Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 3. Error Management Design Patterns

Our new Constitution is now established, and has an appearance that promises permanency; but in this world nothing can be said to be certain, except death and taxes.

Benjamin Franklin

That's what Benjamin Franklin, one of the Founders of the United States, wrote in a letter to French physicist Jean-Baptiste Le Roy in 1789. If Franklin had been a data engineer in our day, he probably would have written something like this:

In this world, nothing can be said to be certain, except errors and data quality issues.

Sad but true, but your data engineering life will rarely be a bed of roses. Remember that data is dynamic and your expectations from today will not remain the same for the whole lifecycle. That's why you will need to expect the worst and adapt accordingly.

Besides, keep in mind that you're processing data generated by others. You directly inherit their data or software engineering issues, such as unreliable networks that lead to late delivery or temporary crashes that cause retried deliveries and subsequent duplicates in the dataset.

The design patterns presented in this chapter focus on managing errors, which is the next logical step in a data engineering project after the data ingestion cycle explained in Chapter 2. Design patterns address issues you need to deal with as a data and infrastructure consumer. You'll find here patterns discussing poor upstream data, such as unprocessable records, late data, and even duplicates. You'll also find patterns addressing hardware issues, like streaming job failures.

With all this context set up, it's time to discover the first group of data-related error management design patterns!

Unprocessable Records

Data quality is a recurrent problem in data projects, and that's why the first issue you'll have to deal with is unprocessable records. Often, they cause fatal failures that stop the data processing job. However, maintaining this *fail-fast* approach won't always be possible, especially for long-running streaming jobs.

Pattern: Dead-Letter

An easy solution is to ignore the bad records and continue processing the correct ones. It's easy to do if you can simply skip the invalid events and thus lose them forever. If this isn't an option, you can opt for another approach and save the bad records elsewhere for further investigation.

Problem

Your stream processing job writes visit events from an Apache Kafka topic to an object store. Bad luck: recently, data producers have started to generate unprocessable records and your job has started failing. For three consecutive days, you have been solving the issue manually by relaunching the job and altering the processed offsets in the checkpoint files. But you're tired. Instead of continuing this tedious action, you are looking now for a better solution to mitigate the issue. The solution should keep the pipeline running even for the occasional failed records and give you an opportunity to investigate the errors later.

Transient Versus Nontransient Errors

While processing the data, you'll face two kinds of errors. The first are *transient errors* that are often temporary and will eventually recover automatically in the future. One example is a short database unavailability mitigated with automatic connection retries.

Another type is *nontransient errors* that by definition are not temporary and will never recover by themselves. One example is unprocessable records, also known as *poison pill messages*. They are fatal issues that stop the application and require your manual intervention.

Solution

If your job can't process one particular record but can continue processing the others, it can be a breath of fresh air during a hard daily maintenance routine. The Dead-Letter pattern makes it possible.

The pattern starts by identifying places in the code where your job can fail. It can be a custom mapping function or even an error-safe transformation fully managed by your data processing framework. Next, you need to add some safety controls over the likely fail spots that have been identified. If you're using the mapping function, the most common safety control will be a try-catch block. If you're using the error-safe transformation, it will rely on an if-else condition. Additionally, you should include the failed message as the metadata to help you better understand the failure at the post-analysis stage. For that, you can leverage the Metadata Decorator pattern.

After identifying the places and decorating your processing part, you need to configure a different output for the erroneous events. The destination can be of the same or a different type than for the successfully processed records. While choosing a destination, you should consider the following:

- Resiliency, so that you don't need to think about a dead-letter strategy for your dead-letter storage.
- Monitoring ease, because it's the key success factor of this implementation. After all, you want to know when your job starts dealing with erroneous records and, especially, how many of them have been written recently. By analyzing these two

metrics you should be able to better understand whether the errors are only occasional issues or whether the whole system is going down.

- Writing performance, since writing the unprocessed records to an extra place will incur some cost in the overall job execution time.

Good candidates for the dead-letter stores are object stores in the cloud or streaming brokers since they're highly available, fast, and easy to monitor.

Finally, you can complete the error handling part of the pattern with the replay pipeline that ingests the failed records into the main data flow. This is an optional step if you don't worry about past data.

Overall, an architecture implementing the Dead-Letter pattern should contain the error handling logic, the dead-letter storage, the monitoring layer, and eventually, the replay pipeline (see Figure 3-1).

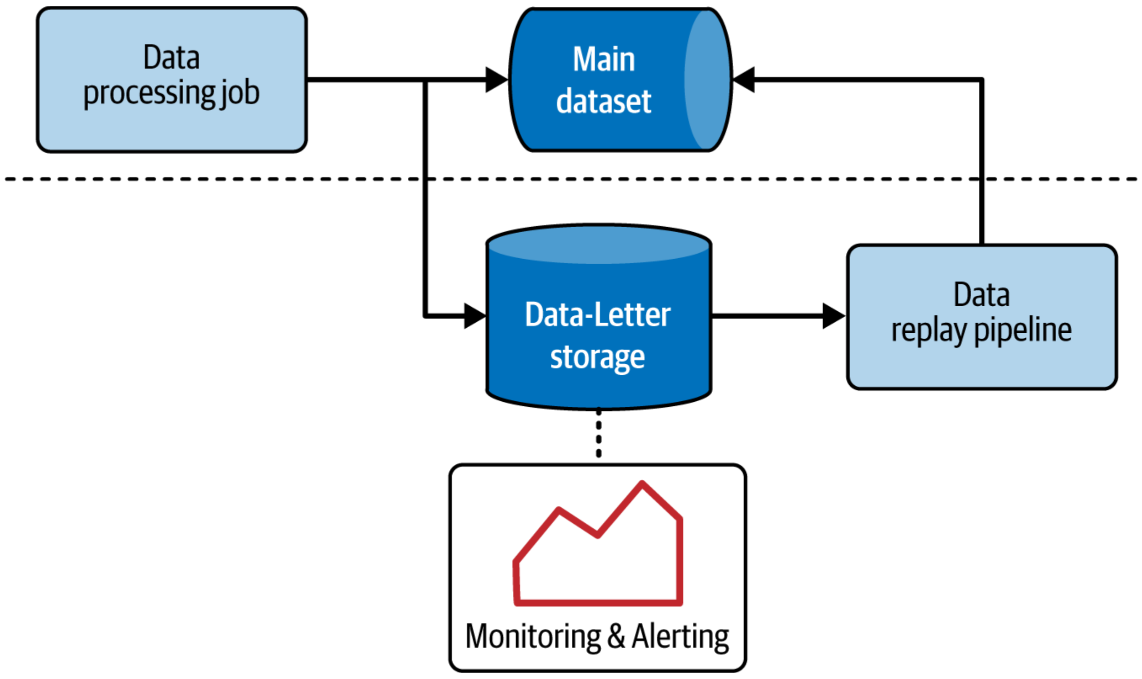


Figure 3-1. Components involved in the Dead-Letter pattern

The Dead-Letter pattern, even though it's often quoted in the context of stream processing, is also widely supported in batch workloads. The difference between stream processing and batch workloads comes from the data perception. Streaming operates on one record at a time and thus can write an individual record to dead-letter storage. Batch works on a bunch of data, and very often, it will write a subset of the erroneous records at once to dead-letter storage. You will see examples of both processing modes in the Examples section.

Consequences

Despite its beneficial impact on daily maintenance efforts, ignoring errors can have some serious consequences that you are going to discover in this section.

Snowball backfilling effect

The good thing about the fail-fast approach is its simplicity for the whole system. In this approach, if your pipeline stops, consumers won't get new data and probably won't run. On the other hand, if your job doesn't follow the fail-fast strategy, consumers will continue processing data that might be partial. Things become even more complicated when it comes to using the optional replay pipeline.

If you decide to run the replay pipeline, the ingested records can belong to the partitions already processed by your downstream consumers. That would require a backfilling action on their part and start a *snowball backfilling effect*, where their downstream consumers must reprocess the data as well. Figure 3-2 shows the beginning of the snowball backfilling effect.

