 1)
>>> temperatures = ssc.queueStream([
[[('Madrid', [(20210801,10, 35),(20210831, 20,25)]),('Moscow', [(20210801,-1,
24),(20210831, 20,30)]],
[[('Madrid', [(20210930,10, 20),(20211001, 10,15)]),('Moscow', [(20210901,20,
26),(20210930, -1,15)]],
[[('Madrid', [(20211102,5, 15),(20211130, 7,20)]),('Moscow', [(20211001, -1,16),(20211030,
-10,10)]]])
>>> min_max_temperatures = temperatures.updateStateByKey(lambda x, y:
store_min_max_temperatures(x,y))
>>> min_max_temperatures.pprint()
-------------------------------------------
Time: 2021-12-15 00:29:35
-------------------------------------------
('Madrid', (20210831,10,35))
('Moscow', (20210831,-1,30))
```

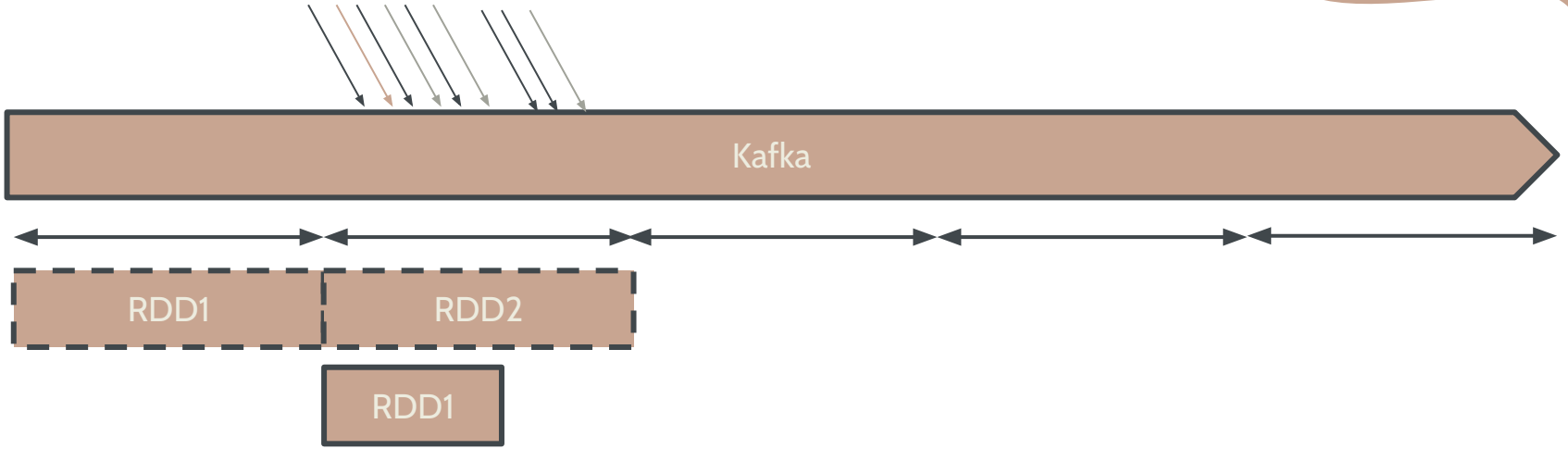
# DStream- operaciones con estado

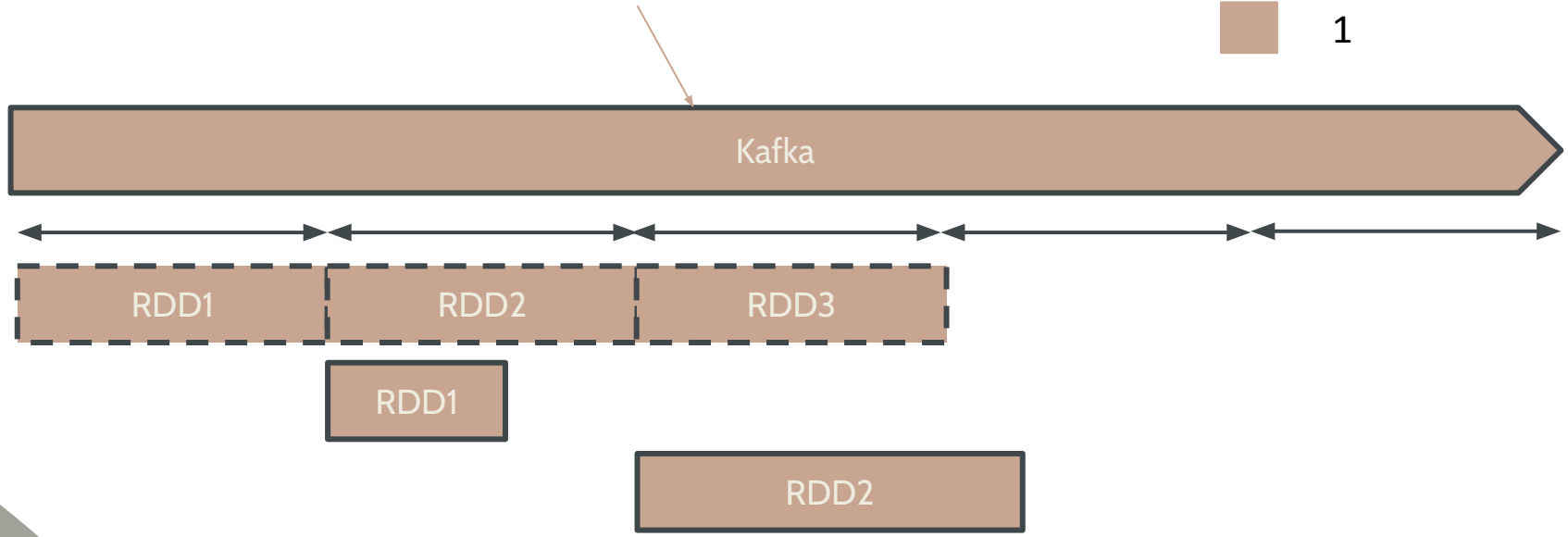
```
-------------------------------------------
Time: 2021-12-15 00:29:35
-------------------------------------------
('Madrid', (20210831,10,35))
('Moscow', (20210831,-1,30))
-------------------------------------------
Time: 2021-12-15 00:29:36
-------------------------------------------
('Moscow', (20210930,-1,30))
