

Register Now for Current 2022: The Next Generation of Kafka Summit, let's explore the future of data streaming!



[APACHE KAFKA](#)

Should You Put Several Event Types in the Same Kafka Topic?



MARTIN KLEPPMANN

JAN 18, 2018

If you adopt a streaming platform such as [Apache Kafka](#), one of the most important questions to answer is: *what topics are you going to use?* In particular, if you have a bunch of different events that you want to publish to Kafka as messages, do you put them in the same topic, or do you split them across different topics?

The most important function of a topic is to allow a consumer to specify which subset of messages it wants to consume. At the one extreme, putting absolutely all your data in a single topic is probably a bad idea, since

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookie Notice](#)

Cookies Settings

Reject All

Accept All Cookies

[Register Now for Current 2022: The Next Generation of Kafka Summit](#), let's explore the future of data streaming!

That performance argument provides some guidance for designing your topic structure: if you're finding yourself with many thousands of topics, it would be advisable to merge some of the fine-grained, low-throughput topics into coarser-grained topics, and thus reduce the proliferation of partitions.

However, performance is not the end of the story. Even more important, in my opinion, are the data integrity and data modelling aspects of your topic structure. We will discuss those in the rest of this article.

Topic = collection of events of the same type?

The common wisdom (according to several conversations I've had, and according to a [mailing list thread](#)) seems to be: put all events of the same type in the same topic, and use different topics for different event types. That line of thinking is reminiscent of relational databases, where a table is a collection of records with the same type (i.e. the same set of columns), so we have an analogy between a relational table and a Kafka topic.

The [Confluent Schema Registry](#) has traditionally reinforced this pattern, because it encourages you to use the same Avro schema for all messages in a topic. That schema can be evolved while maintaining compatibility (e.g. by adding optional fields), but ultimately all messages have been expected to conform to a certain record type. We'll revisit this later in the post, after we've covered some more background.

For some types of streaming data, such as logged activity events, it makes sense to require that all messages in the same topic conform to the same schema. However, some people are using Kafka for more database-like purposes, such as [event sourcing](#), or [exchanging data between microservices](#). In this context, I believe it's less important to define a topic as a grouping of messages with the same schema. Much more important is the fact that Kafka maintains **ordering** of messages within a topic-partition.

Imagine a scenario in which you have some entity (say, a customer), and many different things can happen to that entity: a customer is created, a customer changes their address, a customer adds a new credit card to

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookie Notice](#)

Cookies Settings

Reject All

Accept All Cookies

[Register Now for Current 2022: The Next Generation of Kafka Summit](#), let's explore the future of data streaming!

If you did use different topics for (say) the `customerCreated`, `customerAddressChanged`, and `customerInvoicePaid` events, then a consumer of those topics may see the events in a nonsensical order. For example, the consumer may see an address change for a customer that does not exist (because it has not yet been created, since the corresponding `customerCreated` event has been delayed).

The risk of reordering is particularly high if a consumer is shut down for a while, perhaps for maintenance or to deploy a new version. While the consumer is stopped, events continue to be published, and those events are stored in the selected topic-partition on the Kafka brokers. When the consumer starts up again, it consumes the backlog of events from all of its input partitions. If the consumer has only one input, that's no problem: the pending events are simply processed sequentially in the order they are stored. But if the consumer has several input topics, it will pick input topics to read in some arbitrary order. It may read all of the pending events from one input topic before it reads the backlog on another input topic, or it may interleave the inputs in some way.

Thus, if you put the `customerCreated`, `customerAddressChanged`, and `customerInvoicePaid` events in three separate topics, the consumer may well see all of the `customerAddressChanged` events before it sees any of the `customerCreated` events. And so it is likely that the consumer will see a `customerAddressChanged` event for a customer that, according to its view of the world, has not yet been created.

You might be tempted to attach a timestamp to each message and use that for event ordering. That might just about work if you are importing events into a data warehouse, where you can order the events after the fact. But in a stream process, timestamps are not enough: if you get an event with a certain timestamp, you don't know whether you still need to wait for some previous event with a lower timestamp, or if all previous events have arrived and you're ready to process the event. And relying on clock synchronisation generally leads to nightmares; for more detail on the problems with clocks, I refer you to Chapter 8 of [my book](#).

When to split topics, when to combine?

