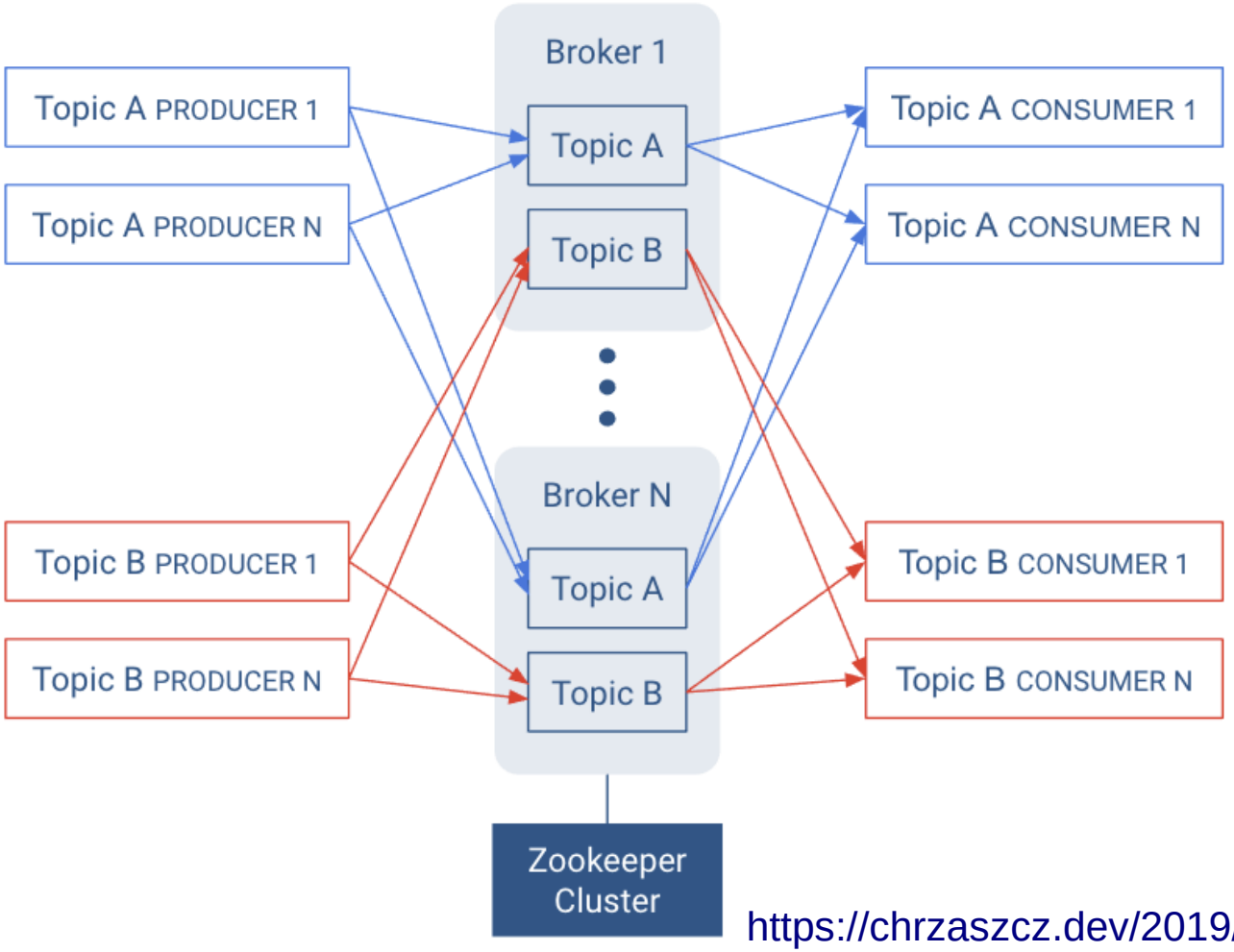
Apache Kafka

Kafka 101

- Kafka is a distributed system, which allows you to construct asynchronous processing of events, publish-subscribe mechanism, or distribute work evenly among worker servvices.



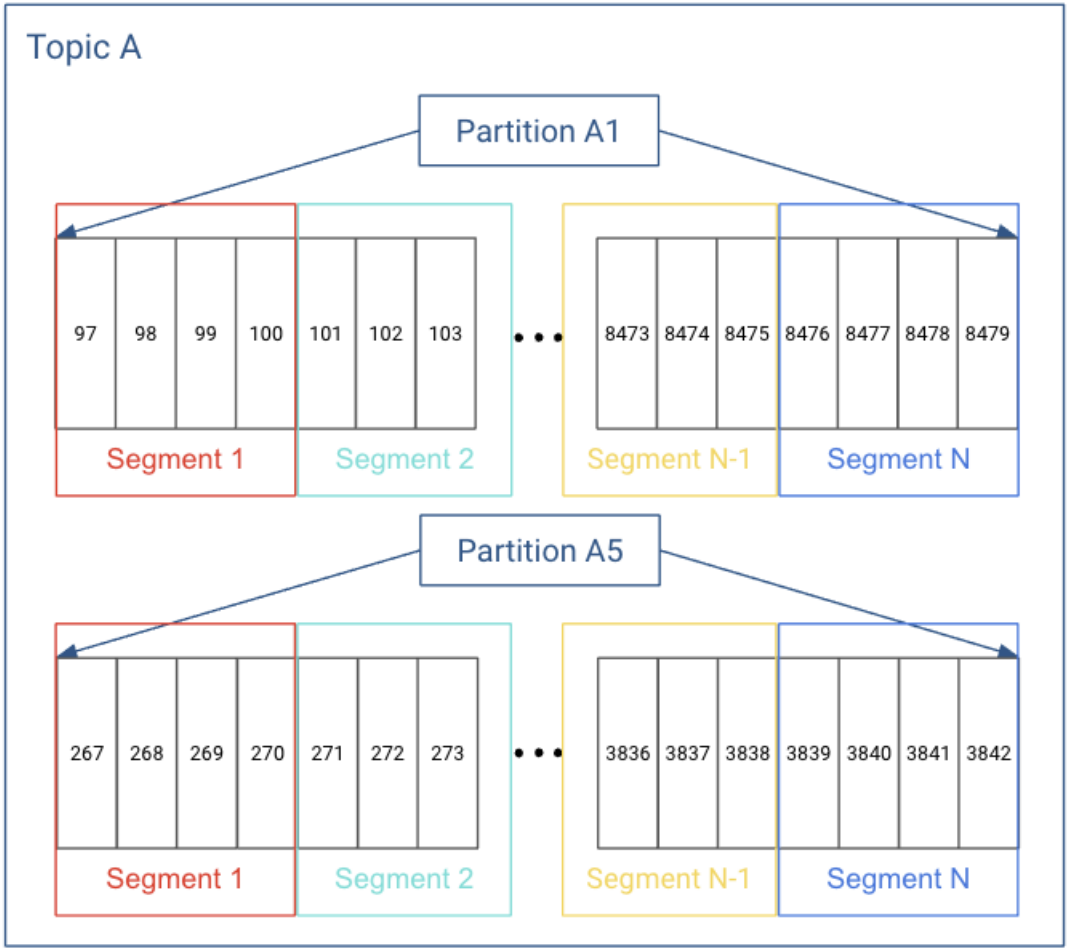
https://chrzaszcz.dev/2019/05/26/kafka-101/

Basic concepts

- Broker
  - server (onprem, virtual, container), on which runs Kafka process
- Message (Event)
  - Message contains a pair (key – value)
    - Key and value can have any (even different) data type – scalar value or even more complex object like JSON.
    - Key isn’t something like primary key in relational database, but for example name of sensor, ID / name of business operation etc. It can be NULL, but for various reasons it’s a good idea if it’s present.
    - Message is immutable

Basic concepts

- Topic
  - Messages are stored to „topic“ (something like log)
- Partition
  - Topic consists of „partitions“
    - Order of messages is guaranteed only within one partition!
      - Each partition has offsets numbered from zero!
- Segment
  - Data aren’t stored into one big file, but are stored in log segments (physically segments are files on disc)



Nice overview where exactly Kafka stores data:
https://rohithsankepally.github.io/Kafka-Storage-Internals/

Basic concepts

- When key == null, then messages are stored into partitions using round robin algorithm. If messages have key, then it’s guaranteed, that messages with the same key will be in the same partition and will be ordered.
  - Internally key is transformed to hash and based on it’s value the message is stored to appropriate partition. The Producer is responsible for hashing and storing to partition.
- Topic is append only, can only seek by offset. Topics are durable, retention is configurable.
  - Default retention is time retention (7 days) and is configurable. Retention also can be based on Topics size:
    - https://medium.com/@sunny_81705/kafka-log-retention-and-cleanup-policies-c8d9cb7e09f8
- Replication
  - One partition is lead replica, others are follow replicas. Replication factor specifies number of replicas, default value is 1 (meaning that replication is by default disabled, data are only in lead replica)

Cluster

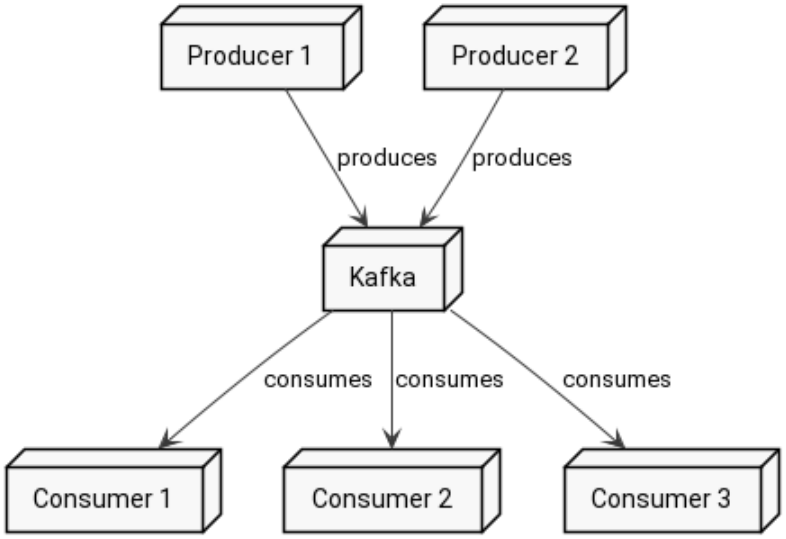
- Kafka uses Apache Zookeeper to keep list of Brokers, which are currently members of cluster. Each Broker has unique ID, which if not specified is automatically generated during Broker startup.
- How to get list of all active brokers in cluster:
  - zookeeper-shell localhost:2181 ls /brokers/ids
- Information about broker with KAFKA_BROKER_ID = 10 in cluster:
  - zookeeper-shell localhost:2181 get /brokers/ids/10
- https://www.baeldung.com/ops/kafka-list-active-brokers-in-cluster
- Controller
  - The first of the Kafka Brokers to join the cluster is co-called Controller and is in charge of allocating partition leaders. When Controller stops working, another working Broker is set as Controller.

Replication

- There are two kinds of replicas:
  - Leader replica
    - Every partition has one replica, which is leader replica. All requests of all Producers and Consumers go through leader for data consistency.
  - Follower replica
    - All other replicas are followers. Followers only job is to replicate messages from leader and to keep up-to-date with state of leader. When leader replica crashes, one of the follower replicas will become the new leader.
- Leader replica has one more role and that’s to control whether follower replicas are in-sync. Replicas don’t have to be in-sync for example due to networking issue, or when broker crashes and data aren’t yet replicated to other brokers.
- Kafka guarantees that every replica is on different broker, thus maximum value of replication factor is equal to number of brokers.

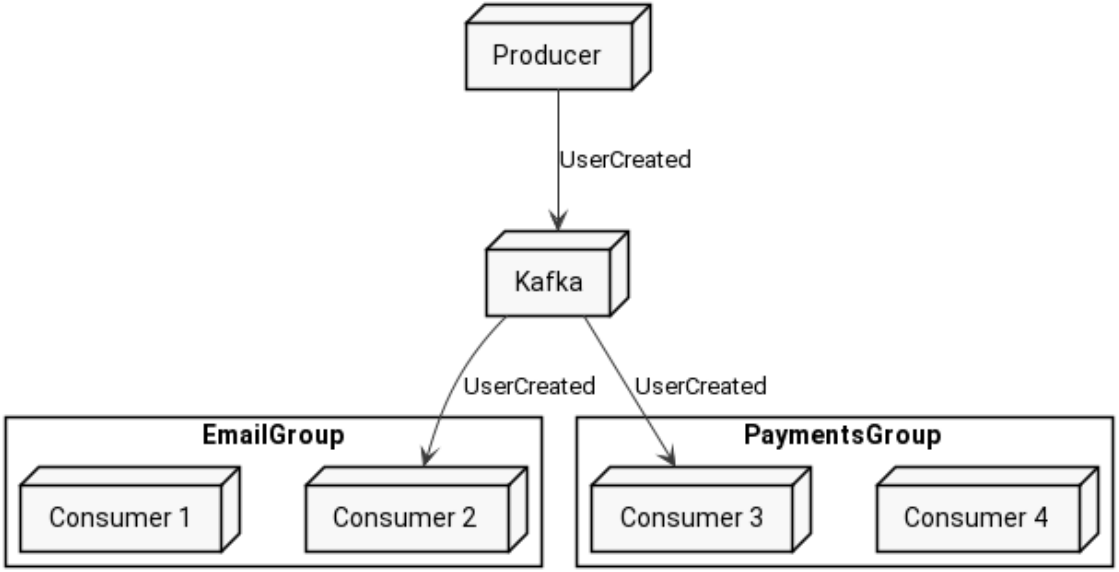
Producer & Consumer

- Producer
  - Producer produces messages to topics
- Consumer
  - Consumer consumes messages from topics
    - Consumers can be grouped to groups (using group.id).
    - Consumer doesn’t remove message from topic (big difference from message-queue), but it keeps offset by which it knows what message to read from.
      - Offsets are stored in topic __consumer_offsets



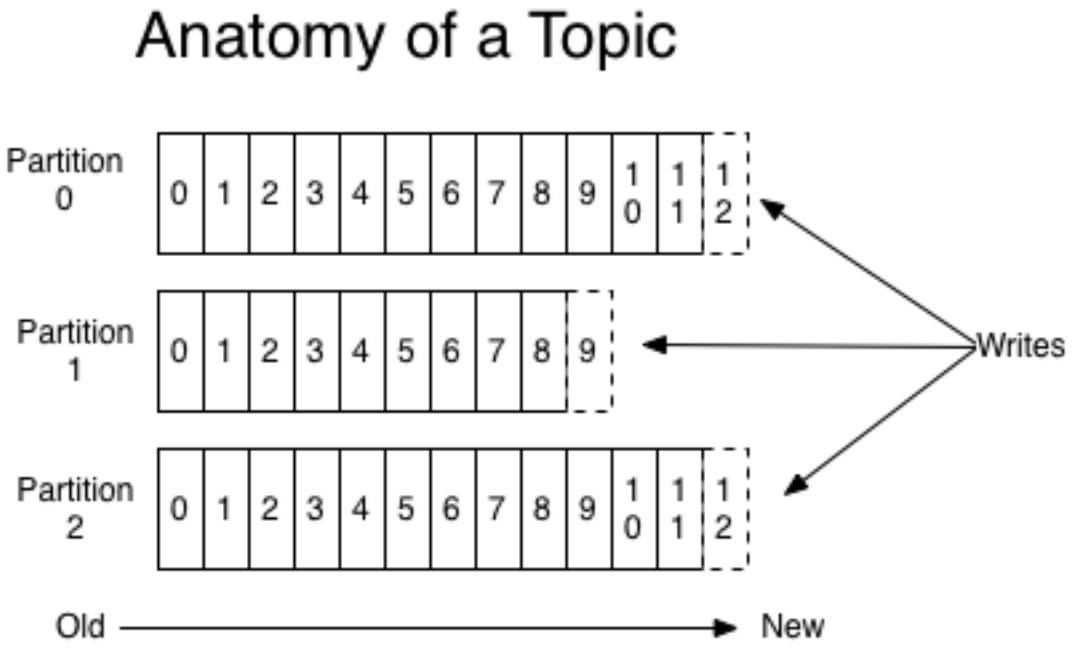
Consumer Group

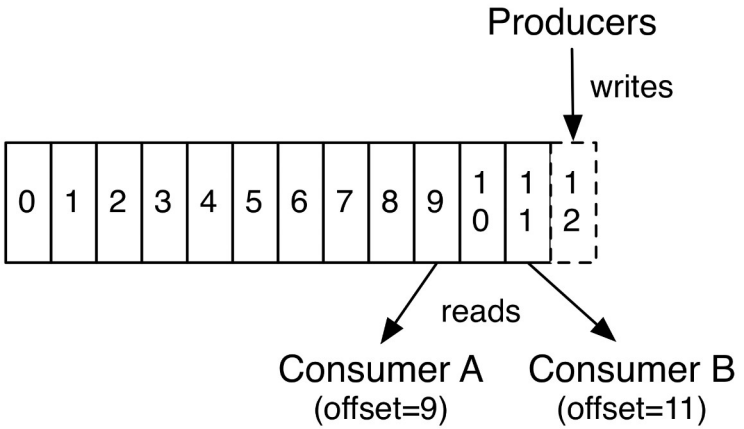
- Consumer Group
  - There can be more Consumers, which have same group.id (that is known as Consumer Group) and thus works horizontal scaling of consumers. Kafka will automatically distribute load between these consumers. It’s not possible to have more active Consumers than number of partitions, but from a single topic can read any number of consumers with different group.id.
    - Each message will be sent to every Consumer group, but inside that group it will be consumed by just one Consumer.



