Keeping your State at Bay: Patterns to Limit Kafka Streams Store Sizes

Why did the stream processor become a philosopher?

...because it realized time is just an ever-flowing state of events.

Hartmut Armbruster

Software Architect, Developer,
Independent Consultant

Architecture • Data • Cloud-native • Distributed Systems • High-load/Scalability • Stream Processing • Backend • Web Front-End • Reactive Programming • Kotlin/Java • TS/JS • Vue.js • Nuxt.js • Kubernetes • GitOps

Agenda

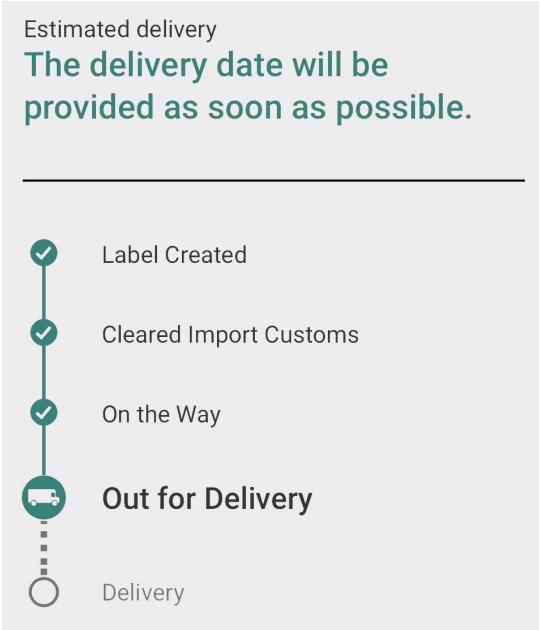
- Subject: "Limiting Store Sizes"
- Key Challenges (Growing State)
- Data Expiration Requirements
- TTL Patterns in Kafka Streams
- Dealing with Large(!) State
- Summary

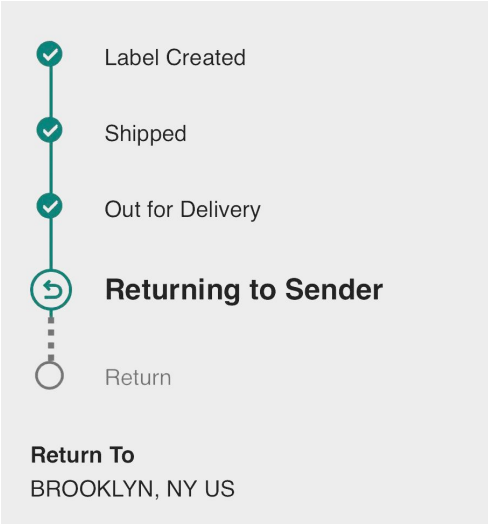
by Dall-E 3

Subject: "Limiting Store Sizes"

"Time is of the essence"

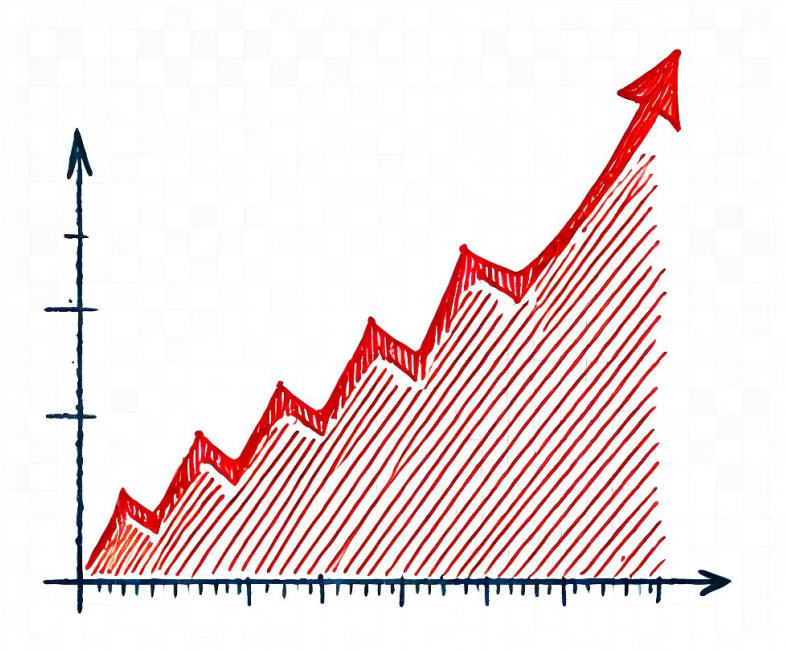
- Data loses relevance over time
- *Windows* keep focus on recent data
- Outdated events are evicted
- **Challenge: dynamic expiry,** based on predicates (not just time-based)



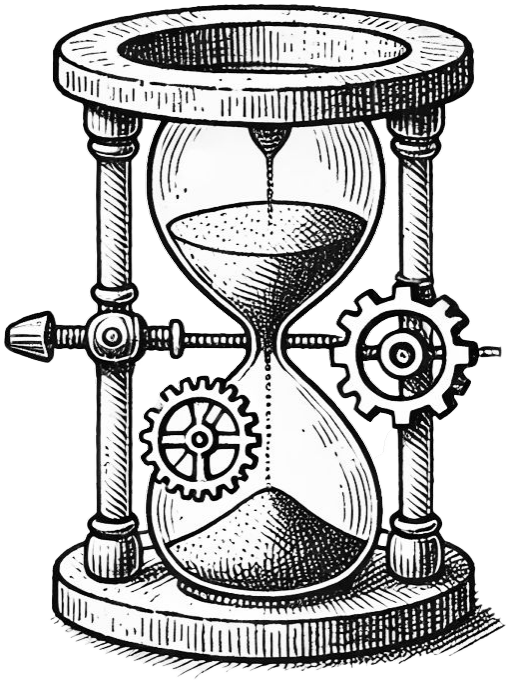


Key Challenges (Growing State)

- Resource Usage
- Scaling Issues
- Rebalancing Delays
- Longer Recovery (State Restoration)
- Operational Overhead



Data Expiration Requirements



- Dynamic Expiry Rules
  (Predicate -> Business Logic)
- Customisable TTL
  & Cleanup Interval
- Efficient Eviction Mechanism
- No/Little Impact on Stream Processing
  (NFR -> Latency)

