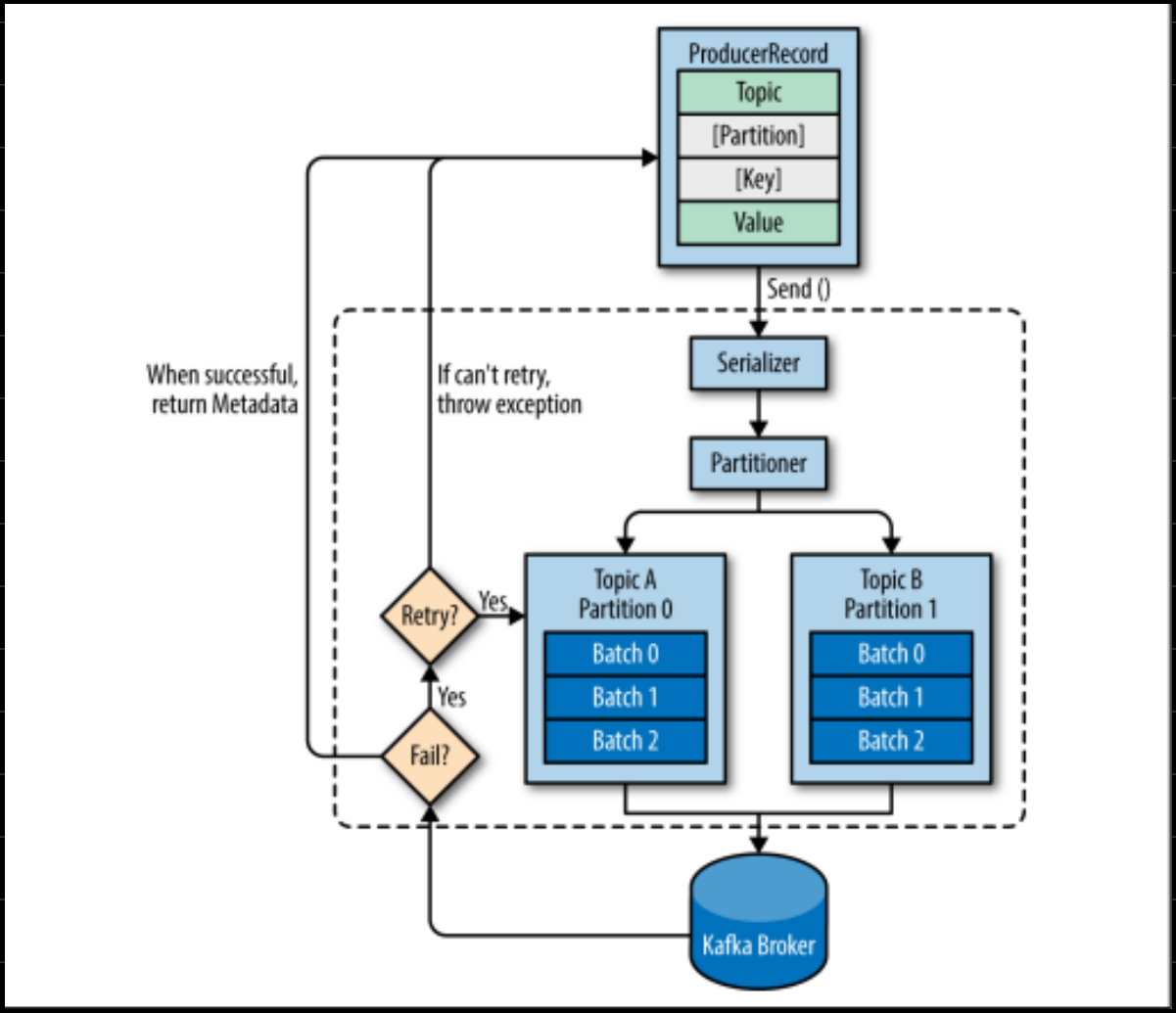
Kafka Book Notes

Producers

→ [KafkaProducer, ProducerRecord]

↑ Important API for Producer.

→

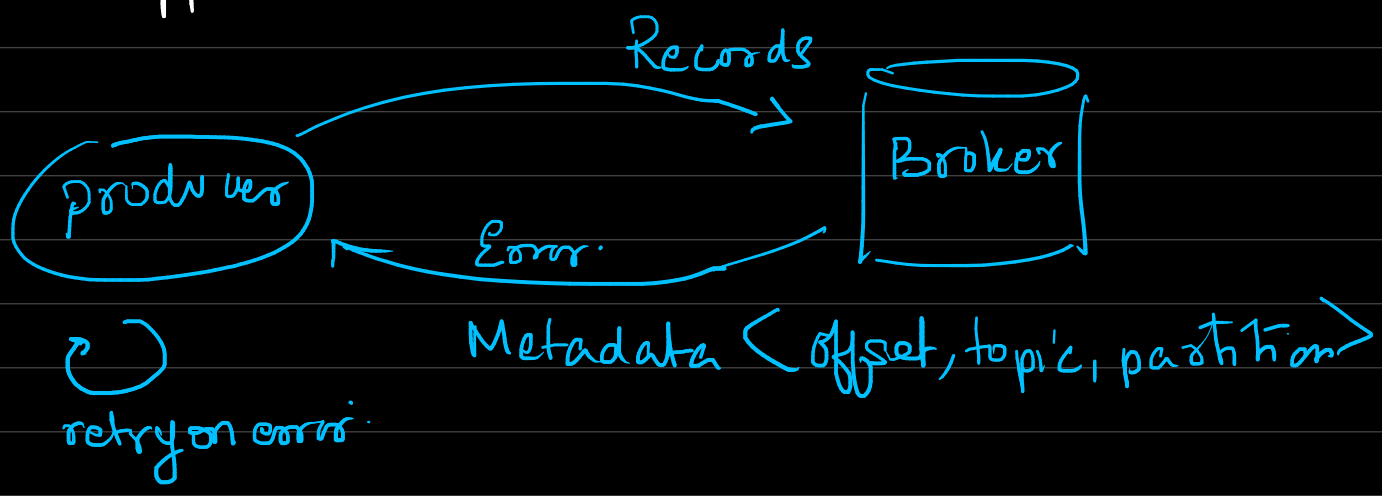


→ High level overview of the producer API

→ Partitioner can be custom. If not given it will be taken as

[hash(Key) % total_partition]

→ A batch of Record i.e keyvalue pair is sent at once to Kafka Broker. This process takes place in a seppparate thread.



Producer Setup.

```
private Properties kafkaProps = new Properties();
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");
kafkaProps.put("key.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps);
```

Send in 3 ways
- → Fire and Forget
- → synchronous – send().get() → Future
- → Asynchronous.
  - ↓
  - send(...., () → {
    - ...
    - })

→ Producer Configuration

- acks = 0 = No acknowledgement / Fast Produce
  Message maybe lost
  1 = Leader ack only
  2 = Once Replication is done ack Leader.

buffer.Memory / compression.type / retries / batch.size.

- buffer.Memory → Controls memory usage for unsent messages
- retries → 100ms default → can use backoff
- batch.size → Controls how many max bytes to use for a single batch
  - low value: frequent flush
  - high value: high write speed

linger-ms → wait period for flushing of current batch.