Offset

- Every message when stored to Partition will have allocated an offset. Offset of the first message is 0, second message 1 etc.
- Every message can be uniquely identified by this combination: (topic_name, partition_number, offset)
- Consumer uses offset to specify position in log. There are two kinds of offsets. First is stored in Kafka (commited offset), second is local and is used by consumer for polling (consumer position).





Commited Offset vs. Consumer Position

- Commited Offset
  - Stored to Kafka when Consumer performs commit.
  - Used when Consumer crashes.
  - Example:
    - Consumer C1 just started consuming new topic. It performed fetch of 10 messages from offset 0. After their processing it wants to give information to Kafka that they’re processed. This process is called „commit“. So it performs a commit, then Consumer C1 crashes. Afterwards starts up Consumer C2, which looks to Kafka what has been processed so far and starts processing messages from 10th message.

Commited Offset vs. Consumer Position

- Consumer Position
  - Consumer normally does polling (that’s what it does). During pollingu is not automatically performed commit. That’s performed either peridocally during Xth call of poll method, or mannually calling commit method.
  - In the background, during polling Consumer remembers the current offset of the last processed message in something called Consumer Position.
  - Consumer Position can be changed by calling one of these methods:
    - consumer.seek(somePartition, newOffset)
    - consumer.seekToBeginning(somePartitions)
    - consumer.seekToEnd(somePartitions)

Initial Offset

- When a brand new Consumer Group is added, it has no commited offset. What is the default? We have to choose:
  - Earliest
    - Smallest (oldest) available offset
    - In Consumeru you need to set auto.offset.reset=earliest
  - Latest
    - Largest (newest) available offset
    - This is default
  - None
    - We have to manually specify concrete offset.
- Warning! In Broker is property offsets.retention.minutes (default 24h), which means, that when Consumer Group drops out for more than 24h and is setup latest offset, then will be processed items, which were delivered after Consumer startup Consumera (initial offset will be created), therefore data will be lost.
  - https://dzone.com/articles/apache-kafka-consumer-group-offset-retention

Kafka vs. RabbitMQ

- Traditionally, RabbitMQ is a message broker that delivers messages (like postman), while Kafka is a distributed log. Today, however, it offers more or less the same thing (especially since the release of RabbitMQ 3.9, which contains new data structure: Streams).
- Today I would see main differences especially in ecosystem around these two products (Kafka has Kafka Streams, Kafka Connect and Debezium, ksqlDB etc.).
- From performance perspective Kafka has higher throughput, while RabbitMQ has lower message delivery latency.
- https://qr.ae/pG0g2J
- https://tanzu.vmware.com/developer/blog/understanding-the-differences-between-rabbitmq-vs-kafka/

Hello World Kafka (landoop)

- Apache Kafka startup:

```
docker run --rm -it -p 2181:2181 -p 3030:3030 \
-p 8081:8081 -p 8082:8082 -p 8083:8083 -p 9092:9092 \
-e ADV_HOST=127.0.0.1 landoop/fast-data-dev
```

- How to get to command line tools:

```
docker run --rm -it --net=host landoop/fast-data-dev bash
```

- Dashboard:
  - http://localhost:3030/

Hello World Kafka (confluent)

- Apache Kafka startup:

```
git clone https://github.com/confluentinc/cp-all-in-one

cd cp-all-in-one/cp-all-in-one

docker compose up -d
```

- How to get to command line tools:

```
docker run -it --rm --net=host \
confluentinc/cp-zookeeper:5.0.0-beta30 bash
```

- Dashboard:
  - http://localhost:9021

- https://docs.confluent.io/platform/current/quickstart/ce-docker-quickstart.html

kafka-topics

- Create topic:

```
kafka-topics --zookeeper localhost:2181 --create \
--topic first_topic --partitions 3 --replication-factor 1
```

- Display list of topics:

```
kafka-topics --zookeeper localhost:2181 --list
```

- More detailed information about some topic:

```
kafka-topics --zookeeper localhost:2181 --describe --topic first_topic
```

- Delete topic:

```
kafka-topics --zookeeper localhost:2181 --delete --topic first_topic
```

Kafka Broker Configuration

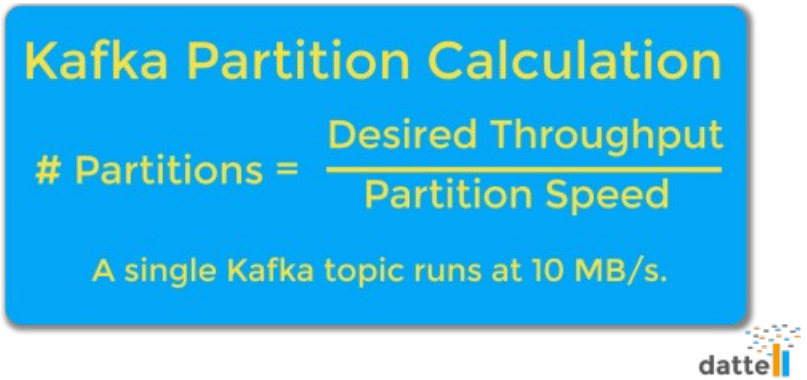
- Kafka Broker defaults to reconsider:
  - auto.create.topics.enable (default: true)
    - It’s not a best practice for an application to automatically create a Topic the first time it is used, especially since each Topic may have a different configuration:
    - log.retention.hours (default: 7 dní): Log retention in Kafka
    - min.insync.replicas (default: 1): When Producer has acks = all, min.insync.replicas specifies minimum number of replicas, that return information that they successfully wrote the message. Correct value is (replication-factor – 1)
    - replication.factor (default: 1): Number of topic replicas. Depends on how many replicas we want to have and number of Brokers, because you cannot have more replikas than brokers.
    - num.partitions (default: 1): Number of partitions. This value determines parallelism.
  - offsets.retention.minutes (default: 24 hodin): After how long unused Consumer offsets in Kafka will be deleted.
- https://www.javierholguera.com/2018/06/13/kafka-defaults-that-you-should-re-consider-i/

Partitions: Throughput

- Topic Partition is a unit of parallelization in Kafka. Both Producer and Consumer can work with different partitions in a fully parallel way. Kafka will send data from one partition only to single Consumer (within one Consumer Group). Thus degree of parallelization(withing one Consumer Group) is limited by number of partitions. Generally, the greater the number of partitions, the greater the throughput.
- https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster/
- Maximal number of partitions in Kafka:
  - Broker: 4 000 partitions
  - Cluster: 200 000 partitions
    - https://blogs.apache.org/kafka/entry/apache-kafka-supports-more-partitions

Number of Partitions

- How many partitions?
  - There are different opinions on this. I like this the most: 10 partitions, or dependding on what is required throughput computed by this formula:



    - https://dattell.com/data-architecture-blog/kafka-optimization-how-many-partitions-are-needed/
  - Consequences of a bad setup:
    - https://docs.cloudera.com/runtime/7.2.10/kafka-performance-tuning/topics/kafka-tune-sizing-partition-number.html

Kafka REST Proxy

- Kafka REST Proxy offers a REST interface for controlling Kafka, which can do:
  - CRUD operations on Topics
  - Simple Producer & Consumer
- https://github.com/confluentinc/kafka-rest
- List of topics:
  - http://localhost:8082/topics

Kafka UI

https://towardsdatascience.com/overview-of-ui-tools-for-monitoring-and-management-of-apache-kafka-clusters-8c383f897e80

- kafdrop:
  - https://hub.docker.com/r/obsidiandynamics/kafdrop
- kowl
  - https://github.com/cloudhut/kowl
- akhq
  - https://github.com/tchiotludo/akhq
- kafka-ui
  - https://github.com/provectus/kafka-ui

Programmatic access to topic

- You can programmatically create / delete / … topics using AdminClient:
  - https://stackoverflow.com/a/45122955/894643
- How to get list of topics:
  - KafkaConsumer#listTopics()
  - OR:
  - AdminClient#listTopics()

kafka-console-producer

- Easily add messages to a topic from console:

```
kafka-console-producer --broker-list localhost:9092 --topic first_topic
```

- Notes:
  - 1 line = 1 message
  - End: CTRL + C
  - Messages go to random partition (because key is not specified)
  - When we add message to non-existent topic, new topic is automatically created (if this feature is not disabled)
  - How to specify key:

```
kafka-console-producer --broker-list localhost:9092 --topic first_topic \
--property "parse.key=true" --property "key.separator=:"
```

kafka-console-consumer

- Easily read messages from topic (output is to console):

```
kafka-console-consumer --bootstrap-server localhost:9092 \
--topic first_topic

kafka-console-consumer --bootstrap-server localhost:9092 \
--topic first_topic --from-beginning

kafka-console-consumer --bootstrap-server localhost:9092 \
--topic first_topic --from-beginning --partition 0
```

- Other useful parameters:

`--from-beginning` (reads messages from oldest offset)

`--partition X` (reads messages from partition X)

`--offset Y` (reads messages from offset Y)

`--consumer-property group.id=mygroup1` (sets group.id)

Custom Deserializer

- Kafka-console-consumer (and other tools) can work only with some data formats (kafka-console-consumer can practically work only with Strings). If you use different formats, it may be needed to add custom deserializer. For example when values are in Double:

```
kafka-console-consumer --bootstrap-server kafka:9092 \
  --group ConsoleConsumer --from-beginning --topic TODO_TOPIC_NAME \
  --property \
    value.deserializer=org.apache.kafka.common.serialization.DoubleDeserializer
```

- https://stackoverflow.com/questions/44530773/kafka-stream-giving-weird-output

kafkacat

- Kafkacat is an alternative to kafka-console-producer/consumer tools:
  - https://docs.confluent.io/platform/current/app-development/kafkacat-usage.html
- Examples:
  - List of topics:
    - `docker run --rm --tty --network kafka_net confluentinc/cp-kafkacat kafkacat -b kafka:9092 -L`
  - Kafkacat acting as a consumer of topic „first_topic“:
    - `docker run --rm --tty --network kafka_net confluentinc/cp-kafkacat kafkacat -b kafka:9092 -t first_topic`
  - Kafkacat acting as a producer to topic „first_topic“:
    - `docker run --rm --tty --network kafka_net confluentinc/cp-kafkacat kafkacat -b kafka:9092 -t first_topic -P`
      - Input should be confirmed by CTRL+D, but for some reason it doesn’t work for me :-(

pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>3.1.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

MyProducer

acks values: 0, 1, -1 (all)

```java
public class MyProducer {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ProducerConfig.ACKS_CONFIG, "1");
        properties.setProperty(ProducerConfig.RETRIES_CONFIG, "3");

        try(Producer<String, String> producer = new KafkaProducer<>(properties)) {
            ProducerRecord<String, String> producerRecord
               = new ProducerRecord<>("first_topic", "mykey", "myvalue");
            producer.send(producerRecord);
            producer.flush(); // can be replaced with properties.setProperty("linger.ms", "1");
        }
    }

}
```

How Producer works

1) Producer establishes a connection with one of the bootstrap servers (Kafka Broker), the best practice is to set up at least two bootstrap servers in the Producer's configuration.
2) The Bootstrap server returns a list of all brokers in the cluster and metadata such as topics, partitions, replication factors etc.
3) Based on this Producer identifies the leader broker, which has the leader partition and writes messages to this partition.

- https://dzone.com/articles/kafka-producer-overview
- Producer performs all these operations in thread kafka-producer-network-thread (Sender), which is daemon thread:
  - https://jaceklaskowski.gitbooks.io/apache-kafka/content/kafka-producer-internals-Sender.html
- Producer IS thread safe!!!
  - https://stackoverflow.com/questions/36191933/using-kafka-producer-by-different-threads

Producer Configuration I.

- acks
  - The number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent.
  - The default value is „all“. This setting refers to the min.insync.replicas setting (in Topicu), which sets the number of brokers which must log a message before an „acknowledge“ is sent to the client. The default value of min.insync.replicas is 1. The correct settings depends on „replication factor“. If the replication factor is 3, then the correct value of min.insync.replicas is 2.
  - https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_acks
- https://strimzi.io/blog/2020/10/15/producer-tuning/
- https://www.javierholguera.com/2018/06/13/kafka-defaults-that-you-should-re-consider-i/

Producer Configuration II.

- retries
  - How many times the Producer will try to deliver the message to Kafka broker. The default value is 0. When retries is set to a higher value than 0, then max.in.flight.requests.per.connection = 1 should be set, otherwise the message that is sent in retry mode could be delivered in wrong order.
    - https://docs.confluent.io/platform/current/installation/configuration/producer-configs.html#producerconfigs_retries
- compression.type
  - Compression is useful for increasing throughput and reducing storage, but may not be suitable for low-latency applications, because compression and decompression will inevitably increase latenci.
  - Supported types: gzip, snappy, lz4, zstd

Producer Configuration III.

- Batching of messages will increase throughput. Important settings:
  - batch.size
    - Maximum batch size
  - linger.ms
    - Maximum time until the batch is sent
  - Messages will be sent if one of these two parameters is reached
- buffer.memory:
  - When Kafka Producer cannot send data to the Broker (for example due to a Broker outage), it will store messages up to buffer.memory. Once the buffer.memory is full (default value: 32 MB), it will wait max.block.ms (default: 60s) if the buffer was empties. If this time elaspes and the buffer is not emptied, then an exception is thrown.