TTL Patterns in Kafka Streams

Fundamentals: Processor API

(aka PAPI)

```
public interface Processor<KIn, VIn, KOut, VOut> {

    default void init(final ProcessorContext<KOut, VOut> context) {}

    void process(Record<KIn, VIn> record);

    default void close() {}
}
```

- Maximum flexibility
- Access to *state stores* for custom stateful operations
- Schedule a *punctuator*
- Manual *forward* and *commit*

Fundamentals: State Stores

Store Variants

- Key-Value
- Window
- Session

Key-Value Store Types

- Persistent (RocksDB)
- InMemory
- LRUCache

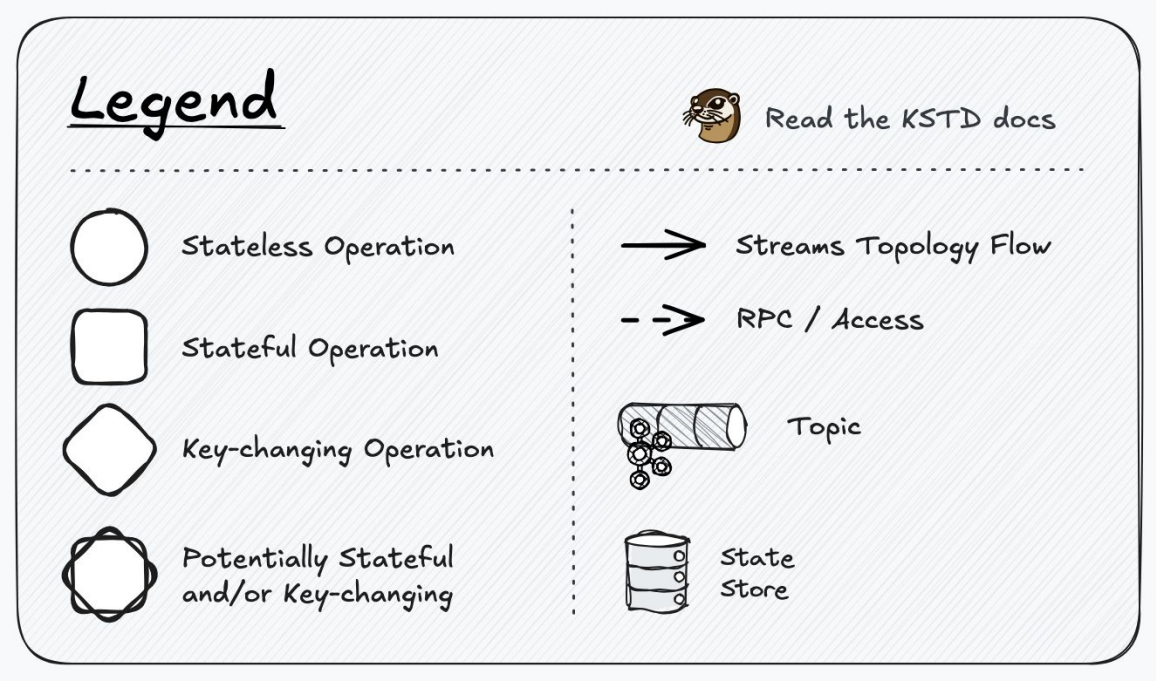
```
builder.addStateStore(
  Stores.keyValueStoreBuilder(
      Stores.persistentKeyValueStore(STATE_STORE),
      Serdes.String(),
      Serdes.Long())
);
```

Fundamentals: Punctuator

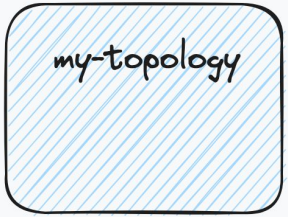
```java
/**
 * ProcessorContext#schedule(Duration, PunctuationType, Punctuator)
 */
public interface Punctuator {

    /**
     * @param timestamp when the operation is being called
     */
    void punctuate(long timestamp);
}
```

- Periodic tasks within processors
- Scheduled based on
  - *Stream-Time* or
  - *Wall-Clock-Time*
- Access to ProcessorContext