- 0 = immediate flush
  No batching.

max.in.flight.requests per connection

- max.request.size → Caps off largest produce request

Optimization Using config :
http://www.pattersonconsultingtn.com/blog/throughput_testing_kafka.html

→ [in.flight.request.per.session] → 1 → ordering guaranteed

→ higher → ordering not guaranteed

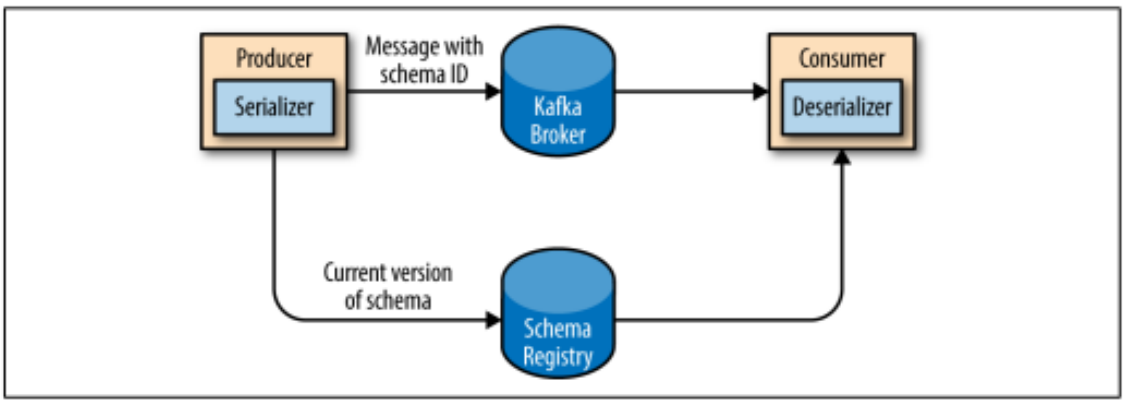


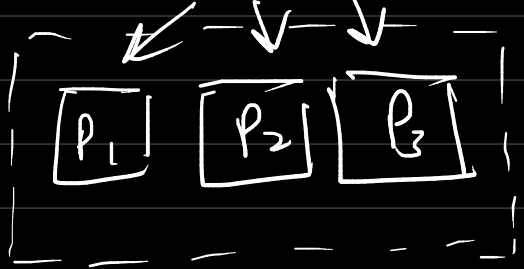
Figure 3-2. Flow diagram of serialization and deserialization of Avro records

← Registry usage.

Partitioning = <key, value> — hash(key) % p_count

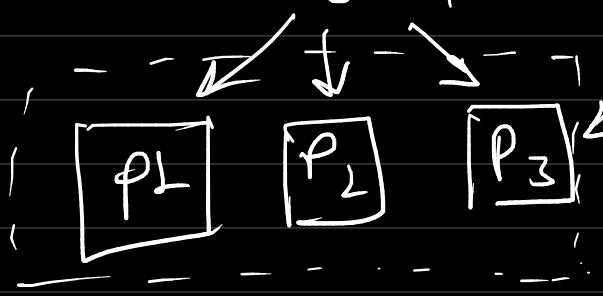
<null, value>

(round robin)



← Topic

- random partition



← Topic

- key based partition

Custom Partitioner

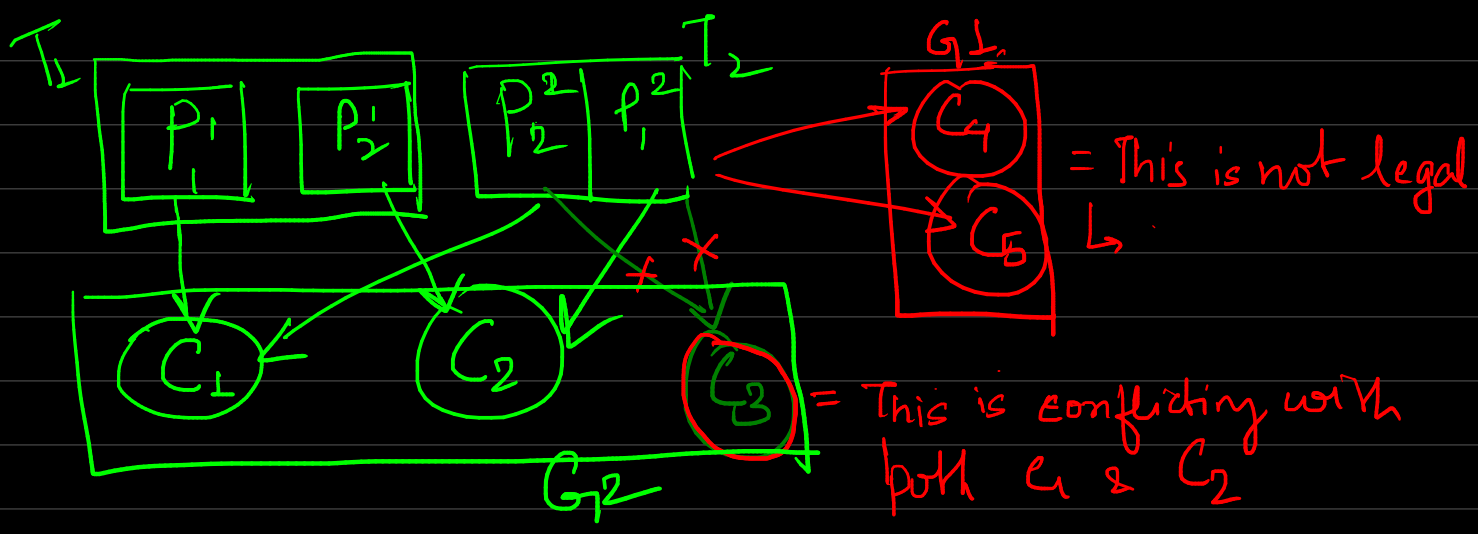
```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;


public class BananaPartitioner implements Partitioner {
        public void configure(Map<String, ?> configs) {}

        public int partition(String topic, Object key, byte[] keyBytes,
                             Object value, byte[] valueBytes,
                             Cluster cluster) {
                List<PartitionInfo> partitions =
                    cluster.partitionsForTopic(topic);
                int numPartitions = partitions.size();
                if ((keyBytes == null) || (!(key instanceOf String)))
                        throw new InvalidRecordException("We expect all messagesto have custom
                        if (((String) key).equals("Banana"))
                                return numPartitions;
                // Banana will always go to lastpartition
                // Other records will get hashed to the rest of the partitions
                return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
        }

        public void close() {}
}
```

Consumer

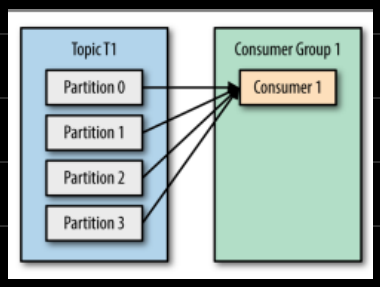
→ Important class to know → [Kafka Consumer]

Consumer & Consumer Group.

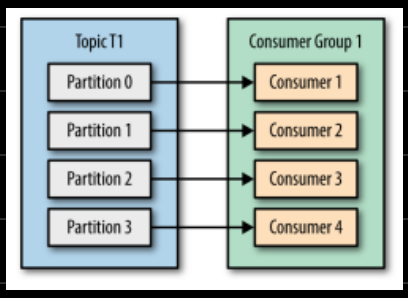
→ Multiple Consumer can read from same topic but not from same partition.