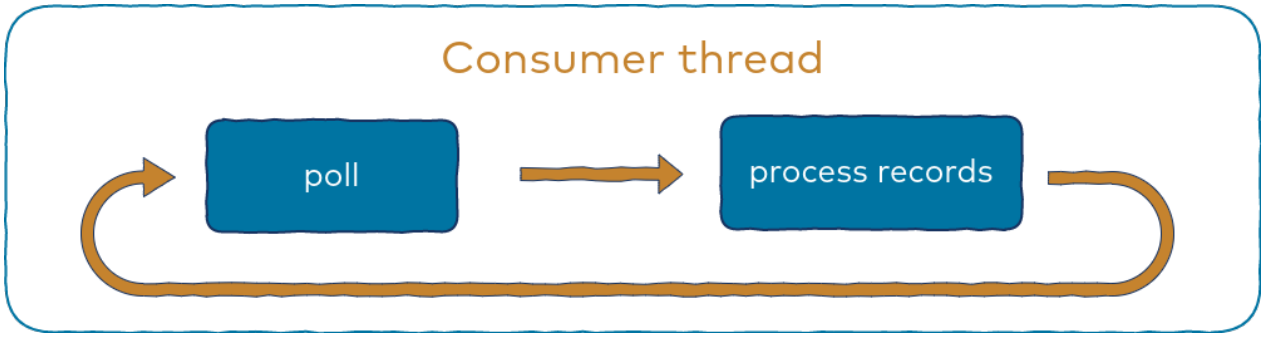
MyConsumer

```java
public class MyConsumer {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.setProperty(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.setProperty(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.setProperty(ConsumerConfig.GROUP_ID_CONFIG, "test");
        properties.setProperty(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        properties.setProperty(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
        properties.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);
        consumer.subscribe(Arrays.asList("first_topic"));
        while(true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofSeconds(1));
            records.forEach(record -> {
                System.out.println(record.key() + ":" + record.value());
            });
        }
    }

}
```

Alternative to:
`consumer.commitSync();`

How Consumer Works I.

- Consumer performs polling:



- By default, Consumer stores offsets in Kafka. You can use `auto.commit.interval.ms` to change commit frequency. Offsets are not commited in separate thread, but are commited in the same thread in which the polling is performed. Only offsets that were processed in the previous call of poll method are commited. Because processing of messages is done between calls of poll methods, offsets of unprocessed messages will never be processed. This guarantees at-least-once delivery.
- Because messages are retrieved and processed in the same thread, they are processed in the same order in which they were written to the partition. This guarantees the order of processing.
- Consumer IS NOT thread safe!!! https://chrzaszcz.dev/2019/06/16/kafka-consumer-poll/
  - https://www.confluent.io/blog/kafka-consumer-multi-threaded-messaging/

How Consumer Works II.

- An interesting configuration of Consumer is `max.poll.records` (default: 500 messages), which sets the maximum number of messages returned in a single poll() metod call. If zou increase this value or your messages in Kafka are larger, then this setting is also important: `max.partition.fetch.bytes` (default: 1 MB). This specifies maximum number of bytes that the server will return per partition.
  - Why these restrictions? In order to protect the Consumer against a situation where are a large number of record in Kafka, or those records have large size.
- Another configuration: `fetch.min.bytes`, which sets minimum number of bytes that Kafka should have in Topic in order to return some data to client (default: 1 byte). This settings increases throughput, but also increases latency. When using it, it may also make sense to set up `fetch.max.wait.ms`, thanks to which Kafka doesn’t wait only for min.bytes, but after wait.ms time has elapsed, the Broker will send data to the client (default: 500 ms).

How Consumer Works III.

- Only records that have been written to all in-sync replicas are sent to the Consumer. This is for data consistency and thanks to this setting there cannot be a situation, where the leader would receive the message, the client would read it, the leader would crash before it could replicate the message to replicas, some replica is promoted to leader and suddenly there wouldn’t be message in Topic, but it would be consumed by client.
  - This also means, that if for some reason data replication between brokers is slow, then it will také longer for messages to reach clients (because first broker waits for them to be replicated to another brokers before it sends them to the clients).

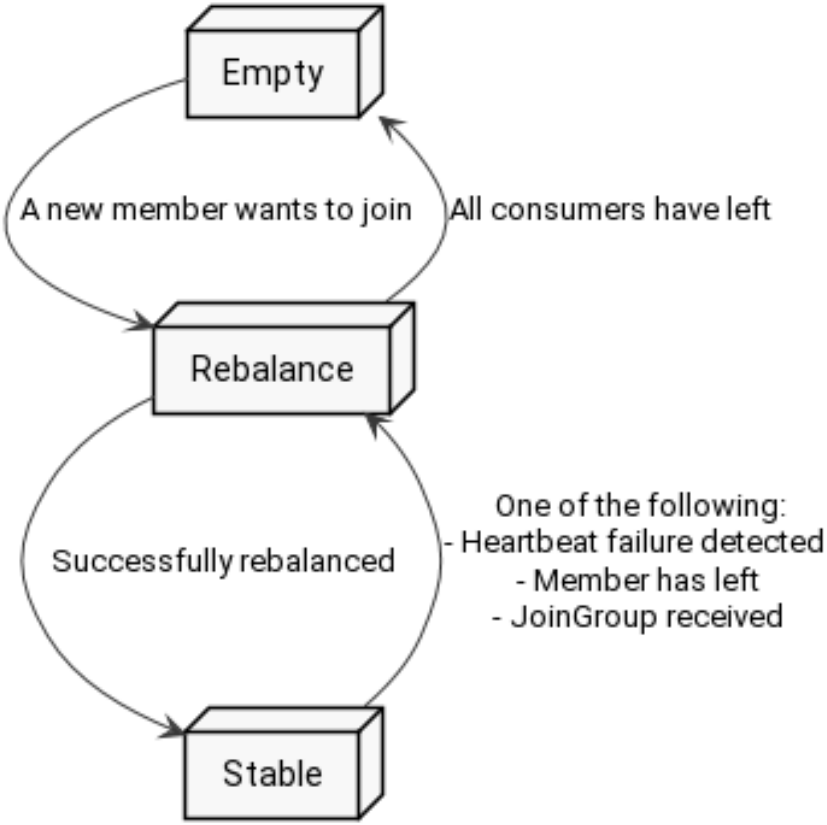
Consumer: Group Leader vs. Followers

- Group Leader is one of the Consumers (in practice the first Consumer to join the group), which functions as a normal Consumer and normally consumes data from Kafka, but in addition, when rebalancing, it decides which Consumer will consume which partition.
- Partition assignment can be changed using property partition.assignment.strategy:
  - https://medium.com/streamthoughts/understanding-kafka-partition-assignment-strategies-and-how-to-write-your-own-custom-assignor-ebeda1fc06f3
- Kafka has 3 build-in strategies:
  - Range (default)
  - RoundRobin
  - StickyAssignor
- If you change the default strategy, then it is important that all Consumers in the Consumer Group must have the same strategy!

How Consumer Works: Group Rebalancing I.

- When new Consumer joins a Consumer Group, an existing Consumer leaves the group, or a partition is added to one of the subscribed topics, „Group Rebalancing“ is started.
- Consumer Group can be in one of these states:
  - Empty: Consumer Group exists, but is empty
  - Stable: Rebalancing has taken place and Consumers and consuming messages
  - Preparing Rebalance & Completing Rebalance
  - Dead

Simplified graph of states and transitions between them:

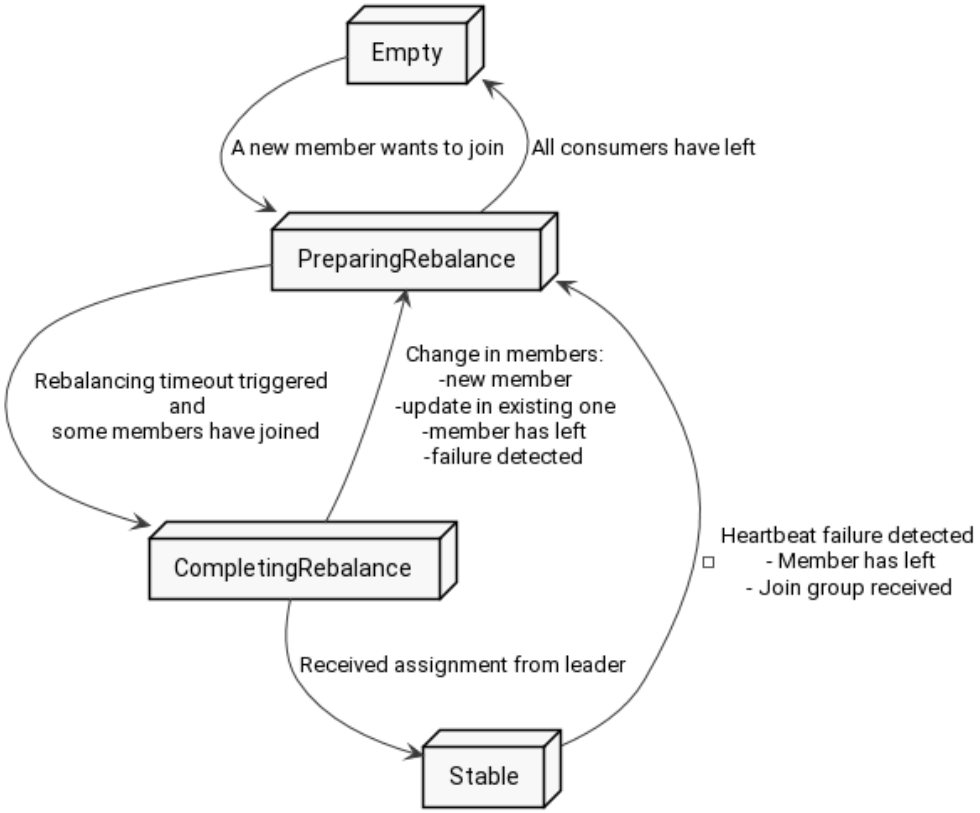


- https://chrzaszcz.dev/2019/06/kafka-rebalancing/

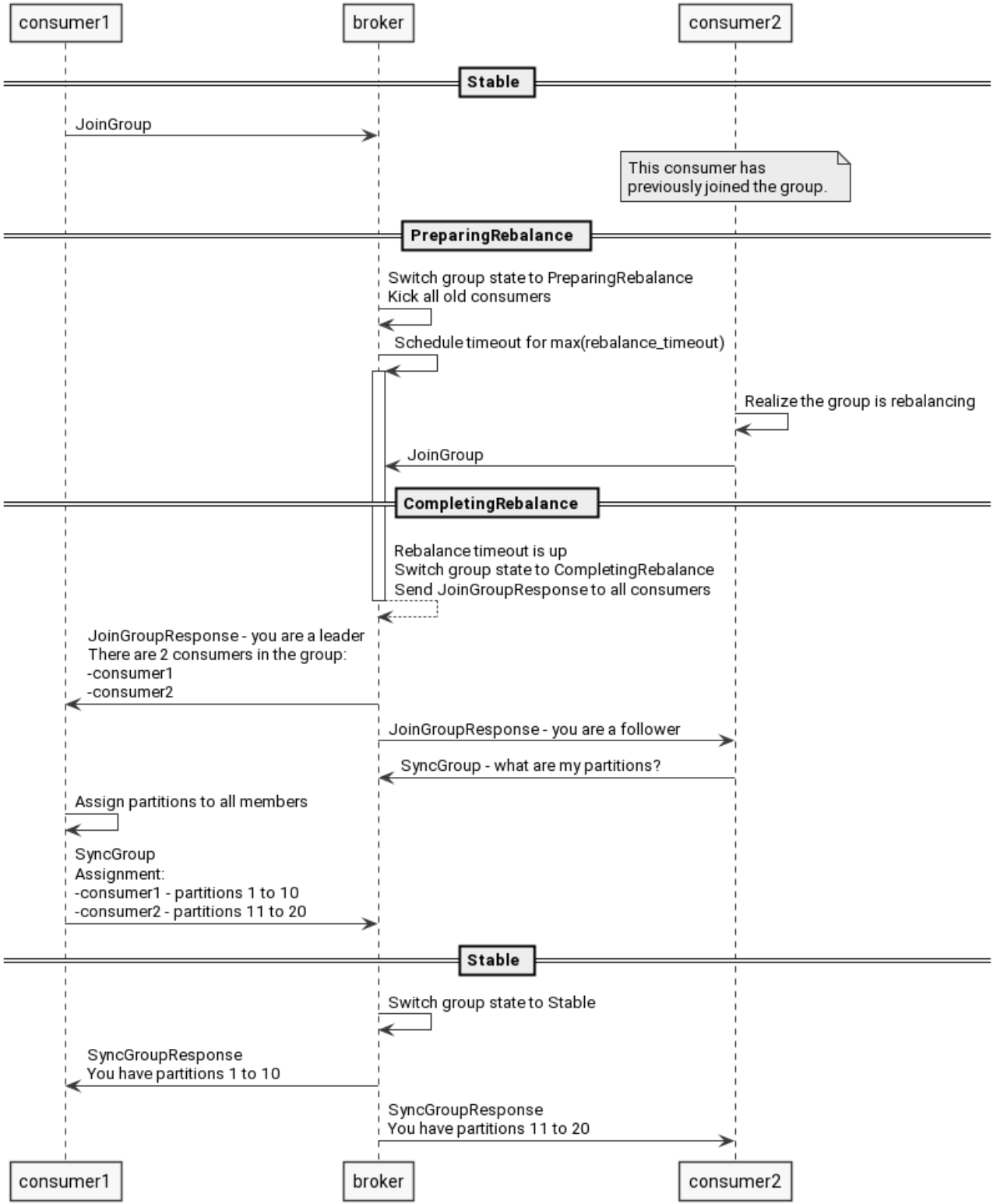
https://www.confluent.io/blog/kafka-consumer-multi-threaded-messaging/#group-rebalancing

How Consumer Works: Group Rebalancing II.

- Rebalancing (from high-level perspective):
  - New Consumer sends a request to one of the bootstrap servers that it wants to join the Consumer Group.
  - Kafka changes state of group to PreparingRebalancing, disconnects all current members and waits until they reconnect.
  - Kafka:
    - Selects one Consumer as Group Leader
    - Sends all Consumers info that they successfully joined group
    - To Leader sends list of followers
  - Followers send request to partitions, which will be processing
  - Leader decides which partitions will be sent to whom and sends this information to Kafka
  - Kafka receives list of partitions from leader and sends to leader and follower which partitions they should process