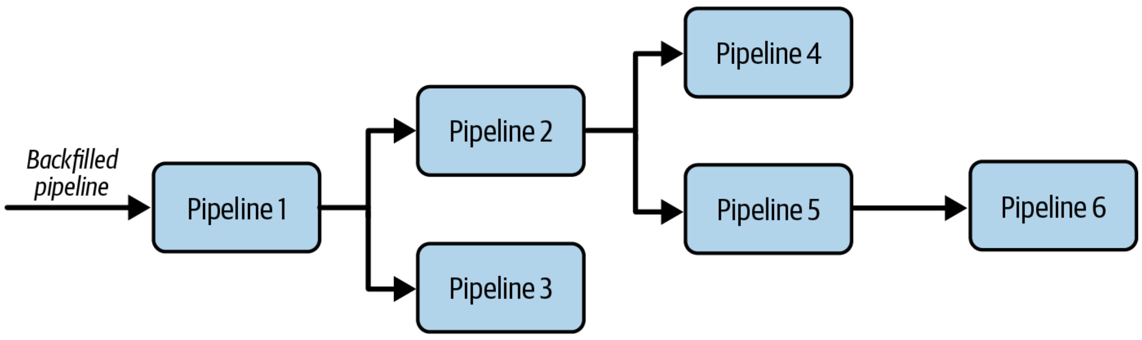


Figure 3-2. Snowball backfilling effect where backfilling Pipeline 1 triggers the same process for all downstream consumers

Mitigating this issue is not easy because each solution comes with its own trade-offs. You can avoid replaying the failed records and thus not trigger the backfilling run. This is the simplest approach, but it has the downside of losing the dead-letterred data. That's why you can decide to trigger the backfilling process. In that configuration, your dataset will be complete but might become inconsistent if you have downstream consumers who don't run or simply can't run a similar backfilling process.

Dead-lettered records identification

Integrating the dead-lettered records with the main data store has another implication: you may want to distinguish them from the rows added in the normal ingestion pipeline. It can be useful to implement a filtering condition skipping replayed records in the downstream consumers or to simply track the origin of each row.

Many solutions exist that you should adapt to your use case. You can add a boolean column or an attribute called was_dead_lettered to indicate each record produced by the Dead-Letter replay job. Or you can use more complete metadata to annotate those records with the job name, version, and replay time. This last approach fits perfectly with the data decoration patterns.

Ordering and consistency

The pattern can break data consistency. Let's learn more about this with an example of Internet of Things (IoT) sensors sending events every minute. One of your consumers builds sessions on top of that data, and the closing rule is based on the five-minute inactivity window. If, for whatever reason, events land in dead-letter storage for five consecutive minutes, the consumer will close the session. As a result, the session will be partial and inconsistent with reality.

This is also true for the ordered data delivery requirement. In that case, any replayed failed delivery will break the ordering consistency. Let's take a look at a quick example of three records to be delivered exactly in this order: 10:00, 10:01, and 10:02. If only the first and last records are correctly written and you decide to replay the failed one, your output data store will return 10:00, 10:02, and 10:01.

Error-safe functions

Error-safe functions greatly reduce the risk of runtime issues in the code because instead of throwing a runtime exception in case of an error, they return a NULL value. Moreover, all this logic is fully managed by your framework or database. However, these error-hiding functions make the Dead-Letter pattern implementation more challenging.

When you use error-safe functions, instead of capturing the exception, you'll need to compare the output value with the input. If the input is present but the function returns a NULL value, it might represent a processing error and thus an unprocessable record.

Moreover, to use error-safe functions, you need to understand their error-safety semantics, which may differ from one function to another. That's why implementing the Dead-Letter pattern on top of them, although possible, is challenging.

Error or failure?

Although the pattern keeps the processing job up, it hides errors. Therefore, it can hide a fatal failure that should stop the pipeline. For that reason, you should complete the code implementation with an appropriate alerting layer that, in case of too many dropped events, could stop the job to avoid propagating potentially wrong data to your system.

Examples

The Dead-Letter pattern is often quoted in the context of stream processing because it allows the pipeline to run despite erroneous records. For that reason, let's see first how to implement it with an Apache Flink job in Example 3-1. The implementation relies on a feature called *side outputs*, which are additional destinations where you can write processed records.

Example 3-1. Dead-Letter component for Apache Flink

```
# source omitted for brevity


invalid_data_output: OutputTag = OutputTag('invalid_visits', Types.STRING())

visits: DataStream = data_source.map(MapJsonToReducedVisit(invalid_data_output),

 Types.STRING())
```

Example 3-2 shows what happens after the output declaration step. The job interacts with the side output in the mapping function called map_rows that implements the dead-letter logic as the try-catch block. As a result, the function will write any rows intercepted in the except part to the side output object. In the end, the job captures all side output entries by calling the get_side_output function and writes them to a dedicated Apache Kafka topic.

Example 3-2. Side output writing and reading in Apache Flink

```
# MapJsonToReducedVisit snippet w/ reference to the side output

def map_rows(self, json_payload: str) -> str:

try:

  evt = json.loads(json_payload)

  evt_time = int(datetime.datetime.fromisoformat(evt['event_time']).

  yield json.dumps({'visit_id': evt['visit_id'], 'event_time': evt_time,

                'page': evt['page']})

except Exception as e:

  yield self.invalid_data_output, _wrap_input_with_error(json_payload, e)


kafka_sink_valid_data: KafkaSink = ...

kafka_sink_invalid_data: KafkaSink = ...


visits.get_side_output(invalid_data_output).sink_to(kafka_sink_invalid_data)

visits.sink_to(kafka_sink_valid_data)
```

But streaming is not the only place where you can implement the Dead-Letter pattern. Let's see now how to adapt it to Apache Spark SQL and Delta Lake with an error-safe CONCAT data transformation, which will never fail but will eventually return null if there is any issue, such as a missing value for one of the concatenated columns. Example 3-3 shows the first building block with the SQL query composed of the following:

- A subquery with the transformation logic using the CONCAT function. The function is one of the error-safe transformations because if any of the combined columns is null, it returns null without throwing an exception.
- The top-level query that validates the concatenation result.

Example 3-3. Dead-Letter pattern for error-safe transformations: the query

```
spark_session.sql('''

 SELECT type, full_name, version, name_with_version,

  WHEN (full_name IS NOT NULL OR version IS NOT NULL)

    AND name_with_version IS NULL THEN false ELSE true

  END AS is_valid

 FROM (SELECT type, full_name, version, CONCAT(full_name, version) AS name_w_version
FROM devices_to_load)''')
```

Example 3-4 shows the second part. Here, the code starts with the .persist() invocation that prevents the query from being executed twice. Next, it applies a filter on top of the cached dataset to write correctly and incorrectly transformed results in two different places.

Example 3-4. Dead-Letter pattern for error-safe transformations: the writing

```
devices_to_load_with_validity_flag.persist()


(devices_to_load_with_validity_flag.filter('is_valid IS TRUE')
.drop('is_valid').write.mode('overwrite')
.format('delta').save(f'{base_dir}/output/devices-table'))


(devices_to_load_with_validity_flag.filter('is_valid IS FALSE')
.drop('is_valid').write.mode('overwrite')
.format('delta').save(f'{base_dir}/output/devices-dead-letter-table'))
```

This example shows that implementing the pattern without explicit failures in declarative languages like SQL is challenging. After all, you need to write a custom try-catch logic, and in the end, you get a very verbose query that might be hard to maintain over time.

Duplicated Records

Capturing unprocessable records is only the first step in the data error management quest. The next part concerns delivery semantics. It's very rare to see records delivered exactly once because achieving this outcome is very challenging in distributed systems. More often, you'll work with a more relaxed environment where records can arrive at least once. That's fine, but what if your data logic must process each occurrence only once?

Pattern: Windowed Deduplicator

Data deduplication is the most common answer to ensuring that your data logic processes each occurrence only once. But how to do that for both batch and streaming pipelines? The key is to consider the data to be limited. For streaming jobs, the limits will be time-based windows, while the batch jobs will reduce the scope to the currently processed dataset.

Automatic Retries

Exactly-once processing works only if you don't encounter runtime errors. Otherwise, the restarted job execution may reprocess already processed records, despite the deduplication logic. This is often an accepted trade-off between automated transient errror managent and deduplication.

Problem

Your batch job needs to process visit events synchronized from the streaming layer to an object store. The job exposes the data directly to the business users, and hence, it must guarantee exactly-once processing for each distinct record. The problem is, the streaming layer often has duplicated events due to the automatic retries of the data producers.