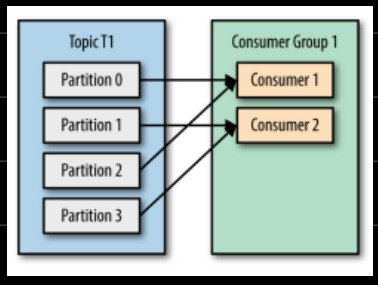


→

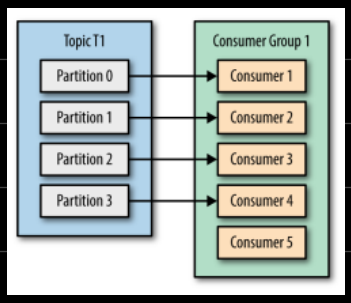


Ideal cases

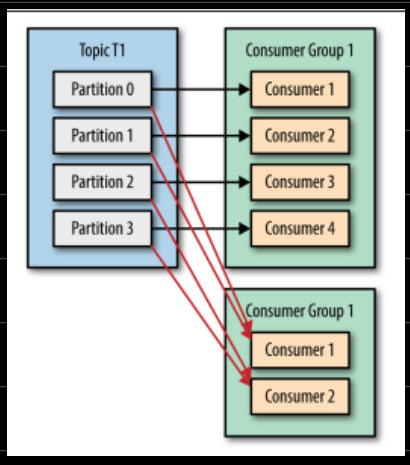




can lead to idle consumer NOT IDEAL!



Multi group scenario



→ Consumer leaving & joining can cause rebalance of partition.

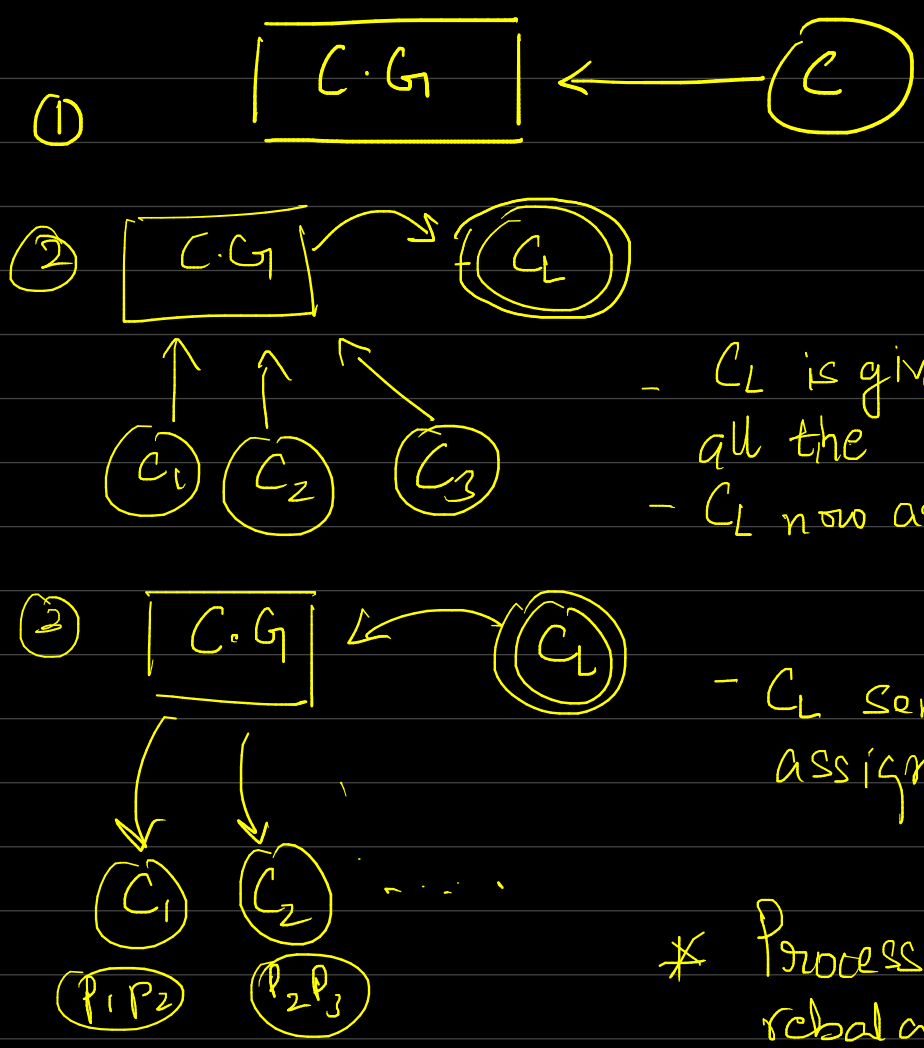
→ Consumer maintains memberships and group ownership by heartbeat to the broker.

→ [Partition Assignment strategy for Consumer]

How Does the Process of Assigning Partitions to Brokers Work?

When a consumer wants to join a group, it sends a `JoinGroup` request to the group coordinator. The first consumer to join the group becomes the group *leader*. The leader receives a list of all consumers in the group from the group coordinator (this will include all consumers that sent a heartbeat recently and which are therefore considered alive) and is responsible for assigning a subset of partitions to each consumer. It uses an implementation of `PartitionAssignor` to decide which partitions should be handled by which consumer.

Kafka has two built-in partition assignment policies, which we will discuss in more depth in the configuration section. After deciding on the partition assignment, the consumer leader sends the list of assignments to the `GroupCoordinator`, which sends this information to all the consumers. Each consumer only sees his own assignment—the leader is the only client process that has the full list of consumers in the group and their assignments. This process repeats every time a rebalance happens.



Elected Leader Because first in group.

- C_L is given the information of all the consumer.
- C_L now assigns the partition.

- C_L sends back the partition assignment to Group Coordinator

* Process repeats for every rebalance.

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer","org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer","org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String,String>(props);


consumer.subscribe(Collections.singletonList("customerCountries"));

// can provide regular expression as well for the topics
```

```
try {
        while (true) {
                ConsumerRecords<String, String> records = consumer.poll(100);
                for (ConsumerRecord<String, String> record : records)
                {
                        log.debug("topic = %s, partition = %s, offset = %d,
                        customer = %s, country = %s\n",
                        record.topic(), record.partition(), record.offset(),
                        record.key(), record.value());
                        int updatedCount = 1;
                        if (custCountryMap.countainsValue(record.value())) {
                                updatedCount = custCountryMap.get(record.value()) + 1;
                        }
                        custCountryMap.put(record.value(), updatedCount)
                        JSONObject json = new JSONObject(custCountryMap);
                        System.out.println(json.toString(4))
                }
        }
} finally {
        consumer.close();
}
```

Consumer loop = Poll.

handles → poll
↳ partition
↳ Rebalance
↳ heartbeat.

→ Configs

* fetch.min.bytes
  ↓
  min data to flush to a consumer
  ↳ Useful to reduce the back & forth.

* fetch.max.wait.ms
  ↳ Reduce latency i.e timer on flush.

* max.partition.fetch.bytes => per partition bytes.

* session.timeout.ms => how long poll can be held.

* auto.offset.reset → earliest → entry 0
  → latest → entry after joining

enable.auto.commit => if commit of offset is done automatically

[partition strategy] => Range / Round Robin
(uneven) (equal)

→ Commits and Offsets.

↳

How does a consumer commit an offset? It produces a message to Kafka, to a special `__consumer_offsets` topic, with the committed offset for each partition. As long as all your consumers are up, running, and churning away, this will have no impact. However, if a consumer crashes or a new consumer joins the consumer group, this will *trigger a rebalance*. After a rebalance, each consumer may be assigned a new set of partitions than the one it processed before. In order to know where to pick up the work, the consumer will read the latest committed offset of each partition and continue from there.

→ use commitSync() for manual control.
→ use commitAsync() for not waiting on commit and go ahead with poll.
→

Retrying Async Commits

A simple pattern to get commit order right for asynchronous retries is to use a monotonically increasing sequence number. Increase the sequence number every time you commit and add the sequence number at the time of the commit to the `commitAsync` callback. When you're getting ready to send a retry, check if the commit sequence number the callback got is equal to the instance variable; if it is, there was no newer commit and it is safe to retry. If the instance sequence number is higher, don't retry because a newer commit was already sent.

→ You can also commit per partition value manually

↳ Continuos commit in case of large processing time

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =new HashMap<>();
int count = 0;
....
while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records)
        {
                System.out.printf("topic = %s, partition = %s, offset = %d,
                customer = %s, country = %s\n",
                record.topic(), record.partition(), record.offset(),
                record.key(), record.value());
                currentOffsets.put(new TopicPartition(record.topic(),
                record.partition()), new
                OffsetAndMetadata(record.offset()+1, "no metadata"));
                if (count % 1000 == 0)
                consumer.commitAsync(currentOffsets, null);
                count++;
        }
}
```