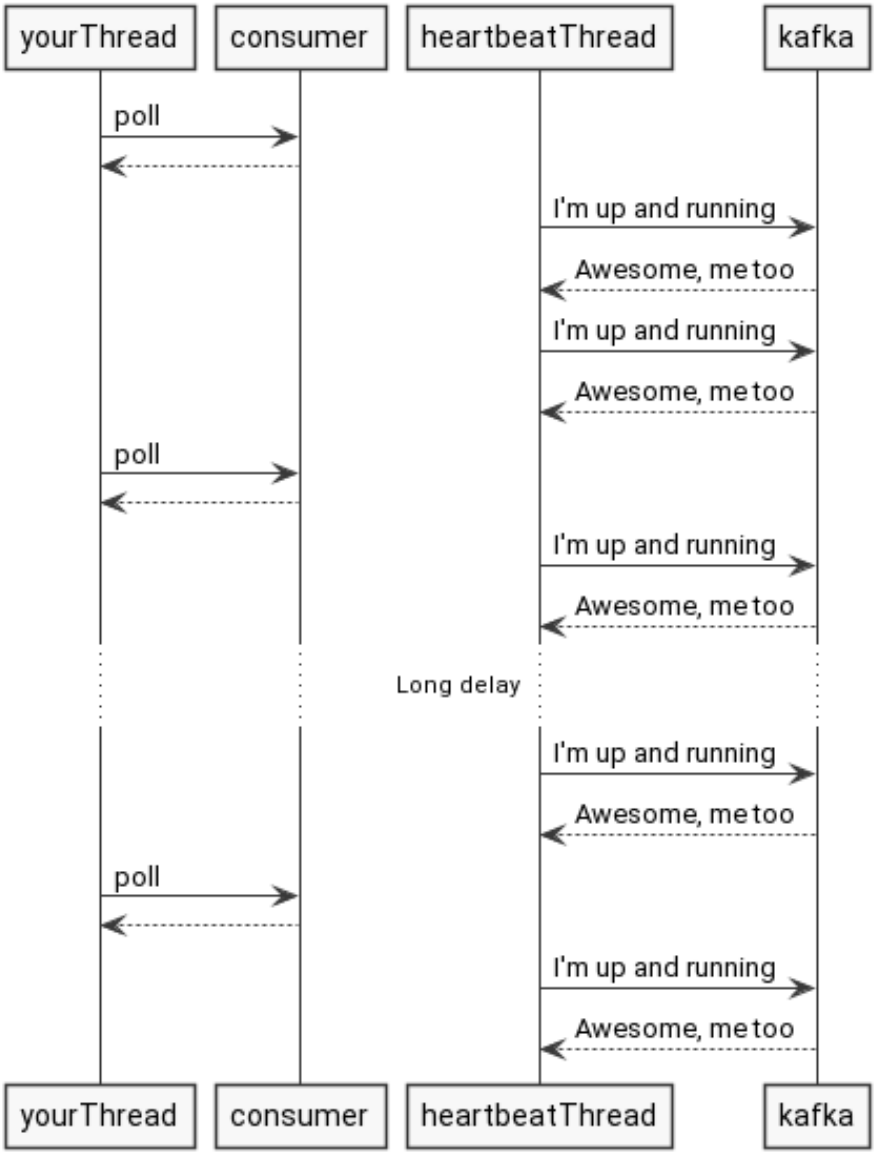


https://chrzaszcz.dev/2019/06/kafka-rebalancing/


consumer1
broker
consumer2
Stable
JoinGroup
This consumer has previously joined the group.
PreparingRebalance
Switch group state to PreparingRebalance
Kick all old consumers
Schedule timeout for max(rebalance_timeout)
Realize the group is rebalancing
JoinGroup
CompletingRebalance
Rebalance timeout is up
Switch group state to CompletingRebalance
Send JoinGroupResponse to all consumers
JoinGroupResponse - you are a leader
There are 2 consumers in the group:
-consumer1
-consumer2
JoinGroupResponse - you are a follower
SyncGroup - what are my partitions?
Assign partitions to all members
SyncGroup
Assignment:
-consumer1 - partitions 1 to 10
-consumer2 - partitions 11 to 20
Stable
Switch group state to Stable
SyncGroupResponse
You have partitions 1 to 10
SyncGroupResponse
You have partitions 11 to 20
consumer1
broker
consumer2

How Consumer Works: Heartbeat

- Consumer has thread kafka-coordinator-heartbeat-thread, which determines if the connection to Kafka works.
  - https://chrzaszcz.dev/2019/06/kafka-heartbeat-thread/
- Heartbeat configuration parameters:
  - `heartbeat.interval.ms`: Frekvency of sending heartbeat requests, default: 3s
  - `session.timeout.ms`: Time interval, in which the Broker must obtain at least one request from Consumer, otherwise it considers Consumer dead, default: 10s
- https://stackoverflow.com/questions/43881877/difference-between-heartbeat-interval-ms-and-session-timeout-ms-in-kafka-consume



How Consumer Works: Heartbeat

- Another significant konfiguration: `max.poll.interval.ms` (default: 5 minut). If data processing takes longer than this setting, then the Broker considers Consumer dead, kicks him out of the Consumer Group and performs rebalancing.

Spring + Kafka

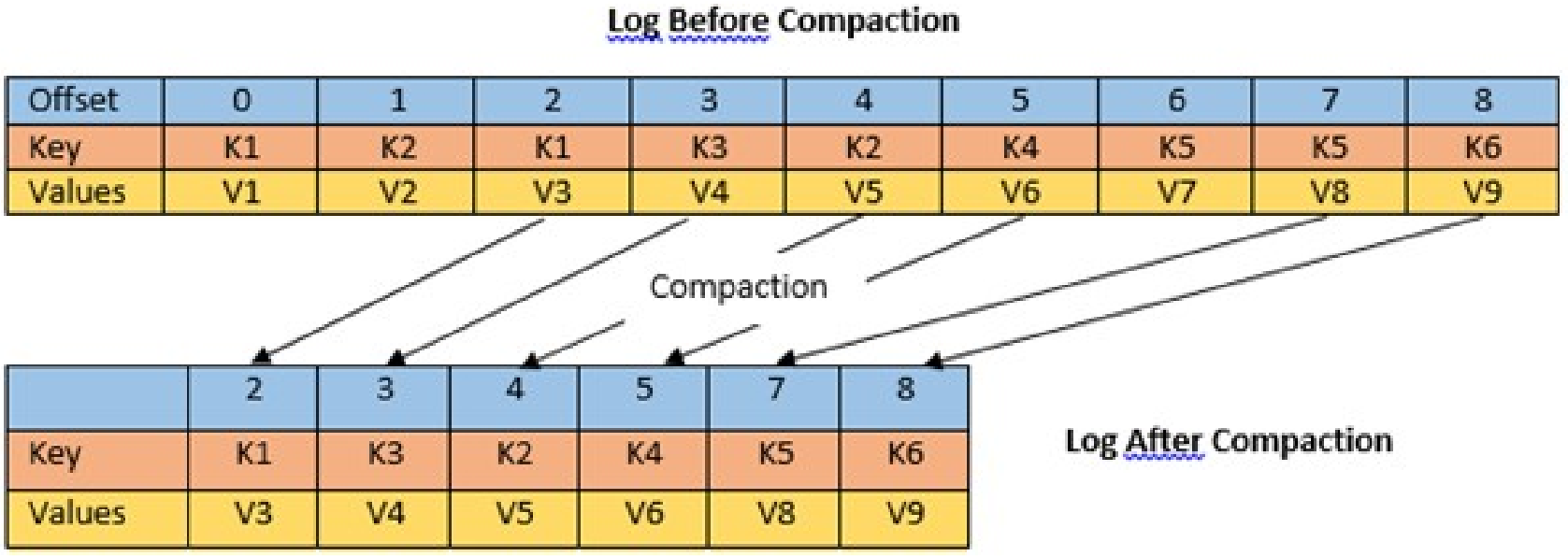
- Documentation:
  - https://spring.io/projects/spring-kafka
- KafkaTemplate IS thread-safe!
- Working project:
  - https://memorynotfound.com/spring-kafka-json-serializer-deserializer-example/
- Spring + Avro:
  - https://www.codenotfound.com/spring-kafka-avro-bijection-example.html

Spring Cloud Streams

- Even greater abstraction than Spring Kafka.
- https://piotrminkowski.com/2021/11/11/kafka-streams-with-spring-cloud-stream/
- https://medium.com/geekculture/spring-cloud-streams-with-functional-programming-model-93d49696584c
- https://www.baeldung.com/spring-cloud-stream-kafka-avro-confluent

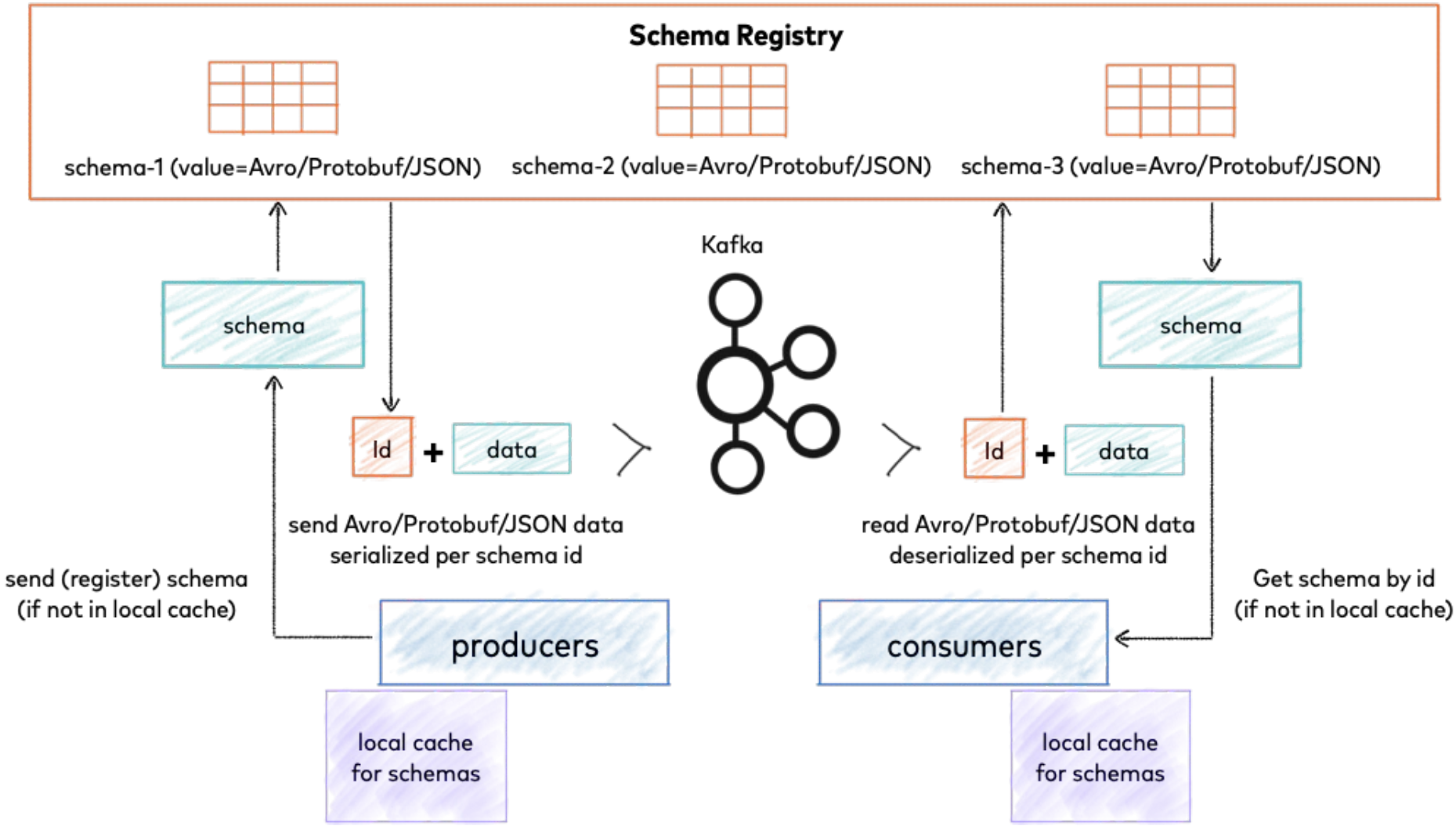
Log Compaction

- Sometimes you may want to use Topic Log Compaction:
  - https://medium.com/swlh/introduction-to-topic-log-compaction-in-apache-kafka-3e4d4afd2262
- It is a mechanism, that selectively deletes old records for which there are newer records with the same key. Beware! If you don't use key (it is NULL), then you cannot use Log Compaction!



Schema Registry + Avro

Schema Registry



Avro + Schema Registry II.

- V případě, že chceme použít jako formát dat Avro, pak musíme specifikovat schéma. Schémata se nachází ve Schema Registry:
  - Landoop: http://localhost:3030/api/schema-registry/
  - Confluent: http://localhost:8081
- Konfigurace:
  - https://docs.confluent.io/platform/current/schema-registry/index.html
    - Schémata jsou uložena v Kafce v topicu `_schemas`
    - Názvy schémat mají následující formát: `<topic>-key, <topic>-value`
- Schema Registry obsahuje REST API, pomocí kterého se s ním dá pracovat:
  - https://github.com/confluentinc/schema-registry
- Nebo přes Maven:
  - https://docs.confluent.io/platform/current/schema-registry/develop/maven-plugin.html
- Seznam všech schémat:
  - http://localhost:8081/schemas

Poznámka: Schéma v registry se nazývá „subject“, tohle vrátí jejich seznam: http://localhost:8081/subjects

Avro + Schema Registry I.

- Message in Kafka can have any format (String, JSON, XML, binary). Kafka is agnostic to the serialization format. For Kafka key and value are just an array of bytes. However, this has the disadvantage, that we must ensure that Producer a Consumer understand each other.
- The solution to this problem is a schema. Default schema in Kafka is Avro. Supported are also JSON Schema and Protobuf. Another advantage of schema is that data can be stored more efficiently in Kafka. Specifically, when using Afro schema, attribute names are not stored internally in the data, it has a binary format and thus is more compact than for example JSON.
  - https://www.baeldung.com/java-apache-avro
  - https://www.confluent.io/blog/avro-kafka-data/
  - http://avro.apache.org/docs/current/
    - There is also Maven plugin, which generates Java classes from .avsc files.
  - https://medium.com/slalom-technology/introduction-to-schema-registry-in-kafka-915ccf06b902

Schema Compatibility

- Schema can have one of the compatibility types:
  - NONE
  - FORWARD
  - BACKWARDS (default)
  - FULL
- For default BACKWARDS compatibility, these changes cause a breaking change:
  - Adding or removing field without a default value
  - Change of name or type of a field
  - Change of enum values if there’s not default enum symbol
- Notes:
  - Compatibility can be set globally, or per schema
  - Change of schema is done using REST API

kafka-avro-console-producer/consumer

```
kafka-avro-console-producer --broker-list localhost:9092 --topic first_topic_avro \
  --property parse.key=true \
  --property key.schema='{"type":"string"}' \
  --property value.schema='{"type":"record","name":"myrecord","fields":[{"name" : "name",
"type" : "string"}, {"name" : "age", "type" : "int"}]}'
```

```
Messages:

"jirka"<TAB>{ "name" : "Jirka Pinkas", "age" : 40 }
```

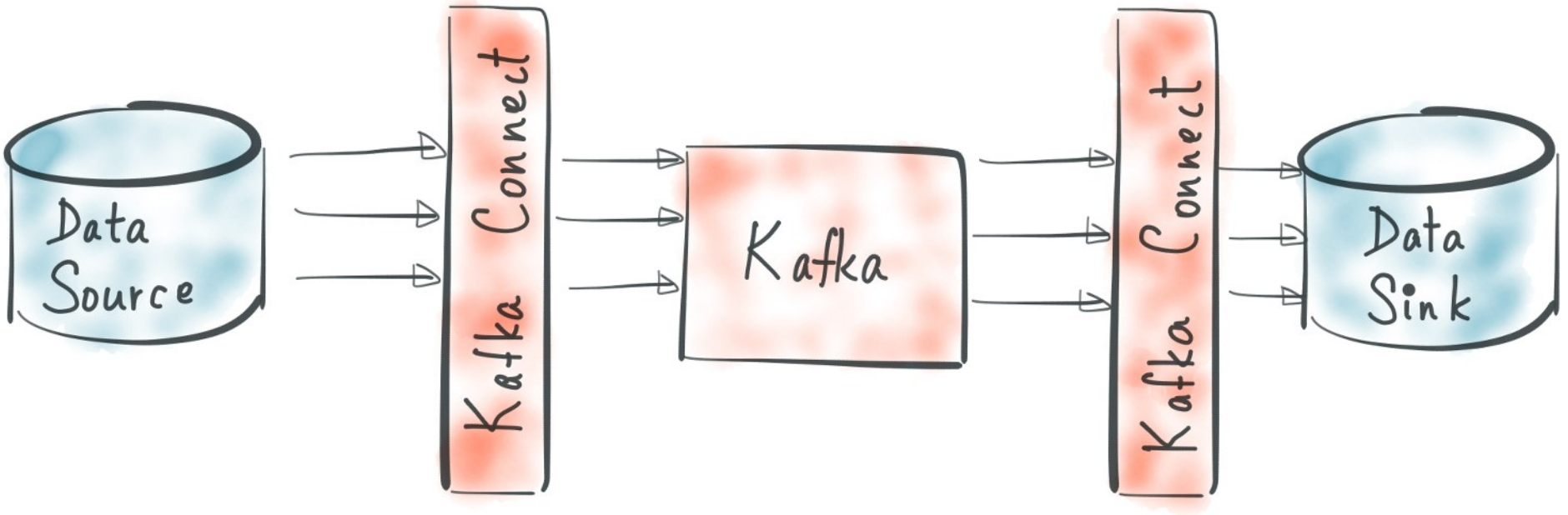
Press TAB, because key is separated from value with tabulator

```
kafka-avro-console-consumer --topic first_topic_avro --bootstrap-server localhost:9092
```