Kafka Streams Topology Design - Notation



Topology

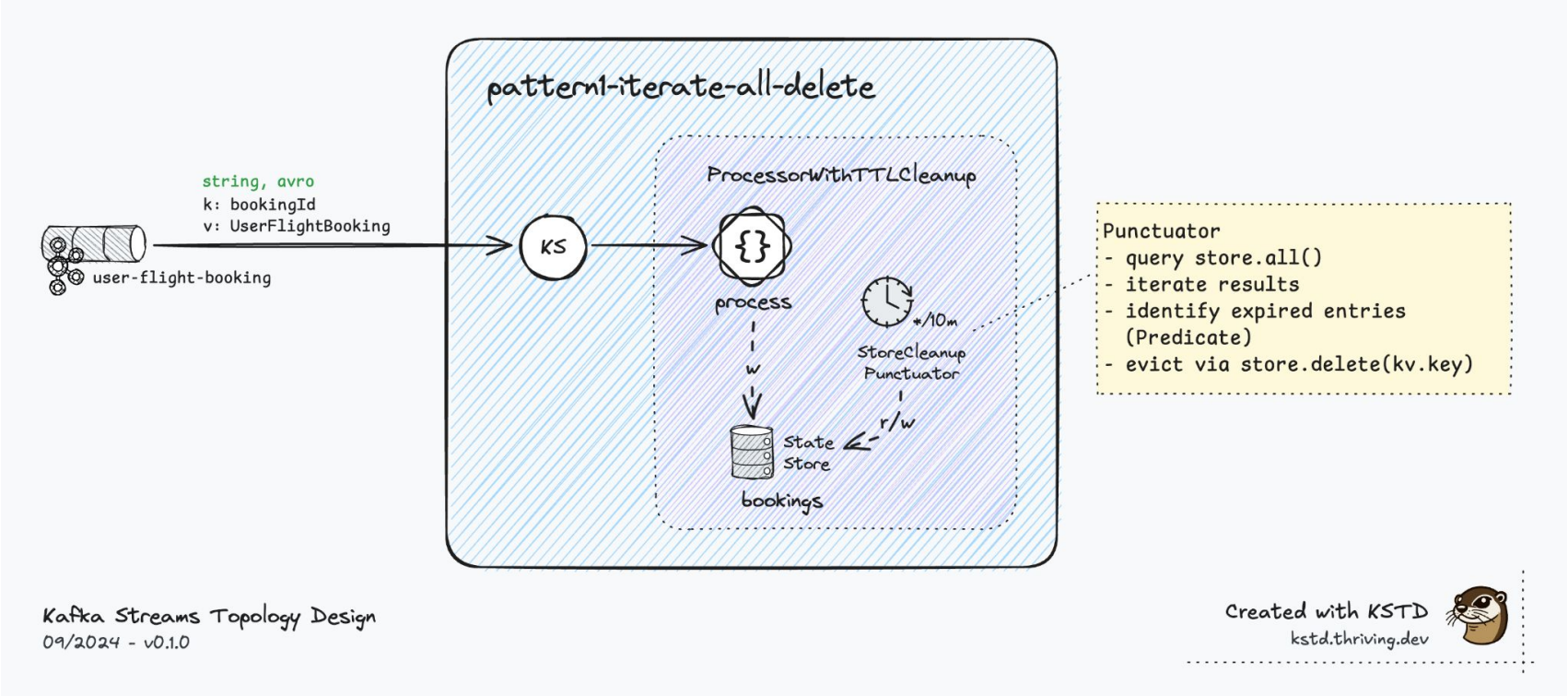


Processor Context

Key-Value Record

```
string, avro
k: userId
v: UserEvent
```

Pattern 1: Processor, punctuate, iterate all(), delete

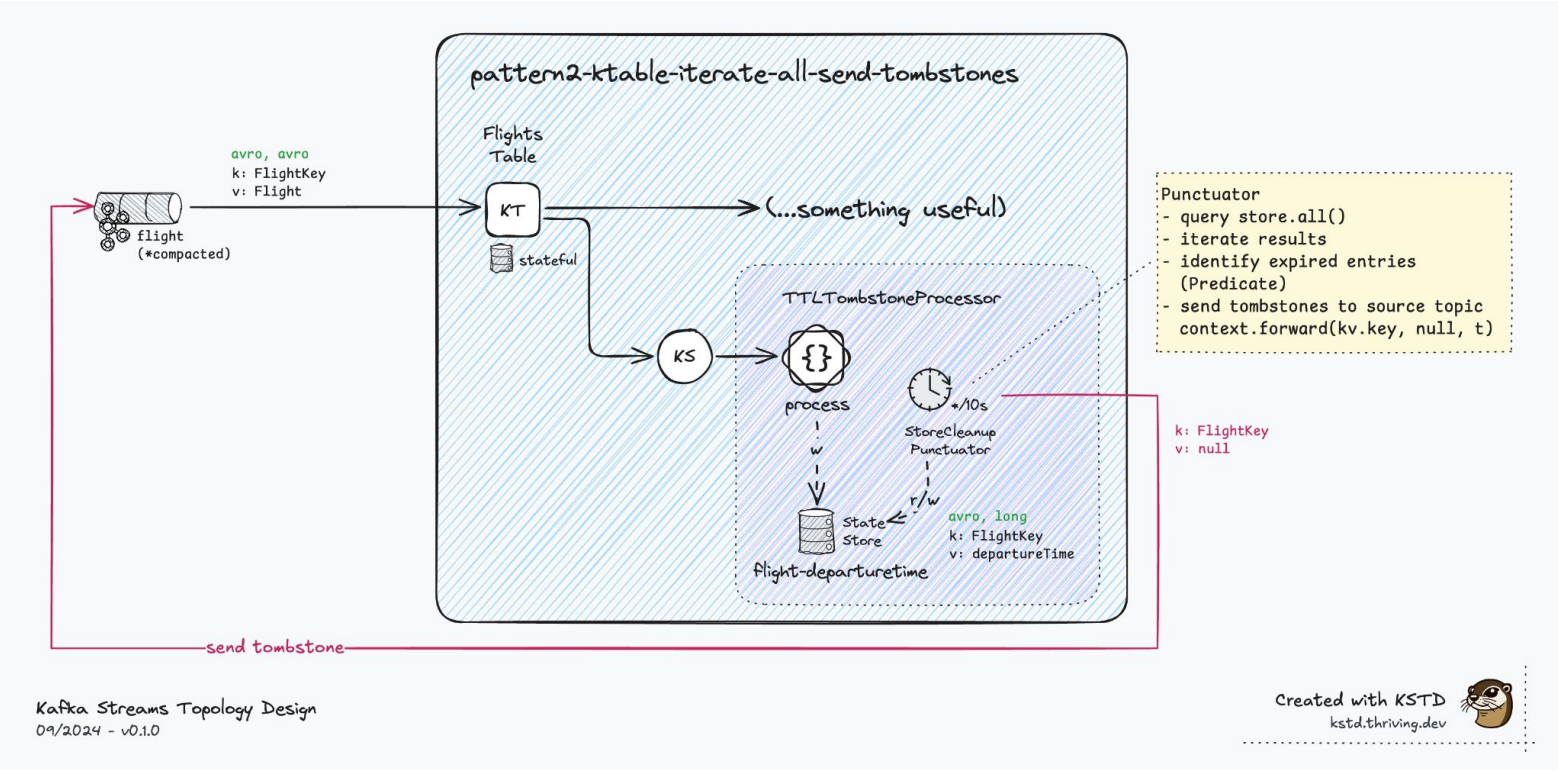


Pattern 1: Processor, punctuate, iterate all(), delete

```
builder.addStateStore(Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore(STATE_STORE),
        stringSerde,
        userFlightBookingAvroSerde
));

builder.<String, UserFlightBooking>stream(INPUT_TOPIC)
        .process(ProcessorWithTTLCleanup::new, STATE_STORE);
```

```
@Override
public void init(ProcessorContext<String, UserFlightBooking> context) {
    store = context.getStateStore(STATE_STORE);
    context.schedule(ofSeconds(10), WALL_CLOCK_TIME, timestamp -> {
        store.flush(); // flush first, to allow deletes while iterating

        // query and iterate all store entries
        try (KeyValueIterator<String, UserFlightBooking> iter = store.all()) {
            iter.forEachRemaining(kv -> {
                LocalDate departureDate = parseDate(kv.value.getDepartureDate());
                // delete records 3 days after departureDate
                if (departureDate.isBefore(now().minusDays(3))) {
                    store.delete(kv.key);
                }
            });
        }
    });
}
```