↖ Offset manual commits

→ Rebalance Listeners

↳ Allows you to handle commits before rebalance occurs

* ↳ Use "consumer.wakeup()" to exit out of polling

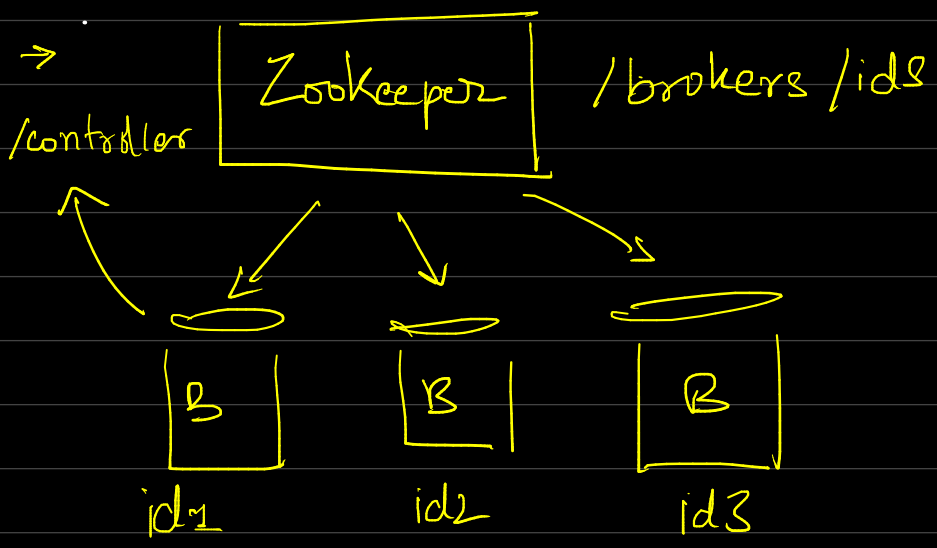
↓

throws exception

Kafka Internals

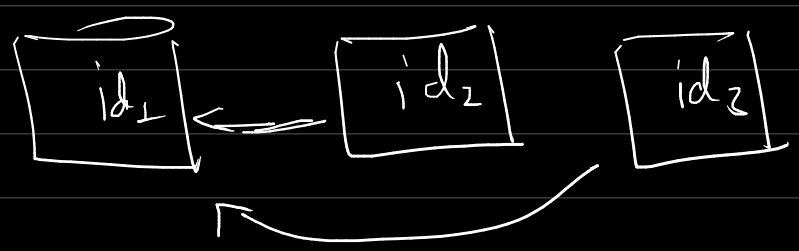
Controller

→ /controller



Controller = first to join cluster

→



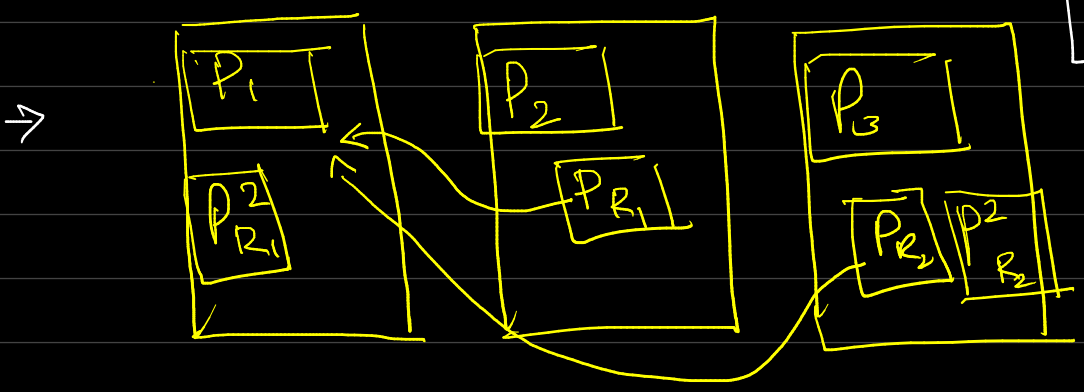
- listen on controller for any changes.

→ Controller keeps track of active broker. If some broker leaves or dies then it triggers leader election for replica

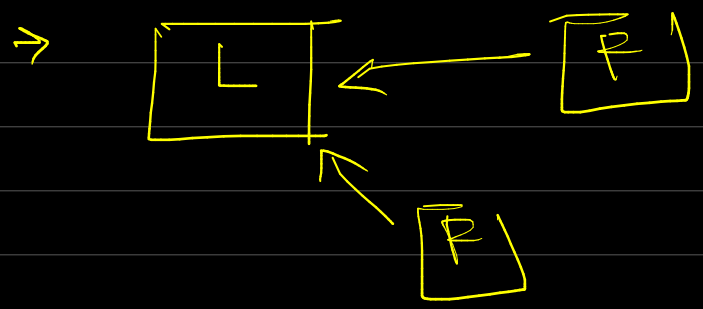
→ Controller uses epoch to fix "Split brain" scenario.

→

Replication



P = Partition (Leader)
PR = Replicas (Followers)



fetch request with offset.

→ If a replica is out of sync it cannot become the new leader

→

Finding the Preferred Leaders

The best way to identify the current preferred leader is by looking at the list of replicas for a partition (You can see details of partitions and replicas in the output of the `kafka-topics.sh` tool. We'll discuss this and other admin tools in Chapter 10.) The first replica in the list is always the preferred leader. This is true no matter who is the current leader and even if the replicas were reassigned to different brokers using the replica reassignment tool. In fact, if you manually reassign replicas, it is important to remember that the replica you specify first will be the preferred replica, so make sure you spread those around different brokers to avoid overloading some brokers with leaders while other brokers are not handling their fair share of the work.

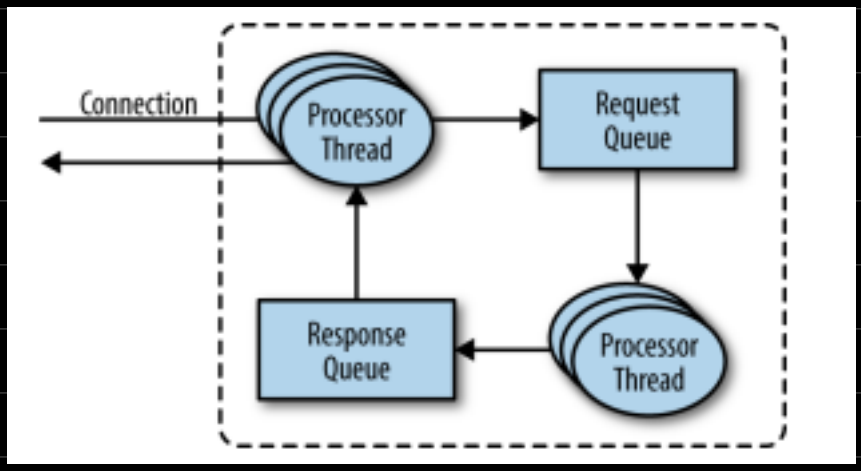
Request Processing

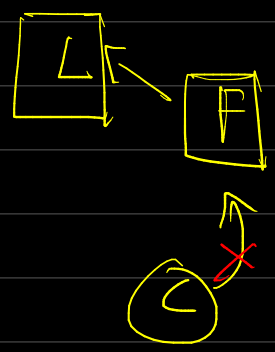
→ All request sent will be processed in order of their delivery → This is the garuntee provided by Kafka.

