Multitype Messages in the Same Kafka Topic

Options, Challenges, and Techniques, Apr 2021

Agenda

Prelude

- Kafka, Schema Registry, Avro, Java
- We compile Avro schemas into Java classes - `SpecificRecord`
- One-topic-one-type
    - just like in database, one-table-one-entity

👉 In this slide, we use `SpecificRecord` to refer to those generated model classes

Why Multitype in the Same Topic

- Business nature
  - Events are naturally multityped within business domains
  - Events evolve as business evolve
- **Ordering !!!**
  - Ordering is CRITICAL to event based processing
    - have-your-cake ⇾ eat-it ⇾ 🤷‍♂️
    - eat-it ⇾ have-your-cake ⇾ 🎉
  - Kafka reserves the order of events ONLY if they are in the same partition
    - Being in the same topic is a prerequisite for being in the same partition

👉 There are other prerequisites for guaranteeing ordering in Kafka

Options

- JSON
- Tuple of KVP
- Map
- Union
- Subject Naming Strategy

Schemaless

$\Downarrow$

Semi-schema

$\Downarrow$

Schemaful

JSON

- 🏌 Zero Schema
- 👍 Super flexible
- 👎 Type is implicit
- 🤞 Good luck on producers & consumers
- This is effectively similar to using Compatibility Type `NONE` with Avro

👉 This is not saying one can't enforce schema with JSON

Tuple & Map

- 🏌️ One Schema for All
- 👍 Unique keys guarantee (Map)
- 👎 Potentially non-unique keys (Tuple)
- 🤹 Challenge supporting various value types
- 🏋️ Tricky on nested structure (especially for Tuple)
- 👎 Type is implicit
- 🤞 Responsibilities on producers & consumers

👉 GraphQL does NOT support Map

Union

- In computer science, a union is a value that may have any of several representations or formats within the same position in memory. — Wikipedia
- In Avro, a union indicates that a field might have more than one data type. E.g.

```
{
   "type" : "record",
   "name" : "Candidate",
   "fields" :
   [
      {
         "name" : "experience",
         "type": ["null", "int"]
      }
   ]
}
```

Union for Multitype

In below schema, `payload` holds either a `Sms` or `Email` type

```
{
  "type": "record",
  "name": "Event",
  "namespace": "demo.model",
  "fields": [
    {
      "name" : "eventId", "type" : "string"
    },{
      "name" : "payload",
      "type" : [
        {
          "type" : "record",
          "name" : "Sms",
          "fields" : [
            {
              "name" : "phoneNumber", "type" : "string"
            },{
              "name" : "text", "type" : "string"
            }
          ]
        },{
          "type" : "record",
          "name" : "Email",
          "fields" : [
            {
              "name" : "address", "type" : "string"
            },{
              "name" : "title", "type" : "string"
            }
          ]
        }
      ]
    }
  ]
}
```

Subject Naming Strategy

- What is **Subject**?
  - 🅠 : Can I use schema-registry without Kafka broker independently? (Schema Registry Issue #533)
  - 🅐 : Definitely. That's why we use slightly different terminology in the schema registry ("subjects") than we use in Kafka ("topics").
- Often people want one-topic-one-type, meaning one-topic-one-subject
  - Hence the default strategy is: `TopicNameStrategy`
- Two other strategies
  - `RecordNameStrategy`
  - `TopicRecordNameStrategy`
  - 👍 Types evolve independently

Naming Strategies in Action

- Producer publishes customer activities to topic `activity`
  - `TopicNameStrategy` (default)
    - `activity-value`, only one type of message allowed
  - `RecordNameStrategy`
    - `demo.model.MonetaryActivity`
    - `demo.model.NonMonetaryActivity`
    - …
  - `TopicRecordNameStrategy`
    - `activity-demo.model.MonetaryActivity`
    - `activity-demo.model.NonMonetaryActivity`
    - …

