Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 5. Data Value Design Patterns

It may be an unpopular opinion, but data sitting somewhere in your storage is not a real asset. Most of the time, after it's ingested into your system, it'll be poor and have various quality issues. Let's take an example of the visit events ingested to the streaming broker from our use case architecture.

The data producer for the streaming layer is a web browser, which means it can get any valuable technical information about the browser version, language, or operating system of the user. That would be enough if you wanted to analyze the technical part of each visit in your system. But what if you need to know more, like what the visitors using a specific browser have in common? Each visit event is ingested as a distinct item without any explicit relationship, so correlating the data is impossible without extra effort.

This is a typical scenario where data value design patterns are helpful. Their purpose is to augment the dataset to improve its usefulness for end users. How? There are different solutions that you're going to learn about in this chapter.

You'll see how to add extra value by either combining two datasets or computing the individual attributes with the Data Enrichment and Data Decoration patterns, which are both covered in the next sections. That said, they're great for extending the context, but they won't help if you have a huge volume of data and need an overview, as in the example quoted previously. That's why next, you'll see the Data Aggregation or Sessionization patterns. To complete the picture, you'll also learn about the data ordering patterns you'll use in situations where the order of the records matters most.

Are you curious about how to solve our web browser issue and many others? Let's see the data value design patterns in action!

Data Enrichment

Very often, raw data will be poor because of technical constraints. Events are the best examples here. They're a perfect representation of time and space attributes, but frequently, they lack an extended context. Sure, they can identify the author of the event, but they'll rarely be capable of providing extra information, such as the last connection date or a user profile score in the context of our visits example. Data enrichment patterns overcome this limitation and make data more useful for different stakeholders.

Pattern: Static Joiner

If your enrichment dataset has a static nature, which is by far the easiest situation, you'll opt for the Static Joiner pattern presented here.

Problem

The datasets developed by your team are extensively used by business stakeholders. In a new project, you've been asked to create a dataset to simplify understanding of the dependency between the registration date of a user and day-to-day activity.

Unfortunately, your raw dataset doesn't include the user context. You can find it only in a static user reference dataset. To answer the business demand, you want to find a way to bring the reference data to the user's activity.

Solution

The at-rest character of the joined dataset presents the perfect condition for using the Static Joiner pattern. Surprisingly, despite the at-rest nature of this data, the pattern also works for streaming pipelines.

The implementation requires a list of attributes from both datasets that may be used to combine the datasets. In our example, it could be any field identifying a user in the visits and users datasets, for example, the user_id field.

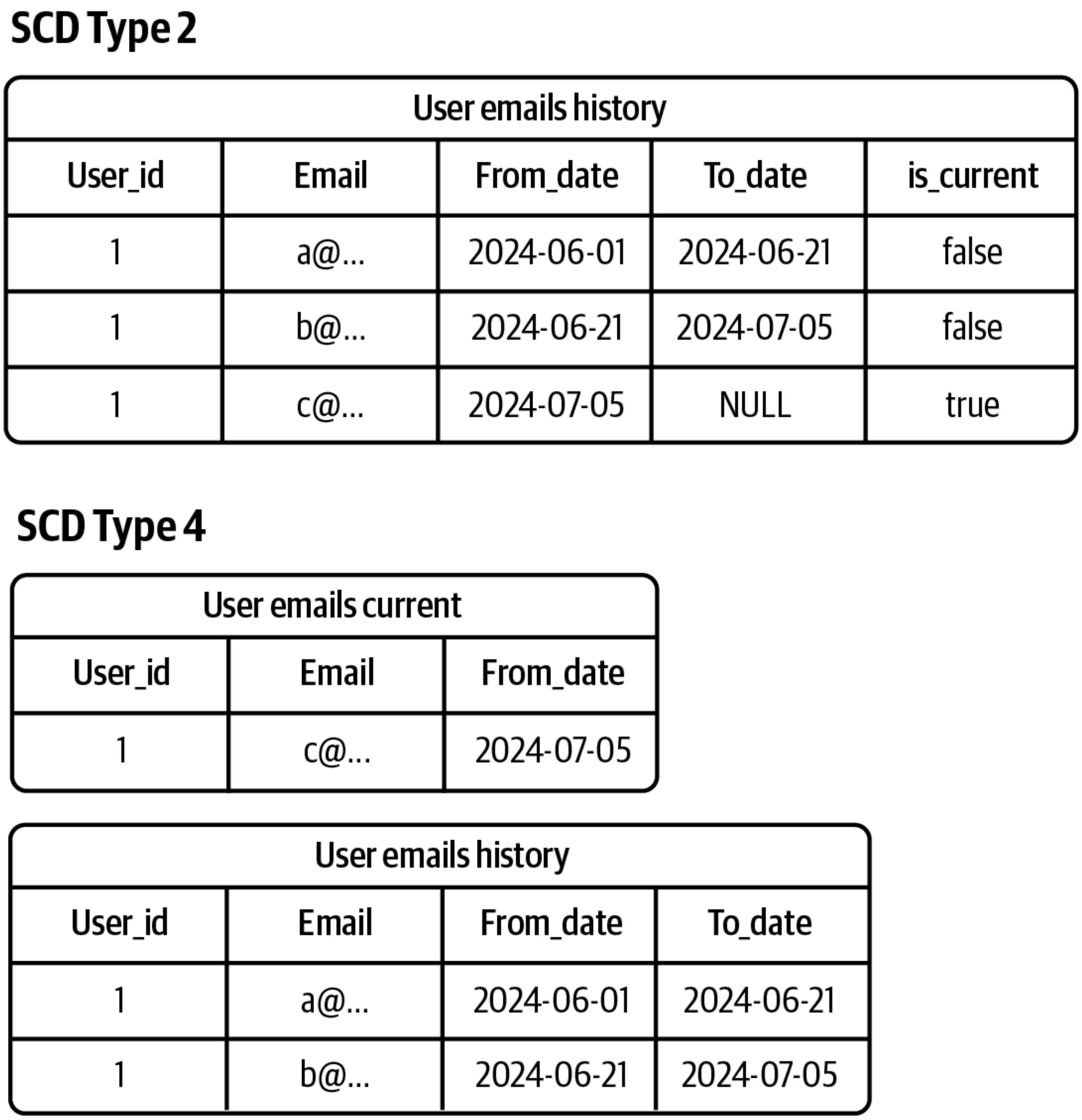
Besides this keyed condition, the combination may also expect some time constraints, especially when the enrichment dataset implements some form of slowly changing dimensions. In that case, you could implement a time-sensitive static joiner variation of the initial pattern.

Slowly Changing Dimensions

If you need to use a time-sensitive data enrichment, you can implement it as one form of *slowly changing dimensions* (SCD), which is a data modeling strategy for slowly changing datasets that may support an entity's evolution over time.

In our solution, the enrichment dataset should implement SCD type 2 or 4. These types track row evolution over time, and the only difference between them is the technical implementation.

SCD type 2 manages tracking with validity dates, and each current value has the end date empty. SCD type 4 relies on two tables. The first table stores the current value for each entity, while the second table stores all historical values, including the current one. Figure 5-1 shows how both SCD types can be used to record the history of user email changes.



SCD Type 2

User emails history

User_id	Email	From_date	To_date	is_current
1	a@...	2024-06-01	2024-06-21	false
1	b@...	2024-06-21	2024-07-05	false
1	c@...	2024-07-05	NULL	true

SCD Type 4

User emails current

User_id	Email	From_date
1	c@...	2024-07-05

User emails history

User_id	Email	From_date	To_date
1	a@...	2024-06-01	2024-06-21
1	b@...	2024-06-21	2024-07-05

Figure 5-1. SCD types 2 and 4: user emails

We'll dig into a technical explanation of these SCD types in the Examples section later in this chapter.

When it comes to code implementation, the enrichment is often expressed with a SQL JOIN statement. The solution is universal as it supports modern data processing frameworks and more classical data warehouses.

Besides this declarative manner with SQL, you can enrich your dataset from a programmatic API. Here, the easiest way is to use an HTTP library to enable communication between your data processing layer and an external API.

But the fact of exposing some data behind an API doesn't mean the dataset can't be accessed from a data storage layer such as a table. If you need to, you can also materialize this API-exposed dataset as a table in a pre-processing step and use it as part of the JOIN statement. Figure 5-2 depicts both approaches.

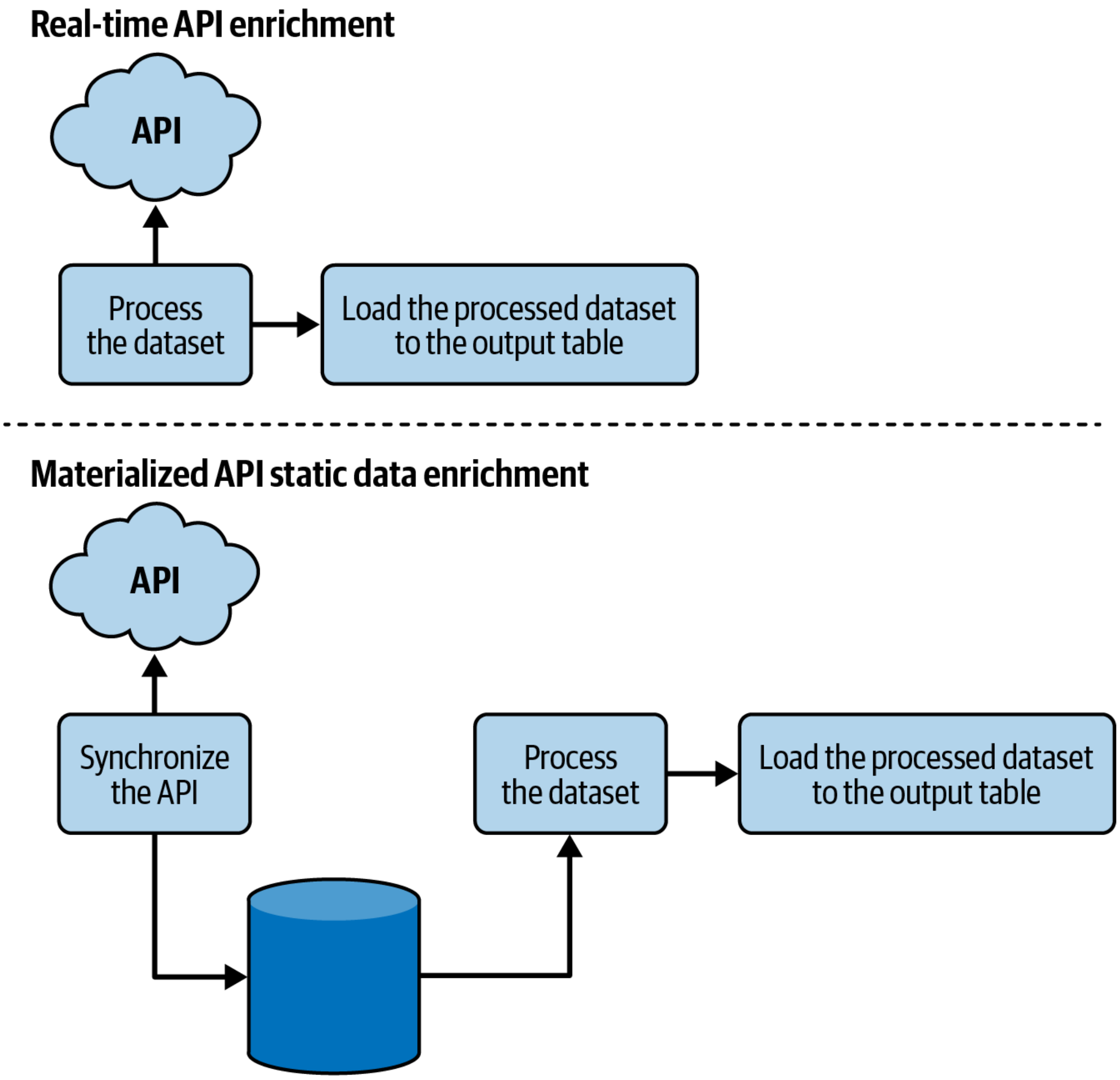

Real-time API enrichment
API
Process the dataset
Load the processed dataset to the output table
Materialized API static data enrichment
API
Synchronize the API
Process the dataset
Load the processed dataset to the output table

Figure 5-2. Static data enrichment with real-time and synchronized API calls

When you care about idempotency, you should opt for the solution materializing the API and leverage one of the SCD forms. That guarantees that whenever you replay the processing logic, you will always use the same dataset. By the way, this is one of the topics we develop more in the next section.

Consequences

Even though the implementation looks simple, there are some gotchas, like the aforementioned idempotency.

Late data and consistency

To help you understand this point, let's take a look at the example of visits and users datasets. In an ideal scenario, users would evolve at the same pace as events are produced (i.e., if a user performs a profile change and immediately thereafter performs a navigation action, the processed visit should include the most recent changes). Unfortunately, as you learned in Chapter 3 and in "Late Data", this scenario may not happen.