Pattern 2: KTable, punctuate, iterate all(), send tombstones

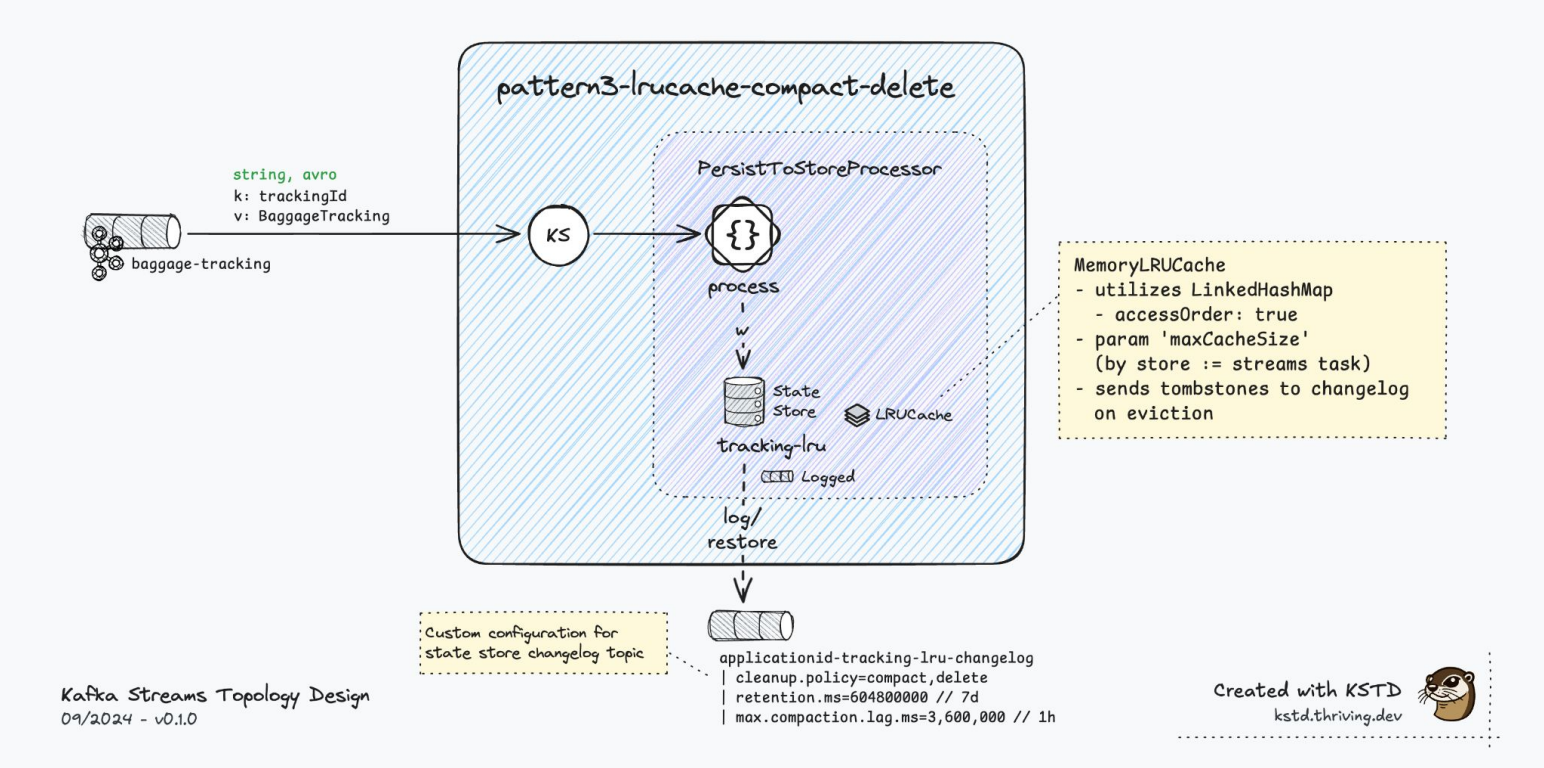


Pattern 2: KTable, punctuate, iterate all(), send tombstones

```
builder.addStateStore(Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore(STATE_STORE_TTL),
        flightKeySerde,
        longSerde
));

KTable<FlightKey, Flight> table = builder.table(INPUT_TOPIC);

table.toStream()
        .process(ProcessorWithTTLCleanup::new, STATE_STORE_TTL)
        .to(INPUT_TOPIC, Produced.with(flightKeySerde, flightSerde));
```

```
@Override
public void init(ProcessorContext<FlightKey, Flight> context) {
    store = context.getStateStore(STATE_STORE_TTL);
    context.schedule(PUNCTUATOR_INTERVAL, WALL_CLOCK_TIME, timestamp -> {
        try (KeyValueIterator<FlightKey, Long> iter = store.all()) {
            iter.forEachRemaining(kv -> {
                if (kv.value < timestamp - TTL_MILLIS) {
                    // send tombstone
                    context.forward(new Record<>(kv.key, null, timestamp));
                }
            });
        }
    });
}

@Override
public void process(Record<FlightKey, Flight> record) {
    if (record.value() == null) {
        store.delete(record.key());
    } else {
        Long departureTimeMillis = parseToEpochMilli(record.value().getDepartureTime());
        store.put(record.key(), departureTimeMillis);
    }
}
```

Pattern 3: LRUCache, changelog retention=[compact,delete]



Pattern 3: LRUCache, changelog retention=[compact,delete]

```
builder.addStateStore(Stores.keyValueStoreBuilder(
        Stores.lruMap(STATE_STORE, 10_000), // maxCacheSize
        stringSerde,
        baggageTrackingSerde
).withLoggingEnabled(Map.of(
        TopicConfig.CLEANUP_POLICY_CONFIG, "compact,delete",
        TopicConfig.RETENTION_MS_CONFIG, "300000", // 5m
        TopicConfig.MAX_COMPACTION_LAG_MS_CONFIG, "300000" // 5m
)));
```

- maxCacheSize := by stream task
- InMemory...