→ Each broker listens on an acceptor thread. Creates a connection on request and hands it over to the processor thread for handling.

→ Processor thread count is configurable.

→





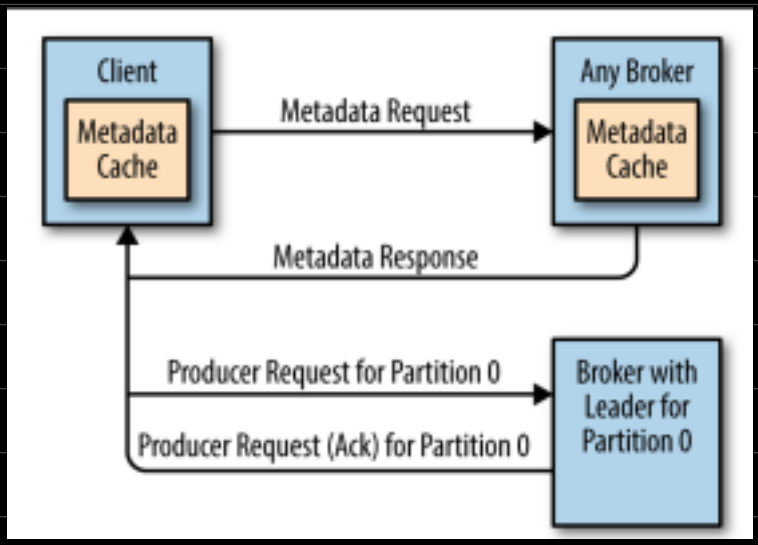
↳ Error

Follower can not take requests

→ Metadata Requests

↳ Tells the topic to position mapping

Standard Req-Res Cycle for new client

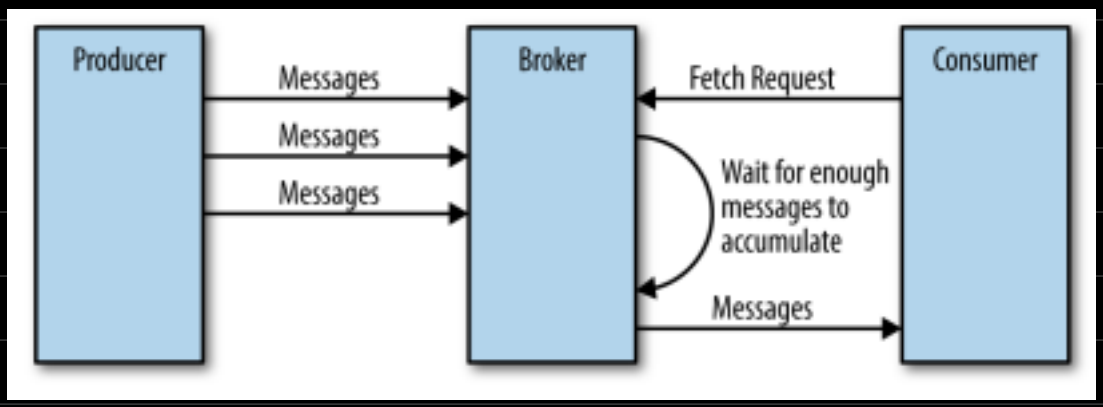


→ Validation Ran for each produce

↳ Write Privilege present / ack belongs 0, 1, or 2 / Are there enough in-sync replicas.

→ Fetch are processed by [Zero Copy Writes / Reads]

→



Broker delaying message until buffer is filled

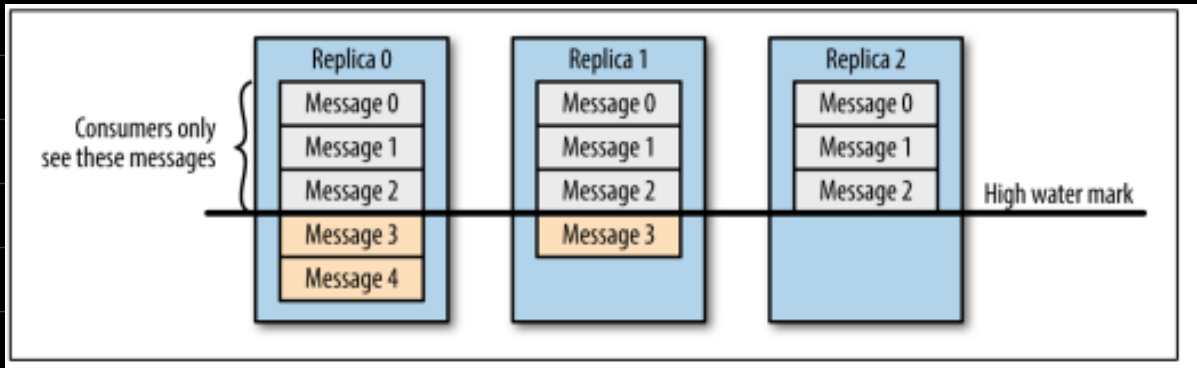


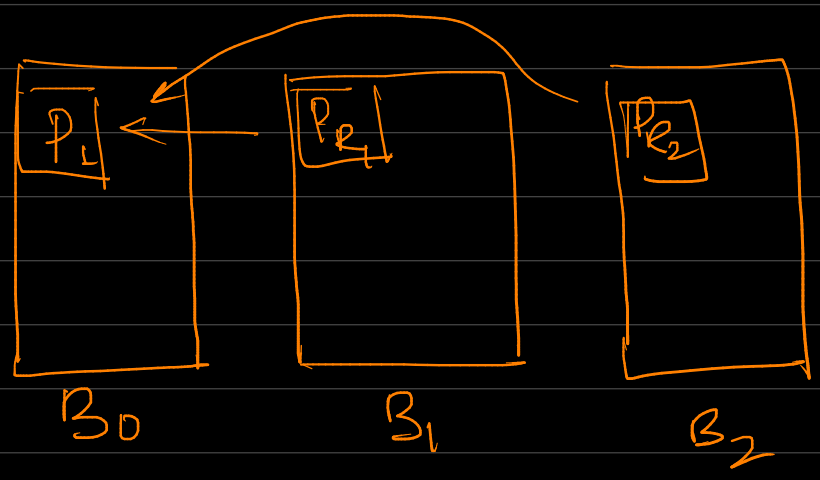
Figure 5-4. Consumers only see messages that were replicated to in-sync replicas

No consumer can read beyond the high water mark. This is for making sure only committed events are transmitted.

[Physical Storage]

→ Partition file cannot be split across disk i.e max size of partition is same as mount point of the current disk.