Note: kafka-avro-console-producer will create two schemas:
- first_topic_avro**-key**
- first_topic_avro**-value**

http://localhost:3030/schema-registry-ui/

Kafka Connect


Data Source
Kafka Connect
Kafka
Kafka Connect
Data Sink

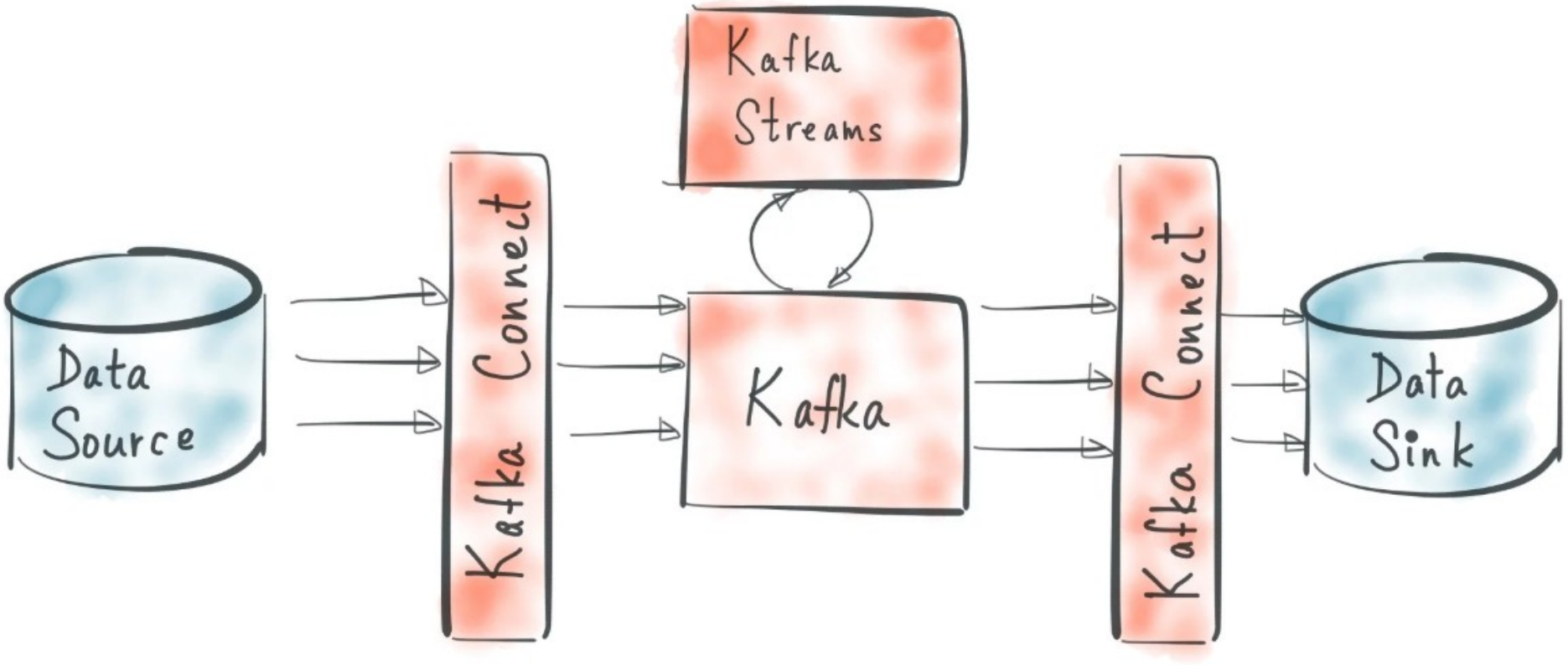
Basic concepts

- Source => Kafka = Kafka Connect Source
- Kafka => Sink = Kafka Connect Sink
- Kafka Connect contains Connectors
- Connectors + Configuration => Tasks
- Tasks are performed by processes called „Workers“ (servers):
  - Worker = Java process
  - Worker can be:
    - Standalone – one (only for testing, state of the worker is stored locally)
    - Distributed – whole cluster (ideally everywhere, not only for production, but also for testing, because configuration of Standalone worker is different than configuration of Distributed worker. State of the worker is stored in Kafka topic)

Debezium

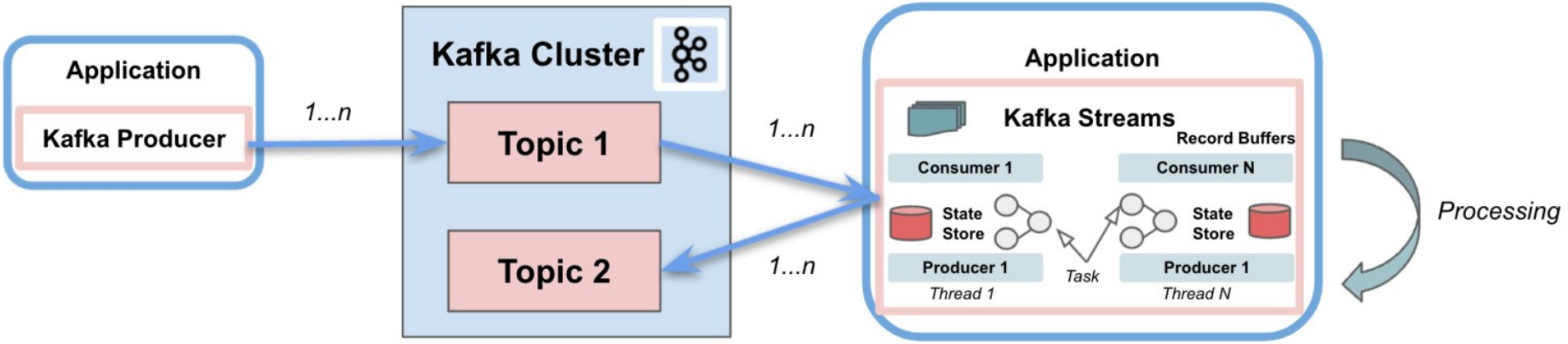
- Debezium is a project build on Kafka Connect and implements CDC (Change Data Capture)

Kafka Streams


Kafka
Streams
Data
Source
Kafka Connect
Kafka
Kafka Connect
Data
Sink

Kafka Streams

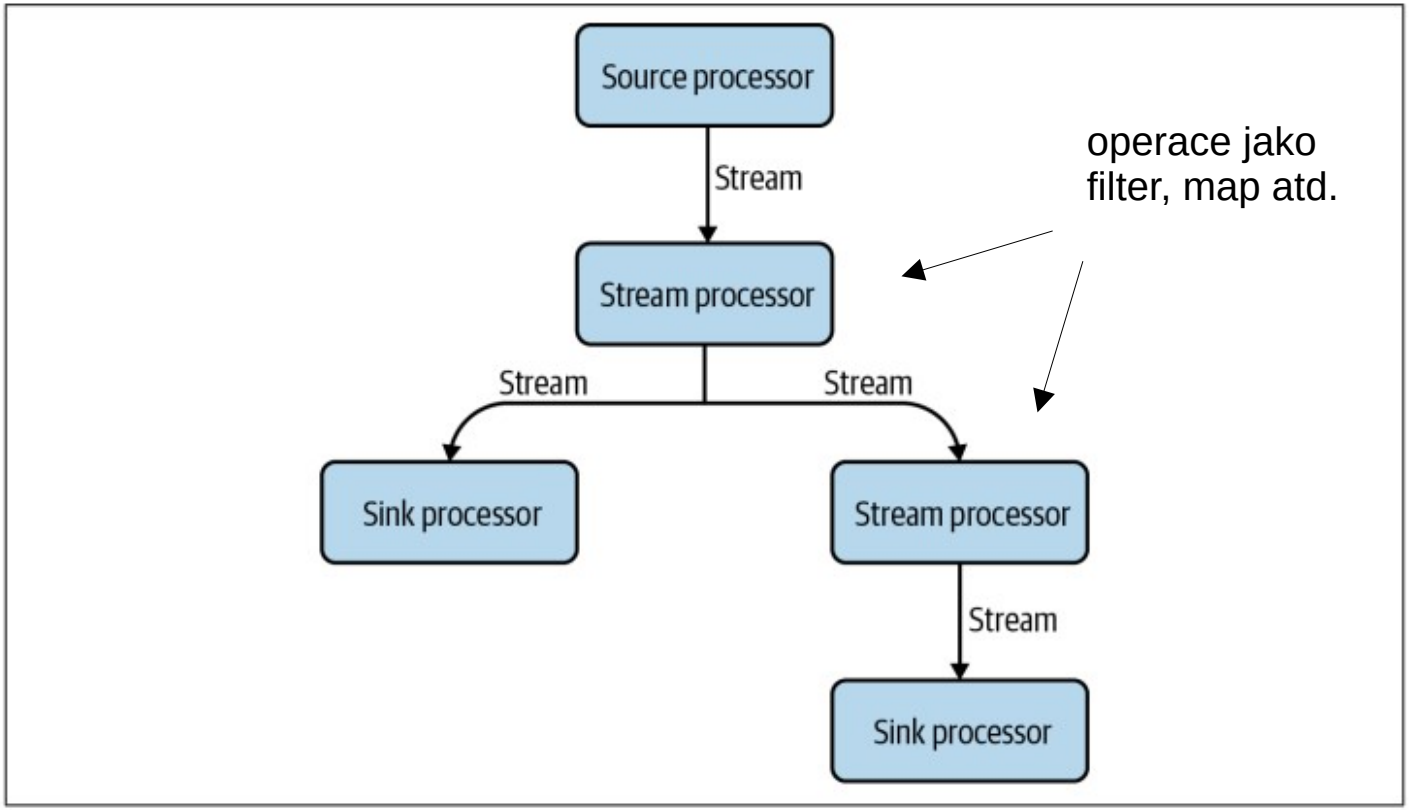
- Kafka Streams je knihovna, která slouží pro získání dat z Kafka Topicu, jejich následnou transformaci a uložení do jiného Kafka Topicu.
- Kafka Streams jsou postavené na Kafka klientských knihovnách (Kafka Consumer & Producer) a v dnešní době to je typicky Spring Boot konzolová aplikace, která obvykle běží v našem Kubernetes clusteru:



- https://www.baeldung.com/java-kafka-streams-vs-kafka-consumer
- Kafka Consumer API vs. Streams API:
  - https://stackoverflow.com/questions/44014975/kafka-consumer-api-vs-streams-api

Processor Topology

- Kafka Stream aplikace je strukturovaná jako directed acyclic graph (DAG). Uzel je „processing step“ a koncové uzly jsou buď vstup z topicu (source), nebo výstupy (sink):



pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>2.0.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Hello World

Tohle je jako group.id
u consumera

Záznamy se prochází
od posledního offsetu
(nebo 0)

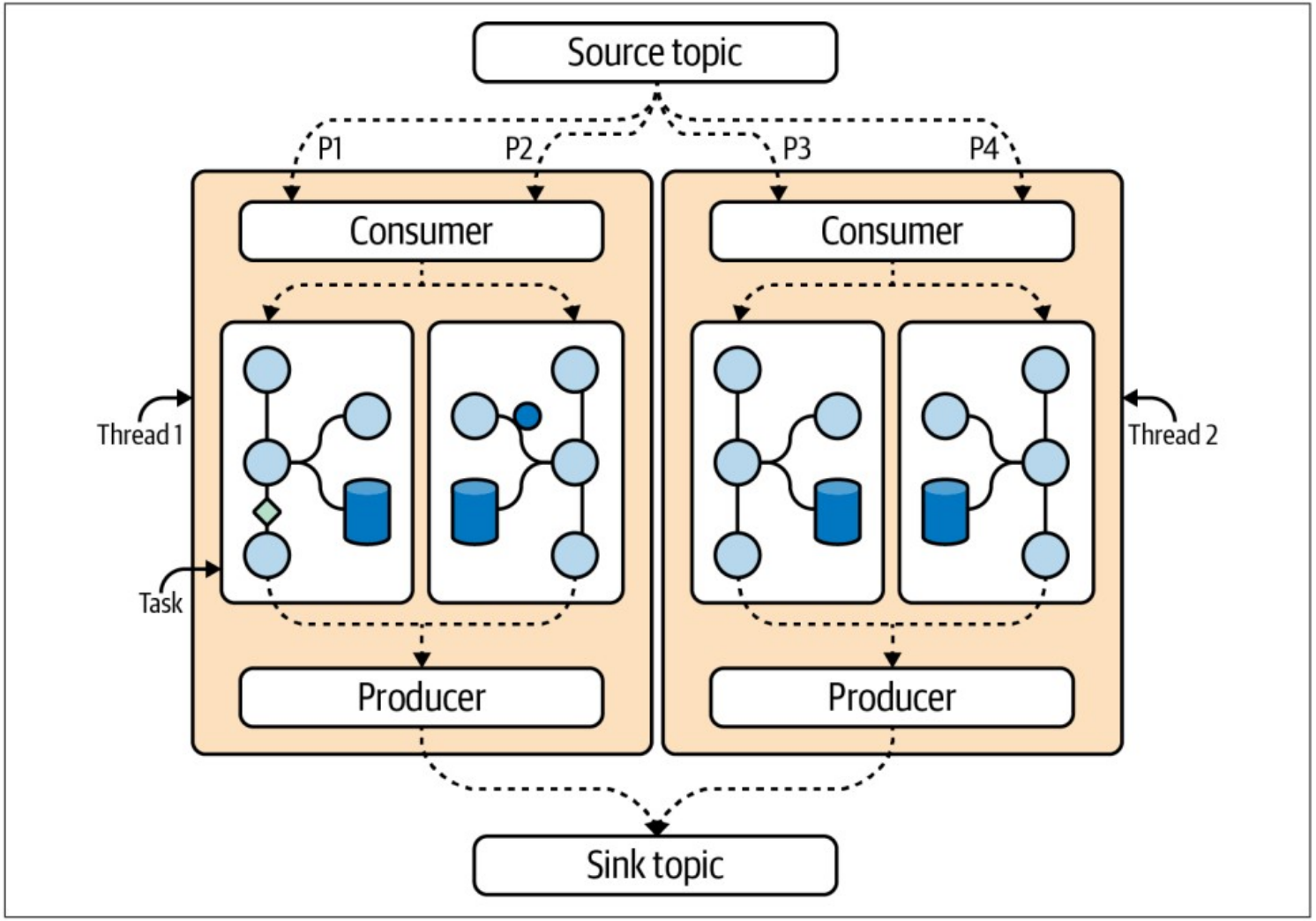
```java
public class Main {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(StreamsConfig.APPLICATION_ID_CONFIG, "kafka-streams-app");
        properties.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        properties.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        properties.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, String> stream = builder.stream("first_topic");
        stream
                .mapValues(s -> s.toLowerCase())
                .to("first_topic_lowercase");

        KafkaStreams streams = new KafkaStreams(builder.build(), properties);
        streams.start();
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }
}
```

Graceful shutdown

Poznámka: Pokud chcete něco jako --from-beginning, tak musíte buď:
- změnit application id
- anebo před streams.start() zavolat streams.cleanUp()