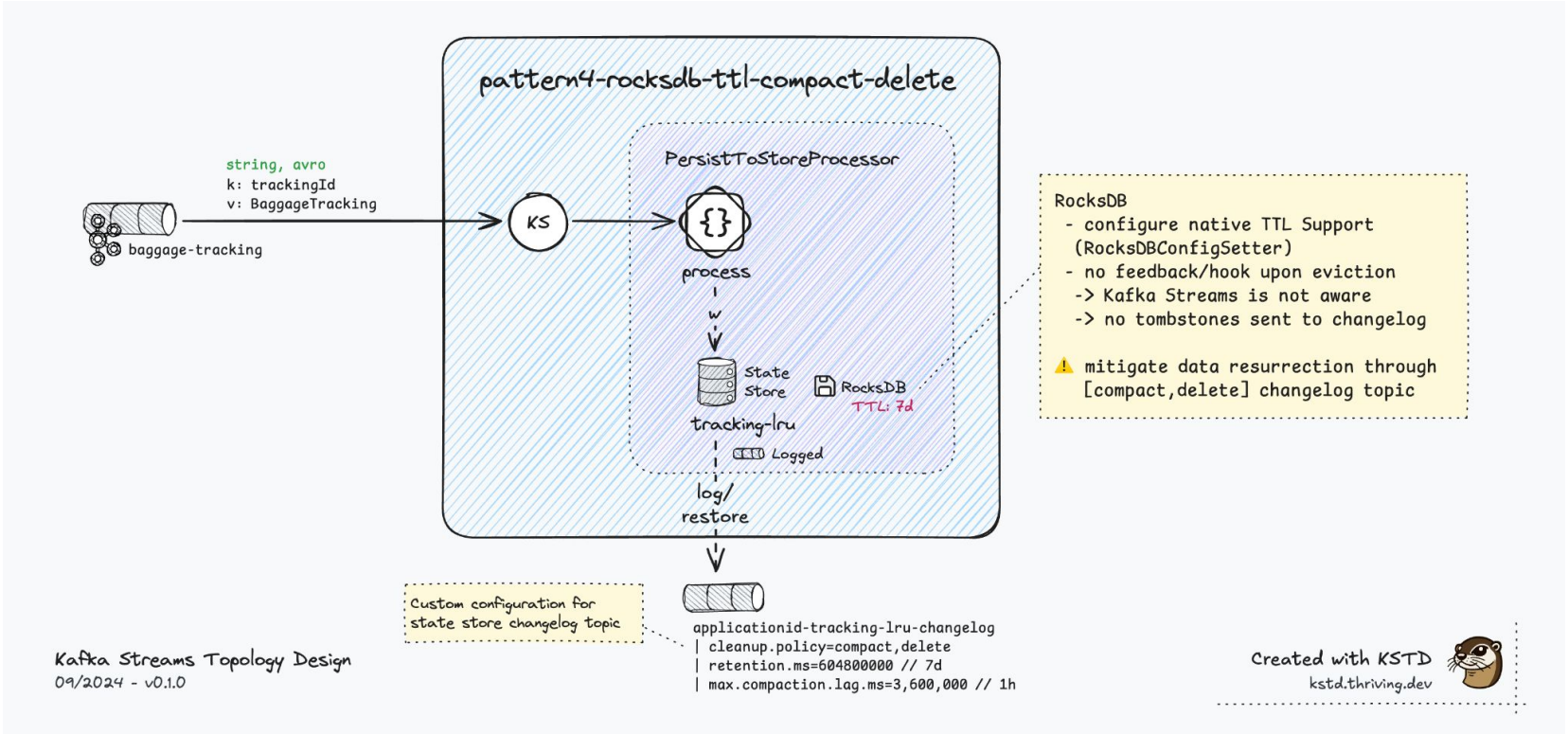
  `LinkedHashMap<Bytes, byte[]>`
- Entries are serialized

⚠️ Know your data (volumes)

- add sufficient buffer
- caution: risk of data loss in *exceptional situations*

  (e.g. increased no. of events due to unforeseeable reasons)

Pattern 4: RocksDB TTL, changelog retention=[compact,delete]



Pattern 4: RocksDB TTL, changelog retention=[compact,delete]

(1) RocksDB can be opened with Time to Live support

- *insertion* based (not *update*, not *access*)
- RocksDB docs:
  - non-strict 'ttl'
  - values are deleted in compaction only
  - get/Iterator may return expired entries

(2) Mitigate data resurrection through [compact,delete] changelog topic

(3) RocksDBConfigSetter is configured once, and therefore applies to the entire topology...

Pattern 4: RocksDB TTL, changelog retention=[compact,delete]