-------------------------------------------
Time: 2021-12-15 00:29:37
-------------------------------------------
('Moscow', (20211030,-10,30))
-------------------------------------------
Time: 2021-12-15 00:29:38
-------------------------------------------
('Moscow', (20211030,-10,30))
```

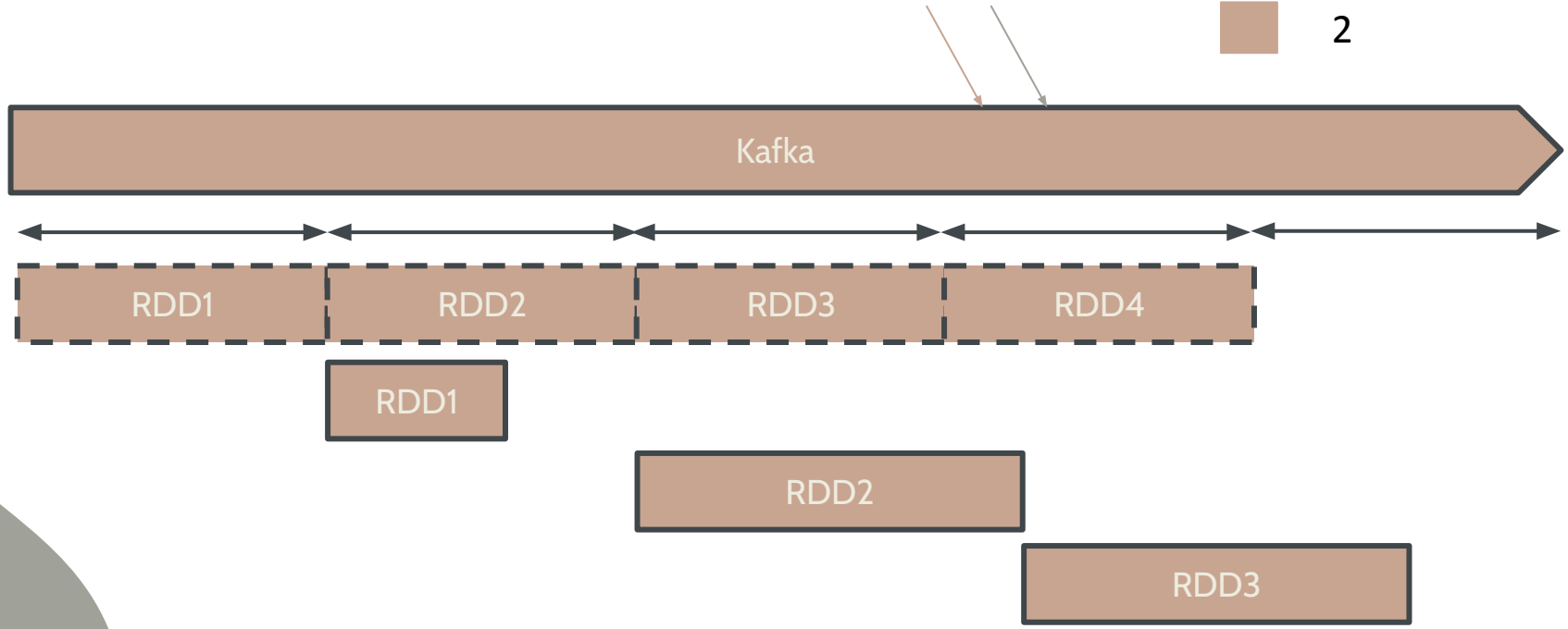

Kafka
RDD1

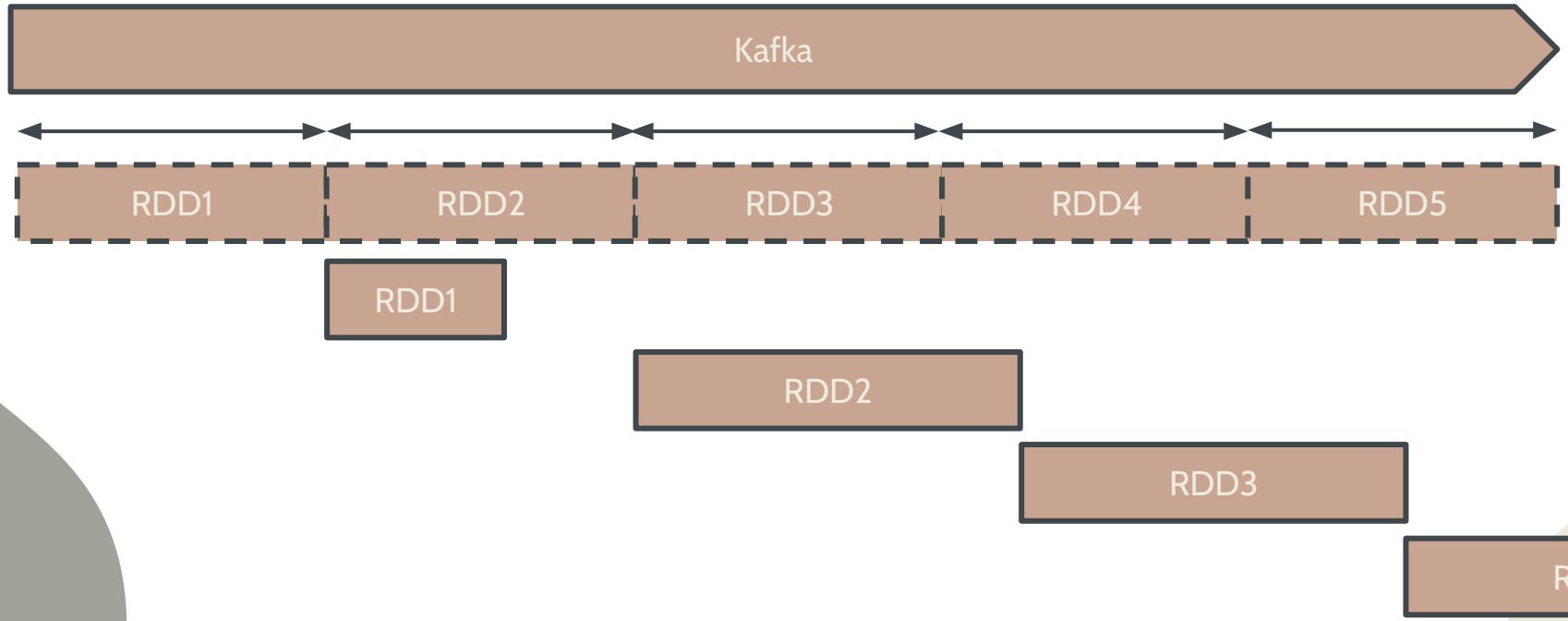
1

2




4
7
1
Kafka
RDD1
RDD2
RDD3
RDD1
RDD2


4
7
2
Kafka
RDD1
RDD2
RDD3
RDD4
RDD1
RDD2
RDD3


5
7
3
Kafka
RDD1
RDD2
RDD3
RDD4
RDD5
RDD1
RDD2
RDD3
RDD4

# SparkStreaming