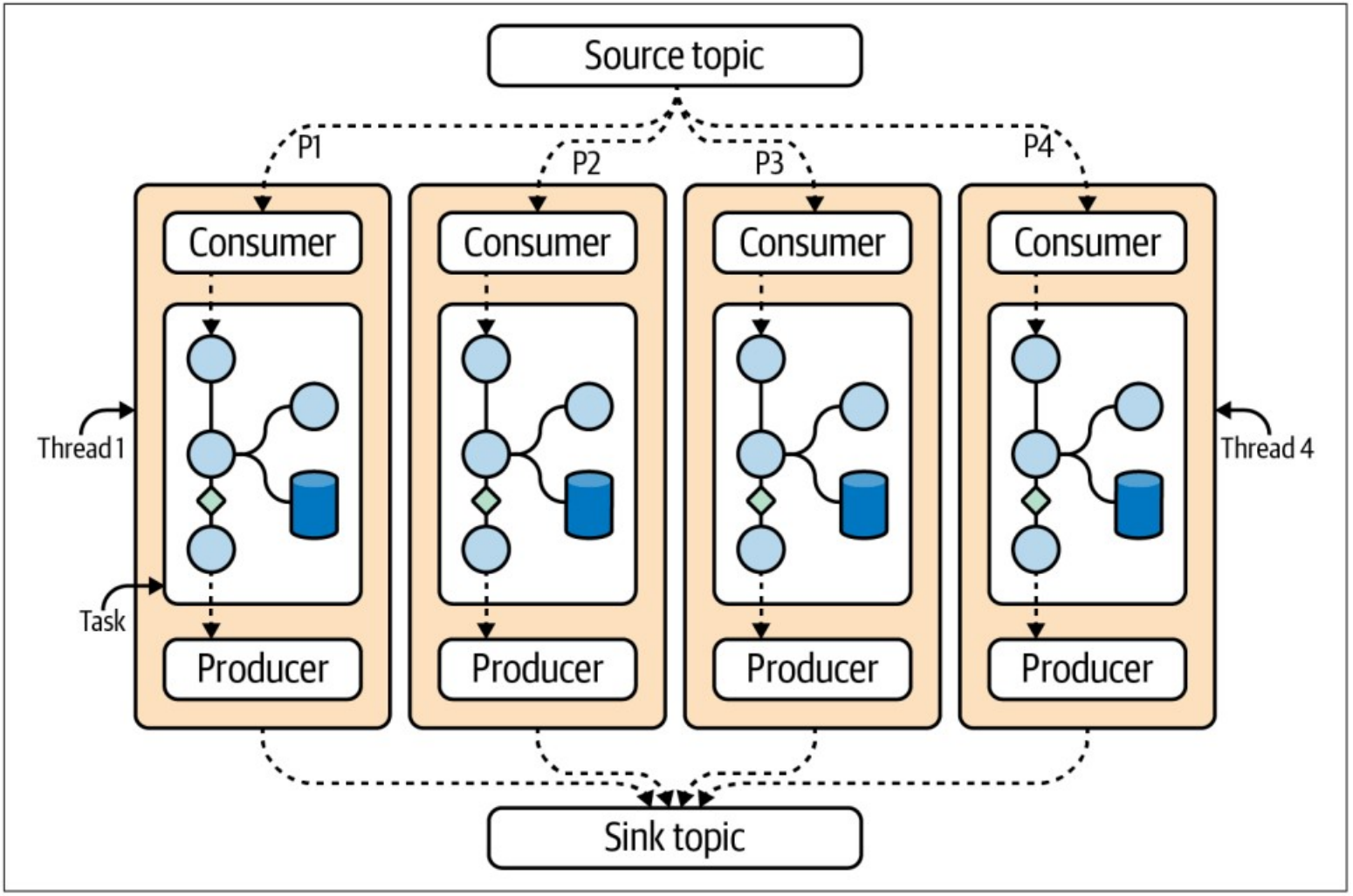
Kafka Streams Internals

- 4 Kafka Streams tasky běžící ve dvou threadech:



Kafka Streams Internals

- 4 Kafka Streams tasky běžící ve čtyřech threadech:



Tasks

- Task = nejmenší „unit of work“ Kafka Streams aplikace. Počet tasků je odvozen od počtu input partitions. Pokud máme Kafka Streams aplikaci, která používá pouze jeden input topic, pak je počet tasků roven počtu partitions toho topicu. Pokud používá více input topiců, pak je počet tasků roven maximálnímu počtu partitions, které má nějaký z topiců.
- Tasky se přiřazují StreamThreadu, který je vykonává. Ve výchozím nastavení je v aplikaci jeden StreamThread. Když bude 5 tasků, pak se nejprve vykoná logika jednoho, až skončí tak druhého atd. Když bude 5 threadů (nebo 5 instancí aplikace se stejným application.id), pak se budou tasky vykonávat paralelně.
- https://developer.confluent.io/learn-kafka/kafka-streams/internals/

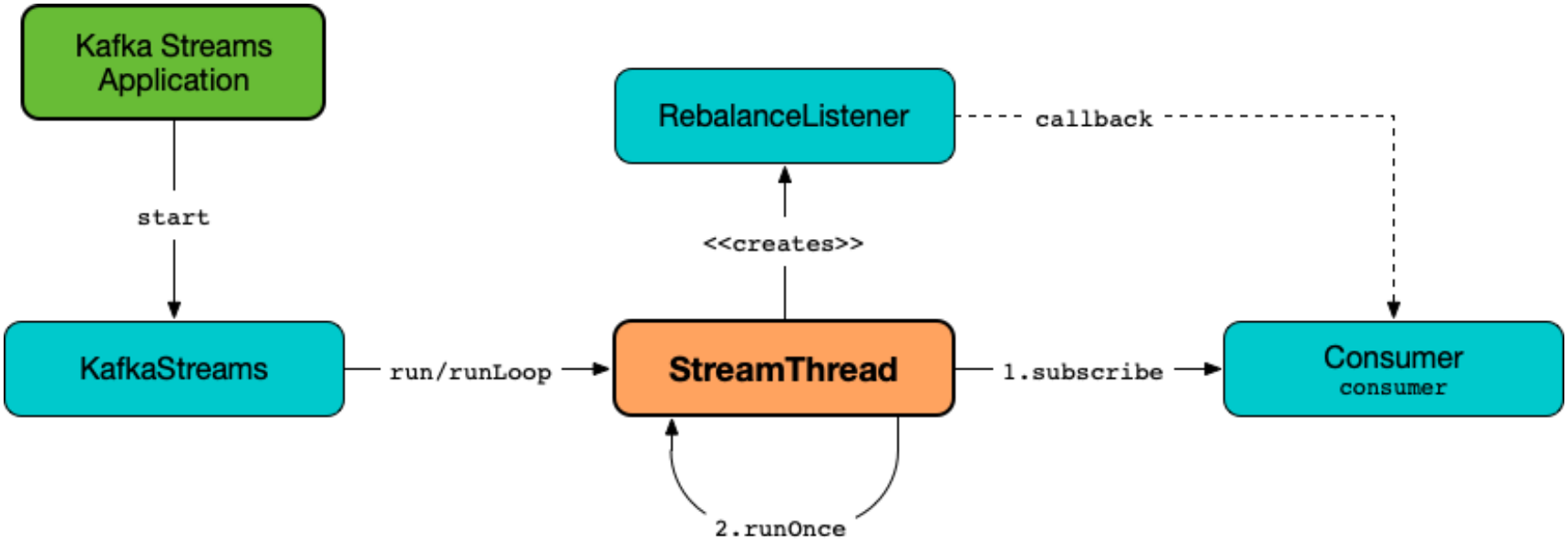
Stream & Threading

- Metoda KafkaStreams#start() spustí Kafka Stream, který se ve výchozím nastavení skládá z následujících vláken:
  - Počet těchto threadů se dá ovlivnit property num.stream.threads (default 1 vlákno):
    - ClientId-StreamThread
      - Hlavní thread, který provede subscribe na Consumera. Pokud chceme, aby právě X vláken získávalo data z input topicu, pak musí mít topic právě X partitions!
    - kafka-coordinator-heartbeat-thread
      - Consumer heartbeat thread
    - kafka-producer-network-thread
      - Kafka Producer thread
  - ClientId-CleanupThread
    - Používá se pro čištění StateDirectory
  - kafka-admin-client-thread
    - Tento thread používá Kafka AdminClient a vytváří StreamThread

StreamThread

- StreamThread
  - https://medium.com/@andy.bryant/kafka-streams-work-allocation-4f31c24753cc
- Jak zjistit počet běžících Consumerů:

```
kafka-consumer-groups --bootstrap-server kafka:9092 --describe --group STREAM_APPLICATION_ID
```



Stream Threads

- Mezi vlákny není žádný sdílený stav. Z pohledu Kafka Streams je jedno, jestli paralelismu u streamů docílíme zvýšením num.stream.threads, nebo spuštěním většího množství instancí Kafka Streams aplikace se stejným application.id.
- Osobně bych ale preferoval škálování pomocí většího množství instancí z následujících důvodů:
  - Větší throughput
  - Lepší vytěžování zdrojů
  - Větší odolnost vůči výpadkům jednotlivých instancí

Depth first processing

- Když Stream začne ze vstupního topicu procesovat nějaký záznam, tak ten záznam musí projít celou topologii, aby se mohl zpracovávat další záznam. Výhoda je, že chování streamu je jednoduché na pochopení. Nevýhoda je, že když je nějaká processing operace pomalá, tak bude blokovat zprocesování dalších záznamů.
  - **Pozor!!!** U sub-topologie toto taky platí, ale pro každou sub-topologii zvlášť a každá se musí vykonávat v dedikovaném vlákně. Když jsou 2 sub-topologie, tak musí být minimálně buď 1 instance streams aplikace s num.stream.threads=2, nebo 2 instance s num.stream.threads=1
    - Sub-topology je, když v rámci topologie se uloží data do sink processoru a pak se načtou dalším source processorem.
    - Jak získat Topology Description:

```
Topology topology = streamsBuilder.build();
log.info("Topology description {}", topology.describe());
```

    - Topology vizualizer:
      - https://zz85.github.io/kafka-streams-viz/
  - https://medium.com/@andy.bryant/kafka-streams-work-allocation-4f31c24753cc

Stateless operations

- Stateless Kafka Streams aplikace jsou nejjednodušší streamy. Nevyžadují žádnou znalost předcházejících eventů, neukládají si žádný stav, jenom vezmou vstupní data, udělají jejich processing a uloží je do výstupních topiců.

mapValues(), map()

- mapValues()
  - Provede transformaci každé hodnoty ve streamu.
- map()
  - Provádí transformaci klíčů i hodnot
- Pokud není nutné provést transformaci klíče, pak bychom měli vždy preferovat operaci mapValues(), protože nezpůsobí re-partitioning.
- Podobné operace jsou transformValues() a transform(), které ale používají low-level Processor API:
  - https://docs.confluent.io/platform/current/streams/developer-guide/processor-api.html

filter(), filterNot()

- filter()
  - Filtrování záznamů ze streamu
  - Neprovádí změny v klíčích / hodnotách
- filterNot()
  - Negace filter()

foreach(), peek(), print()

- foreach() je klasický foreach cyklus :-), jedná se o terminální operaci.
- peek() dělá prakticky to samé jako foreach, ale jedná se o intermediate operaci. Peek MUSÍ být idempotentní (nesmí mít side-efekty)!
- print() je pouze pro debugování a pro vypsání každého záznamu ve streamu, jedná se o terminální operaci.

branch(), merge()

- branch() rozdělí stream na základě 1 .. N predikátů do pole streamů.
- Od Kafka 2.8.0 se používá ve spojení s operací split()
- Podobného výsledku bychom docílili pomocí aplikování většího množství filter() operací.
- Opačnou operací je operace merge(), která kombinuje eventy dvou streamů dohromady.

selectKey()

- selectKey() provede transformaci klíče na novou hodnotu.

flatMapValues(), flatMap()

- Vstupem je jeden záznam, výstupem je 0, 1 nebo N záznamů.
- flatMapValues()
  - Neprovádí změny v klíčích
  - Pouze pro KStream
- flatMap()
  - Umožňuje změny v klíčích
  - Pouze pro KStream

to(), through(), repartition()

- KStream nebo KTable výstup je možné uložit do jakéhokoli topicu metodami:
  - to(): terminální operace – zapíše data do topicu a ukončí stream
  - through(): zapíše data do topicu a vrátí ho ve formátu nového KStream nebo Ktable. Od Kafka 2.6 deprecated.
    - repartition(): novější náhrada through()
    - Jak through(), tak repartition() je zkrácená varianta KStream#to() následované StreamsBuilder#stream()
      - Proč používat through() / repartition()? Vzhledem k tomu, že se data zapíšou do topicu, tak dojde k rozdělení jedné topologie do dvou sub-topologií a tím se zvýší paralelismus Kafka Streams aplikace.
      - U through() se musel předem ručně vytvořit topic, do kterého se budou ukládat data, u repartition() se topic vytvoří automaticky, má název KSTREAM-REPARTITION-XXXXX-repartition a dá se lehce v Kafka Streams aplikaci nastavit počet partitions tohoto topicu.

Stateful operations

- Use-cases:
  - Joining data
  - Aggregating data
  - Windowing Data
- Stav je uchováván ve „state store“. Kafka Streams aplikace může obsahovat větší množství state stores. State store je na úrovni Kafka Streams aplikace. Výchozí implementace je RocksDB.
- Pro zajištění failover je stav také v Kafce v topicu s názvem: <application.id>-<storeName>-changelog

RocksDB

- RocksDB je key-value store na straně Kafka Streams aplikace.
- https://www.confluent.io/blog/how-to-tune-rocksdb-kafka-streams-state-stores-performance/

KStream vs. KTable vs. GlobalKTable

- KStream je jako log, jede záznam po záznamu.
  - Příklad: máme 2 záznamy ve streamu:
    - (KEY, VALUE): (alice, 1), (alice, 3).
    - Když se provede SUM operace, bude výsledek: (alice, 4)
  - KStream je stateless
- KTable bere v úvahu poslední hodnotu záznamu s určitým klíčem
  - U předcházejícího příkladu by SUM operace vrátila (alice, 3)
  - KTable je stateful, interně je KTable implementována pomocí RocksDB a topicem v Kafkou. RocksDB obsahuje aktuální data tabulky a je uložena na disku Kafka Streams aplikace. RocksDB není fault-tolerant, proto jsou data také uložena v Kafka topicu.
    - https://stackoverflow.com/questions/52488070/difference-between-ktable-and-local-store
  - Pozor! KTable oproti KStream nezískává data real-time z topicu, ale načte poslední stav (key-value) do cache a ve výchozím nastavení po 30 vteřinách se provede flush cache a načte se aktuální stav (property commit.interval.ms). Nebo když je plná cache (ve výchozím nastavení 10 mb: property cache.max.bytes.buffering).
- GlobalKTable je jako KTable, ale data čte ze všech partition.

join, left join, outer join

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.