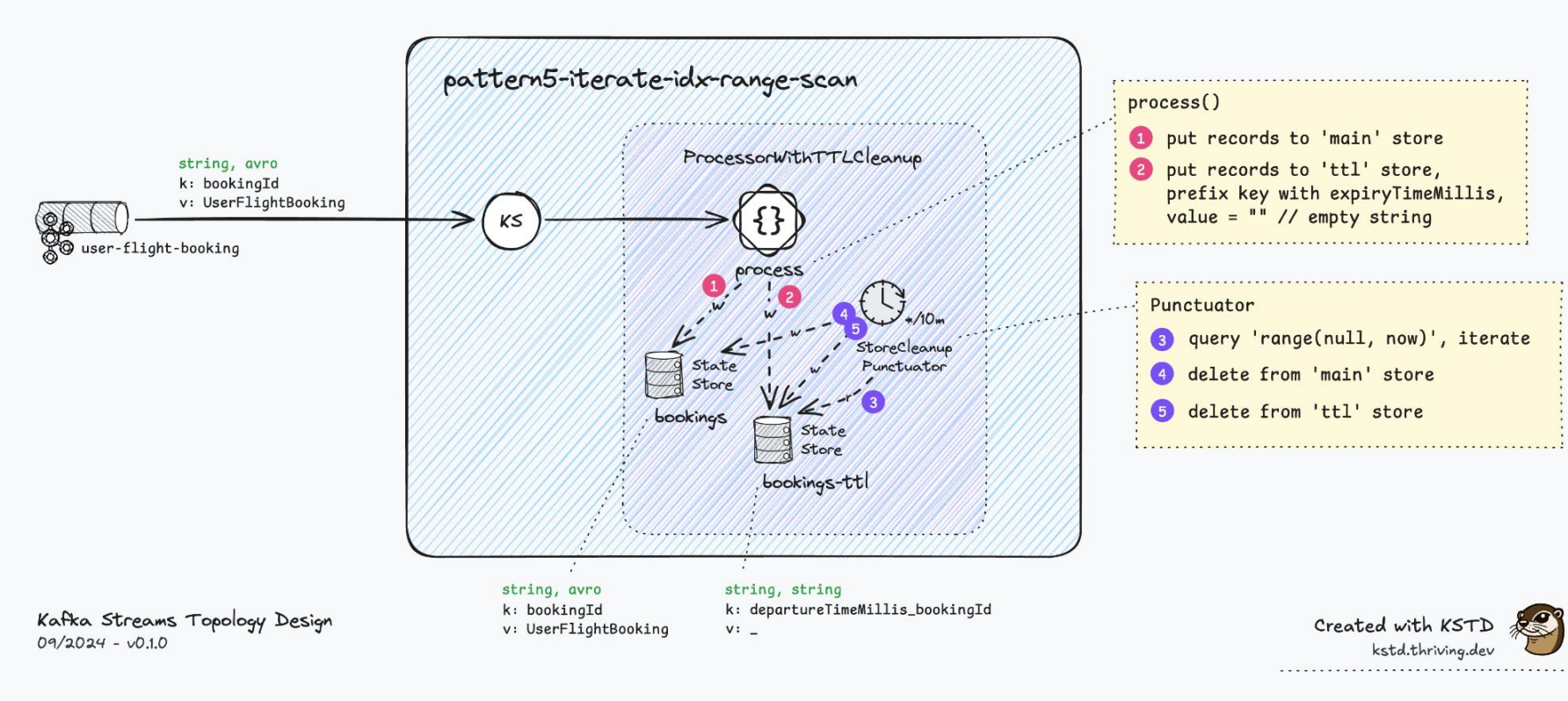
```
props.put(ROCKSDB_CONFIG_SETTER_CLASS_CONFIG, "dev.thriving.poc.RocksConfigSetter");
```

```
public class RocksConfigSetter implements RocksDBConfigSetter {

    @Override
    public void setConfig(final String storeName, final Options options, final Map<String, Object> configs) {
        BlockBasedTableConfigWithAccessibleCache tableConfig =
                (BlockBasedTableConfigWithAccessibleCache) options.tableFormatConfig();
        options.setTableFormatConfig(tableConfig);

        if (storeName.equals(KStreamsTopologyFactory.STORE_NAME)) {
            // TTL in seconds
            options.setTtl(86400); // 24h
        }
    }

    @Override
    public void close(final String storeName, final Options options) {}
}
```

Dealing with Large(!) State

- punctuation pauses the stream (task)
  -> latency spikes
- iterating store entries takes time
  -> grows linearly with no. of entries
- entries (kv) are deserialized
  -> cpu, memory, GC

Pattern 5: Processor, punctuate, idx store, range(), delete



Pattern 5: Processor, punctuate, idx store, range(), delete

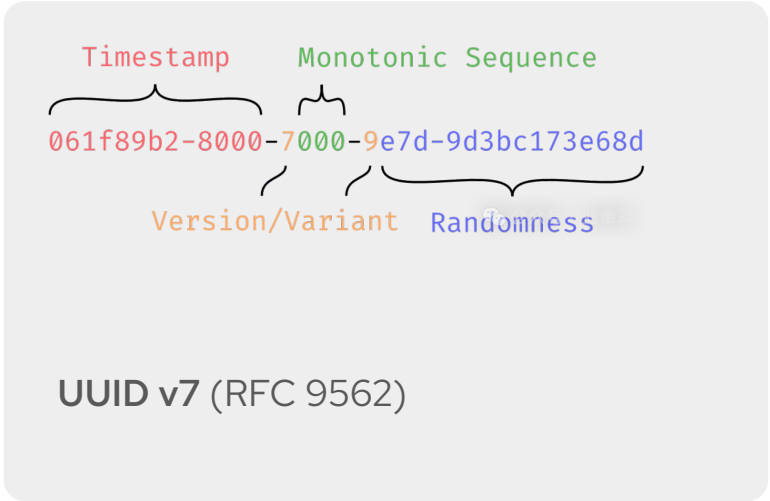
- Use separate 'ttl-store' to *index* keys for eviction
- Prefix keys with timestamp
  (epoch timestamp, or ISO 8601)
- Also works for time-based keys (ULID, UUID v7)
- Challenging if time-to-delete can change over time
  (deleting the previous *entry*)

Pattern 5: Processor, punctuate, idx store, range(), delete

```java
@Override
public void init(ProcessorContext<String, UserFlightBooking> context) {
    mainStore = context.getStateStore(KStreamsTopologyFactory.STATE_STORE);
    deleteAtStore = context.getStateStore(KStreamsTopologyFactory.STATE_STORE_DELETION_IDX);

    context.schedule(Duration.ofSeconds(10), PunctuationType.WALL_CLOCK_TIME, timestamp -> {
        ArrayList<String> keysToRemove = new ArrayList<>();
        String rangeTo = Long.toString(timestamp);
        try (KeyValueIterator<String, String> iter = deleteAtStore.range(null, rangeTo)) {
            iter.forEachRemaining(kv -> keysToRemove.add(kv.key));
        }
        keysToRemove.forEach(timestampedKey -> {
            String bookingId = timestampedKey.substring(timestampedKey.indexOf(DELIMITER) + 1);
            mainStore.delete(bookingId);
            deleteAtStore.delete(timestampedKey);
        });
    });
}
```

Pattern 6: Example UUID v7 keys, iterate, lazy-value-deserializer

- Time-based IDs. Iterate state stores; evict based on key
- ☝ Value is still read into memory as byte[]