To mitigate the latency issue in streaming pipelines, you can use the Dynamic Joiner pattern presented next. It considers the enrichment dataset to be a dynamic one and uses adapted join conditions in that context. The mitigation is simpler for batch pipelines, where you can rely on the orchestration to wait for the enrichment dataset to be present (for example, by leveraging the Readiness Marker pattern discussed in Chapter 2).

Idempotency

Besides consistency, you should also consider idempotency. If you backfill a batch pipeline, you should ask yourself whether the outcome must be idempotent for the enrichment dataset. If that's the case and the data provider doesn't let you perform any time-based queries, you may need to bring the enrichment dataset into your data layer to control the time aspects before doing the join.

The situation is even trickier when it comes to the external datasets hidden behind an API. Here too, ideally, you should be able to issue time-based queries, but this may not be possible. The solution could be adding this temporality into your internal data store and writing all enrichment records there, as you saw in Figure 5-2.

For both cases, using SCD should be a good option.

Examples

Let's start this section by focusing on the SCDs, as they require more effort than simple join conditions. The first SCD we'll analyze is type 2. We'll use the two tables in Example 5-1.

Example 5-1. Two tables demonstrating SCD type 2

```
CREATE TABLE dedp.users ( # ...
 id TEXT NOT NULL,
 login VARCHAR(45) NOT NULL,
 start_date TIMESTAMP NOT NULL DEFAULT NOW(),
 end_date TIMESTAMP NOT NULL DEFAULT '9999-12-31'::timestamp
 PRIMARY KEY(id, start_date)
);
CREATE TABLE dedp.visits ( # ...
 visit_id CHAR(36) NOT NULL,
 event_time TIMESTAMP NOT NULL,
 PRIMARY KEY(visit_id, event_time)
);
```

The users table in Example 5-1 is a slowly evolving reference dataset that enriches a more dynamic website visits dataset. The start_date and end_date columns define the validity of the attributes for each user. Therefore, to combine the visits and users, you need to use these dates as shown in Example 5-2.

Example 5-2. Example of SCD type 2 join

```
SELECT v.visit_id, v.event_time, v.page, u.id, u.login, u.email
```

```
FROM dedp.visits v JOIN dedp.users u ON u.id = v.user_id
  AND NOW() BETWEEN start_date AND end_date;
```

The condition in Example 5-2 uses the NOW() function as part of an ad hoc query to get the current state of the dataset. You can use any other date here that fits your needs. For example, it could be the execution time provided by your data orchestrator, which is an immutable property related to the pipeline run that helps enforce idempotency.

We're omitting SCD type 4 here as it relies on the same query as type 2. The only difference is that type 4 stores current values in a separate table while in type 2, both current and past rows are present in the same dataset. You will find this example in the GitHub repo.

To complete this list of examples, let's see how to combine batch and streaming datasets in Apache Spark. Technically, the operation is not rocket science as it relies on the same API as for regular batch joins (see Example 5-3).

Example 5-3. Stream-to-batch join in PySpark

```
devices: DataFrame = spark.read.format('delta').load(...)
visits: DataFrame = (spark.readStream.format('kafka').load()...

(visits.join(devices_table, [visits.device_type == devices.type,
  visits.device_version == devices.version], 'left_outer'))
```

The operation in Example 5-3 combines the static devices reference dataset with the visits. It doesn't include any temporal condition because the devices table is an insert-only table, and not having the matched records is fine (the left join is used). But this relaxed condition hides a tricky point. The static dataset and the streaming job have separate lifecycles. Put differently, the streaming job doesn't wait for the static dataset to update. Therefore, this might lead to not only join misses but also, in the case of a full rewriting, to joining with an empty reference table.

This could happen if the static dataset were written with a raw file format such as JSON or CSV. In our case, the devices table relies on a table file format that provides atomicity and consistency guarantees. Unless the consumer decides to read uncommitted files, there is no risk of processing empty tables because of the concurrency issue.

Finally, you can also combine the raw dataset with a dataset exposed from an API. One of our recommendations here is to leverage bulk operations, where you can ask for information on multiple items at once. This optimizes network throughput, which often is the most expensive operation in data enrichment. In Example 5-4, you can find an example of an Apache Spark writer calling a locally managed IP mapping service.

Example 5-4. PySpark writer with data enrichment

```
class KafkaWriterWithEnricher:
 BUFFER_THRESHOLD = 100
# ...
```

```
def process(self, row):
   if len(self.buffered_to_enrich) == self.BUFFER_THRESHOLD:
     self._enrich_ips()
     self._flush_records()
   else:
     self.buffered_to_enrich.append(row)


def _enrich_ips(self):
  ips = (','.join(set(visit.ip for visit in self.buffered_to_enrich
     if visit.ip not in self.enriched_ips)))
  fetched_ips = requests.get(f'http://localhost:8080/geolocation/fetch?ips={ips}',
                 headers={'Content-Type': 'application/json','Charset': 'UTF-8'})
  if fetched_ips.status_code == 200:
    mapped_ips = json.loads(fetched_ips.content)['mapped']
    self.enriched_ips.update(mapped_ips)
```

As you can see in Example 5-4, the code buffers the records, and once the buffer size reaches the threshold, the writer makes the API call with a unique list of IPs.

Pattern: Dynamic Joiner

As you might have guessed, the Static Joiner pattern, due to its static character, isn't the best fit for combining two streaming datasets. The problem lies in the data perception. Streaming stands for a continuously moving dataset, with as-soon-as-possible processing, while static batch workloads operate on more slowly evolving data. But the good news is this section will show you an alternative which is better adapted to these dynamic environments.

Problem

Even though you've implemented the Static Joiner for the users-to-visits use case, you're still not satisfied with the final outcome. With thousands of new users coming online each week, the number of profile changes has increased. As a result, the enriched dataset is irrelevant and becomes problematic for your downstream consumers. Since each user change is registered to a streaming broker from the Change Data Capture pattern, you're looking for a better way to combine events for both sides.

Solution

As both datasets are in motion, you can't use the Static Joiner. Instead, you should consider its alternative, the Dynamic Joiner pattern, which is better suited for that kind of data.

Even though the implementation shares some points with the Static Joiner—namely, the identification of the keys and the definition of the join method—there is one extra requirement: *time boundaries*. Without this dedicated time management strategy, there's a risk that many of the joins will be empty. Why? It's simply because the two datasets may have different latencies. In other words, the enrichment dataset can be late compared with the enriched dataset or vice versa. To mitigate this issue, dynamic joins are often completed with additional time conditions.

Defining these time conditions implies having a time-bounded buffer for joined records on both streams. That way, the faster data source can align its time semantics with the slower data source. As a result, the buffer gives some extra time for joins to happen. This extra time is often an allowed latency difference between the data sources. For example, if the users from our example are one hour late compared with the visits, the buffer will store the visits for an extra hour, hoping to find a match once the user stream catches up.

Besides improving the join's outcome, the buffer involves a streaming aspect called the *garbage collection* (GC) *watermark*. Even though technically, you can always decide to keep events from both streams forever, this will require significant hardware resources and will fail sooner or later if you cannot scale your infrastructure indefinitely. A better approach is to define when events that are too old should go away from each buffer, meaning when you should use a GC watermark. This obviously means losing the join if one of the records comes really late, but that's the trade-off for having a manageable size buffer.

The component that's responsible for cleaning the buffer of the elements that are too old is the GC watermark. Figure 5-3 shows an example.

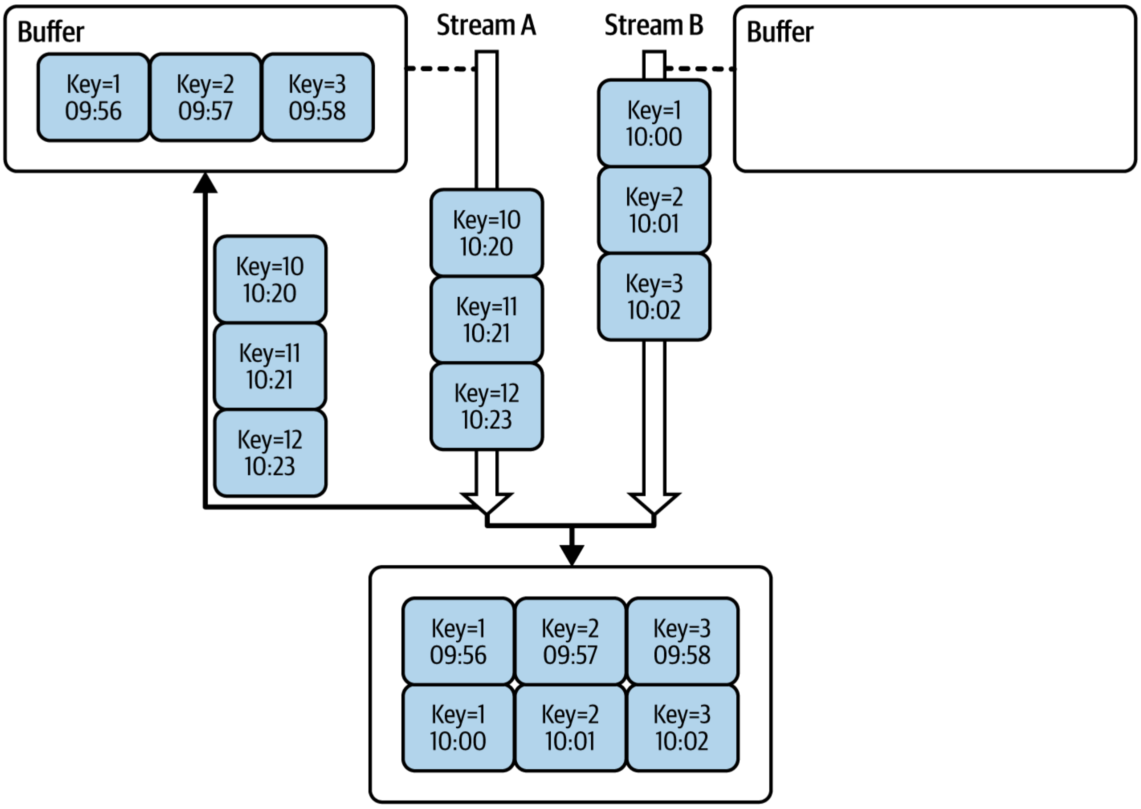


Figure 5-3. Dynamic Joiner with buffering for two streams of different latencies

Here, you can see two streams where Stream A is several minutes faster than Stream B. Due to this difference, to maximize the success rate of the joined records, Stream A buffers all unmatched keys for a period time. Then, when Stream B catches up, Stream A tries to find the corresponding rows either from the incoming data or directly from the buffer. If there is no match, the GC watermark removes records that are older than Stream B's oldest event time. (You learned about watermarks in the Late Data Detector pattern.)

Even though it sounds complicated, you won't need to deal with the entire buffering logic as it's natively implemented by data processing frameworks.

Consequences

Even though it addresses one of the Static Joiner's major issues, the Dynamic Joiner has its own drawbacks.

Space versus exactness trade-off

Due to the GC watermark and time boundaries, you may not be able to get all the joins that are possible. You can optimize efficency by increasing buffer space, but it'll cost you more hardware resources. On the other hand, reducing space optimizes storage but may reduce the likelihood of matching if the latency difference is too big.

For that reason, you will always need to balance these two factors, and unfortunately, there is no one-size-fits-all formula. Each solution will depend on your business requirements and the usual latency difference between joined datasets.

Late data

Late data is another reason for missed joins. Stream processing, due to its inherently lower latency processing semantics, has a weaker tolerance for late data integration in the pipelines. For example, our users stream could encounter temporary connectivity issues in some areas, leading to delayed delivery of a subset of events. As a result, the GC watermark will move on and invalidate the buffered state, and these late events will be ignored.

But neither of the two data enrichment patterns presented here will give you a 100% guarantee of the join results without any extra effort, due to this late data arrival issue. To overcome this limitation, you'll need to track and integrate late data, as explained in Chapter 3.

Examples

Unsurprisingly, to see this pattern in action, you'll need a streaming job. Let's begin with Apache Spark Structured Streaming. In Example 5-5, you can see a join condition used to combine visits with displayed ads.

Example 5-5. Time-based condition for Apache Spark Structured Streaming

```
visits_from_kafka: DataFrame = (visits_data_stream # ...
 .withWatermark('event_time', '10 minutes'))


ads_from_kafka: DataFrame = (ads_data_stream # ...
 .withWatermark('display_time', '10 minutes'))
```

```
visits_with_ads = visits_from_kafka.join(ads_from_kafka, F.expr('''
   page = visit_page AND
   display_time BETWEEN event_time AND event_time + INTERVAL 2 minutes
  '''), 'left_outer')
```

Currently, the join condition has business and technical meanings. It makes explicit the business rule that says an ad can be displayed at most two minutes after a user visits a page. From a technical standpoint, this also means there is room left for late data. The withWatermark expression allows late records to be up to 10 minutes late on both sides.

Even though this kind of join is also supported in Apache Flink, the framework also offers an alternative and more managed way to combine streaming data sources called *temporal table joins*. Let's take a look at Example 5-6. The code is written in Java this time due to the lack of support for temporal table joins in the Python API.