Type	Windowed	Operators	Co-partitioning required
`KStream-KStream`	Yes[a]	• `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

- Při join operacích musí vstupní data (topicy) splňovat co-partitioning podmínku.
- Co je co-partitioning?
  - Input topic vlevo i vpravo musí mít stejný počet partitions.
  - Input topicy musí používat stejnou partitioning strategii (out-of-the-box tomu tak naštěstí je).
- Proč je co-partitioning vyžadován? Protože joiny se provádí na základě klíčů (musí se „spárovat“ klíč vlevo i vpravo). A na základě klíče se rozhoduje, v jaké partition daný záznam bude (z klíče se vypočítává hash).
- GlobalKTable nevyžaduje co-partitioning.

groupByKey, groupBy

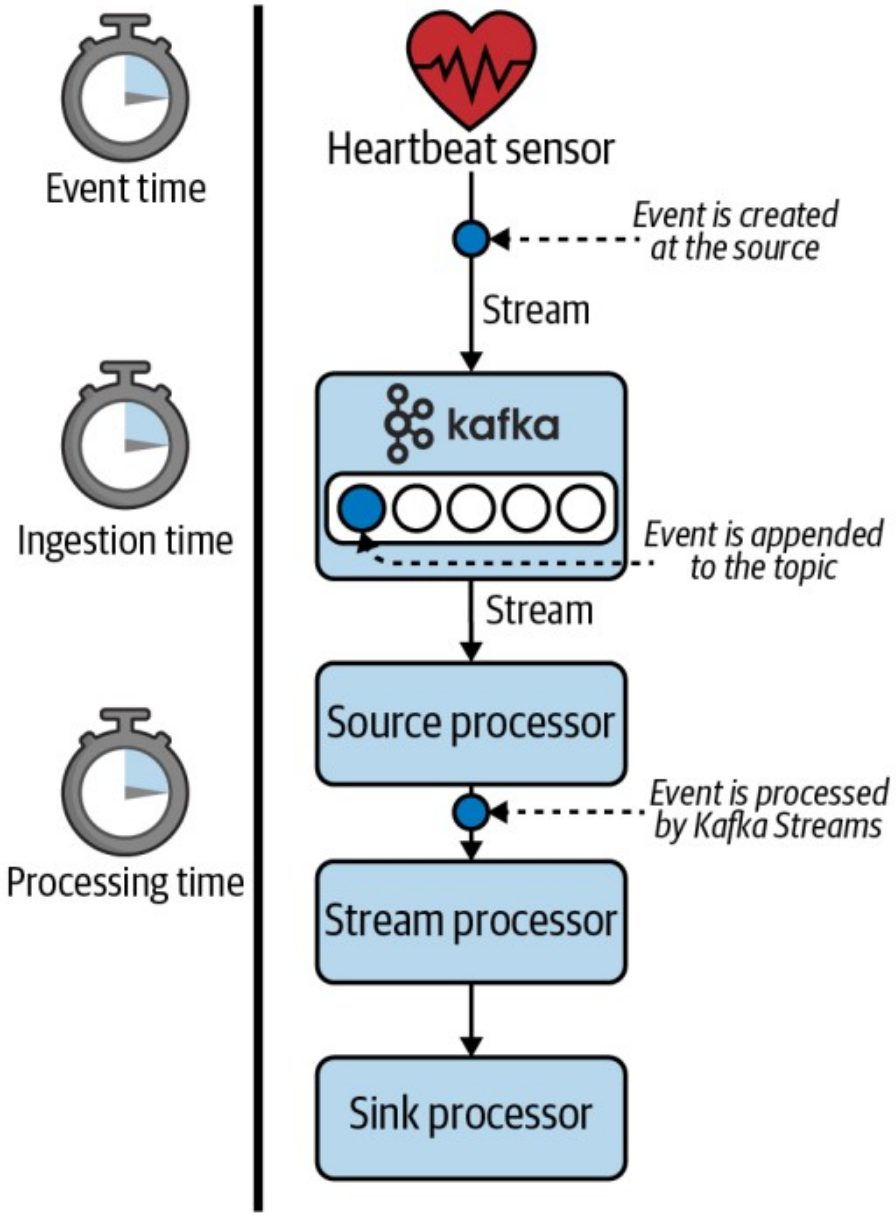
- Předtím, než je možné provádět agregace je nutné nejprve KTable nebo KStream, který se bude agregovat, sgrupovat:
  - groupByKey provede sgrupování bez transformace klíče
    - Efektivnější, neprovádí repartitioning
    - Je možné provést pouze na KStream
  - groupBy provede přemapování klíče a teprve poté sgrupování
    - Provádí repartitioning
    - Je možné provést na KStream i KTable

Aggregations

- Operace:
  - count
    - Klasický count.
  - reduce
    - Reduce algoritmus, oproti aggregate se liší v tom, že výsledný typ musí být stejný jako vstupní typ.
    - Dá se s ní implementovat min, max, sum.
  - aggregate
    - Dá se s ní implementovat to samé co s reduce operací, ale navíc pokročilejším způsobem. Plus například average.
- https://mail-narayank.medium.com/stream-aggregation-in-kafka-e57aff20d8ad

Timestamp

- Ve výchozím nastavení (od Kafka 0.10.0) když Producer pošle záznam do Kafky, tak je mu přiřazen timestamp (ten čas nastavuje Producer, jedná se o „Event time“).
- Když se vytváří ProducerRecord, tak se ten čas dá specifikovat (je datového typu long):
  - `new ProducerRecord<>(TOPIC, null, TIMESTAMP, KEY, VALUE);`
- Dá se také nastavit aby tento čas byl časem zařazení záznamu do topicu (pak by mu byl přiřazen čas „Ingestion time“):
  - https://kafka.apache.org/documentation/#log.message.timestamp.type



TimestampExtractor

- Out-of-the-box se při práci s časem používá výchozí timestamp, který byl probrán na předcházejícím snímku. V případě že je zapotřebí použít jiný (například definovaný v záznamu), pak je nutné implementovat TimestampExtractor, který je pak možné použít:
  - Globálně v Kafka Streams konfiguraci:
    - `streamsConfiguration.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG, MyTimestampExtractor.class);`
  - Nebo ad-hoc:
    - `Consumed.with(Serdes.String(), temperatureValueSerde)`
      `.withTimestampExtractor(new MyTimestampExtractor());`

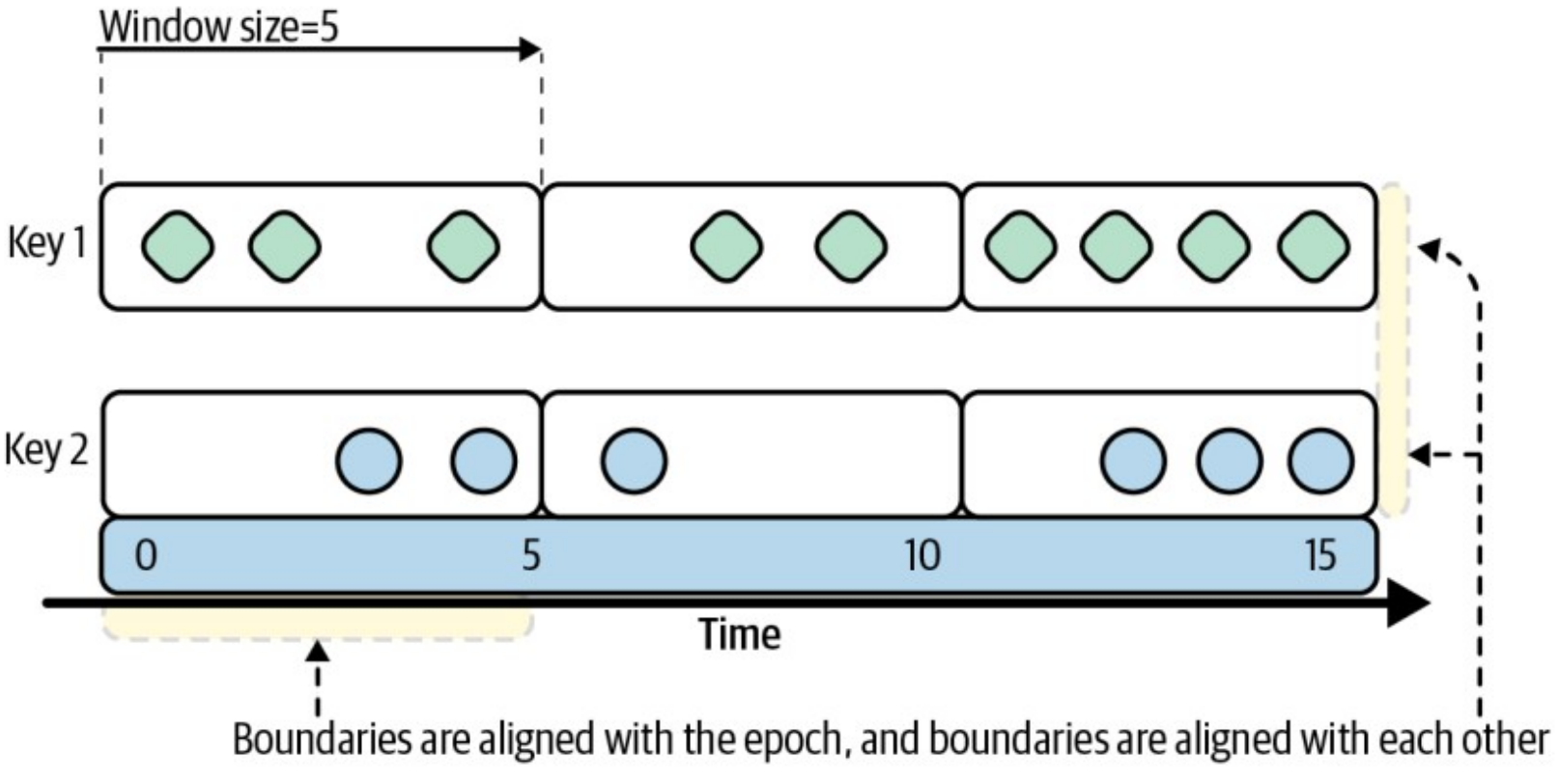
Windowing Streams

- Agregace nebo join operace pracují nad všemi záznamy. Často ale chceme výsledky za nějaké časové období. K tomu se používá koncept Windows. TimeWindow objekt je srovnaný s epoch time (tzn. okno o velikosti 60 000 ms bude mít hranice [0, 60 000), [60 000, 120 000) …
  - Poznámka: [= inclusive,) = exclusive
- Grace period
  - Velice důležitý koncept jak dlouho se bude čekat na záznamy pro nějaké okno, i když už to okno není platné. Pokud záznam přijde po ukončení grace periody, pak bude zahozen a nebude v dané okně již zprocesován.
- Supress
  - Další velice důležitý koncept. Ve výchozím nastavení se jednou za 30s provádí commit KTable. Abychom neměli v KTable mezivýsledky, je možné použít operaci supress(). Pozor! Tato operace nepoužívá state store, ale bufferuje záznamy v operační paměti.

Tumbling Window

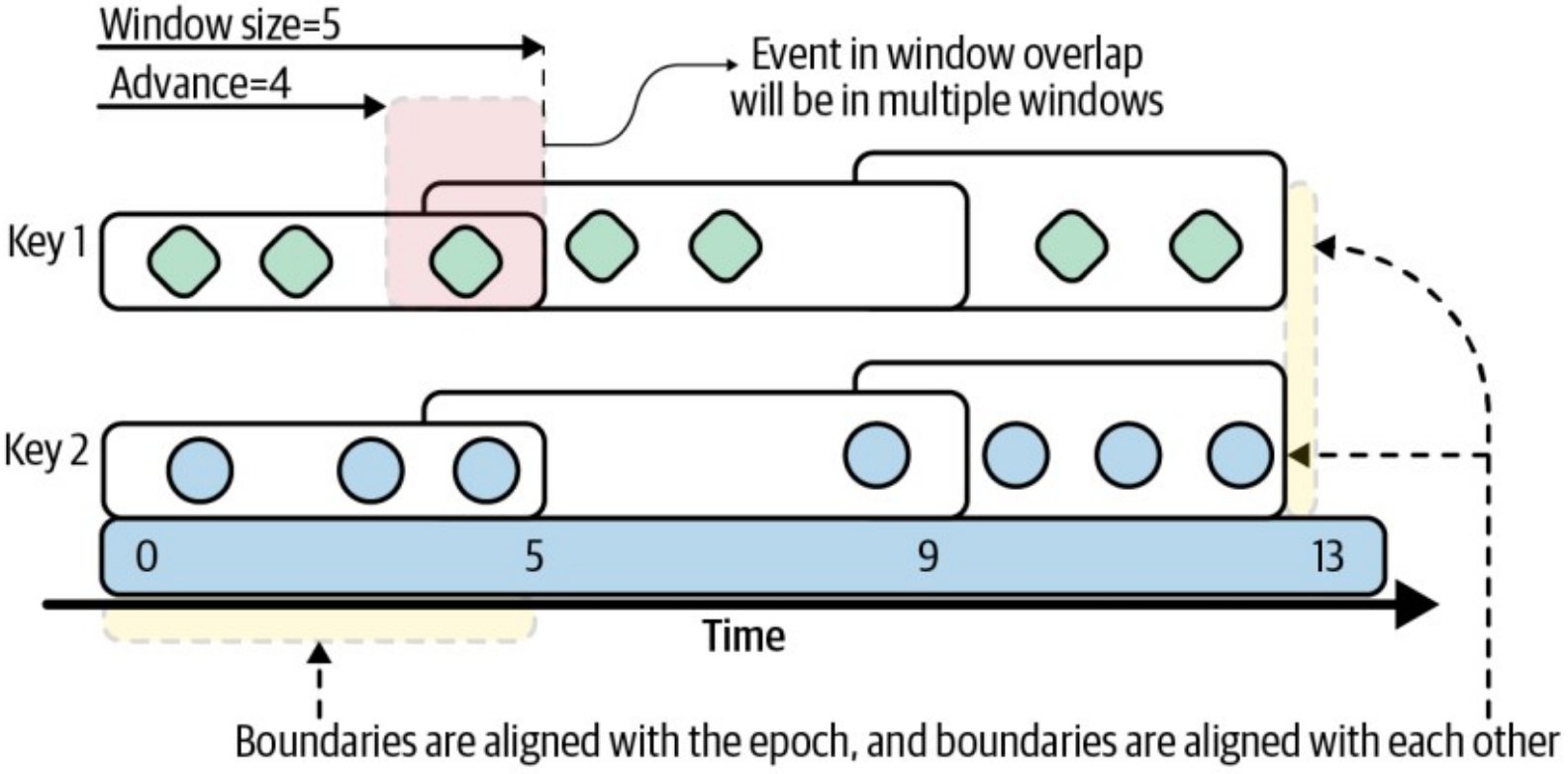
- Okna (Windows), která se nikdy nepřekrývají a jsou definovaná pomocí jedné property: window size.
  - `TimeWindows tumblingWindow =`