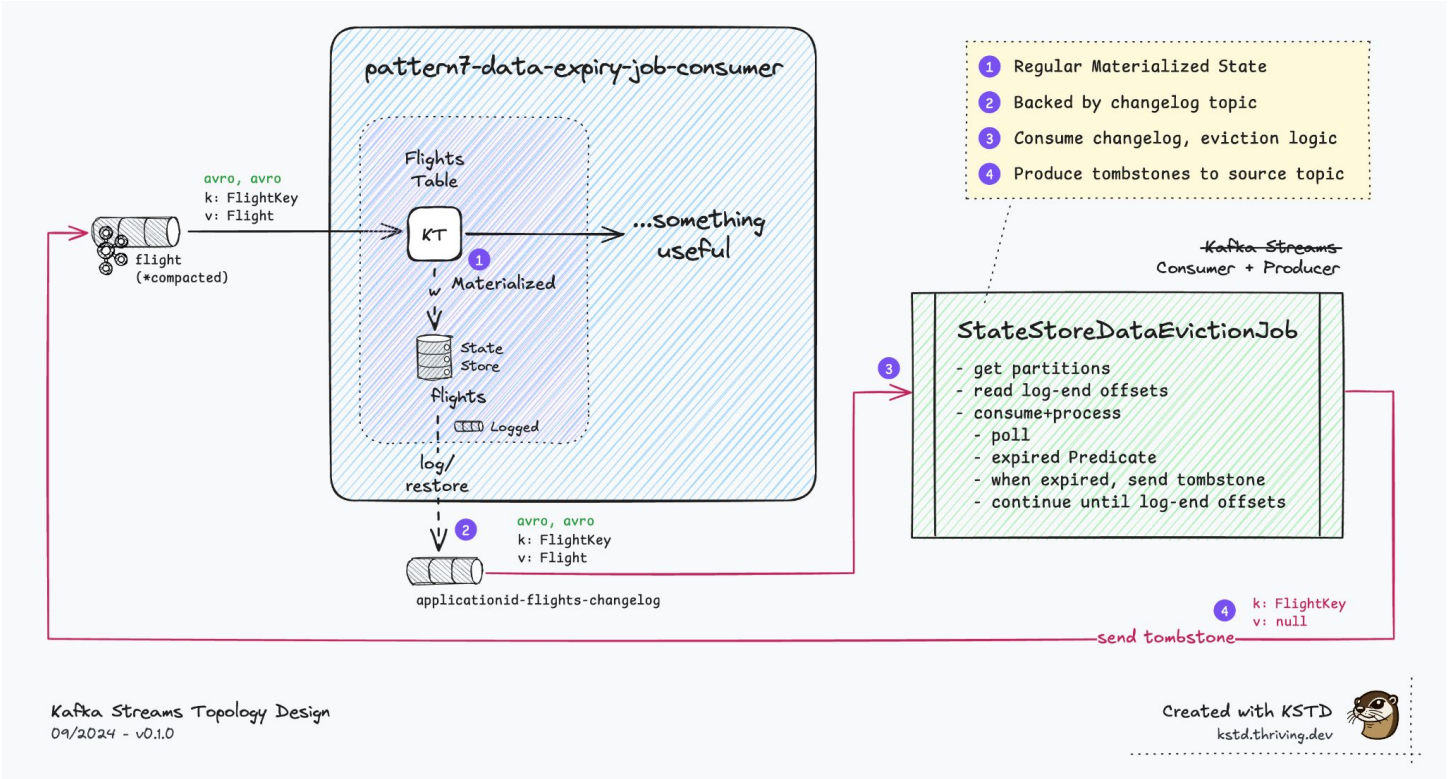
[0]: https://datatracker.ietf.org/doc/rfc9562/
[1]: https://itnext.io/why-uuid7-is-better-than-uuid4-as-clustered-index-edb02bf70056

LazySerde<T> (Value Deserializer)

```java
public class LazySerde<T> implements Serde<LazySerde.Lazy<T>> {

    private final Serde<T> innerSerde;

    public LazySerde(Serde<T> inner) {
        innerSerde = inner;
    }

    @Override
    public Serializer<Lazy<T>> serializer() {
        return (topic, data) -> innerSerde.serializer().serialize(topic, data.get());
    }

    @Override
    public Deserializer<Lazy<T>> deserializer() {
        return (topic, data) ->
                new Lazy<>(data, (t, d) -> innerSerde.deserializer().deserialize(t, d));
    }

    public static class Lazy<T> { ... }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
        innerSerde.configure(configs, isKey);
    }

    @Override
    public void close() { innerSerde.close(); }
}
```

```java
public static class Lazy<T> {
    private final byte[] data;
    private final Deserializer<T> deserializer;
    private Optional<T> cachedValue = Optional.empty();

    public Lazy(byte[] data, Deserializer<T> deserializer) {
        this.data = data;
        this.deserializer = deserializer;
    }

    public Lazy(T value) {
        this.cachedValue = Optional.of(value);
        data = null;
        deserializer = null;
    }

    public T get() {
        if (cachedValue.isEmpty()) {
            cachedValue = Optional.ofNullable(deserializer.deserialize(null, data));
        }
        return cachedValue.orElse(null);
    }

    public boolean isPresent() { return cachedValue.isPresent(); }
}
```

Pattern 7: Separate job, consume changelog, send tombstones



Pattern 7: Separate job, consume changelog, send tombstones (1)

```
1   public static void main(String[] args) {
2       try (KafkaConsumer<FlightKey, Flight> consumer = new KafkaConsumer<>(consumerProps());
3            KafkaProducer<FlightKey, Flight> producer = new KafkaProducer<>(producerProps())) {
4           // Get partitions for the topic
5   (1)     List<TopicPartition> partitions = new ArrayList<>( // we need a mutable list
6                   consumer.partitionsFor(CHANGELOG_TOPIC).stream()
7                           .map(info -> new TopicPartition(info.topic(), info.partition()))
8                           .toList()
9           );
10
11          // Assign partitions to the consumer
12          consumer.assign(partitions);
13
14          // Request end offsets
15  (2)     Map<TopicPartition, Long> endOffsets = consumer.endOffsets(partitions);
16
17          // Seek to the beginning of each partition
18          consumer.seekToBeginning(partitions);
19
20          boolean consuming = true;
21  (3)     while (consuming) { ... }
22
23          // Unsubscribe
24          consumer.unsubscribe();
25      } catch (Exception e) { e.printStackTrace(); }
26  }
```

1. get partitions
2. read log-end offsets
3. consume+process

(continue on next slide)

Pattern 7: Separate job, consume changelog, send tombstones (2)