Example 5-6. Temporal table join in Apache Flink

```
tableEnv
 .createTemporaryTable("visits_tmp_table", TableDescriptor.forConnector("kafka")
 .schema(Schema.newBuilder().fromColumns(SchemaBuilders.forVisits())
   .watermark("event_time", "event_time - INTERVAL '5' MINUTES")
   .build()).option("topic", "visits")
// ...


tableEnv
 .createTemporaryTable("ads_tmp_table", TableDescriptor.forConnector("kafka")
 .schema(Schema.newBuilder().fromColumns(SchemaBuilders.forAds())
   .watermark("update_time", "update_time - INTERVAL '5' MINUTE")
   .build()).option("topic", "ads")
/// ...


TemporalTableFunction adsLookupFunction = adsTable.createTemporalTableFunction(
  $("update_time"), $("ad_page"));


tableEnv.createTemporarySystemFunction("adsLookupFunction", adsLookupFunction);


Table joinResult = visitsTable.joinLateral(call("adsLookupFunction",
 $("event_time")), $("ad_page").isEqual($("page"))).select($("*"));
```

Example 5-6 starts with the tables declaration. As you can see, besides the schema and topic configuration, the command defines the allowed watermark for each topic. Next, the code initializes a TemporalTableFunction used later in the joinLateral command. This function performs the important role of getting the most recent ad for each page. Put differently, it gets an ad whose update_time <= event_time.

Data Decoration

Once the dataset has gained increased value through a data enrichment pattern, the next question to ask is, is that enough? Data is a crucial asset in modern organizations, but in the enrichment scenario, where the data is still raw or unstructured, the data is rarely in its final form. Without additional preparation work, it can be hard to understand and seize. This is where data decoration patterns may help.

Pattern: Wrapper

In software engineering, *wrapping* consists of adding an extra behavior or attribute(s) to an object. This same definition is valid in the data world, where wrapping also helps separate the original parts of a record from transformed parts.

Problem

Your streaming layer processes the visits data. Visits come from different data providers, and as a consequence, they result in different output schemas.

You need to write a job that extracts the different fields and puts them into a single place so that downstream consumers can rely on it for easy processing. The requirement here is to clearly separate these computed values from the original ones to simplify processing logic but keep the original structure for debugging needs.

Solution

The requirement to keep the original record untouched reduces the transformation scope. You can't simply parse the row and generate a new structure because you'll lose the initial values. To preserve them, you should use the Wrapper pattern.

The idea is to add an extra abstraction at the record's level. The abstraction wraps the original values with a high-level envelope. In addition to these initial attributes, the envelope references computed attributes that may come from the input data itself or from the execution context (e.g., processing time or job processing version). For example, our input visit event could transform into an event composed of *raw* and *computed* fields, corresponding respectively to the two sections.

In addition to formats used in streaming, such as Apache Avro, Protobuf, or JSON, the pattern is supported in structured formats, like tables. In this context, you can implement it either as a separate table that you can join with the original one if needed or as extra columns within the same table. Figure 5-4 depicts possible implementations.

As you can see in the figure, there are four different wrapping implementations for structured data:

- Implementation 1 stores the original row in a flat structure and all computed columns as nested attributes.
- Implementation 2 does the opposite (i.e., it stores computed rows as a single flat structure).
- Implementation 3 stores all columns in a flat structure at the same level.
- Implementation 4 stores the data in two separate tables that can be joined later by a unique key.

The first two implementations use a denormalization approach that may be faster at reading. The third one uses the normalized approach, which may be slower at reading but can be a better choice if you need to logically isolate the datasets or when you simply can't change the original structure. All these approaches share the need for schema management that you will discover along with the schema consistency patterns in Chapter 9.

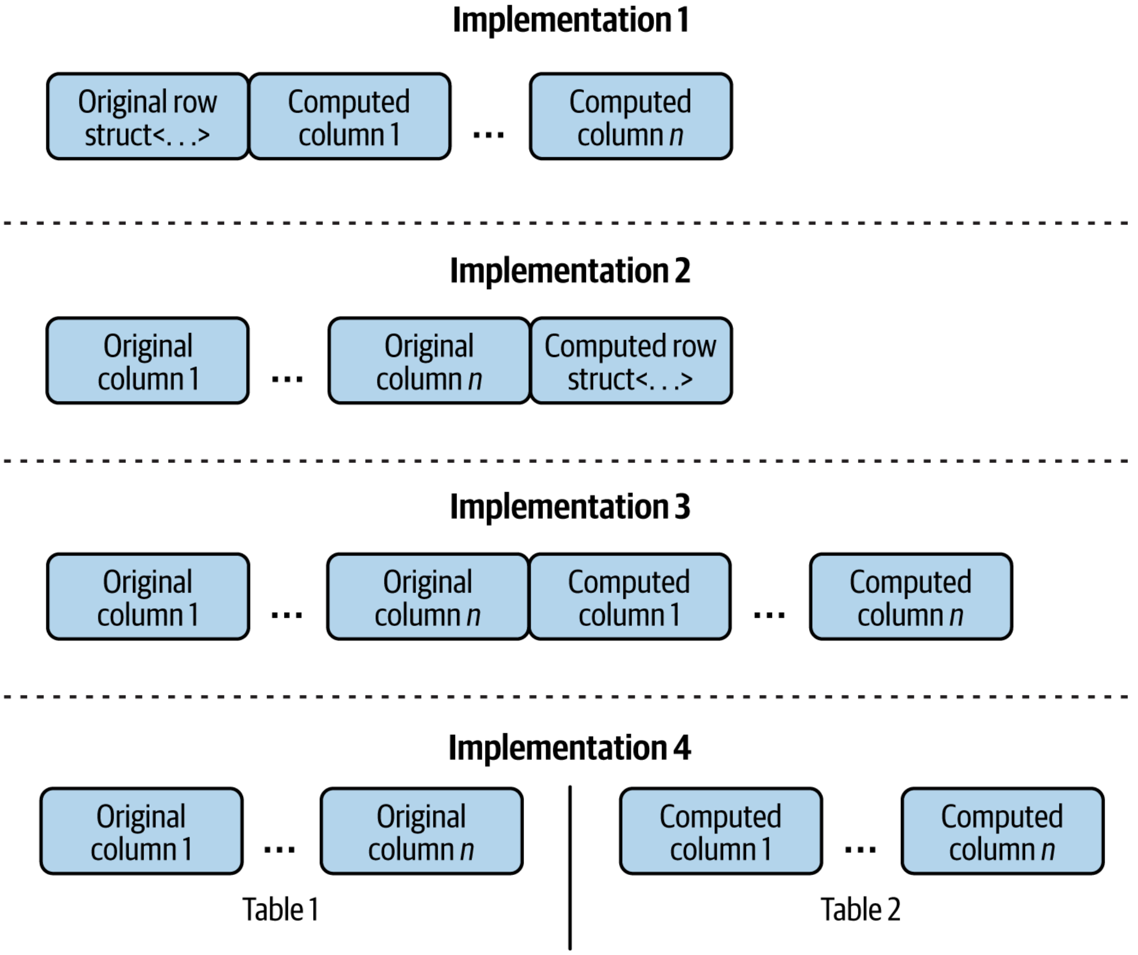


Figure 5-4. Possible Wrapper implementations for structured data

A Wrapper in a Table?

Even though the wrapper envelope is not directly visible in structured formats, it's still there. In fact, the envelope is a row of a table. Therefore, there is no need to break from the normal columnar format and, for example, put fields from multiple columns into one column with nested attributes.

Consequences

This logical and physical separation may have some serious consequences for your dataset, however, mostly related to the storage footprint.

Domain split

This is the logical implication because the pattern divides attributes for a given domain. Let's take a look at an example of a user. If you implement the Wrapper, you'll find user-related fields in two different high-level structures: raw and computed. Although this approach has some advantages, such as making a clear distinction between transformed and nontransformed values, it also makes data retrieval more complicated. Consumers must be aware that user data is in these two locations.

As a trade-off, you could consider the wrapped data to be the data belonging to the first storage layers of your system, like the Silver layer from our use case, and not the final data exposed to the users, for whom this separation may be confusing.

Size

Decorated values form an intrinsic part of the processed record, and therefore, they impact the overall size and network traffic. This differs from the behavior of the Metadata Decorator pattern, which you'll see in the next section.

When it comes to the size impact in the Wrapper pattern, you can mitigate the limitations if your data storage format supports data source projection. With this feature, you can select the columns you are interested in and ask the data source to physically access only them. This practice is very common for all solutions leveraging columnar data storage, such as data warehouses (e.g., AWS Redshift, GCP BigQuery).

Examples

The Wrapper supports both business and technical attributes. The second category includes the metadata values of the execution context, such as the job version or execution time, that may help with debugging production issues later. Let's take a look at Example 5-7 which shows how to integrate them into the row with Apache Spark.

Example 5-7. Wrapping metadata with PySpark

```
visits_w_processing_context = (visits.withColumn('processing_context', F.struct(
 F.lit(job_version).alias('job_version'), F.lit(batch_number).alias('batch_version')
)))


visits_to_save = (visits_w_processing_context.withColumn('value', F.to_json(
 F.struct(F.col('value').cast('string').alias('raw_data'),
 F.col('processing_context')))))
```

Example 5-7 defines the metadata as a new column of the input dataset. Next, it includes these extra attributes with the help of a new structure composed of the initial value (raw_data) and the technical context (processing_context). The whole is then transformed into a JSON document and written to the output database.

Although the code uses the PySpark API, it's also possible to do the wrapping in Apache Spark SQL. Example 5-8 shows a query enriching the input rows with an additional structure called decorated.

Example 5-8. Wrapping with an extra struct in SQL

```
SELECT *, NAMED_STRUCT(
 'is_connected',
 CASE WHEN context.user.connected_since IS NULL
   THEN false ELSE true END,
 'page_referral_key', CONCAT_WS('-', page, context.referral)
) AS decorated FROM input_visits
```

The difference with the query in Example 5-8 is the NAMED_STRUCT function, which provides a convenient way to alter the struct key names with values such as key1, value1, key2, value2, keyn, and valuen. Alternatively, you can consider the decorated data to be first-class table columns and all raw data to be additional context, as demonstrated in Example 5-9.

Example 5-9. Wrapping with the raw value struct in SQL

```
SELECT
 CASE WHEN context.user.connected_since IS NULL
  THEN false ELSE true END AS is_connected,
 CONCAT_WS('-', page, context.referral) AS page_referral_key,
 STRUCT(visit_id, event_time, user_id, page, context) AS raw
FROM input_visits
```

As you can see, Example 5-9 promotes all computed values as the table columns and relegates the raw values to a column of a struct type. Finally, you can also have a table with all computed columns at the same level as the input ones. But you must be aware that in that context, the end users may not be able to distinguish the raw values from the computed ones if the names don't clearly differentiate raw from computed values.

Pattern: Metadata Decorator

The Wrapper pattern is universal since it's always possible to wrap a record. However, when the added values shouldn't be directly exposed to end users, it can be misleading to consumers. After all, your end users won't care whether the record was generated by job version 1 or 2. To overcome this problem, you can hide the extra records in the metadata layer of your data store.

Problem

Your streaming jobs evolve quite often, and you release a new version almost once a week.

Although your deployment process is smooth, you lack some visibility into the impact of the released version on the generated data. To simplify your maintenance activity, you need to add some technical context to each generated record, such as the job version. However, you don't want to include this information in the records sent to end users.

Solution

Including the technical context in the record with the Wrapper pattern is not an option here. This information may not be relevant to your consumers since they're not interested in your internal data processing details. Instead, you can leverage the metadata layer of your data store to apply the Metadata Decorator pattern.

The implementation will depend on your data store capabilities for handling metadata, though. If it supports the metadata out of the box, you will be able to associate each written record with a dedicated metadata attribute. Since this attribute is a native part of the data producer's capabilities, the implementation is relatively straightforward. This is the case with Apache Kafka, which, besides the key and value attributes, supports a list of optional header key-value pairs for each record.

If you work with object stores and your metadata decoration applies to all rows present in a given file, you can define the metadata attributes as tags associated with the file.

If tags are unavailable, other data stores may not support metadata decoration natively. That's the case with relational or NoSQL databases. With them, you can simulate the decoration by including the metadata within the data part but without publicly exposing it to end users. To help you understand how to do this, let's take a look at an example of a data warehouse table. If you want to track the metadata individually for each record, you can write it to a dedicated column and either expose the table from a view without this technical information or use permissions to block reading of that column by nontechnical users (see "Pattern: Fine-Grained Accessor for Tables"). Table 5-1 shows an example of this type of schema.

event_id	event_time	...	processing_context
1	2023-06-10T10:00:59Z		{"job_version": "v1.0.3", "processing_time":"2023-06-10T10:02:00Z"}

Table 5-1. A table with a Metadata Decorator column, which consumers should access via a view or with privileges for reading all but the metadata columns

In addition to the row-based approach shown in Table 5-1, you can opt for an alternative approach and store the processing context in a dedicated table that will be joined with the dataset. Here, the implementation is even simpler as you can literally hide this table in a schema available only to the technical members of your team. Tables 5-2 and 5-3 show this approach.

event_id	event_time	...	processing_context_id
1	2023-06-10T10:00:59Z		1

Table 5-2. Processing context table approach, data table

processing_context_id	job_version	...	processing_time
1	v1.0.3		2023-06-10T10:02:00Z

Table 5-3. Processing context table approach, technical table

These tables present another implementation that's possible where the metadata context is normalized as a separate table; therefore, it's not duplicated for each row. Again, users shouldn't have access to the technical table or the processing_context_id field from the data table, to avoid any confusion.