👉 Control Center UI supports `TopicNameStrategy` only

Naming Strategy in Code

- Producer

```
props.put("value.subject.name.strategy", TopicRecordNameStrategy.class);
KafkaProducer producer = new KafkaProducer(props);

// Subject: activity-demo.model.MonetaryActivity
producer.send(new ProducerRecord("activity", "key1", new MonetaryActivity()));

// Subject: activity-demo.model.NonMonetaryActivity  (same topic as above)
producer.send(new ProducerRecord("activity", "key2", new NonMonetaryActivity()));
```

- Consumer

```
ConsumerRecords records = consumer.poll(ONE_SECOND);
for (ConsumerRecord record : records) {
    Object activity = record.value();
    if (activity instanceof MonetaryActivity) { ... }
    if (activity instanceof NonMonetaryActivity) { ... }
}
```

👉 Generics omitted for brevity

Challenges in Union and Subject Naming Strategy

- ❌ Message Filtering
  - Similar to `SELECT...WHERE...` in `SQL`, and Message Selector in JMS
  - There is NO BUILT-IN way to filter out unwanted messages
- Add a new Type
  - Breaks existing consumers if
    - Using `SpecificRecord`
    - AND filtering not implemented
- Deprecate a Type
  - Breaks new consumers for the same reason
- 😕 Evolve types inside Union

Message Filtering

- Because Kafka brokers are 'DUMB'
- and Kafka requires that a consumer to take ALL messages it receives
    - Consumer's Life:

      ```
      1. fetch messages in bytes
      2. deserialize into objects
      3. process them
      4. commit offset
      5. goto step 1
      ```

    - 😐 Consumer cannot say no to unwanted bytes
- 👎 Consequently, Introducing new types BREAKS existing consumers using `SpecificRecord`
    - Unless the consumer upgrades to newer schema or implements custom filtering

👉 This applies to both the Union and Subject Naming Strategy approaches

Techniques - GenericRecord

- Consumer use `GenericRecord` rather than `SpecificRecord`
- This is quite similar to Map approach
- 👍 Deserialisation always SUCCEEDS

👉 One can have producers publishing messages of new type using `SpecificRecord` while not breaking existing consumers from consuming them in `GenericRecord`

Kafka Headers

- **Header** exists in many places, TCP, HTTP, HTML, JMS, SOAP, etc.
- Headers are **Metadata**, serve purposes like routing, authentication, etc.
- Header support added to Kafka in version 0.11 in 2017

```
public interface Header {
  String key();
  byte[] value();
}
```

- Available to Producer and Consumer

```
record.headers().add("meta1", "important-data".getBytes(UTF_8)); // Producer

for (Header header : record.headers()) { ... } // Consumer
```

- Header support added to `Deserialiser` interface in 2018

```
public interface Deserializer<T> {
  void configure(Map<String, ?> configs, boolean isKey);
  T deserialize(String topic, byte[] data);
  T deserialize(String topic, Headers headers, byte[] data);
}
```

Techniques - Swallow But Not Digest with Header Filtering

- 🤝 Filter out unwanted messages
  - Producer tags a message's type via header

    ```
    record.headers().add("Type", "MonetaryActivity".getBytes());
    producer.send(record);
    ```

  - Deserializer converts them into `null` as per tag value

    ```
    KafkaAvroDeserializer worker;

    public Object deserialize(String topic, Headers headers, byte[] data) {
        if (!isKnown(headers)) return null;

        return worker.deserialize(topic, headers, data);
    }
    ```

  - Consumer ignores `null` messages and continue

    ```
    props.put("value.deserializer", TolerantDeserializer.class);
    props.put("tolerant.headerName", "Type");
    props.put("tolerant.headerValueRegex", "MonetaryActivity|NonMonetaryActivity");

    for (ConsumerRecord<String, SpecificRecordBase> record : records) {
        if (Objects.isNull(record.value())) continue;
    }
    ```

👉 This technique can be applied to both Subject Naming Strategy and Union approaches

Safely Add New Types with Defensive Consumers

Producers	**Topic**	**Consumers**
ProducerA: E1		ConsumerA: E1 ☔
ProducerB: E2	=> multi-events-topic =>	ConsumerB: E1 | E2 ☔
ProducerC: E-new		ConsumerC: E1 | E-new

Conclusion

- It is Complicated!
- It Depends.