→



Brokers are always trying to distribute the workload/Replicas equally

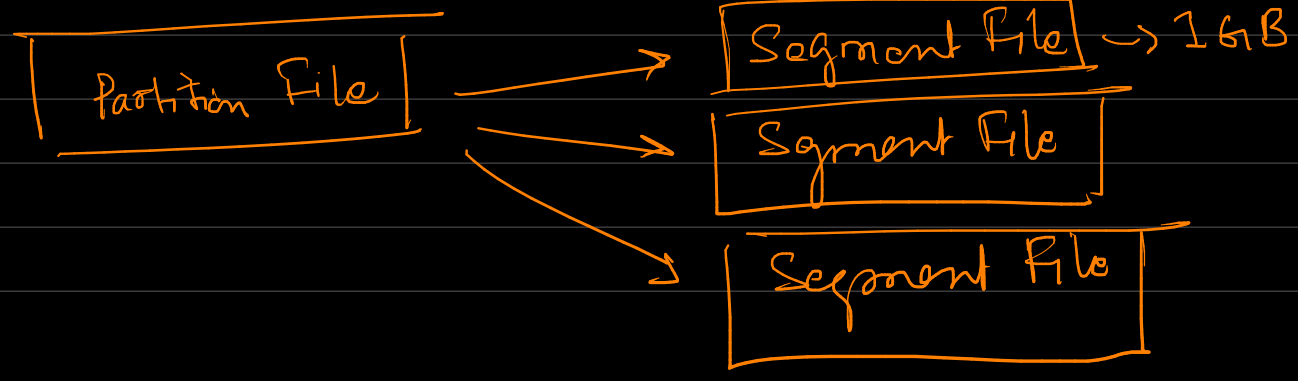
* Replication is done in a rack aware manner.

*

Mind the Disk Space

Note that the allocation of partitions to brokers does not take available space or existing load into account, and that allocation of partitions to disks takes the number of partitions into account, but not the size of the partitions. This means that if some brokers have more disk space than others (perhaps because the cluster is a mix of older and newer servers), some partitions are abnormally large, or you have disks of different sizes on the same broker, you need to be careful with the partition allocation.

*



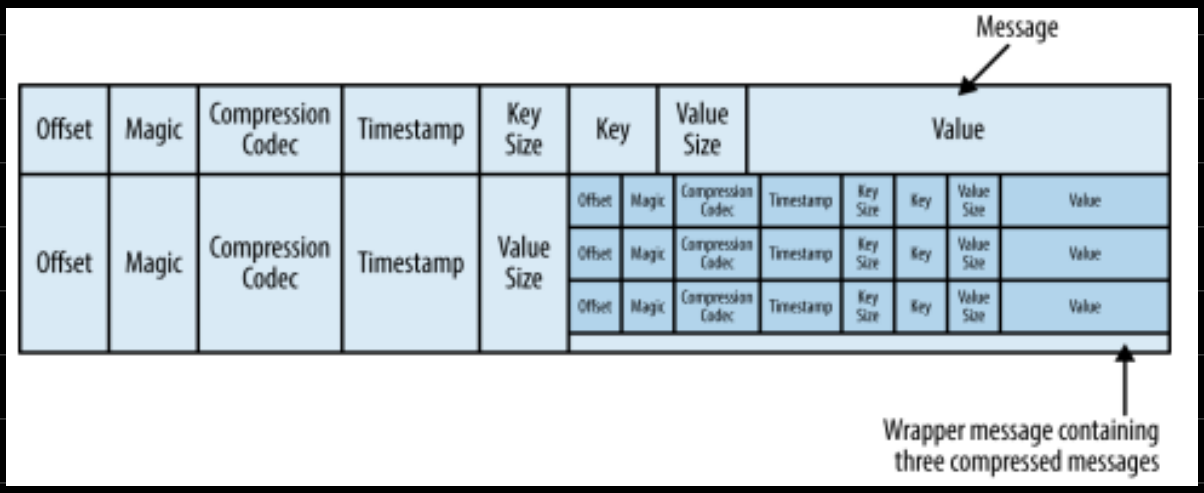
currently in write

→ If [active segment] contains past 5 days data then past 5 days data will be retained since you cannot delete active segments.

→ Kafka keeps file handles to inactive segments as well.

→ Kafka uses same format of write used by consumers this allows the zero copy mechanism.

→



Standard message structure.

→ Index: Kafka maintains an index of offset since we want to allow reading of an available offset.

index contains info of

offset → <file, file offset>

→ Compaction: Is done to reduce file space.

K_1	K_1	K_2	K_2	K_4	K_3	K_3
0	1	2	3	4	5	6

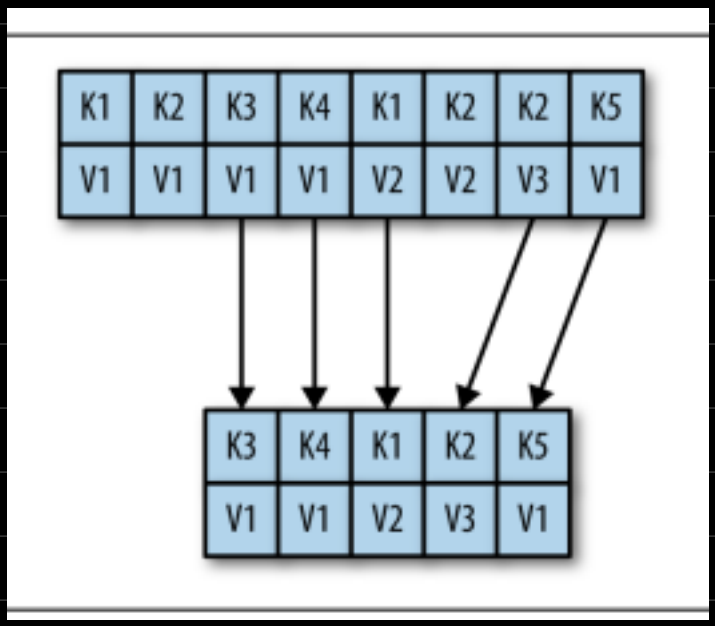
compacted = 2 entry

K_1	K_2	K_2	K_1	K_3	K_3
1	2	3	4	5	6

compd = 1 entry

K_2	K_4	K_3
3	4	6

* This is done in a seppate thread.



Tombstone message : <Key : null> ⇒ Instruct Kafka to clean all logs for the key.

[Reliable Data Delivery]

Reliability Tenets

- Ordering in a single partition
- Committed once written in all "in-sync-Replicas". not necessarily in the disk.
- Only committed messages can be read.

Out-of-Sync Replicas

Seeing one or more replicas rapidly flip between in-sync and out-of-sync status is a sure sign that something is wrong with the cluster. The cause is often a misconfiguration of Java's garbage collection on a broker. Misconfigured garbage collection can cause the broker to pause for a few seconds, during which it will lose connectivity to Zookeeper. When a broker loses connectivity to Zookeeper, it is considered out-of-sync with the cluster, which causes the flipping behavior.

- Kafka does atleast-once-delivery
  If you want exactly once then use an external Key-Value store for same

-

[PIPELINES - EVENT STREAMS]

Properties of Event Streams

→ Ordered / Immutable data record / Replayable

→

Processing Concepts

→ Problem with Time

- If R_1 & R_2 are of different zone then time data might be different.
- A common agreement of time representation is must.