Wrapper and Metadata Semantics

This metadata decoration is similar to the one you saw for the Wrapper pattern. The only difference is the semantics. Metadata is not supposed to be exposed publicly to business users because by definition, it's a description of the data. That's not the case with the Wrapper because it's intended to decorate business attributes in addition to technical ones.

Consequences

The support of data stores for metadata handling will probably be your biggest limitation in implementing the pattern. But it's not the only one.

Implementation

Even the streaming brokers from the problem statement may lack native metadata support, making implementation impossible for them. For example, Amazon Kinesis Data Streams doesn't support headers.

Implementation can also be challenging for table datasets, where, as demonstrated before, you'll often need to define an extra column or table to handle the metadata information. Although this works, it requires more effort than for data stores that natively support metadata decoration.

Data

Even though there is no technical limitation on what type of information you can put into the metadata layer, you should avoid writing business-related attributes there, such as shipment addresses or invoice amounts. Otherwise, they remain hidden to consumers, most of whom will never think about querying the metadata part. By definition, metadata is data about data, and any external dataset users will primarily be interested in the data from the second part of the sentence.

Examples

For each new concept, it's always better to start with an easy example, so let's take a look at Example 5-10 to see how to add metadata to an Apache Kafka record written from an Apache Spark Structured Streaming job.

Example 5-10. Adding a metadata header for Apache Kafka in PySpark

```
visits_with_metadata = (visits_to_save.withColumn('headers', F.array(
 F.struct(F.lit('job_version').alias('key'), F.lit(job_version).alias('value')),
 F.struct(F.lit('batch_version').alias('key'),
 F.lit(str(batch_number).encode('UTF-8')).alias('value'))
)))
(visits_with_metadata.write.format('kafka')
 .option('kafka.bootstrap.servers', 'localhost:9094')
 .option('includeHeaders', True).option('topic', 'visits-decorated')
.save())
```

Example 5-10 has two important parts. First, the metadata is an array of key-value pairs. Second, the includeHeaders option commands Apache Spark to include the headers column in the generated record out of the box.

Another implementation of the Metadata Decorator is an external metadata table whose schema is present in Example 5-11.

Example 5-11. Metadata table initialization

```
CREATE TABLE dedp.visits_context (
  execution_date_time TIMESTAMPTZ NOT NULL,
  loading_time TIMESTAMPTZ NOT NULL,
  code_version VARCHAR(15) NOT NULL,
  loading_attempt SMALLINT NOT NULL,
  PRIMARY KEY (execution_date_time)
)
```

The context table is later referenced as a part of the job loading input data to the user-exposed table. The loading script first inserts new execution context and later adds the primary key of the visits_context to the visits weekly table. As a result, you'll be able to combine the visits with the metadata. The full code is present in Example 5-12.

Example 5-12. Inserting new visits with a metadata table

```
{% set weekly_table = get_weekly_table_name(execution_date) %}

INSERT INTO dedp.visits_context

 (execution_date_time, loading_time, code_version, loading_attempt)

VALUES ('{{ execution_date }}', '{{ dag_run.start_date }}',

 '{{ params.code_version }}', {{ task_instance.try_number }});


INSERT INTO {{ weekly_table }} (SELECT tmp_devices.*,

  '{{ execution_date }}' AS visits_context_execution_date_time FROM tmp_devices);
```

Data Aggregation

So far, you have been “adding” information. But can you imagine that removing it is also a way to generate data value? If not, the two patterns from this section should prove you wrong.

Pattern: Distributed Aggregator

The first pattern leverages distributed data processing frameworks. One of their great features is the capability to combine multiple physically isolated but logically similar items.

Problem

You’ve written a job that cleans the raw visit events from the Bronze layer of our use case architecture and writes them to the Silver layer. From that place, many consumers implement various final business use cases.

One of the use cases requires building an *online analytical processing* (OLAP) *cube*, thereby reducing all visits to an aggregated format that’s well suited to your dashboarding scenarios. The result should include basic statistics (count, average duration, etc.) across multiple axes (user geography, devices, etc.).

The dataset is stored in daily event time partitions, and the analytics cubes should represent daily and weekly views.

Solution

For datasets that fit into a single machine or container, you don’t need any specific tool to perform the aggregation. The native group by function of your programming language, followed by a reduced logic, should be enough. However, in the big data era, when related records can be split across multiple physical places, this requirement doesn’t hold every time. That’s where the Distributed Aggregator pattern helps.

The pattern leverages multiple machines that together form a single execution unit called a *cluster*. These servers individually don’t have enough capacity to process the whole input dataset, but together, they divide the work and can handle this scenario.

Despite this hardware difference, the code-based implementation may remain the same as for small, local datasets. This means you can still use a grouping function to bring the related rows together and later apply a reduce function on top of them. The first step is optional, though, as you can also combine the whole dataset as is (for example, to get a total count of rows or a global average across all rows).

But the devil is in the details. Although the API might look the same, under the hood, execution of the Distributed Aggregator involves a step to exchange records that were initially loaded into different machines, across the network. As a result, the reduce function can operate on all necessary collocated rows, as you can see in the schema in Figure 5-5.

Here, the action depicted is called a *shuffle*. Often, it is one of the first latency troublemakers because of the network traffic cost.

Even though the example shows a raw record exchange, not all record exchanges are raw. Any aggregation that supports partial generation can be optimized by performing a partial aggregation locally, before the shuffle. As a result, the exchanged records will be smaller and the whole operation should be faster. A great example here is the count operation. Instead of shuffling all the raw records and counting them on the final reduce nodes, the compute layer can perform a partial count on each initial node for all keys that are present locally and shuffle only the numbers to sum in the end.

MapReduce

The Distributed Aggregator pattern is a typical example of the MapReduce programming model, which greatly contributed to simplifying distributed data processing back in 2004. Over the years, its implementations have evolved from disk-based Hadoop MapReduce to memory-first Apache Spark.

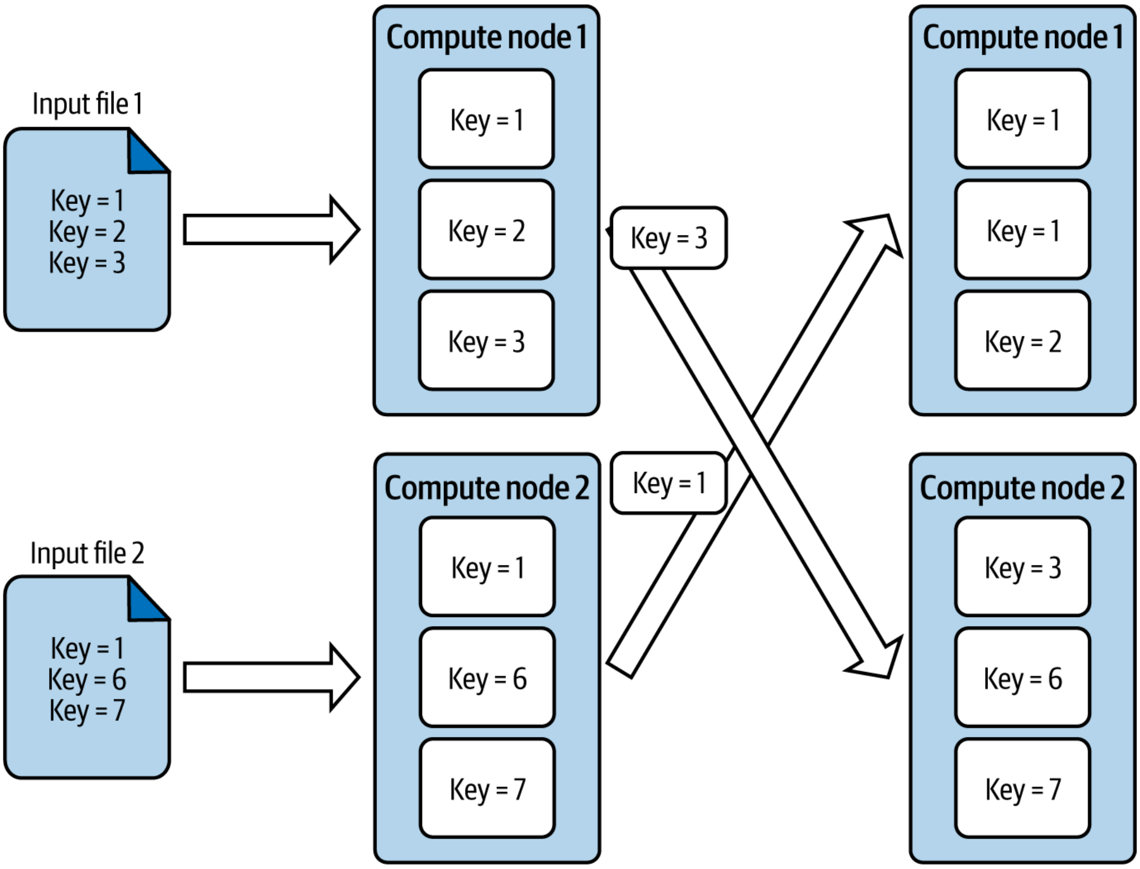


Figure 5-5. Data exchange in Distributed Aggregator

Consequences

Even though frameworks fully implement the pattern, with their high-level API and low-level cluster orchestration, there are some gotchas related to the data itself.

Additional network exchange

The pattern involves two network exchanges. The first brings input data to each node. This is difficult to avoid as nowadays, storage and compute colocation is not a common pattern. The second network exchange comes from the Distributed Aggregator pattern because it's a required step to gather related data on the same server. This is one of the possible latency issues to look at when problems arise.

Unlike the input part, this shuffle can be avoided under specific conditions, which are all explained in the next section on the Local Aggregator pattern.

Data skew

Data skew is a term describing unbalanced datasets in which at least one key has way more occurrences than the others. In that case, the cost of moving it across the network and processing it in a single node will be the highest. Thankfully, some techniques exist to prevent the skew, such as *salting*, which consists of adding an extra value (aka *salt*) to the grouping key and performing the first grouping operation on the salted column. Next, if you need to get the results for the original grouping key, you'll need to aggregate the outcome of the salted column's aggregation again (see Example 5-13).

Example 5-13. Salting example in PySpark for skewed column column_a

```
dataset.withColumn('salt', (rand()*3).cast("int"))
 .groupBy('group_key', 'salt').agg(...)
 .groupBy('group_key').agg(...)
```

Also, data processing frameworks may have some native data skew mitigation, like Apache Spark with Adaptive Query Execution.[1]

Scaling

In addition to this network traffic impact, shuffle has also another implication: scaling. If, for whatever reason, a node has completed all planned reduce operations, it may still be in use by the hardware layer for fault tolerance reasons. If the whole reduce computation fails and gets restarted, this data won't need to be reshuffled again—but when there is no failure, the node will still be there but will not be reclaimed as long as the processing is running. If you want to avoid keeping it for all that time, you can opt for a component called shuffle service.

Shuffle service is an additional compute component that is responsible for storing and serving only shuffle data. Therefore, if one node is not used anymore, the compute layer can free it at any time, even when the job is still running. Among the implementations here, you'll find Apache Spark's External Shuffle Service and GCP Dataflow's Shuffle.

Examples

The best way to understand the Distributed Aggregator and have a demo that’s easily reproducible locally at the same time is to use two different data stores. In our case, we’ll use PostgreSQL and a local file system with JSON files. Although the data stores are physically isolated, Apache Spark can combine them with the code in Example 5-14.

Example 5-14. Aggregation of two physically isolated data stores in PySpark

```
visits: DataFrame = spark_session.read.json(f'{base_dir}/input-visits')

devices: DataFrame = spark_session.read.jdbc(url='jdbc:postgresql:dedp',

  table='dedp.devices', properties={'user': 'dedp_test',

 'password': 'dedp_test', 'driver': 'org.postgresql.Driver'})

visits_with_devices = visits.join(devices,

 [devices.type == visits.context.technical.dev_type,

  devices.version == visits.context.technical.dev_version],

 'inner')
```

Apache Spark provides a convenient way to check whether the operation contains shuffle or not. To perform this verification, you need to call the explain() method on top of your DataFrame and look for the Exchange hashpartitioning node. Example 5-15 shows output from the explain() command for the join between the PostgreSQL and JSON datasets from Example 5-14. As you’ll notice, the job performs local operations such as filtering before preparing the data to be exchanged across the network.