Solution

Duplicated data can often lead to inconsistent results and mislead end users. If you want to avoid degrading the quality of the dataset, use the Windowed Deduplicator pattern.

The first step requires identifying the deduplication attributes that guarantee the uniqueness of each record. Once you get them, you need to define the deduplication scope. For batch jobs, it'll often be the currently processed dataset. Even though it's possible to extend it by including datasets processed in the past, you must be aware that it will require more compute power and eventually be slower.

When it comes to streaming jobs, by definition, they work on an unbounded set of records. It's therefore difficult to reason in terms of completed datasets, like for batch workloads. Instead, the pattern simulates them by creating time-based windows where the job will retain already processed keys composed of deduplication attributes.

Windows in Batch

Batch processing doesn't imply an explicit window that you might define from code. Instead, it relies on an implicit global window which encompasses the whole processed dataset. The pattern's name doesn't relate to the underlying implementation details but to this dataset consideration in terms of windows.

Regarding code implementation, to eliminate duplicates, batch jobs will either use a DISTINCT expression or a WINDOW function alongside the condition on the row_number(). Streaming jobs will be different because they will need to store already processed records for the deduplication window duration. This involves then keeping some state about the past, thus the name of this store, the *state store*. Consequently, the streaming logic will be more complex than for the batch systems. It will require interacting with the state store to verify whether a record has already been seen or not. The interaction is present in Figure 3-3.

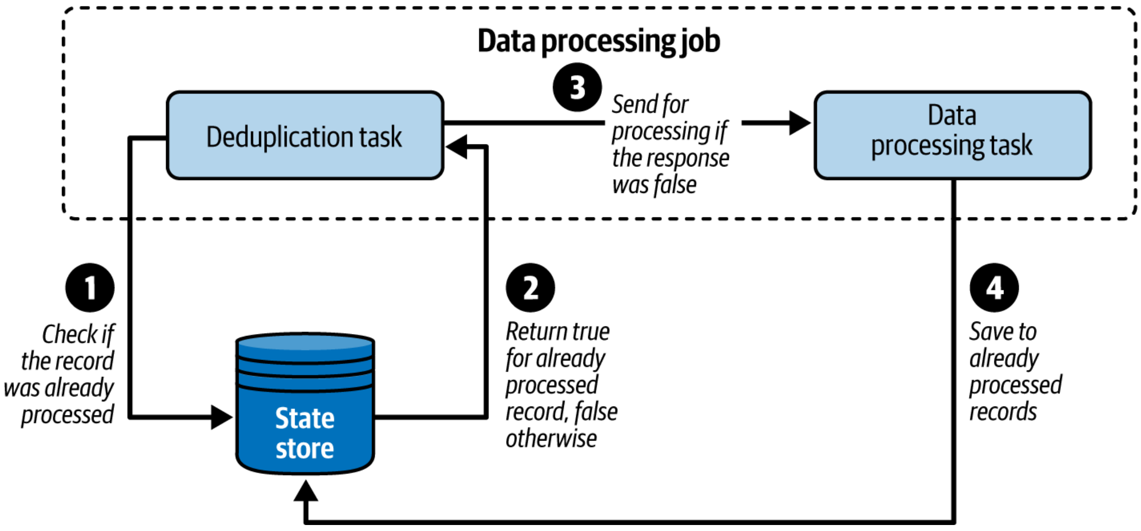

Data processing job
Deduplication task
3 Send for processing if the response was false
Data processing task
1 Check if the record was already processed
State store
2 Return true for already processed record, false otherwise
4 Save to already processed records

Figure 3-3. Windowed Deduplicator for a streaming job

State Stores

There are three different types of state stores. They're all trade-offs between performance and data consistency:

Local

Here, the state data lives only in memory. It's the fastest solution, but it might not be a good choice for production systems if you can't accept losing the state in case of failure.

Local with fault-tolerance

Here, the state still primarily lives in memory, which makes the access fast. But additionally, the job persists it to a remote storage for fault tolerance reasons. However, the persistence action has a cost in terms of processing time or consistency. If it occurs at each iteration, such as after updating a value in a window or microbatch, consistency should be fine but the job will be slower. In the opposite implementation, you will sacrifice consistency for time.

Remote

Here, the state is only present in a remote data store. Although it natively brings fault tolerance, it might negatively impact the latency and/or the overall cost of the pipeline.

Consequences

You may be surprised, but exactly-once processing doesn't guarantee exactly-once delivery or perfect deduplication. The next two points will shed some light on this.

Space versus time trade-off

This gotcha is valid for streaming pipelines because they're long-running applications on top of incremental datasets. Put differently, you don't have all the data at once, and you don't know if in a few minutes, you won't see any duplicates. For that reason, the implementation uses a time-based deduplication window and looks for duplicates only within the specified period.

As a consequence, a short window will probably miss some duplicates, but on the other hand, it will have a small impact on resources. If you extend the window duration, you'll need more resources since there will be more unique keys to persist and manage in the state store.

Idempotent producer

Correctly deduplicating the data doesn't guarantee exactly-once delivery for processed records. Very often, it will not be possible because of transient errors and their automatic solutions, such as retries. You should be aware of that and look for an idempotency pattern from Chapter 4 if you want to ensure exactly-once delivery.

Examples

Let's see how to first implement the pattern with batch pipelines. Apache Spark provides a dropDuplicates function that automatically takes care of duplicates. The function from Example 3-5 deduplicates the records, with the columns list defined in the parameter. If the parameter is missing, it'll use all the columns from the schema.

Example 3-5. Deduplication with dropDuplicates

```
dataset = (session.read.schema('...').format('json').load(f'{base_dir}/input'))


deduplicated = dataset.dropDuplicates(['type', 'full_name', 'version'])
```

However, that kind of native deduplication is not widely available. If you rely on a data processing framework, you'll probably have it. If not, you'll need to design the solution on your own. Thankfully, you can easily leverage grouping for that. In plain SQL, you can combine it with a WINDOW function. The WINDOW-based deduplication shown in Example 3-6 groups all rows by the columns from the PARTITION BY expression and filters out all but the first position (position = 1).

Example 3-6. Deduplication with a WINDOW **function**

```
SELECT type, full_name, version FROM (
 SELECT type, full_name, version,
   ROW_NUMBER() OVER (PARTITION BY type, full_name, version ORDER BY 1) AS position
 FROM duplicated_devices
) WHERE position = 1
```

This window-based approach will also work for streaming pipelines, but yet again, your data processing framework may help you by providing a high-level abstraction for deduplication. This is the case with Apache Spark, where the dropDuplicates function is also available for streaming data sources. Example 3-7 demonstrates how to declare the dropDuplicates function in that context. First, you need to interpret your input data and extract a time-based column. In our example, it'll be visit_time.

Example 3-7. Deduplication with dropDuplicates **in streaming data preparation**

```
event_schema = StructType([StructField("visit_id", StringType()),
  StructField("visit_time", TimestampType())])
deduplicated_visits = (input
  .select(F.from_json("value", event_schema).alias("value_struct"), "value")
  .select("value_struct.visit_time", "value_struct.visit_id", "value")
# see part 2...
```

After preparing the input dataset, you need to configure how long the job will remember already seen records. Example 3-8 shows how to configure it with a feature called a watermark on top of the visit_time column. The *watermark* has two responsibilities in our job. First, it defines the late data arrival boundary, meaning that records that are older than the current watermark value will not integrate into the pipeline. Since it's a building block of the Late Data Detector pattern described in the next section, I won't detail it here. Second, the watermark in the deduplication context also controls how long the job remembers the given key. Once again, all remembered entries older than the watermark will be automatically removed. Keep in mind that the streaming jobs are long running and that having that kind of control prevents their states from growing indefinitely.

Example 3-8. Deduplication with dropDuplicates **in streaming data expiration**

```
# ...part 2

.withWatermark("visit_time", "10 minutes")

.dropDuplicates(["visit_id", "visit_time"])

.drop("visit_time", "visit_id"))
```

Late Data

If you think that you have seen the worst errors with unprocessable and duplicated records, beware because you haven't heard about late data! Although it sounds very innocent, it has a serious impact on your data pipelines.

Pattern: Late Data Detector

In the context of late data, the first aspect you have to deal with is late data detection. It can help in many situations, such as completing already processed partitions or controlling the state in stateful jobs, as you saw before for deduplication in stream processing.