→ Maintain State

- Local
  - Fast
  - Not tolerant
- External → Syn issue
  - Tolerant
  - Slow

→ Stream – Table Duality → Every table is formed by sequence of events i.e stream

→ Such transformation of aggregation requires windowing

→

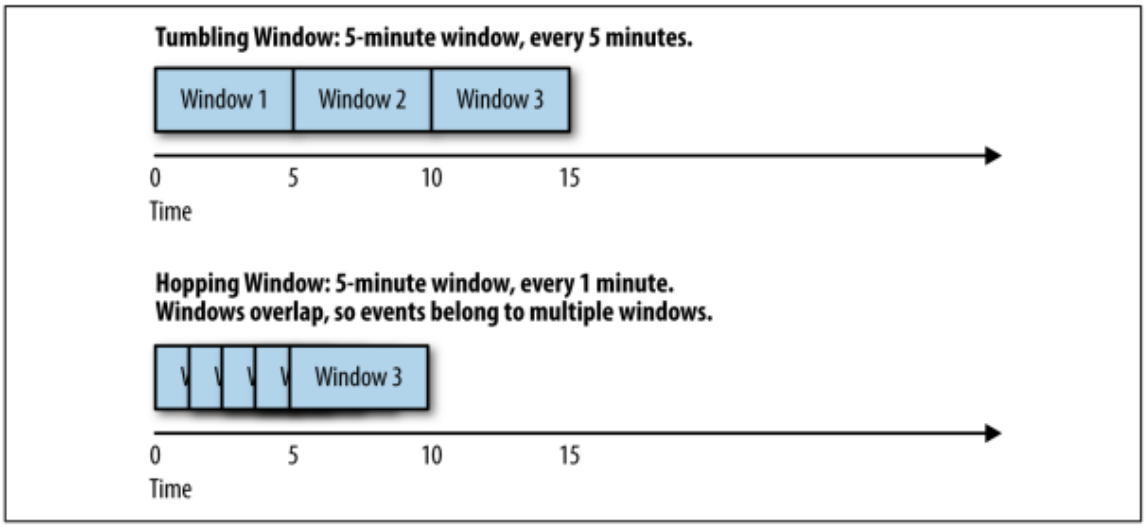


Figure 11-2. Tumbling window versus Hopping window

DESIGN PATTERNS

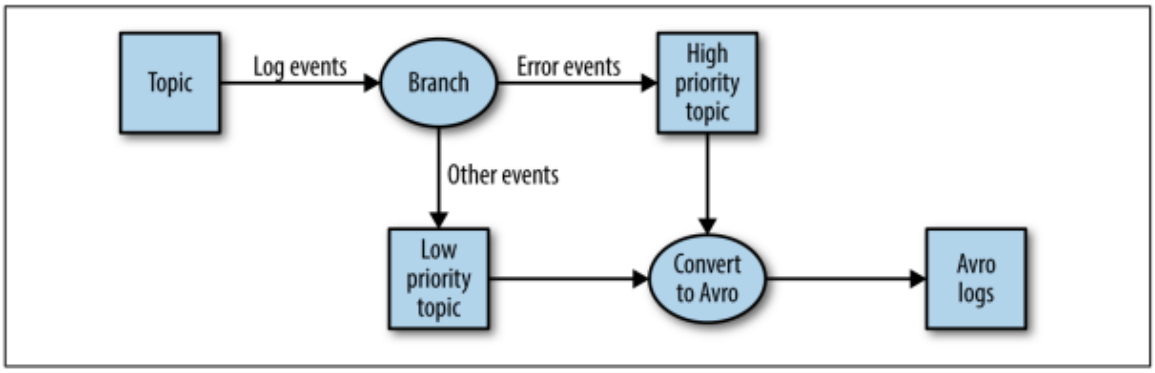


Figure 11-3. Single-event processing topology

↑

Common Consumer Example, Advantage is easy scale & load balancing management of various topic

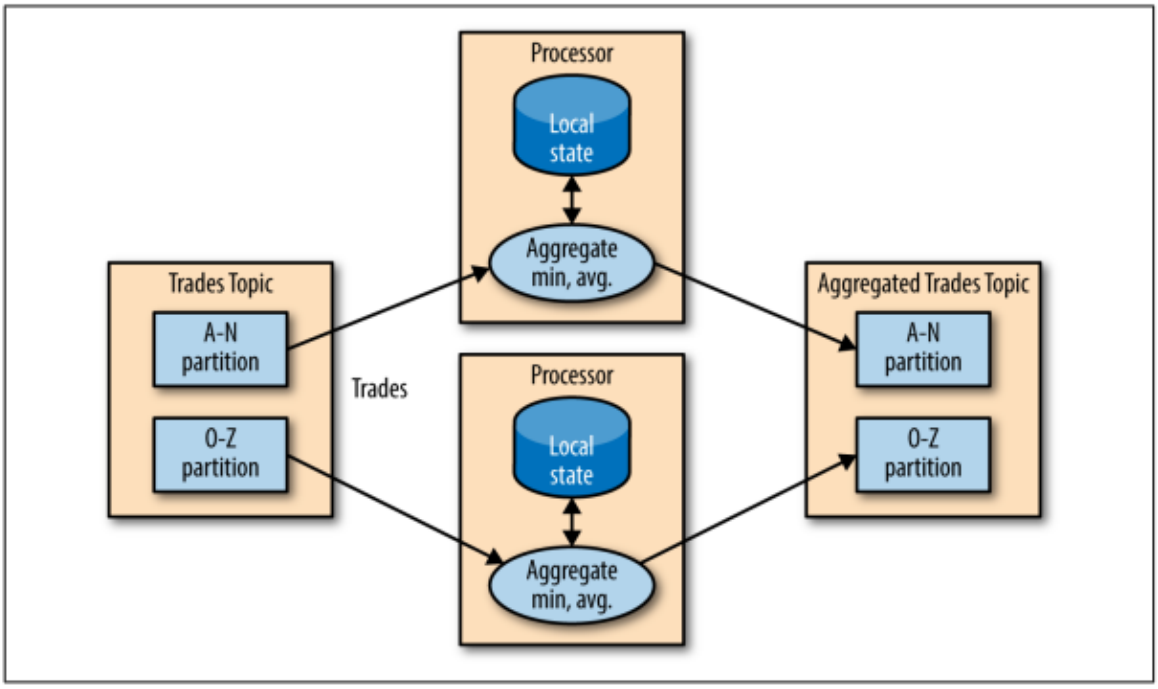


Figure 11-4. Topology for event processing with local state

Useful for aggregations.

Issues

- Memory usage – memory can explode
- Persistence – pods must be ephemeral i.e data recovery issue
- Rebalancing – On rebalance how is data exchanged.