Example 5-15. The execution plan for our devices-to-users join

```
== Physical Plan ==

AdaptiveSparkPlan isFinalPlan=false

+- SortMergeJoin [ctx#8.technical.dev_type, ctx#8.technical.dev_version],..

 :- Sort [ctx#8.technical.dev_type ASC NULLS FIRST, ctx#8.technical.dev_version..

 : +- Exchange hashpartitioning(ctx#8.technical.dev_type, ...

 :   +-Filter (....))

 :     +- FileScan json [.....

 +- Sort [type#20 ASC NULLS FIRST, version#22 ASC NULLS FIRST], false, 0

  +- Exchange hashpartitioning(type#20, version#22, 200), ENSURE_REQUIREMENTS,..

    +- Scan JDBCRelation(....
```

In addition to a distributed data processing framework, the pattern works for databases, which can read datasets from a different storage location. For example, in GCP BigQuery, a serverless data warehouse offering on the public cloud, you can combine a table with files stored on the Google Cloud Storage (GCS) service. This looks similar to our example with Apache Spark, by the way.

To enable this combination, you need to declare the GCS objects to be an *external table* and later just reference them as any regular BigQuery objects in the queries. By the way, external tables are also supported in other data warehousing systems, including AWS Redshift, Azure Synapse Analytics, and Snowflake.

Pattern: Local Aggregator

Network exchange may not be necessary if the data is correctly partitioned in the input or when the dataset fits into a single machine. Both scenarios can be addressed with a new aggregation pattern.

Problem

You have a streaming job that generates windows for incoming visits stored in a partitioned streaming broker. The data volume is static, and you don't expect any sudden variations or changes in the underlying partitioning. As a result, the partitions number will never change. You're looking for a way to optimize the job and remove the grouping shuffle step that's added automatically by your data processing framework.

Solution

A costly shuffle, static data source partitioning, and related attributes stored together are three factors in favor of the alternative to the Distributed Aggregator, which is the Local Aggregator pattern.

Although on the surface, the pattern still performs some aggregations, it does so locally with the single network exchange of reading the input data. This solution works thanks to the fixed partitioning schema and correct input data distribution. All records that are relevant for a given grouping key are already present in the same input partition, so there's no need to load them from other places.

In addition to the lack of shuffle, this implementation brings another advantage. The tasks can be *fully isolated*, meaning they won't need to wait for the data on other tasks and can move forward. This is especially useful for streaming applications, where some slower processing units may delay the whole execution.

The implementation effort should focus here on the producer side. It must guarantee to write a record with a particular grouping key to the same physical partition. This can be achieved with a static per-record partitioning key and an immutable number of partitions.

On the consumer side, some of the tools provide facility methods to adapt the pre-partitioned dataset to its shuffle format. This is the case with Kafka Streams and its groupByKey method. Apache Spark doesn't provide any hints or methods for avoiding shuffle, but thanks to its per-partition operations, such as mapPartitions and foreach Partition, you can leverage your programming language's API capability to perform local aggregations. In addition to these convenience methods, Apache Spark avoids shuffle for the datasets that are saved in buckets with the same key and the same number of buckets.

About the Buckets

Bucketing (aka *clustering*) is a way of partitioning the partitions. You'll learn more about it in the Bucket pattern in Chapter 8.

The logic we've just presented applies to the partitioned data sources whose volume is too big to be processed in a single machine. However, if you are working on a nonpartitioned or partitioned but small dataset, you don't need to worry about static numbers of partitions. That's because you can load the whole dataset into your processing node and leverage the shared memory to perform any aggregations.

Consequences

The pattern requires some immutability, which may not always be possible, for the storage and grouping keys.

Scaling

Scaling is the most visible issue. The pattern depends on the static nature of the data source and consistent partitioning (i.e., a guarantee that a given key will always be available from only one processing partition). If you can't guarantee one of these conditions, the pattern won't work correctly because it'll create one or multiple groups for a given key whenever you change storage partitions.

If you needed to scale and adapt the organization, you could do it with a dedicated data storage reorganization task, which would regenerate the partition assignments for all the records. This operation may be costly as it requires processing the whole dataset. Also, it's even trickier to achieve in streaming applications since it would involve a stop-the-world event to enable processing all remaining data on the old partitions before the data producers can write records to the newly organized partitions.

As you can see, Local Aggregator avoids an extra shuffle but makes this important scaling part a lot more challenging.

Grouping keys

For partitioned data sources with static numbers of partitions, the pattern also expects one grouping key logic for all consumers. Unfortunately, this may not be easy to achieve because it would involve writing the same record in multiple places, each time with a different grouping key.

For example, if your application that reads user profile changes groups them by change type while other consumers perform some user ID–based aggregation, you'll need to opt for the Distributed Aggregator pattern for one of those customers.

Examples

Let's start the examples with Kafka Streams, which is a Java-written data processing layer that processes data from Apache Kafka. The library has a groupByKey method that implements the Local Aggregator pattern. Example 5-16 shows how.

Example 5-16. Local aggregation in Kafka Streams

```
KStream<String, String> visitsSource = streamsBuilder.stream("visits");

KGroupedStream<String, String> groupedVisits = visitsSource.groupByKey();

KStream<String, AggregatedVisits> aggregatedVisits = groupedVisits

  .aggregate(AggregatedVisits::new, new AggregatedVisitsAggregator(),
    Materialized.with(Serdes.String(), new JsonSerializer<>())).toStream();
aggregatedVisits.to("visits-aggregated", Produced.with(new Serdes.StringSerde(),
  new JsonSerializer<>()));
```

Example 5-16 starts by declaring the visits topic to be the KStream abstraction. In the next line, it calls the groupByKey to perform local aggregation. How is that possible? There is no mention of any key, after all. The method uses the key attribute associated with each incoming record to combine all records sharing the same key without a network exchange.

If this API capability is not explicitly provided by your library, there may be an implicit way to use it. To understand this better, let's take a look at an Apache Spark code that aggregates visit events without a network exchange. Example 5-17 shows the job logic. It sorts the visit events in each partition by the visit_id and event_time fields.

Example 5-17. Local Aggregator for visits in PySpark

```
sorted_visits: DataFrame = (visits_to_save
 .sortWithinPartitions(['visit_id', 'event_time']))
def write_records_from_spark_partition_to_kafka_topic(visits):
  kafka_writer = KafkaWriter(...)
  for visit in visits:
    kafka_writer.process(visit)
  kafka_writer.close()


sorted_visits.foreachPartition(write_records_from_spark_partition_to_kafka_topic)
```

After ordering the records, still separately for each partition, Apache Spark calls the KafkaWriter from Example 5-18. This class uses the sorted rows to generate an aggregate of the pages visited in a session. Whenever the visit_id is different from the currently buffered visit, the writer sends the aggregation result to the output Kafka topic.

Example 5-18. Local Aggregator for visits in PySpark: partition-based writer

```
class KafkaWriter:
  def __init__(self, bootstrap_server: str, output_topic: str):
    self.in_flight_visit = {'visit_id': None}


  def process(self, row):
    if row.visit_id != self.in_flight_visit['visit_id']:
      send_visit_to_kafka(self.in_flight_visit)
      self.in_flight_visit = {'visit_id': row.visit_id, 'pages': [],...}
    self.in_flight_visit['pages'].append(row.page)
    # ...
```

In addition to open source tools, cloud services provide local aggregation capability. That's the case with the AWS Redshift distribution types presented in Example 5-19. The code configures a users table with the ALL distribution. That way, Redshift copies the users table to all nodes in the cluster. As a result, any join operation with the users table will be performed locally, without shuffle. Another possible configuration is the KEY distribution applied in the example to the visits table. The KEY distribution groups all rows sharing the same DISTKEY on the same storage node. That way, any combination involving the DISTKEY column can be performed locally.

Example 5-19. Storage distribution in AWS Redshift

```
CREATE TABLE visits (
 visit_id INT,
 user_id INT, ...
) ...
DISTSTYLE KEY,
DISTKEY(visit_id);


CREATE TABLE users (
 user_id INT,  ...
) ...
DISTSTYLE ALL;
```

Sessionization

Sessions are special kinds of aggregators since they combine events related to the same activity. Sessions are also popular. You generate them in your daily life whenever you watch streaming videos, work out, or even cook. In each of those activities, you create a session composed of a starting point, session events, and an ending point. In a data engineering context, depending on the nature of your data, which may be at rest or in motion, you can choose from two available sessionization patterns that you are going to learn about in this section.

Pattern: Incremental Sessionizer

A session sounds like a real-time component. Indeed, most of the time, it results from real-time data, but the generation method also supports batch processing. The pattern you're going to see right now is adapted to batch pipelines.

Problem

The data ingestion team stores visit events from our use case architecture in an hourly partitioned location. You want to aggregate them into sessions that start with the first visit and end if there are no new visits from the given user within two hours.

The typical duration of a visit session is between several minutes and three hours. As a result, one visit can spread across at most three different partitions. Data analysts from your team are struggling to get the sessions right because each time, they need to process many consecutive partitions for a given user. They've explained to you what the problems are, and thankfully, you know about Incremental Loader design pattern and have an idea about how to leverage it for the sessionization use case.

Solution

Since records for one session may be present in multiple consecutive partitions, the problem belongs to the incremental processing family. To solve it, you can leverage the Incremental Sessionizer pattern.

The implementation requires setting up the following three storage spaces:

Input dataset storage

This stores the raw events you need to correlate in the sessionization pipeline. There, you'll find the hourly partitioned visits from the problem statement.

Completed sessions storage

This is the place where you'll write all finished sessions. Eventually, you could also write the ongoing sessions here as well, but ideally, you should distinguish them from the completed sessions (for example, by using an attribute like is_final set to false whenever a session is still active).

Pending sessions storage

Here, you store all sessions spread across multiple partitions that will be closed in one of the next executions. There are two differences between this and completed sessions storage. First, it must remain private as this space will belong to you and evolve with your internal logic. End users doesn't need to be aware of the details, and if they were, you would have less evolution flexibility. Second, the data format for the sessions can be different from the format in completed sessions storage. You could include some technical or internals details here, such as execution ID, if it would be helpful in defining the processing logic (for example, in guaranteeing idempotency).

However, storage won't be enough. You also need to define the workflow logic. The logic starts by combining the input dataset with all pending sessions generated in the previous execution. The combination happens for each session entity—such as a user, product, or visit—and it can generate the following:

- A new session if there is no pending session for a given session entity.
- A restored session with new data coming from the read input.
- A restored session without new session data. This session will probably be about to expire in this or the next execution, depending on the expiration rules you've defined.

Once you complete this combination, you'll get session data to process, possibly composed of previous and new records. On top of that, you need to apply sessionization logic that defines three states:

Initialization

When a session starts. For example, it can start when a particular event type occurs such as visiting the home page of your blog.

Accumulation

When a session is live. What do you do with new incoming data? For example, you could store the visited pages in order.

Finalization

When a session stops. A session can finish when a particular event type occurs or because of a period of inactivity (for example, when a user closes the browser or goes away for three hours).

Figure 5-6 explains this in visual form.

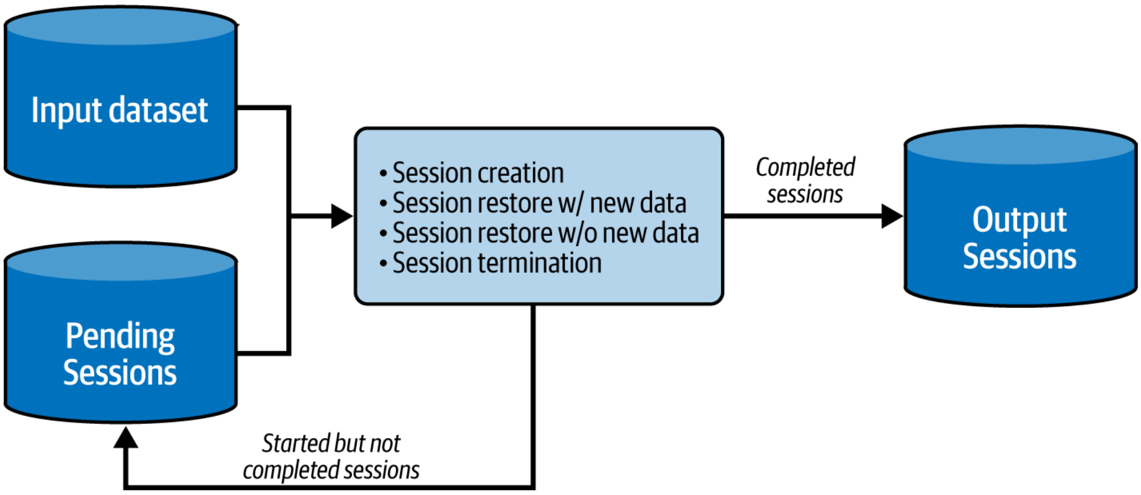


Figure 5-6. Incremental Sessionizer with three storage spaces

In the schema, you can see the execution flow with all involved storage spaces. The transformation loads pending sessions created in the previous run and new data available in the input dataset. Afterward, all completed sessions go into the publicly exposed storage while all pending sessions are written elsewhere to keep them alive and available for the next job execution.

You can define the processing logic with a WINDOW function or a GROUP BY expression, followed by a custom mapping function in the programming language of your choice. Depending on the logic's complexity, the implementation may be easier to express with a programmatic API than with SQL.