Problem

Most of the time, visitors to your blogging platform (as shown back in Figure 1-1 from Chapter 1) generate visit events in near real time so that your system can get them within 15 seconds of their creation. However, sometimes the users lose their network connection, and as a result, they buffer the visits locally before flushing them once the connection is restored. Your data processing jobs should detect these late events in order to apply a dedicated strategy for each use case, such as ignoring them.

Solution

The first step when dealing with data arrival issues is their detection with the Late Data Detector pattern. Since the problem is related to the time, the pattern requires defining one time-based attribute to track late data. The attribute should describe when a given event happened. Otherwise, it might be impossible to classify the incoming records as being late or on time.

Event and Processing Time

Data processing has two time concepts: event time and processing time. The event time indicates when a given action happened, while the processing time indicates when the data pipeline interacted with it. Naturally, the processing time will never be late.

In the next step, you need to define a latency aggregation strategy that will apply individually to each partition in your input data store. To avoid a situation in which your processing layer doesn't move on, the latency aggregation strategy must be monotonically increasing. Put differently, the tracked event time must move forward and can never go back. For that reason, the most common aggregation strategy uses the MAX function, taking the greatest event time for each partition. Using the MIN function here would lead to a stuck-in-the-past situation in which your job may never move forward.

After defining the partition-based logic, you need to decide on an additional aggregation strategy that will calculate a single event time for all partitions to represent overall progress.

Unlike in the previous step, where the MAX function is recommended to ensure monotonicity, for this global event time, you can opt to use the following:

- The MIN function if your job needs to follow the slowest upstream dependency. This approach guarantees you're going to consider more data as being on time. However, if your processing logic performs some buffering based on the event time, you'll buffer more since the event time follows the slowest partition.
- The MAX function that follows the fastest upstream dependency. This approach risks skipping records coming from the slowest dependencies, but on the other hand, it reduces the buffer size and thus its storage pressure.
- The MIN and MAX combined at different levels. This approach is possible only if you interact with multiple partitioned data sources. In that case, besides the first aggregation on top of each individual data source, you'll have another aggregation on top of all data sources. You can decide to apply the MIN function to each source and MAX to all sources, or the opposite.

Figure 3-4 shows how to apply the MIN and MAX functions on top of a single data source where each partition returns the MAX event time seen so far. At the bottom of the schema, you can see an example of the combined approach where each data source applies a different aggregation strategy, and in the end, the overall progress is represented with the MIN function.

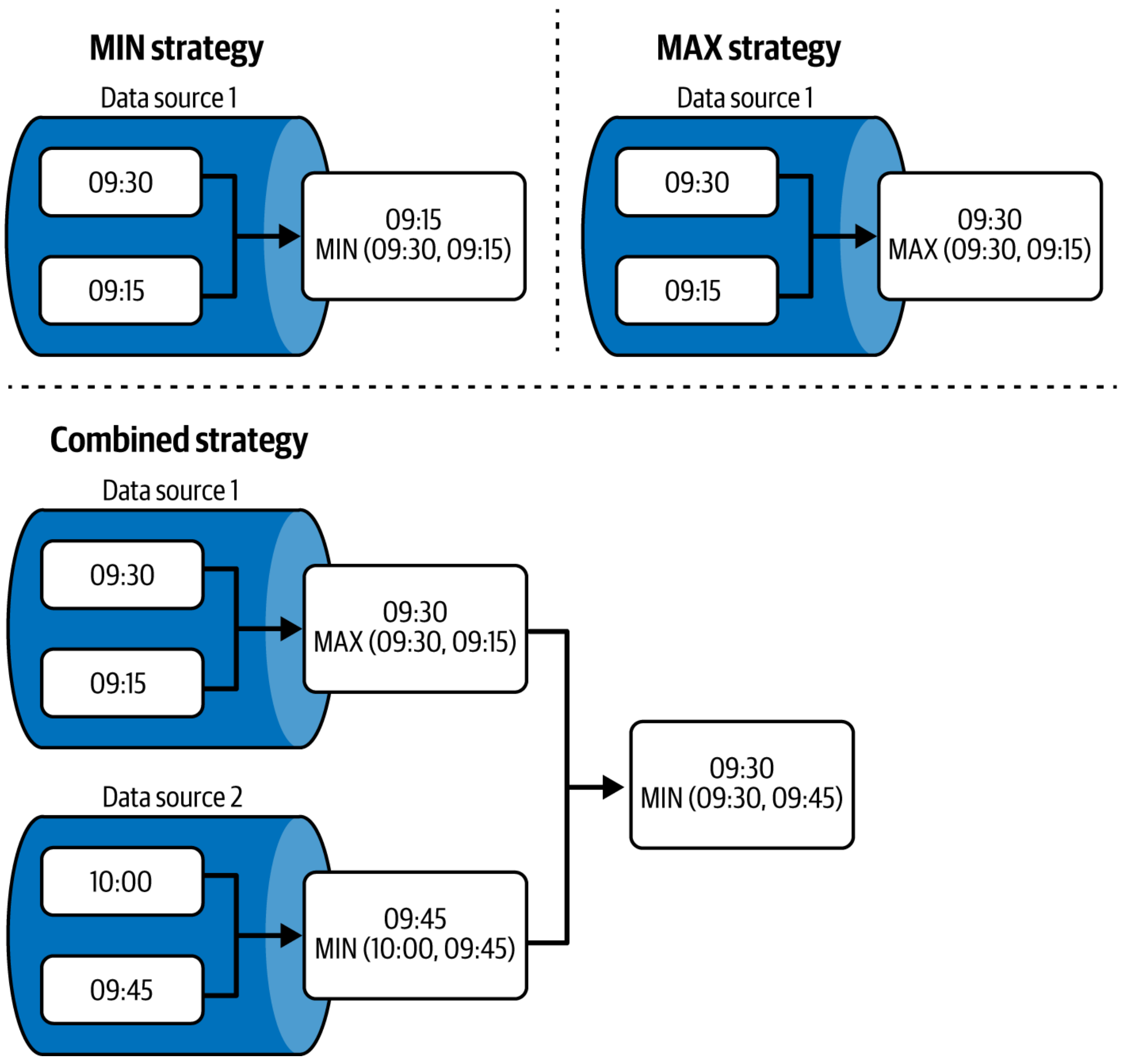


Figure 3-4. Different aggregation strategies applied to a job processing with a single partitioned data source and two partitioned data sources

But the implementation doesn't stop there. Using event time alone would mean you won't accept any data producer issues, such as lost connectivity or a slower network. Unfortunately, this will rarely happen, and your data producers may encounter some difficulties with punctuality in delivering data on the go. For that reason, to allow some extra unexpected latency, the pattern requires an allowed lateness attribute. The Late Data Detector pattern subtracts the allowed lateness value from the workflow's tracked event time as MAX(event time) - allowed lateness. The result of this calculation is called the *watermark*, and it defines the minimum event time to consider an event as on time.

To understand the watermark and the overall Late Data Detection pattern better, let's look at an example. Table 3-1 illustrates data flowing to a streaming system. As you'll notice, in the first row, there is no prior observation to define the watermark. Consequently, the pipeline accepts all the records and defines a new output watermark. The output watermark, by the way, is the same as the watermark candidate here.

This is not the case in the second row, where new data generates a watermark candidate. However, the candidate doesn't need to be taken into account. Simply speaking, if the candidate value is lower than the current input watermark, it's ignored. Otherwise, it would mean that the job, instead of moving forward, will be moving back.

Event times	Input watermark	Watermark candidate	Output watermark	Ignored records
10:00, 10:05, 10:06	-	MAX(10:00, 10:05, 10:06) – 30' = 9:36	MAX(9:36) = 9:36	-
9:20, 9:31, 10:07	9:36	MAX(10:07) – 30' = 09:37	MAX(9:36, 9:37) = 9:37	9:20, 9:31

Table 3-1. Watermark of 30 minutes

What to do once you detect an event as being late? The simplest thing to do is ignore it. However, if even the late records are valuable data assets, you'll need to write them to the system with the Late Data Integrator pattern presented next.

Consequences

Even though the pattern only detects late data, it has some implementation gotchas.