    `TimeWindows.ofSizeWithNoGrace(Duration.ofSeconds(5));`



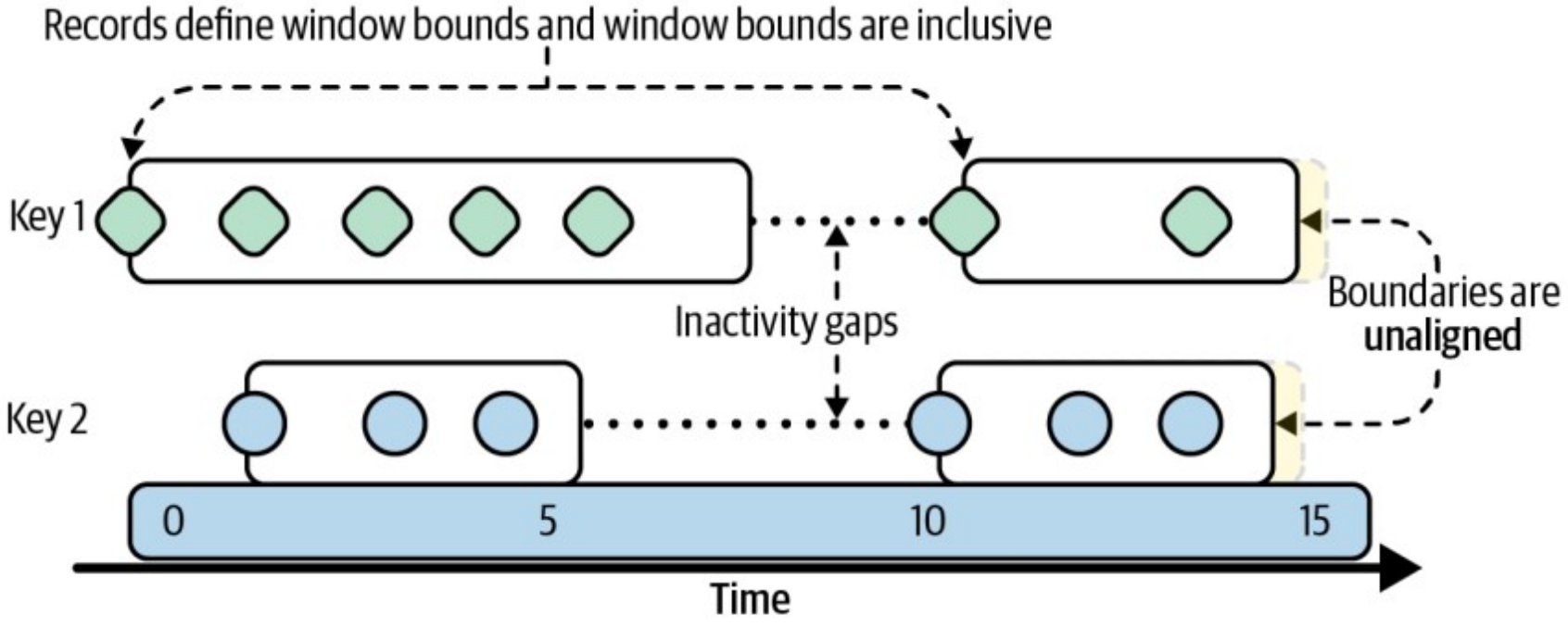
Hopping Window

- Okna, která jsou fixed-size a mohou se překrývat.
  - `TimeWindows hoppingWindow =`
    `TimeWindows.ofSizeWithNoGrace(Duration.ofSeconds(5))`
    `.advanceBy(Duration.ofSeconds(4));`



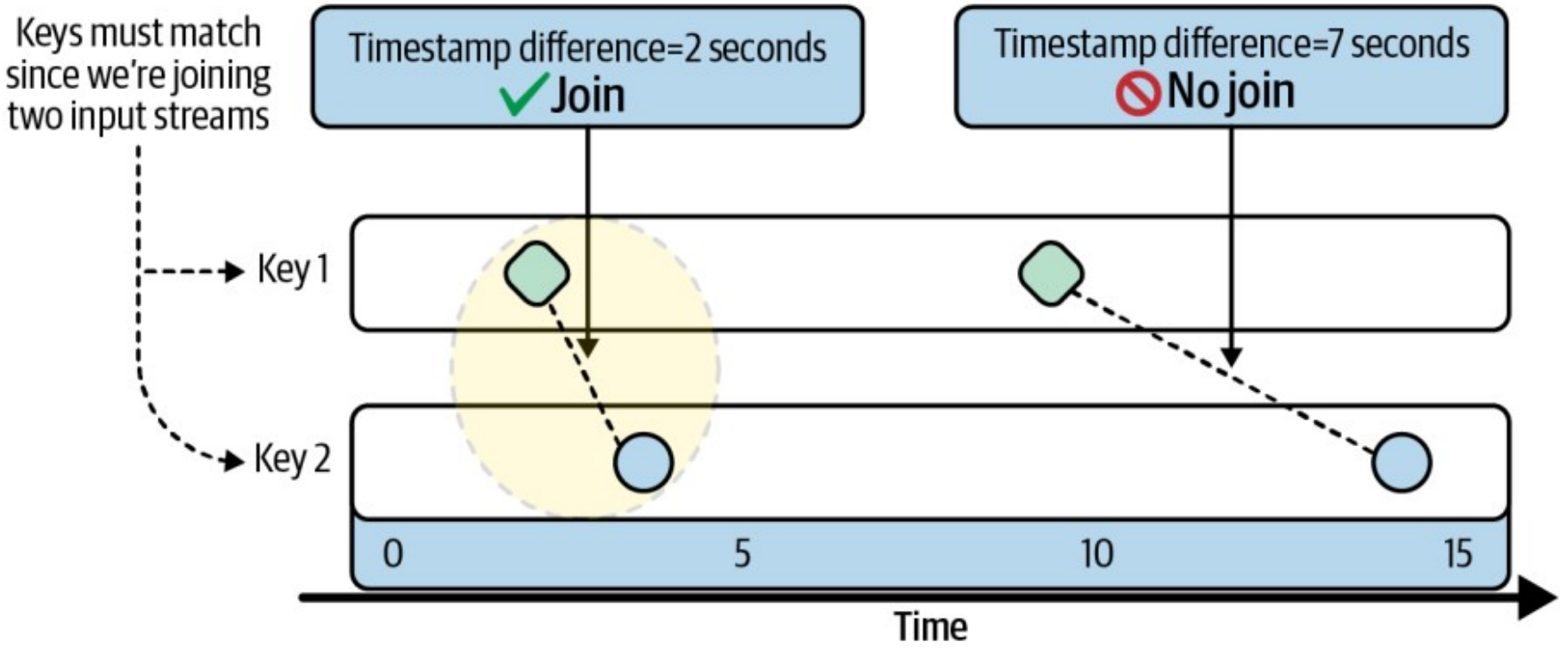
Session Window

- U Session Window se specifikuje inactivity gap. Jakmile dojde k této situaci (po tuto dobu nepřichází žádné zprávy), pak se stávající session window ukončí a při příchodu nové zprávy se započne nové session window.
  - `SessionWindows.ofInactivityGapWithNoGrace(Duration.ofSeconds(5))`



Join Window

- Dva záznamy se spojí, když rozdíl mezi jejich timestamp hodnotami je menší nebo roven window size.
  - `JoinWindows.of(Duration.ofSeconds(5));`
- U agregací je možné použít SlidingWindow, které není navázané na epoch, ale na timestamp záznamu obdobně jako JoinWindow:
  - `SlidingWindows.ofTimeDifferenceWithNoGrace(Duration.ofSeconds(5));`



Guarantees

- Kafka streamy out-of-the box garantují at_least_once Stream processing. Lze to změnit na „exactly once“, což garantuje atomicitu celého streamu:

```
properties.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, "exactly_once_v2");
```

  - https://docs.confluent.io/platform/current/streams/developer-guide/config-streams.html#processing-guarantee
  - Poznámka: Exactly once vyžaduje cluster s minimálně třemi brokery!

Stream Exception Handling

- Kafka Streams v současnosti umožňují následující handlování výjimek:
  - Deserialization Exception Handler umožňuje odchytit výjimky, ke kterým může dojít při deserializaci záznamu.
  - Streams Uncaught Exception Handler odchytává výjimky, ke kterým může dojít při processingu streamu.
  - Production Exception Handler umožňuje odchytit výjimky, ke kterým může dojít při komunikaci s brokerem.

    - https://developer.confluent.io/learn-kafka/kafka-streams/error-handling/
    - https://developer.confluent.io/learn-kafka/kafka-streams/hands-on-error-handling/

Interactive Queries

- Doposud používaný State store se takto dá používat pouze když je jenom jedna instance Kafka Streams aplikace. V případě, že máme větší množství instancí v clusteru, tak musíme buď získat data z lokálního state store, nebo přes REST zavolat jinou Kafka Streams aplikaci, která vrátí data ze svého lokálního state store.
- Kafka Streams nám k tomu zjednodušuje nalezení instance, která obsahuje State Store s daty (discovery). K tomu je nutné nastavit proměnnou application.server. Jinak ale vystavení REST endpointu a komunikaci s ním musíme implementovat sami.
- https://kafka.apache.org/20/documentation/streams/developer-guide/interactive-queries.html
- https://github.com/confluentinc/kafka-streams-examples/tree/7.0.0-post/src/main/java/io/confluent/examples/streams/interactivequeries

State Store & Performance I.

- Z pohledu výkonu je problematický rebalancing, zejména když dojde k výpadku nějaké instance Kafka Streams aplikace. Následně musí dojít ke znovu-vytvoření state store z topiku v Kafce a zejména když je state store velký, tak může dojít k delšímu downtime. K řešení tohoto problému se používá:
  - Standby Replicas
    - Stav state store bude replikován na více instancí Kafka Streams aplikace. To se dá ovlivnit pomocí property num.standby.replicas (výchozí hodnota je 0, při hodnotě 1 bude jedna replika)
      - https://medium.com/transferwise-engineering/achieving-high-availability-with-stateful-kafka-streams-applications-cba429ca7238

State Store & Performance II.

- K rebalancingu dřív nebo později stejně dojde. Jak omezit jeho dopady? Zejména tak, aby ve state store bylo co nejmenší množství dat pomocí:
  - Tombstones: Když už není zapotřebí mít hodnotu ve state store, tak se nastaví na null.
  - Window retention: U windowed stores je možné nastavit retenci dat
  - Agressive topic compaction: KTable má zapnuté compaction, ale data na disku mohou zabírají více místa. Proč? Data jsou fyzicky na disku v segmentech. Vždy existuje aktivní segment, do kterého se zapisují data a až postupem času se z aktivních segmentů stanou neaktivní (když překročí velikost nebo uplyne nějaký čas). Pouze neaktivní segment je možné promazat. Tyto parametry ovlivňují rychlost čištění segmentů:
    - segment.bytes (default: 1GB): Maximální velikost segmentu
    - segment.ms (default: 7 dnů): Maximální doba kdy dojde k vytvoření nového segmentu i když nebylo naplněno segment.bytes
    - min.cleanable.dirty.ratio (default: 0.5): Jak často se bude čistit log. Ve výchozím nastavení se nebude čistit log, když je víc než 50% logu compacted.

Testing

- https://chrzaszcz.dev/2020/08/kafka-testing/
- https://www.baeldung.com/spring-boot-kafka-testing
- https://docs.confluent.io/platform/current/streams/developer-guide/test-streams.html
- https://blog.jdriven.com/2019/12/kafka-streams-topologytestdriver-with-avro/

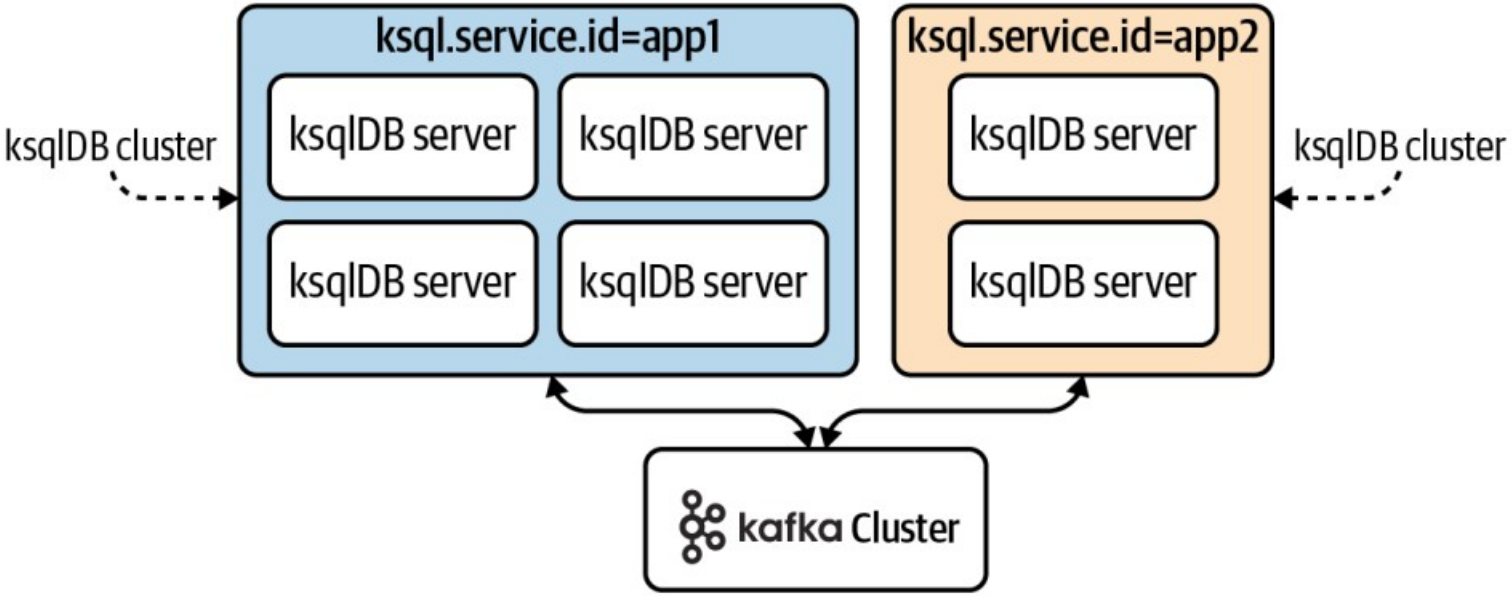
Kafka Streams Examples

- Další příklady od společnosti Confluent:
  - https://github.com/confluentinc/kafka-streams-examples/tree/7.0.0-post/src/main/java/io/confluent/examples/streams

ksqlDB

ksqlDB

- ksqlDB je nadstavba nad Kafka Streams a Kafka Connect. Každá instance ksqlDB funguje jako jedna microservice Kafka Streams aplikace a když se jim nastaví stejné ksql.service.id, pak to je to samé jako když více Kafka Streams instancí má stejné application.id.



Základní konfigurace

- Interactive vs. headless mode:
  - ksqlDB ve výchozím nastavení běží v „interactive mode“, kdy klienti mohou ovládat ksqlDB pomocí REST API. V tomto módu se všechny query ukládají do Kafka topicu _confluent-ksql-default__command_topic.
  - Alternativně se dá spustit v headless módu, ve kterém je vypnuté REST API a všechny query jsou definované při startu v souboru, jehož umístění se nastavuje pomocí property queries.file.
- Kafka Connect integrace má také dva módy:
  - External mode: Kafka Connect cluster může být mimo ksqlDB cluster, poté se jeho umístění nastavuje pomocí property ksql.connect.url.
  - Embedded mode: V tomto módu každá instance ksqlDB serveru plní zároveň roli Kafka Connect workeru. V takovém případě se nastavuje property ksql.connect.worker.config, která odkazuje na properties soubor, ve kterém je konfigurace workeru.

ksqlDB CLI & REST API

- ksqlDB CLI je konzolová aplikace, která slouží k ovládání ksqlDB:
  - https://docs.ksqldb.io/en/latest/operate-and-deploy/installation/cli-config/
- Pro ovládání ksqlDB instance je je také možné použít REST API (pokud neběží v headless módu):
  - https://docs.ksqldb.io/en/latest/developer-guide/api/index.html

use-cases

- Co je možné s ksqlDB dělat?
  - Vytvářet Kafka Connect source & sink connectory
  - Tvořit transient (ad-hoc) query
    - Tyto query nepřežijí restart serveru
  - Tvořit persistent query (query, které získají vstupní data, provedou transformaci a výsledek zapíší do topicu).
    - Tyto query přežijí restart serveru
  - Query mohou být buď ve formátu streamu, nebo ve formátu tabulky.
  - Přidávat záznamy do streamu

Books

- Kafka The Definitive Guide 2nd Edition:
  - https://www.oreilly.com/library/view/kafka-the-definitive/9781492043072/
- Kafka streams & ksqlDB:
  - https://www.amazon.com/Mastering-Kafka-Streams-ksqlDB-Real-Time/dp/1492062499
- Free ebooks (from Confluent):
  - https://www.confluent.io/resources/?language=english&assetType=ebook
  - Včetně výše doporučovaných knih

Kafka & Multiple Datacenters

- https://mbukowicz.github.io/kafka/2020/08/31/kafka-in-multiple-datacenters.html
- https://docs.confluent.io/platform/current/tutorials/examples/multiregion/docs/multiregion.html
- https://cloud.redhat.com/blog/geographically-distributed-stateful-workloads-part-four-kafka