Consequences

The implementation should be clear by now. However, that's just the bright side. The Incremental Sessionizer is no exception to the fact that you may encounter some surprises in any pattern.

Inactivity period

The inactivity period defines how long you can keep a session open. The longer it is, the more late data you can include in the session. On the other hand, you'll need more compute and storage resources to handle the late data. Again, you should find the right balance between the compute requirements and the business logic because it'll be challenging to have both. What that balance is, of course, is entirely dependent on your specific business needs.

A long inactivity period threshold will also keep the sessions in the hidden space for that amount of time. If your users can accept partial session views, you can also emit them to the output sessions storage. But there is a consistency risk your consumers must be aware of.

A partial session is not a completed session, and it may change in subsequent versions. Imagine a partial session for fraud detection in a banking system (which is better to do in real time, but it's a good example for partial sessions). After you process the first partition, the partial session will be classified as "not at risk" and only the next partition will change the status to "risky." If consumers consider partial sessions to be final, they may apply the wrong logic to the first instance.

It's therefore important to flag the ongoing sessions—for example, with an attribute like is_completed: false—to help downstream consumers ignore them if they only care about the finished state.

Data freshness

The Incremental Sessionizer works for batch pipelines, which are still the first choice of processing mode for data teams. As a result, insights often come very late compared to real time. To mitigate this issue and still be able to use batch pipelines, you can create the partial sessions introduced in the previous section.

Late data, event time partitions, and backfilling

If your sessionization logic relies on event time partitioning, late data will be a problem as you may miss sessions for already processed partitions.

Sessions are a specific data asset that is forward dependent. Yeah, that sounds weird at first, but it's true. A session generated for the partition at 09:00 directly impacts the session at 10:00, the one at 10:00 impacts the one at 11:00, and so on. Therefore, even if you manage to integrate late data for one entity, there is no guarantee that this data won't impact the next sessions!

This dependency is also visible in backfilling. If you rerun the session generation logic for one partition, you'll have to do the same for all subsequent partitions. This can become expensive very quickly.

Unfortunately, there is no silver bullet that will keep your codebase easy and your costs optimized. The simple solution of replaying all partitions after the backfilled one is easy for the code but costly. On the other hand, having a smart detection method to find entities to backfill and rerunning only them from a dedicated backfill pipeline optimizes the cost but adds extra complexity.

Examples

Incremental is the key word of the pattern since it implies using a data orchestrator to coordinate the loading work. For that reason, let's start the section with the task list for Apache Airflow presented in Example 5-20.

Example 5-20. Incremental Sessionizer steps

```
clean_previous_runs_sessions = PostgresOperator(...)

clean_previous_runs_pending_sessions = PostgresOperator(...)

generate_sessions = PostgresOperator(...)


([clean_previous_runs_sessions, clean_previous_runs_pending_sessions]

 >> generate_sessions)
```

We omitted the configuration from Example 5-20 as the task names are self-explanatory, and you will find all the missing details in the GitHub repo. The pipeline starts with two simultaneous tasks that clean all completed and pending sessions generated in this and all subsequent executions. As you can see in Example 5-21, they use simple DELETE FROM statements applied to the Apache Airflow's immutable ds parameter.

Example 5-21. Idempotency component for the Incremental Sessionizer

```
DELETE FROM dedp.sessions WHERE execution_time_id >= '{{ ds }}';

DELETE FROM dedp.pending_sessions WHERE execution_time_id >= '{{ ds }}';
```

After this context preparation step, the pipeline runs the session generation query, which is composed of four blocks. Example 5-22 shows the first part, which loads all input data into a temporary and session-scoped visits table. That way, if the session query fails, you won't have to replay the loading step each time.

Example 5-22. Session generation: loading new data

```
CREATE TEMPORARY TABLE visits_{{ ds_nodash }} (# ...);


COPY visits_{{ ds_nodash }} FROM '/data_to_load/date={{ ds_nodash }}/dataset.csv' CSV
```

After loading the data, the session generation logic combines the input data with pending sessions. An important thing here is to use an idempotent property to identify previously generated pending sessions. Our snippet leverages Apache Airflow's execution time for that. The query also applies the WINDOW function to the new data so that the visits get formatted into the same schema as the pending sessions. On top of that, the SELECT statement uses facility methods to get the first nonnull element (COALESCE), get the first or last values (LEAST or GREATEST), or concatenate two arrays (ARRAY_CAT). Finally, it also computes the session expiration time only when the session has new data. The whole query is presented in Example 5-23.

Example 5-23. Session generation: the logic

```
CREATE TEMPORARY TABLE sessions_to_classify AS
 SELECT
  COALESCE(p.session_id, n.session_id) AS session_id,
  # ...
  LEAST(p.start_time, n.start_time) AS start_time,
  GREATEST(p.last_visit_time, n.start_time) AS last_visit_time,
  ARRAY_CAT(p.pages, n.pages) AS pages,
  CASE
   WHEN n.user_id IS NULL THEN p.expiration_batch_id
   ELSE '{{ macros.ds_add(ds, 2) }}'
  END AS expiration_batch_id
 FROM (SELECT ... FROM visits_{{ ds_nodash }}
  WINDOW visits_window AS (PARTITION BY visit_id, user_id ORDER BY event_time)
 ) AS n
 FULL OUTER JOIN (
  SELECT ... FROM dedp.pending_sessions WHERE execution_time_id = '{{ prev_ds }}')
  AS p ON n.session_id = p.session_id;
```

In the end, there are two writing steps, which are shown in Example 5-24. The first INSERT writes all pending sessions, that is, the sessions whose expiration time is different than the current run (expiration_batch_id != '{{ ds }}'). Next, it writes all finished sessions.

Example 5-24. Session generation: the writing component

```
INSERT INTO dedp.pending_sessions (...)
    SELECT ... FROM sessions_to_classify WHERE expiration_batch_id != '{{ ds }}';
INSERT INTO dedp.sessions (...)
    SELECT ... FROM sessions_to_classify WHERE expiration_batch_id = '{{ ds }}';
```

Pattern: Stateful Sessionizer

If data freshness is an issue, the Incremental Sessionizer will not help you. Instead, you should use another sessionization pattern that performs great on top of the stream processing layer, thanks to its more frequent and smaller iterations.

Problem

Stakeholders are now quite happy with the session's availability. However, more and more of them need to access the session in a lower latency. That's impossible to achieve with the

Incremental Sessionizer as the partitions are hourly based and the best latency you can provide is one hour.

The good news is that the visits are also available in your streaming broker within seconds. You are looking for a way to rewrite the batch pipeline and generate sessions in near real time.

Solution

Achieving the "as soon as possible" guarantee for sessions is hard with batch pipelines, but default streaming pipelines won't help either because they are stateless. For that reason, you need to use a more advanced version and solve the problem with the Stateful Sessionizer pattern.

How is that different from stateless streaming pipelines? Stateful pipelines bring an extra component called a *state store* that you discovered when you were learning about the Windowed Deduplicator pattern. In our sessionization context, the state store plays the same role as the pending sessions storage zone in the Incremental Sessionizer (i.e., it persists all in-flight sessions and keeps them available throughout the processing).

But this storage for pending sessions is not the Stateful Sessionizer's only similarity to the Incremental Sessionizer. The Stateful Sessionizer's implementation follows the same workflow as for the Incremental Sessionizer:

- The pattern starts by creating a session, or resuming it, if it's present in the state store.
- Next, the Stateful Sessionizer combines the created or resumed session with new incoming records according to your business logic. For example, it may store visited pages in order.
- In the end, if the session is completed or the partial sessions need to be available to consumers, the pattern transforms and writes the pending session record into the final output. Additionally, if the session is not completed, this step also writes the new state to the state store.

If you implement the pattern on top of a data processing framework, you'll likely get the state store support out of the box. On the other hand, if you implement the job fully on your own, you will need to code these interactions. Figure 5-7 shows a pretty common state store interaction implementation present in Apache Spark Structured Streaming and Apache Flink.

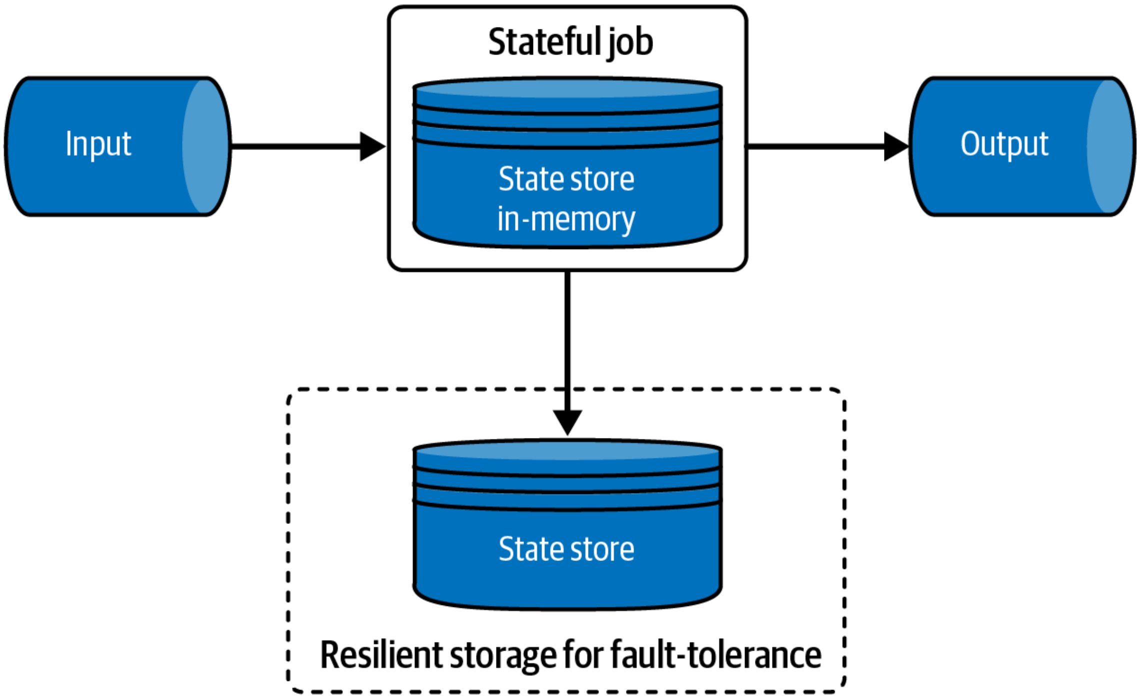


Figure 5-7. Interaction between a data processing job and its state store

The schema illustrates the interaction between a stateful data processing job and its state store. As you can see, there are two flavors of the store. The first one is for fast access. For that reason, it lives in memory, which of course involves volatility. To overcome the risk of losing the state in case of failure or restart, the job synchronizes the state regularly to a more resilient fault tolerance storage.

The data processing logic can rely on the following data processing abstractions:

Session windows

A *session window* is a window created for each session key. Its length is specified by a *gap duration*, which is the maximum allowable period of inactivity between two events with the same session key. If the amount of time between two events with the same session key is greater than the gap duration, a new session window will be created. If not, the new event will be part of the already opened session. Figure 5-8 shows a session window logic with the gap duration of 20 minutes. Two session windows are created once the difference between events is greater than this threshold.

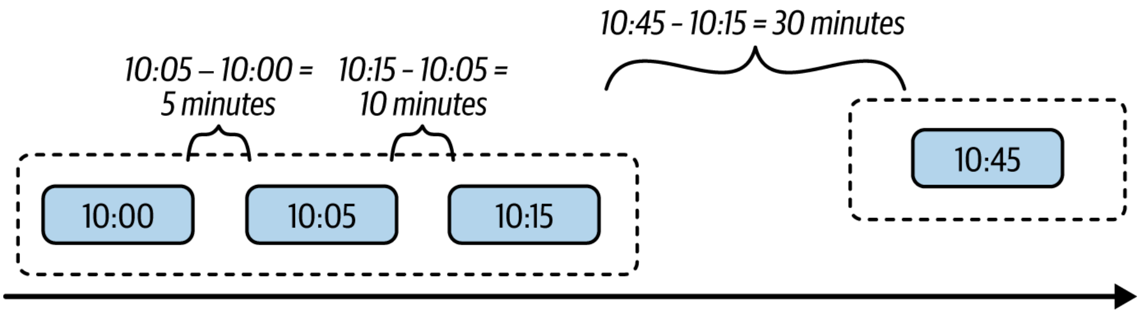

10:45 - 10:15 = 30 minutes
10:05 – 10:00 = 5 minutes
10:15 - 10:05 = 10 minutes
10:00
10:05
10:15
10:45

Figure 5-8. Two session windows for the same key where the gap duration is 20 minutes

Arbitrary stateful processing

This approach requires more implementation effort than for the session window, but it also provides more flexibility. It's up to you to define the gap duration logic, which can be a static timer, as in the previous example, or a dynamic operation, possibly different for each session key. Modern data engineering frameworks such as Apache Spark Structured Streaming, Apache Flink, and even GCP Dataflow provide this capability out of the box.

Consequences

Fault tolerance and the state store, despite their positive impact on the sessionization pipeline, also have some gotchas.