Late data capture

Some of the data processing frameworks either don't support late data detection or don't support late data capture. For example, Apache Spark Structured Streaming has a built-in capability to detect and ignore late events, but it doesn't expose an API to capture them easily. That's not the case with Apache Flink, which provides more flexibility for both capturing and detecting late events.

MIN strategy, stuck-in-the-past situations, and stateful jobs

You learned about this very quickly in the previous section. The partition-based event time tracker doesn't use the MIN function in order to avoid a stuck-in-the-past situation. Let's take a look at an example of a stateful job to help us understand why it's problematic and why even though using this function would technically be possible, you should avoid doing so.

A stateful job accumulates events for a streaming job in a state store, as you discovered in the "Pattern: Windowed Deduplicator". For example, it could count the number of visits for a user visiting a website. The challenge with this stateful accumulation is to know when to stop. Put differently, you should define when the accumulated state is complete. Often, you'll use the current watermark for that. Thus, by emitting each counter, you'll say, "For this user, there shouldn't be more new data."

If you used the MIN strategy to track partition event times, it would imply the following consequences:

Open-close-open infinite loop

This is the semantic implication for the workloads storing some event time–based state. Let's imagine that your watermark moved to 10:30, and you decided to emit all accumulated states older than this time. However, after 10 minutes, you integrated some late records that moved the watermark back to 9:00. Therefore, you will have to reopen all states that are already emitted and thus considered completed.

Stuck in the past

This is the most serious implication. If your pipeline is getting late data over and over again, the watermark may never make any progress. Consequently, your eventual event time–based state will grow because you will not be able to determine the buffered items as completed with regard to the watermark.

To help you avoid the issues from the previous list, the Late Data Detector should be monotonically increasing, which means it should never decrease as is the case with the MIN function. That's why, although technically possible, this function is not commonly used for tracking event time at the partition level.

Max strategy and event skew

The max-based strategy also has a gotcha. In highly skewed environments, it can be too aggressive and consequently drop many records.

Let's imagine the following example with multiple data sources. Four out of five data sources for our pipeline encounter some network issues, and they deliver the data 40 minutes later than the single fully working source. Since the watermark is based on the MAX function, records coming from the late data sources will be considered late, and the consumer will miss them.

Unfortunately, there is no silver bullet for this issue. The best mitigation strategy should rely on appropriate late events monitoring and the possibility of reintegrating late records whenever there is a high event skew. The next two patterns, Static Late Data Integrator and Dynamic Late Data Integrator, will address the replaying aspect.

Examples

The Late Data Detector pattern is mostly present in stream processing. For that reason, let's take a look at the implementation of two major frameworks from that area. To start with, let's use an example from Apache Spark Structured Streaming.

Example 3-9 shows late data detection in Structured Streaming with the withWatermark function. As you can see, it accepts two parameters, one that defines the event time attribute for time tracking and another for the allowed lateness. In our example, the configuration means that the job accepts data that's up to one hour late.

Example 3-9. Late data detection in a stateful job

```
visits_events = (input_data.selectExpr('CAST(value AS STRING)')
 .select(F.from_json('value', 'visit_id INT, event_time TIMESTAMP, page STRING')
 .alias('visit')).selectExpr('visit.*'))


session_window: DataFrame = (visits_events
 .withWatermark('event_time', '1 hour')
 .groupBy(F.window(F.col('event_time'), '10 minutes')).count())
```

To understand what the code is doing, let’s analyze Table 3-2. It shows the buffered and emitted windows for each event time received with incoming records. As you can see, the job ignores the event of 01:50 as it happened earlier than the current watermark (02:15). At the same time, the two windows for three o’clock are pending in the state store. The job emits them only after processing the record from 04:31 since it’s the first time the watermark can move on.

Event time w/o seconds	**Current to new watermark (w/o seconds)**	**Buffered windows**	**Emitted windows**
03:15	1970-01-01T00:00 to 2023-06-30T02:15	[03:10-03:20]	[]
03:00	02:15 to 02:15	[03:00-03:10, 03:10-03:20]	[]
01:50	02:15 to 02:15	[03:00-03:10, 03:10-03:20]	[]
03:11	02:15 to 02:15	[03:00-03:10, 03:10-03:20]	[]
04:31	02:15 to 03:31[a]	[04:30-04:40]	[03:00-03:10, 03:10-03:20]

[a] Technically, Apache Spark Structured Streaming rounds the watermark up to the upper bound of the window. The example here uses a simplified version to facilitate understanding.

Event time w/o seconds	Current to new watermark (w/o seconds)	Buffered windows	Emitted windows

Table 3-2. The impact of the watermark on the late data for the event time of 2023-06-30

Apache Spark Structured Streaming handles all late data on your behalf. This is great as it means you don't need to worry about it. However, in some scenarios, you may need to capture the late records to, for example, write them into a separate storage for reprocessing or further analysis. Although it can be hard to achieve with Apache Spark, it's relatively simple in Apache Flink, which gives access to the current watermark value from the execution context (see Example 3-10).

The first step is to create a timestamp assigner instance that will extract the event time value from the incoming records. In our example, this is the role of the VisitTimestampAssigner. Next, we also have to declare a watermark-aware data processor function. In our example, it's represented by the VisitLateDataProcessor class. As you can see in Example 3-10, the processor compares the extracted event time with the current watermark, and depending on the outcome, it writes the record to a different storage.

Example 3-10. Timestamp assigner and records processor in Apache Flink

```
class VisitTimestampAssigner(TimestampAssigner):
 def extract_timestamp(self, value: Any, record_timestamp: int) -> int:
  event = json.loads(value)
  event_time = datetime.datetime.fromisoformat(event['event_time'])
  return int(event_time.timestamp())


class VisitLateDataProcessor(ProcessFunction):


 def __init__(self, late_data_output: OutputTag):
  self.late_data_output = late_data_output


 def process_element(self, value: Visit, ctx: 'ProcessFunction.Context'):
  current_watermark = ctx.timer_service().current_watermark()
  if current_watermark > value.event_time:
    yield (self.late_data_output, json.dumps(VisitWithStatus(visit=value,
     is_late=True).to_dict()))
  else:
    yield json.dumps(VisitWithStatus(visit=value, is_late=False).to_dict())
```

After declaring the late data handling logic, we need to integrate it with the data processing job. That's the part shown in Example 3-11. You can see there that we allow records to be late (out of order) by at most five seconds. Later comes the processing logic, which is not relevant here, and finally, the side output definition. A *side output* in Apache Flink is a structure you can use to send records to a different storage location than the one configured mainly for the pipeline.

Example 3-11. Using a timestamp assigner and data processor in Apache Flink

```
watermark_strategy = (WatermarkStrategy
 .for_bounded_out_of_orderness(Duration.of_seconds(5))
 .with_timestamp_assigner(VisitTimestampAssigner()))


data_source = env.from_source(source=kafka_source,
 watermark_strategy=watermark_strategy, source_name="Kafka Source"
).uid("Kafka Source").assign_timestamps_and_watermarks(watermark_strategy)


late_data_output: OutputTag = OutputTag('late_events', Types.STRING())
visits: DataStream = (data_source.map(map_json_to_visit)
 .process(VisitLateDataProcessor(late_data_output), Types.STRING()))
kafka_sink_valid_data: KafkaSink = ...
kafka_sink_late_visits: KafkaSink = ...


visits.get_side_output(late_data_output).sink_to(kafka_sink_late_visits)
visits.sink_to(kafka_sink_valid_data)
```

Pattern: Static Late Data Integrator

You already know that you can ignore late data. It'll make your life easy as neither you nor the downstream consumers will need to backfill the jobs impacted by the late data. However, late data may also be valuable, and if it represents a significant percentage of your dataset, losing it won't be an option. If you were an ecommerce store, you wouldn't want to miss half of your orders, would you? That's only one scenario where you'll need to integrate late data after capturing it.

Problem

One of your daily jobs generates various statistics from the websites that refer your blog posts. The statistical results are considered to be approximate for 15 days because that's the maximum delay allowed to integrate late data. Records older than 15 days are skipped.

Your batch only processes the current day and consequently ignores any late data that's not older than the allowed 15 days. You would like to adapt the job and include late data as part of the daily pipeline without having to run 15 individual jobs separately each day.

Solution

A fixed delay for late data ingestion is a perfect scenario where you can leverage the Static Late Data Integrator pattern.