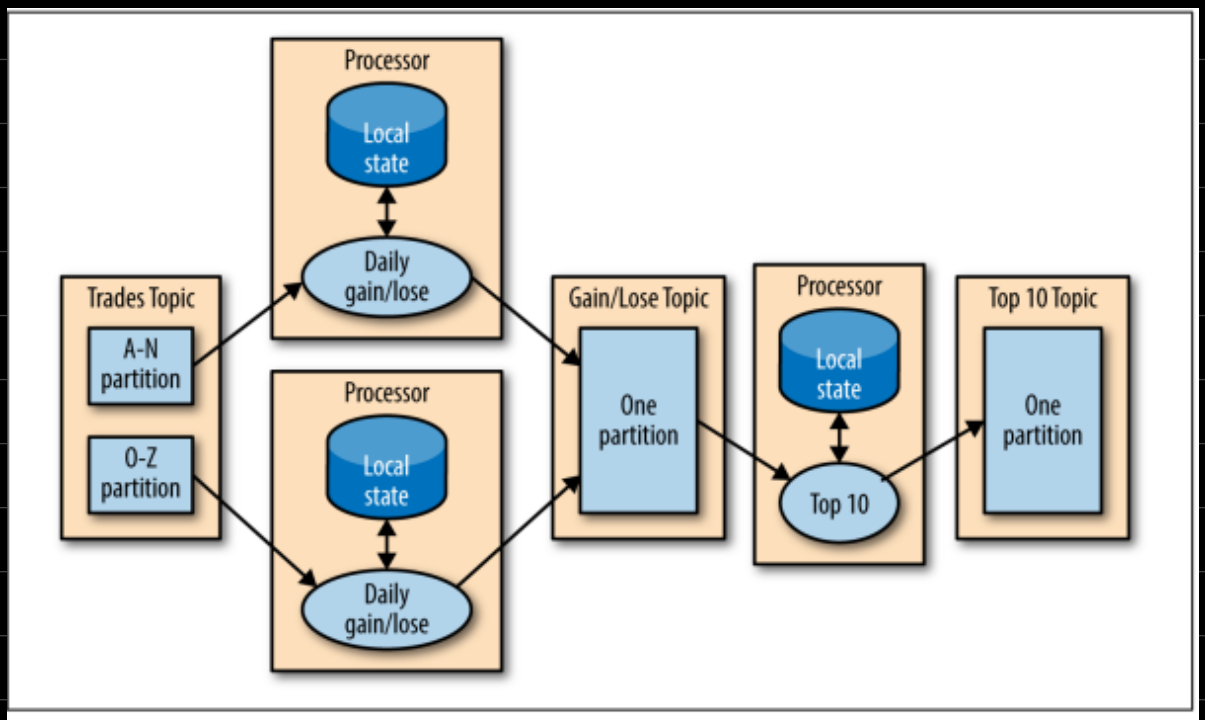


Figure 11-5. Topology that includes both local state and repartitioning steps

Map-Reduce pattern

- Big advantage is you can use same application code everywhere unlike hadoopMR requiring separate instances of app running.
- Flexibility in language
- Easy store and Recovery of operation as well.

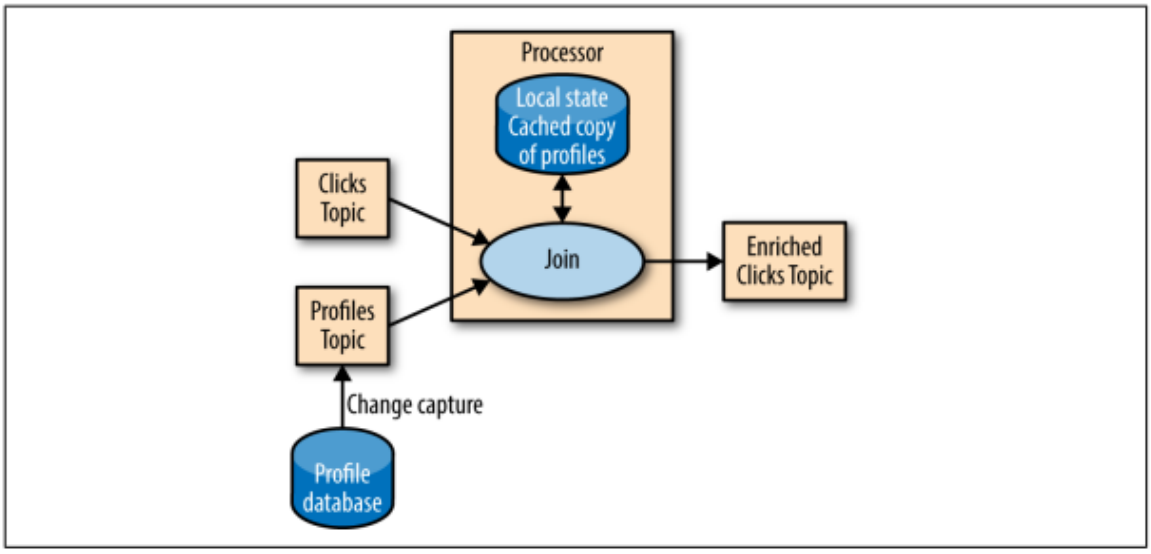


Figure 11-7. Topology joining a table and a stream of events, removing the need to involve an external data source in stream processing

Streaming Joins

→ Useful when you want to enrich data

→ The local state copy is needed since you dont want to bring data form DB over network I/O.

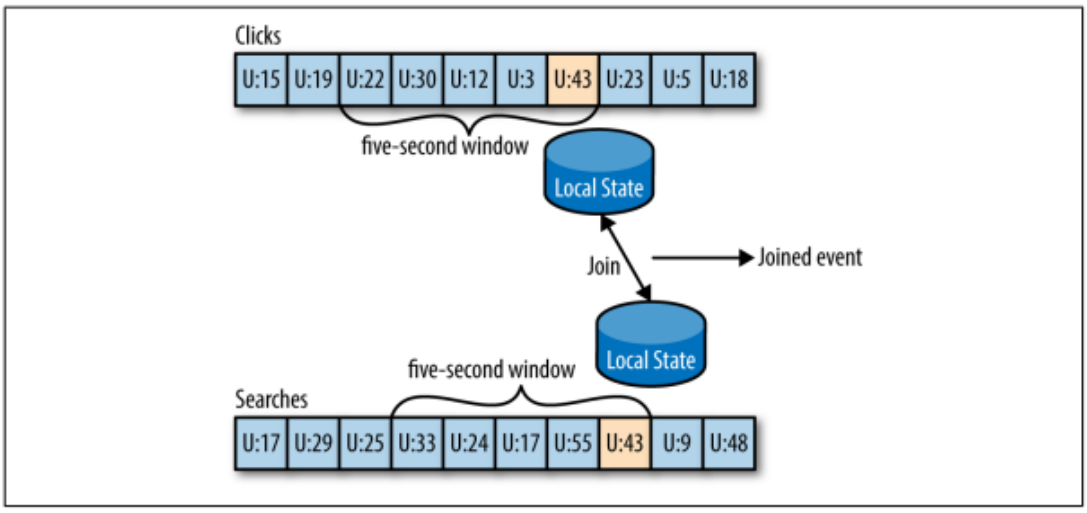


Figure 11-8. Joining two streams of events; these joins always involve a moving time window

Streaming Join (II) → Windowed Join.

→ Keep local data copy & match on some key

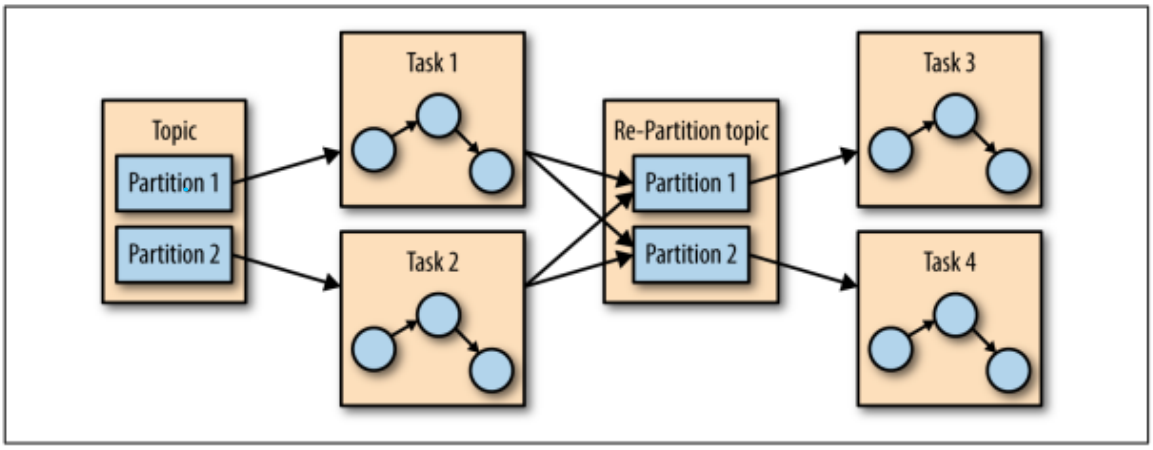


Figure 11-13. Two sets of tasks processing events with a topic for re-partitioning events between them

For people familiar with MR

Kafka = (Shuffle phase and Task distribution)