At-least-once processing

Saving the state on fault tolerance storage doesn't happen during every state update. Instead, the writing process, which is called *checkpointing*, occurs irregularly. As you may deduce, any stopped job restarts from the last successful checkpoint, leading to at-least-once processing. This semantic is not bad in itself, but it's better to be aware of it and avoid any side effects of operations that might impact the session's idempotency, such as relying on an attribute that changes between the runs to generate the session key. An example of that type of attribute is real time, which obviously will be different at each restart.

Scaling

Changing the compute capacity in this stateful context may involve state rebalancing. This means that the job will not be able to process the data as long as the particular state keys are not assigned to new workers. That doesn't make scaling impossible, but it makes it more costly than for stateless jobs.

Inactivity period length

As for the Incremental Sessionizer, here too you'll need to strike the right balance to keep the total cost acceptable and include as many sessions as possible. Having a longer inactivity period implies more hardware pressure and output freshness.

Inactivity period time

Besides emitting the sessions to the output storage, the solution will also need to manage the expiration of the state for all completed sessions. The cleaning relies on two different temporal aspects. The first one is the event time from incoming events, which is more reliable and should be preferred in most cases. But there is also processing time–based expiration, which is a bit dangerous, though.

Processing time–based logic relies on the current time, meaning that any unexpected latency (for example, due to writing retries) may cause sessions to expire too early. Because of this unpredictability, it is always easier to reason in terms of event time in stateful pipelines.

Examples

To help you understand the Stateful Sessionizer, we're going to implement a sessionization job with Apache Spark and arbitrary stateful processing. The key part of the implementation is the method in Example 5-25.

Example 5-25. Stateful mapping in PySpark

```
grouped_visits = (visits_from_kafka.withWatermark('event_time', '1 minute')
 .groupBy(F.col('visit_id')))

visited_pages_type = ArrayType(StructType([StructField("page", StringType()),
 StructField("event_time_as_ms", LongType())]))

sessions = grouped_visits.applyInPandasWithState(
 func=map_visits_to_session,
 outputStructType=StructType([
  StructField("visit_id", StringType()), StructField("user_id", StringType()),
  StructField("start_time", TimestampType()),
  StructField("end_time", TimestampType()),
  StructField("visited_pages", visited_pages_type),
  StructField("duration_in_milliseconds", LongType())]),
 stateStructType=StructType([StructField("visits", visited_pages_type),
  StructField("user_id", StringType())]),
 outputMode="update", timeoutConf="EventTimeTimeout"
)
```

The logic presented in Example 5-25 starts by defining the expiration column (event_time) and the grouping key (visit_id). Next, it calls a stateful mapping function composed of the following:

- The function with the stateful logic (func)
- The structure of the session written to the output location (outputStructType)
- The structure of the pending session interacting with the state store (stateStruct Type)
- The output mode that configures the output generation to the updated rows (outputMode)
- The session expiration configuration based on the event time (timeoutConf)

As you can see, this is a purely declarative part of the code. The real generation and accumulation logic is present in the map_visits_to_session function. You can get a high-level view of it in Example 5-26. The code first detects whether the session state has timed out. If it has, the code aggregates the accumulated values into the final format. I'm omitting that part here for brevity's sake but you can retrieve it in the GitHub repo. If the state is still active, the code applies the accumulation logic explained next.

Example 5-26. Session generation logic: high-level view

```
def map_visits_to_session(visit_id_tuple:Any, input_rows:Iterable[pandas.DataFrame],
    current_state:GroupState) -> Iterable[pandas.DataFrame]:
  session_expiration_time_10min_as_ms = 10 * 60 * 1000
  visit_id = visit_id_tuple[0]
  # ...
  visit_to_return = None
  if current_state.hasTimedOut:
    visits, user_id, = current_state.get
    visit_to_return = get_session_to_return(visits, user_id)
    current_state.remove()
  else:
    # ... accumulation logic


  if visit_to_return:
    yield pandas.DataFrame(visit_to_return)
```

Now comes a two-part accumulation logic. You can see the first part in Example 5-27. It's responsible for detecting the base time for the state expiration. Since the watermark will be missing in the first job iteration, the code uses either the event time or a watermark.

Example 5-27. Expiration base time detection

```
 should_use_event_time_for_watermark = current_state.getCurrentWatermarkMs() == 0
 base_watermark = current_state.getCurrentWatermarkMs()
 new_visits = []
 user_id: Optional[str] = None
 for input_df_for_group in input_rows:
  input_df_for_group['event_time_as_ms'] = input_df_for_group['event_time'] \
    .apply(lambda x: int(pandas.Timestamp(x).timestamp()) * 1000)
  if should_use_event_time_for_watermark:
     base_watermark = int(input_df_for_group['event_time_as_ms'].max())
# ... visits accumulation, omitted for brevity
```

Why might using the event time every time not be the best idea, despite the fact that it greatly simplifies the code logic? Relying on a watermark is a better approach here since it's related to the job progress and thus the .withWatermark('event_time', '1 minute') operation. To help you understand this better, let's take a look at Table 5-4, where the watermark-based and event time–based expiration strategies are shown for a watermark of one minute.

State key	**Event time**	**Expiration times**	**New watermark**
A	10:00	Watermark: 10:10 Event-time: 10:10	09:59
A	10:01	Watermark: 10:09 Event-time: 10:11	10:00
A	10:08	Watermark: 10:10 Event-time: 10:18	10:07
B	10:15	Watermark: 10:17 Event-time: 10:25	10:14

Table 5-4. Event time and watermark expiration strategies for a job with 1-minute watermark and 10-minute state expiration

Table 5-4 shows state accumulation for two keys, A and B. As you can see, A is active for eight minutes, and according to the watermark expiration strategy, it could be emitted right after processing the first element of the state B (10:10 < 10:14). However, that's not the case with the event-time strategy as the expiration time is beyond the new watermark (10:18 > 10:14).

Just after this expiration section comes the second part of the logic. Example 5-28 shows the function that interacts with the state store to load the previous state and update the in-flight session with the new expiration time.

Example 5-28. State and expiration time update

```
visits_so_far = []
if current_state.exists:
  visits_so_far, user_id, = current_state.get
visits_for_state = visits_so_far + new_visits
current_state.update((visits_for_state, user_id,))

timeout_timestamp = base_watermark + session_expiration_time_10min_as_ms
```

```
 current_state.setTimeoutTimestamp(timeout_timestamp)
```

The example relying on the session window is much simpler, so let’s see how to define it with Apache Flink. Example 5-29 shows the transformation section that starts with the code extracting the session key. Next, the code configures the session window with the 10-minute gap duration and an allowed lateness of 15 minutes. The last parameter defines how late the events that are going to be integrated into the already emitted session windows can be. In the end, the VisitToSessionConverter converts all records from the window into the final output structure.

Example 5-29. Session window with Apache Flink

```
sessions: DataStream = (visits_input_data_stream
  .key_by(VisitIdSelector())
  .window(EventTimeSessionWindows.with_gap(Time.minutes(10)))
  .allowed_lateness(Time.minutes(15).to_milliseconds())
  .process(VisitToSessionConverter(), Types.STRING()).uid('sessionizer'))
```

The full snippet is available in the GitHub repo.

Data Ordering

Data transformation resulting from data aggregation and combination is not the only valuable property with which you can enhance your dataset. Another one is order (for example, events delivered chronologically to your downstream consumers). That’s why the last data value patterns you’re going to see are about ordering. Even though this feature is often reduced to an ORDER BY clause in SQL, ordered data delivery is challenging. It provides chronology that is important for many use cases, especially those requiring real-time data insight. Imagine a car fleet tracking system that doesn’t respect the order of the cars’ positions on the road. You will likely see cars jumping over buildings or swimming across rivers instead of following their ordered and real positions on the roads. Depending on your data store, you will likely be providing the ordering guarantee via one of two available patterns.

Pattern: Bin Pack Orderer

One of the nightmares for ordered data delivery at scale is partial commits. Some databases provide a *bulk* API to write multiple items at once and consequently optimize network communication. However, this feature may come with partial commits. In other words, you can get a fully successful, partially successful, or fully failed request. The first and last outcomes are fine, but the second, unfortunately, may break the ordering within your dataset. Despite these challenging semantics, there is a solution.

Problem

Your blogging platform from the use case enables external websites to embed your pages. The visit events generated by these embeddings are arriving in your system, and you’re processing them as your own events. Besides keeping them internally, you need to expose them from an external API to external websites for analytics purposes.

The project is reaching the last stage, and you need to write the synchronization job to feed the external API storage. To optimize the cost, the job must be common for all partners. It has to create a processing time window of 10 minutes with per-minute aggregates, and in the end, it must flush the buffer to different outputs provided by partners. The events must be delivered individually for each minute and provider, and in event time order.

Unfortunately, the API ingestion storage is a streaming broker with partial commit semantics. You need to be particularly careful while implementing the ordering logic to overcome any issues related to retries.

Partial Commits

Beware of partial commits. Unlike classical ones, where only two states (success and failure) are possible, they have one more state for partial failure. In that state, the database manages to ingest only a subset of records.

Why can this partial mode break the ordering? Let's take a look at an example of records timestamped with 10:00, 10:10, and 10:20. The database can write all, two, one, or none of them. The two scenarios in the middle are risky because there is no way to know which records have not been delivered and when they will succeed. For example, if only 10:20 has been written, you'll need to retry 10:00 and 10:10, which will leave you with writing that's out of order.

Although this issue is most visible on streaming systems, ordering semantics can also be important for data-at-rest stores, where a temporarily partially empty dataset can trigger some downstream processing on top of disordered data.

You'll find this partial commit semantic in the PutRecords API from Amazon Kinesis Data Streams, the BatchWriteItem from AWS DynamoDB, and the bulk operation from Elasticsearch.

Solution

You can solve the problem if you deliver each record individually. However, that implies significant network overhead as you'll need to initialize as many requests as there are records. You can mitigate the issue by relying on the bulk operations that together with the Bin Pack Orderer pattern can guarantee ordered delivery in the context of partial commits.

The implementation follows two important steps. The first step is responsible for grouping all related events and sorting them. Typically, you'll implement it as a sort by grouping key and event time operation.

The result will be records sorted within the same entity. Next, you need to pack those rows in bins individually (i.e., you need to group events of different entities sharing the same position together, in the same bin). That way, you create isolated subsets that you can deliver through a bulk API without worrying about completeness, duplicates, and more importantly, partial commits. The process is summarized in Figure 5-9.

The workflow follows the steps from the previous paragraph:

1. First it sorts the records by their grouping key and time.

2. Then, the algorithm places the sorted rows into delivery bins so that there is only one grouping key in each bin. The bins can be arrays or lists in your programming language.

3. Finally, the workflow emits bins sequentially. If there is any retry within the delivery bin, it remains local to the group. Put differently, it doesn't interfere with the ordering for the retried grouping key because there is only a single occurrence per bin. The next bin isn't delivered as long as the current one is not fully written to the output.