Easy Solution for You, but Not for Others

In fact, the easiest solution to the problem is using processing time–based partitions. However, if you do care about the event time somewhere in your system, using the processing time solution simply moves the problem somewhere else. Let's look at an example to help us understand this better.

Let's imagine that your processing time partition for nine o'clock has the following distribution: 80% of the data for nine o'clock, 10% for eight o'clock, and 10% for seven o'clock. One of the downstream consumers uses event time–based partitions. Hence, even though your pipeline doesn't need to deal with late data, it generates late data that will need to be handled by other processes in the system (see Figure 3-5).

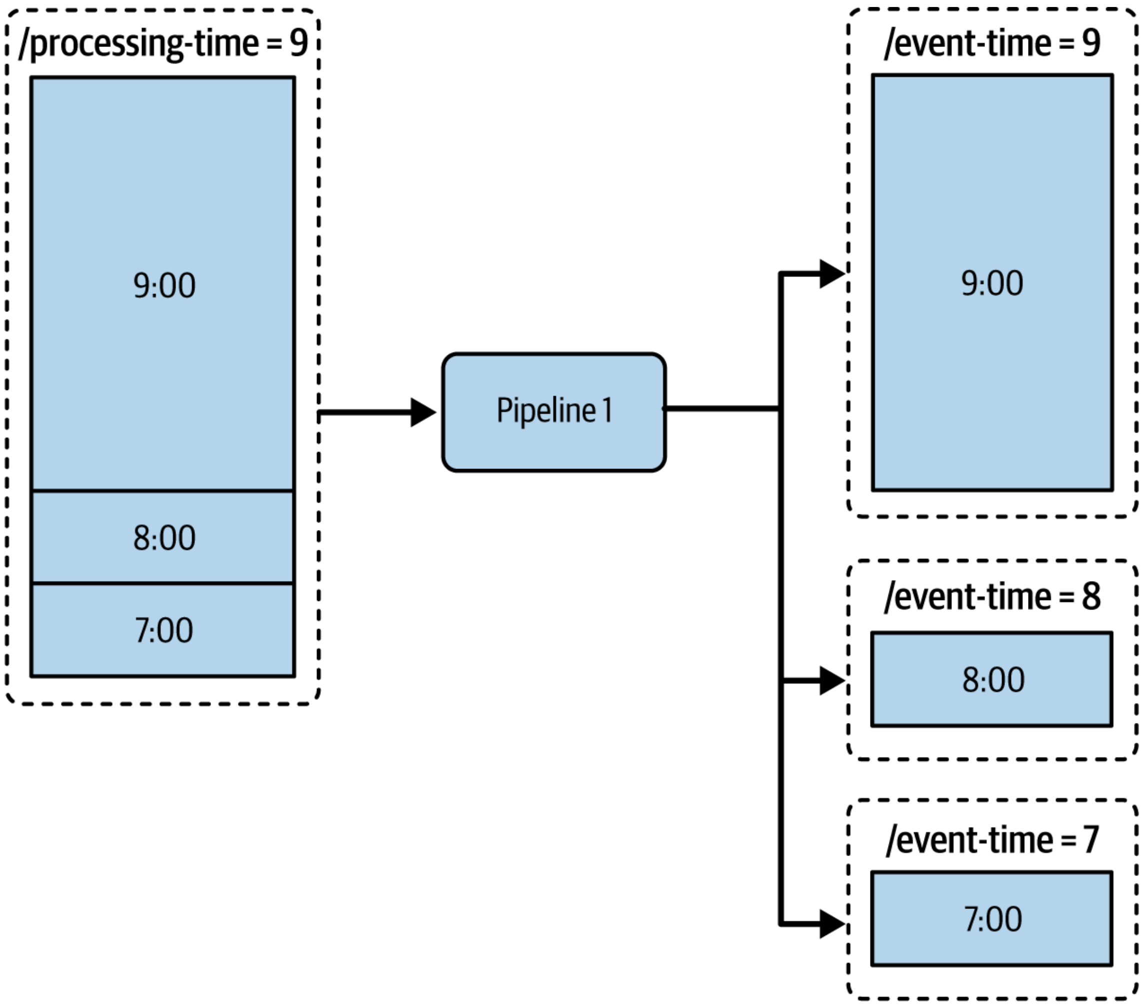


Figure 3-5. Shifting the late data problem

You start the implementation by defining a so-called *static lookback window* (i.e., how far to look back in the past for late data in a given job execution). It's called static because the duration will never change, even though your dataset will receive some late data after the allowed window duration. For example, if your pipeline is about to execute on the dataset from 2024-12-31 and your lookback window is set to 14 days, the current run will reprocess past partitions between 2024-12-17 and 2024-12-30 in addition to the current day. If, for whatever reason, you get some late data for 2024-12-15, it will be ignored.

After defining the lookback window, you need to place the late data integration process in your pipeline. Figure 3-6 shows three different configurations.

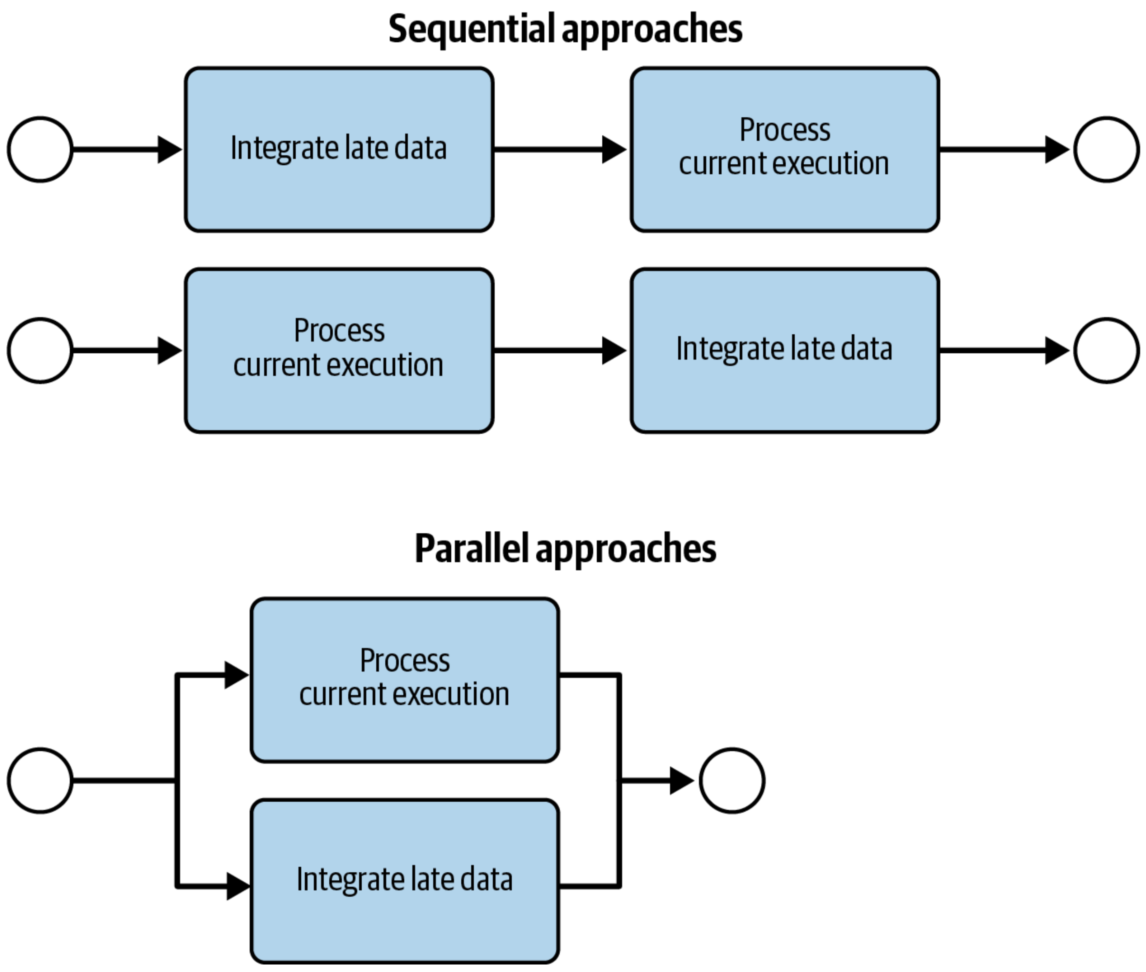


Figure 3-6. Strategies to include late data integration in pipelines

Which strategy you should choose? There is no one-size-fits-all solution, and your choice will depend on the category of the data processing job and the delivery constraints. Here are some hints to help you make a choice:

- For stateful pipelines where the results generated by one execution depend on the results generated by the previous executions, you should opt for the sequential strategy where the late data ingestion is handled as the first step. After all, you need a valid history to generate the current dataset.
- For stateless pipelines, you can use and switch between all three strategies. But if you want to deliver current data first, you should opt for either the second or the third approach, in which late data is handled at the same time or after the current execution time.