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookie Notice](#)

Cookies Settings

Reject All

Accept All Cookies

Register Now for Current 2022: The Next Generation of Kafka Summit, let's explore the future of data streaming!

2. When you have events about different entities, should they go in the same topic or different topics? I would say that if one entity depends on another (e.g. an address belongs to a customer), or if they are often needed together, they might as well go in the same topic. On the other hand, if they are unrelated and managed by different teams, they are better put in separate topics. It also depends on the throughput of events: if one entity type has a much higher rate of events than another entity type, they are better split into separate topics, to avoid overwhelming consumers who only want the entity with low write throughput (see point four). But several entities that all have a low rate of events can easily be merged.
3. What if an event involves several entities? For example, a purchase relates a product and a customer, and a transfer from one account to another involves at least those two accounts. I would recommend initially recording the event as a single atomic message, and not splitting it up into several messages in several topics. It's best to record events exactly as you receive them, in a form that is **as raw as possible**. You can always split up the compound event later, using a stream processor—but it's much harder to reconstruct the original event if you split it up prematurely. Even better, you can give the initial event a unique ID (e.g. a UUID); that way later on when you split the original event into one event for each entity involved, you can carry that ID forward, making the provenance of each event traceable.
4. Look at the number of topics that a consumer needs to subscribe to. If several consumers all read a particular group of topics, this suggests that maybe those topics **should be combined**. If you combine the fine-grained topics into coarser-grained ones, some consumers may receive unwanted events that they need to ignore. That is not a big deal: consuming messages from Kafka is very cheap, so even if a consumer ends up ignoring half of the events, the cost of this overconsumption is probably not significant. Only if the consumer needs to ignore the vast majority of messages (e.g. 99.9% are unwanted) would I recommend splitting the low-volume event stream from the high-volume stream.
5. A changelog topic for a **Kafka Streams state store** (KTable) should be a separate from all other topics. In this case, the topic is managed by Kafka Streams process, and it should not be shared with anything else.
6. Finally, what if none of the rules above tell you whether to put some events in the same topic or in different topics? Then by all means group them by event type, by putting events of the same type in the same topic. However, I think this rule is the least important of all.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookie Notice](#)

Cookies Settings

Reject All

Accept All Cookies

[Register Now for Current 2022: The Next Generation of Kafka Summit](#), let's explore the future of data streaming

and consumers using different schema versions at the same time, and they still remain compatible with each other.

More precisely, when Confluent's Avro serializer registers a schema in the registry, it does so under a *subject name*. By default, that subject is `<topic>-key` for message keys and `<topic>-value` for message values. The schema registry then checks the mutual compatibility of all schemas that are registered under a particular subject.

I have recently [made a patch to the Avro serializer](#) that makes the compatibility check more flexible. The patch adds two new configuration options: `key.subject.name.strategy` (which defines how to construct the subject name for message keys), and `value.subject.name.strategy` (how to construct the subject name for message values). The options can take one of the following values:

- `io.confluent.kafka.serializers.subject.TopicNameStrategy` (default): The subject name for message keys is `<topic>-key`, and `<topic>-value` for message values. This means that the schemas of all messages in the topic must be compatible with each other.
- `io.confluent.kafka.serializers.subject.RecordNameStrategy`: The subject name is the fully-qualified name of the Avro record type of the message. Thus, the schema registry checks the compatibility for a particular record type, regardless of topic. This setting allows any number of different event types in the same topic.
- `io.confluent.kafka.serializers.subject.TopicRecordNameStrategy`: The subject name is `<topic>-<type>`, where `<topic>` is the Kafka topic name, and `<type>` is the fully-qualified name of the Avro record type of the message. This setting also allows any number of event types in the same topic, and further constrains the compatibility check to the current topic only.

With this new feature, you can easily and cleanly put all the different events for a particular entity in the same topic. Now you can freely choose the granularity of topics based on the criteria above, and not be limited to a single event type per topic.

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookie Notice](#)

Cookies Settings

Reject All

Accept All Cookies

Register Now for Current 2022: The Next Generation of Kafka Summit, let's explore the future of data streaming

By clicking "Accept All Cookies", you agree to the storing of cookies on your device to enhance site navigation, analyze site usage, and assist in our marketing efforts.

[Cookie Notice](#)

Cookies Settings

Reject All

Accept All Cookies