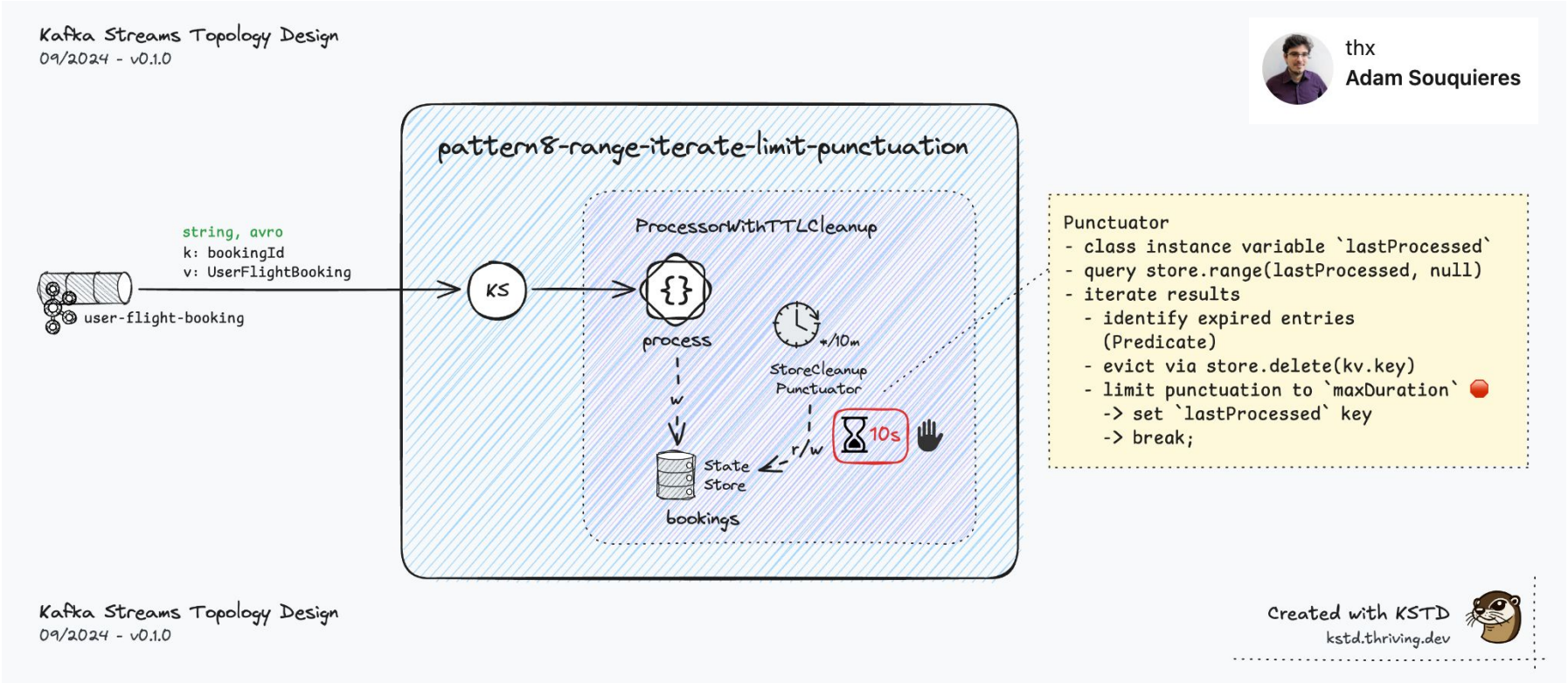
```
1  boolean consuming = true;
2  while (consuming) {
3      // Poll for records
4  1   ConsumerRecords<FlightKey, Flight> records = consumer.poll(ofMillis(100));
5
6      // Exit loop when topic contains no records
7      if (records.isEmpty()) { consuming = false; }
8
9      for (ConsumerRecord<FlightKey, Flight> record : records) {
10 2        if (record.value() != null && hasExpired(record)) {
11 3            producer.send(new ProducerRecord<>(ORIGINAL_TOPIC, record.key(), null));
12         }
13
14         // Check if we've reached the end offset for the partition
15         long endOffset = endOffsets.get(
16                 new TopicPartition(record.topic(), record.partition()));
17         if (record.offset() >= endOffset - 1) {
18             partitions.remove(new TopicPartition(record.topic(), record.partition()));
19         }
20     }
21
22 4   // Exit loop when all partitions are fully consumed
23     if (partitions.isEmpty()) { consuming = false; }
24 }
```

1. poll
2. record expired? (Predicate)
3. when expired, send tombstone
4. continue until reaching log-end offsets

Pattern 7: Separate job, consume changelog, send tombstones (2)

- Changelog topic is not an interface but an *'internal'* topics of the streams app
- Publishing a large number of tombstones over a short period of time might still cause latency spikes.
  - (Option: throttle publishing of tombstones)
- Caution: By default, changelog is not compacted instantly leading to processing of outdated records
- Separate job -> more moving parts to maintain, monitor and operate

Pattern 8: Punctuate, range(), time-box, stop & continue



Pattern 8: Punctuate, range(), time-box, stop & continue

```java
@Override
public void punctuate(long timestamp) {
    long startedAt = System.currentTimeMillis();
    store.flush(); // flush first, to allow deletes while iterating

    // range query and iterate store entries, continue 'from' where we left off (or NULL := beginning)
    try (KeyValueIterator<String, UserFlightBooking> iter = store.range(lastProcessedKey, null)) {
        while (iter.hasNext()) {
            KeyValue<String, UserFlightBooking> kv = iter.next();
            if (hasExpired(kv)) { store.delete(kv.key); }
            if (startedAt + MAX_PUNCTUATION_DURATION_MS < System.currentTimeMillis()) {
                lastProcessedKey = kv.key;
                break;
            }
        }
        if (!iter.hasNext()) { lastProcessedKey = null; } // reached the end, reset
    }
}
```

Summary (take with a pinch of salt)

	Usability	Large State	Risks
Pattern 1: Processor, punctuate, iterate all(), delete	🙂	😐	🙂
Pattern 2: KTable, punctuate, iterate all(), send tombstones	🙂	😐	🙂
Pattern 3: LRUCache, changelog retention=[compact,delete]	😐	☹️	😐
Pattern 4: RocksDB TTL, changelog retention=[compact,delete]	☹️	🙂	😐
Pattern 5: Processor, punctuate, idx store, range(), delete	😐	😐	🙂
Pattern 6: Example UUID v7 keys, iterate, lazy-value-deserializer	☹️	🙂	🙂
Pattern 7: Separate job, consume changelog, send tombstones	😐	🙂	😐
Pattern 8: Punctuate, range(), time-box, stop & continue	😐	🙂	🙂

Questions?

@hartmut-co-uk
@hartmut-co-uk
@TheThrivingDev
@thriving_dev

https://thriving.dev
https://kstd.thriving.dev

https://github.com/thriving-dev/kafka-streams-state-ttl-patterns