Consequences

Although the pattern introduces some data correctness fixes, it does so with an increased complexity cost.

Snowball backfilling effect

The biggest problem you may encounter here is the snowball effect. If you are a data provider and your data consumers care about consistency, they'll inevitably need to replay all partitions with the late data, just as you have done. If they have consumers too, those consumers will also need to run backfilling for these partitions...and in the end, the whole operation may become very compute-intensive. Unfortunately, there is not much you can do here besides notifying your direct consumers of all backfilling actions and letting them decide what to do on their end.

Overlapping executions and backfilling

Due to the static nature of the lookback window, you shouldn't backfill your jobs as you would backfill jobs without the static lookback window. To understand this better, let's take a look at an example of a pipeline with a four-day lookback window that already executed on 2024-10-10, 2024-10-11, and 2024-10-12.

If you replay all three executions, you will generate overlapping runs (see Table 3-3).

Execution date	Executed dates
2024-10-10	2024-10-09, 2024-10-08, 2024-10-07, 2024-10-06
2024-10-11	2024-10-10, 2024-10-09, 2024-10-08, 2024-10-07
2024-10-12	2024-10-11, 2024-10-10, 2024-10-09, 2024-10-08

Table 3-3. Overlapped backfilling examples

For that reason, before starting a backfill, you need to take the lookback window duration into account. As a result, in the previous example, it would be enough to restart only the 2024-10-12 execution.

Pipeline trigger

With the Static Late Data Integrator, your backfilling jobs must be part of the main pipeline. You can't start separated pipelines as part of the lookback window–based backfilling because it'll lead to the same problem explained in the section on overlapping executions and backfilling.

Figure 3-7 depicts both valid and invalid approaches.

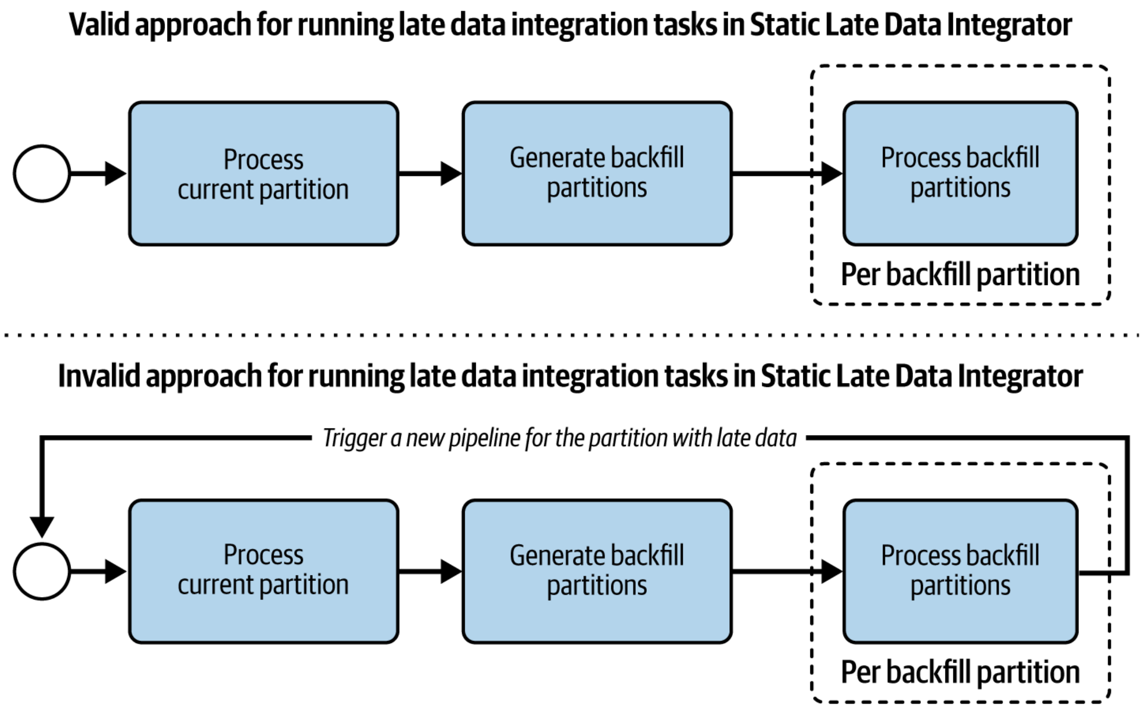

Valid approach for running late data integration tasks in Static Late Data Integrator
Process current partition
Generate backfill partitions
Process backfill partitions
Per backfill partition
Invalid approach for running late data integration tasks in Static Late Data Integrator
Trigger a new pipeline for the partition with late data
Process current partition
Generate backfill partitions
Process backfill partitions
Per backfill partition

Figure 3-7. Valid and invalid approaches for integrating late data in the Static Late Data Integrator pattern

Waste of resources

Fixed periods from the lookback window may not contain late data every time. If you are worried about running the job for a partition without new late data to integrate, you can add a control task to run the integration task only when there is late data, or you can use the Dynamic Late Data Integrator pattern.

Time requirement

If your dataset is not partitioned by time or doesn't have any time concept, you cannot really detect and thus integrate late data. Time partitions from the Static Late Data Integrator pattern are time boundaries that each incoming record is comparing against.

Examples

Let's see how to implement the Static Late Data Integrator pattern in Apache Airflow with a feature called Dynamic Task Mapping. In a nutshell, the feature lets you create tasks dynamically from a data provider function. Since the execution time will change with each new run, Dynamic Task Mapping is a perfect fit for generating late data integration tasks for the static lookback window duration.

The pipeline from our example, which is fully available on GitHub, copies files from an input location to an output location. The workflow runs daily, and after copying the file for the current day, the pipeline backfills two previous dates by subtracting the lookback window size from the execution date (see Example 3-12).

Example 3-12. Generation of backfilling tasks with a static lookback window of two days

```
@task
```

```
def generate_backfilling_runs():
 dr: DagRun = get_current_context()['dag_run']
 backfilling_dates = []
 days_to_backfill = 2
 start_date_to_backfill = (dr.execution_date
  datetime.timedelta(days=days_to_backfill))
 for days_to_add in range(0, days_to_backfill):
  date_to_backfill = start_date_to_backfill + datetime.timedelta(days=days_to_add)
  backfilling_dates.append(date_to_backfill.date().strftime('%Y-%m-%d'))
 return backfilling_dates
```

So generated backfilling_dates are later passed to the integrate_late_data task. Under the hood, thanks to the Dynamic Task Mapping expressed in Example 3-13 with the expand(...) method, Apache Airflow will create one integration task for each date. As you'll notice, other than the expand call, the code doesn't differ a lot from the code you would write without the late data integration.

Example 3-13. Generating tasks for each of the backfilled dates from an expand **method**

```
@task
def integrate_late_data(late_date: str):
 copy_file(late_date)


# ....
integrate_late_data.expand(late_date=generate_backfilling_runs())
```

Dynamically created tasks integrate with regular ones. In our case, we're loading the current day before backfilling two previous dates, which gives us the workflow in Example 3-14.

Example 3-14. Workflow with tasks created dynamically

```
backfilling_runs_generator = generate_backfilling_runs()
(file_to_load_sensor >> load_current_file() >> backfilling_runs_generator >>
  integrate_late_data.expand(late_date=backfilling_runs_generator))
```

Pattern: Dynamic Late Data Integrator

The Static Late Data Integrator is a simple form of late data inclusion because it relies on some fixed period of time. However, having a static tolerance period is not always possible, and sometimes you may need a more dynamic approach that will just load the partitions impacted by the late data.