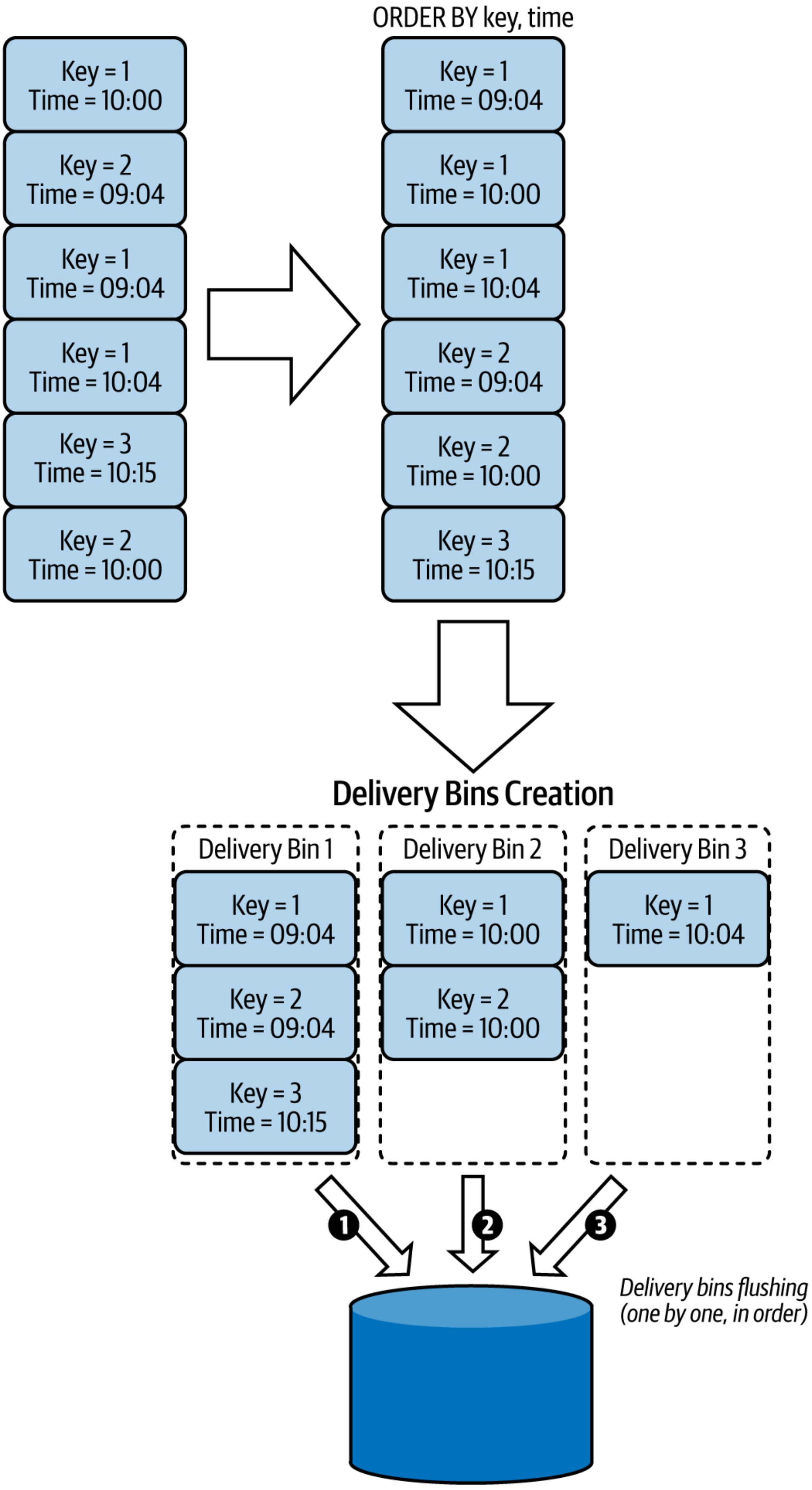

ORDER BY key, time
Key = 1 Time = 10:00
Key = 2 Time = 09:04
Key = 1 Time = 09:04
Key = 1 Time = 10:04
Key = 3 Time = 10:15
Key = 2 Time = 10:00
Key = 1 Time = 09:04
Key = 1 Time = 10:00
Key = 1 Time = 10:04
Key = 2 Time = 09:04
Key = 2 Time = 10:00
Key = 3 Time = 10:15
Delivery Bins Creation
Delivery Bin 1
Key = 1 Time = 09:04
Key = 2 Time = 09:04
Key = 3 Time = 10:15
Delivery Bin 2
Key = 1 Time = 10:00
Key = 2 Time = 10:00
Delivery Bin 3
Key = 1 Time = 10:04
1
2
3
Delivery bins flushing (one by one, in order)

Figure 5-9. Bin packer records delivered with three bulk requests

Consequences

As you might have guessed, the pattern looks great for the ordering guarantee, but if your compute runtime is not adapted, some items may still be out of order. Let us explain.

Retries

The pattern guarantees ordering inside the same execution. If your whole pipeline fails, then the retry will involve already emitted results. Consequently, the overall ordering will be broken despite the fact that you're using the pattern.

Complexity

The bin packer is definitely more difficult to implement than a classical sort. It requires a custom sorting and bin creation logic, whereas performing classical sorting is just a matter of calling an appropriate sorting function.

Examples

Let's see how to implement the pattern with Apache Spark and Amazon Kinesis Data Streams. First, the preparation step uses a local sorting mechanism. The locality consists of partitioning the dataset in each task individually, without involving the network exchange (see Example 5-30).

Example 5-30. Bin packer preparation step

```
(events.sortWithinPartitions([F.col('visit_id'), F.col('event_time')])
 .foreachPartition(lambda rows: write_records_to_kinesis(...))
```

The Bin Pack Orderer pattern sorts the input rows by visit_id and later, inside each group, by event_time. Therefore, prepared rows are later processed via the write_records_to_kinesis method demonstrated in Example 5-31.

Example 5-31. Bin Pack Orderer for Amazon Kinesis Data Streams

```
def write_records_to_kinesis(output_stream, visits_rows):
 producer = boto3.client('kinesis')
 delivery_groups = []
 groups_index = 0
 last_visit_id: Optional[str] = None
 for visit in visits_rows:
  if visit.visit_id != last_visit_id:
   last_visit_id = visit.visit_id
   groups_index = 0
  if len(delivery_groups) <= groups_index:
   delivery_groups.append([])
  delivery_groups[groups_index].append(visit)
  groups_index += 1
```

As you can see, the code in Example 5-31 iterates all input rows and puts each of them into a dedicated bin, as long as the visit_id doesn't change. If it does, the bin's position is reset to 0. After this preparation step comes the delivery, group by group, to the Kinesis output stream. I'm omitting the code for brevity's sake because it's a simple iteration over the groups. Instead, you can view it in the GitHub repo.

Pattern: FIFO Orderer

The Bin Pack Orderer is a pattern that keeps data stores in proper order with partial commit semantics and optimizes throughput thanks to bulk operations. However, there is a simpler alternative that can be good for use cases that don't require low latency or a large volume of data.

Problem

One of your streaming jobs runs on top of the visits dataset. It needs to detect a subset of particular events and forward them in processing order to a different stream. The requirement is to deliver each record as soon as possible, so any buffering to optimize the network traffic is not an option.

Solution

Buffering and bulk requests help reduce network overhead in data transmission. However, in environments with more relaxed delivery constraints, you may want to consider a simpler alternative, like the FIFO Orderer pattern.

The implementation is more straightforward than the one for the Bin Pack Orderer. It doesn't require any specific sorting algorithm since the requirement is to send data in the *first in, first out* (FIFO) manner. Instead, it only detects the records and issues the delivery request. An important thing is to get the delivery acknowledgment for each record before proceeding to the next one. Otherwise, it may lead to issues of data being out of order or lost.

The pattern can be implemented with either an API delivering one record at a time or a bulk API with its concurrency level set to 1. In the first implementation, you'll find AWS Kinesis Data Streams' PutRecord API or Apache Kafka's send(...) function followed by a synchronous flush(...) invocation.

Also, though it may sound surprising at first, you can use the pattern with a bulk API, but only for data stores that support full commit semantics. Here, the only requirement is to avoid too many bulk requests being sent at once, which may lead to data being out of order if the data store fails to write one of them. If you use Apache Kafka, you can achieve this by setting the max.in.flight.requests.per.connection to one or by using the idempotent producer feature that accepts up to five concurrent requests and still guarantees correct ordering.

In-Flight Requests

Using in-flight requests is a great way to optimize throughput. With them, the producer can issue the delivery request for the first bulk API and then, without waiting for the server's response, create and deliver the next one. However, this can also break the ordering. For example, if only the second of two in-flight requests succeeds and the first is retried, the ordering between them will be broken.

Consequences

The pattern's simplicity is appealing, but you shouldn't consider it for all use cases involving FIFO delivery.

I/O overhead and latency

The biggest drawback of this simple implementation, despite the possibility of using bulk API under some conditions in some data stores, is the I/O overhead and resulting increased latency. Instead of sending one network request for many records, the FIFO Orderer pattern sends one request for each input row.

This overhead leads to increased latency as the data store and data producer must handle requests individually. The problem will be particularly visible if you have a lot of data to deliver and a monitoring dashboard to observe the number of delivered records and records to deliver per minute.

You can slightly reduce this impact by leveraging multithreading, meaning by issuing the individual requests from multiple processes of your producer. The only problem is guaranteeing the ordering between these processes because remember, they're isolated and not aware of each other. A good strategy here is to create scopes of ordered records. Put differently, if you need to guarantee the ordering for an entity such as a user or product, you can allocate all records of that entity to the same process. This looks similar to bins, except that each container stores all the records of the same entity and delivers each of them individually from the asynchronous process.

FIFO is not exactly once

Don't get this wrong: FIFO stands only for delivering the oldest records first, and it doesn't guarantee the exactly-once delivery by itself. Let's see why in Example 5-32.

Example 5-32. Snippet showing challenges with exactly-once delivery

```
fifo_messages_to_deliver = ....

for message in fifo_messages_to_deliver:

  producer.send(message)

  consumer.ack(message)
```

Example 5-32 shows what a naive implementation of exactly-once delivery with the FIFO Orderer might look like. First, the producer sends the message, and soon after, the consumer acknowledges it so that the event is not processed again. However, even after a successful send(...), the subsequent ack(...) call can fail. As a result, the exactly-once guarantee will be broken after the code restarts because the producer will try to send already delivered records.

To mitigate this issue, you'll need to rely on one of the idempotency patterns from Chapter 4.

Examples

How do you implement the FIFO Orderer with Apache Kafka? Example 5-33 shows the most basic implementation, in which a producer delivers each record individually.

Example 5-33. FIFO Orderer with individual records delivery

```
producer.produce(...)

producer.flush()
```

It's pretty easy, isn't it? You produce a record, flush the buffer immediately, and wait for the broker to perform the write. But as you know already, it'll be costly in terms of network traffic because you will send one record at a time. Can you do better, then? Yes, there is an alternative with bulk requests, shown in Example 5-34.

Example 5-34. FIFO Orderer with bulk requests

```
producer = Producer({

  'max.in.flight.requests.per.connection': 1,

  'queue.buffering.max.ms': 1000

})

producer.produce(...)
```

Example 5-34 shows an approach that's different from the one in the previous example. It doesn't perform the flush(...). Instead, it delegates the bulk request generation to the producer, who is asked to buffer the records for at most one second. Also, to avoid the issue of concurrent writes, the concurrency flag is set to 1. This approach is indeed more efficient than individual delivery, but the single concurrency flag may still be a slowness factor. To optimize that part, you can use the idempotent producer configuration from Example 5-35.

Example 5-35. FIFO Orderer with idempotent producer

```
producer = Producer({

 'max.in.flight.requests.per.connection': 5,

 'enable.idempotence': True,

 'queue.buffering.max.ms': 2000

})

producer.produce(...)
```

Example 5-35 uses Apache Kafka's idempotent producer,[2] which accepts up to five concurrent requests and still guarantees ordering. The configuration also increases the buffering time so that there is a greater chance to fill the buffer and issue more bulk requests to the broker.

Unfortunately, you may not always have a chance to leverage the bulk API for the FIFO delivery. It won't be possible on data stores with partial commit semantics, where you'll have to rely on individual requests. That's the case with AWS Kinesis Data Streams, where, to guarantee correct ordering, you have to use the PutRecord API and set the SequenceNumberForOrdering as does the code in Example 5-36.

Example 5-36. Using SequenceNumberForOrdering **in AWS Kinesis Data Streams**

```
records_to_deliver = [...]
previous_sequence_number = None
for record in records_to_deliver:
  put_result = client.put_record(StreamName=..., Data=...,
    SequenceNumberForOrdering=previous_sequence_number)
  kinesis_client.put_record(record)
  previous_sequence_number = put_result.sequence_number
```

Without this SequenceNumberForOrdering property, Kinesis may put the records in rough order. Another cloud streaming service, GCP Pub/Sub, also has a built-in ordering guarantee mechanism, but it works only within the same producer writing to the single region. Besides, the feature requires setting an ordering_key attribute (see Example 5-37).

Example 5-37. Using ordering_key **for GCP Pub/Sub**

```
records_to_deliver = [Record(data=..., ordering_key="a"),
  Record(data=..., ordering_key="b"), Record(data=..., ordering_key="c"),
  Record(data=..., ordering_key="a")]
previous_sequence_number = None
for record in records_to_deliver:
  publisher.publish(..., data=..., ordering_key=record.ordering_key)
```

Example 5-37 shows the ordering_key attribute. As you can see, unlike the SequenceNumberForOrdering, it's more of a grouping key used by Pub/Sub's publisher to ensure all records sharing it are delivered in the FIFO manner.

Summary

In this chapter, you discovered common ways to increase the value of your dataset. Generally, two improvement scenarios are possible:

- When you add information to the input data
- When you reduce the information to make it more understandable

In the first scenario, you can use the data enrichment and data decoration patterns. Data enrichment patterns are a great candidate for combining datasets. You saw that this is possible not only for homogeneous pipelines but also for heterogeneous ones, where streaming meets the batch world. When it comes to data decoration patterns, you learned about two approaches to annotating raw records. The first approach wraps the input data and computes extra attributes separately, which is a great way to introduce data consistency in terms of data representation. Additionally, in the second approach, you can hide these extra attributes with the hidden metadata layer if they are not relevant to end users.

In the second scenario, information is reduced with the goal of making the data more understandable. Here, the data aggregation patterns work in a distributed or local environment. In addition, this is the place where you'll be able to summarize your users' experience with Sessionization patterns that are adapted to incremental batch workloads or real-time streaming pipelines.

Finally, you learned about two solutions to preserve correct order in data ordering patterns. The Bin Pack Orderer shows how to address this requirement in the context of data stores with partial commit semantics. The second pattern, FIFO Orderer, is an alternative that sacrifices network exchange for simplicity.

But your journey doesn't stop here. You've already learned how to ingest data and generate value with data value patterns supported by error management and idempotency patterns. However, there is still one point missing: how do you connect them all? The good news is that the next chapter is all about that topic!

[1] You can learn more about Adaptive Query Execution in the Apache Spark documentation.

[2] The idempontent producer was introduced as a part of KIP-98.