Problem

The Static Late Data Integrator pattern that you implemented to handle the last 15 days of data is not enough anymore. Your product owner wants to enrich the statistics with all late data, even beyond the initial 15-day window. Now you need to adapt your pipeline to this new business requirement and avoid blindly replaying only the two previous weeks.

Solution

To handle variability and integrate only the partitions with late data, you can use the Dynamic Late Data Integrator pattern.

The implementation leverages a lookback window that is dynamic, which means that all the backfilled partitions really contain late data. To make this happen, the dynamic approach requires an additional data structure to store the last execution time, and eventually,**1** the last update time for each partition. You can find an example of this structure in Table 3-4.

Partition	**Last processed time**	**Last update time**
2024-12-17	2024-12-17T10:20	2024-12-17T03:00
2024-12-18	2024-12-18T09:55	2024-12-20T10:12

Table 3-4. Data structure (aka state table) for Dynamic Late Data Integrator

Based on this table, you can perform a query to get the partitions to backfill (see Example 3-15).

Example 3-15. Query to get the list of late partitions

```
SELECT partition FROM state_table WHERE

`Last update time` > `Last processed time` AND `Partition` < `Processed partition`
```

However, adding the state table is not enough. You also need to define a place in your pipeline where you will run the query. Typically, after you successfully process your data, you will need to update the last processed time (see Figure 3-8).

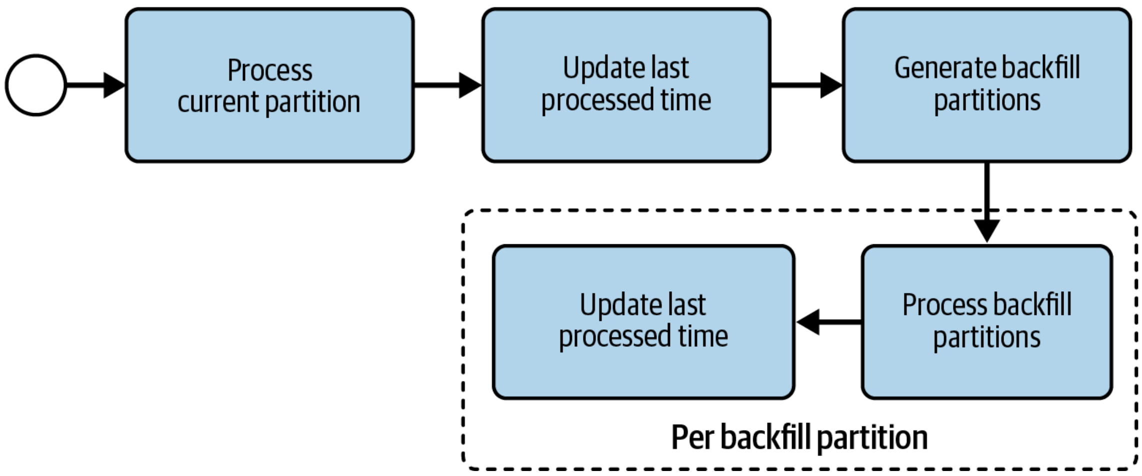

Process current partition
Update last processed time
Generate backfill partitions
Process backfill partitions
Update last processed time
Per backfill partition

Figure 3-8. Interaction with state table integrated with the main pipeline

Even though you've created the state table and integrated it into the pipeline, one question remains: how can you get the last update time for the partitions? Some data stores provide this fine-grained level of detail out of the box. In this category, you'll find BigQuery, which exposes an INFORMATION_SCHEMA.PARTITIONS view with the last_modified_timestamp attribute for each partition, or Apache Iceberg, which includes a last_updated_at column for the partitions metadata table. For the data stores without this information, you'll need to find a way to generate it from existing data, as we demonstrate in the code in the Examples section.

Not Whole Partitions

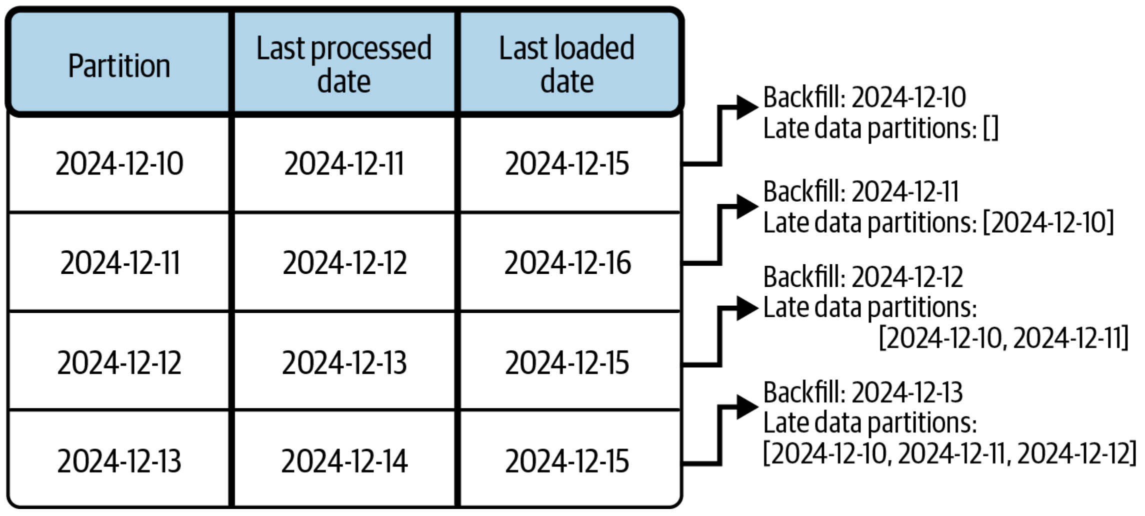
If you can isolate the entities impacted by the late data, you don't need to backfill full partitions. Instead, you can only overwrite the data generated for the impacted entities. This approach optimizes resources usage but also is more challenging to implement.

Consequences

The Dynamic Late Data Integrator pattern solves the resources waste issue and fixes the lookback window, but it also has its own shortcomings.

Concurrency

If your pipeline supports concurrent executions, dynamic late data integration may generate duplicated late data integration runs. Figure 3-9 shows what could happen in a pipeline running four different jobs in parallel, with late data in each processed partition. As you'll notice, partitions for 2024-12-10 and 2024-12-11 would be executed more than once.



Partition	Last processed date	Last loaded date
2024-12-10	2024-12-11	2024-12-15
2024-12-11	2024-12-12	2024-12-16
2024-12-12	2024-12-13	2024-12-15
2024-12-13	2024-12-14	2024-12-15

Backfill: 2024-12-10
Late data partitions: []
Backfill: 2024-12-11
Late data partitions: [2024-12-10]
Backfill: 2024-12-12
Late data partitions: [2024-12-10, 2024-12-11]
Backfill: 2024-12-13
Late data partitions: [2024-12-10, 2024-12-11, 2024-12-12]

Figure 3-9. Late data integration concurrency problem

To avoid this issue, you need to add an extra column to the state table that will keep the partition status either as already processed or as being processed. Consequently, the query retrieving the partitions to backfill should add this column as an extra filtering condition to ignore the partitions already planned for late data integration. The new query looks like the one in Example 3-16.

Example 3-16. Query to get the partitions to backfill in the concurrent Late Data Integrator pattern

```
SELECT partition FROM state_table WHERE
`Last update time` > `Last processed time` AND
`Partition` < `Processed partition` AND
`Is processed` = false
```

In addition to this query change, you will need to adapt the pipeline with the following adjustments:

- Each pipeline needs to start with the task that updates the Is processed column of the currently processed partition. That way, you can avoid having the next execution generate the current partition as the one to backfill. Also, this task should run only if the execution of the previous run succeeded. In our example, this task for 2024-12-11 would start only if the same task successfully completed for the run of 2024-12-10.

- The task that generates the partitions to backfill should now also update all retrieved partitions as having been processed. In addition, it should run only if its previous execution succeeded. This dependency on the past runs helps avoid race conditions and triggering the same partitions in two different runs.

- The task updating the last processed time should additionally set the Is processed flag to false. That way, if the partition gets new late data, it can still be replayed.

After applying all these changes, you can extend the execution mode, and instead of executing the late data integration tasks as part of the main pipeline, you can trigger complete late data integration pipelines. Both execution modes are depicted in Figure 3-10, where only the processes in the lighter boxes can run in parallel.

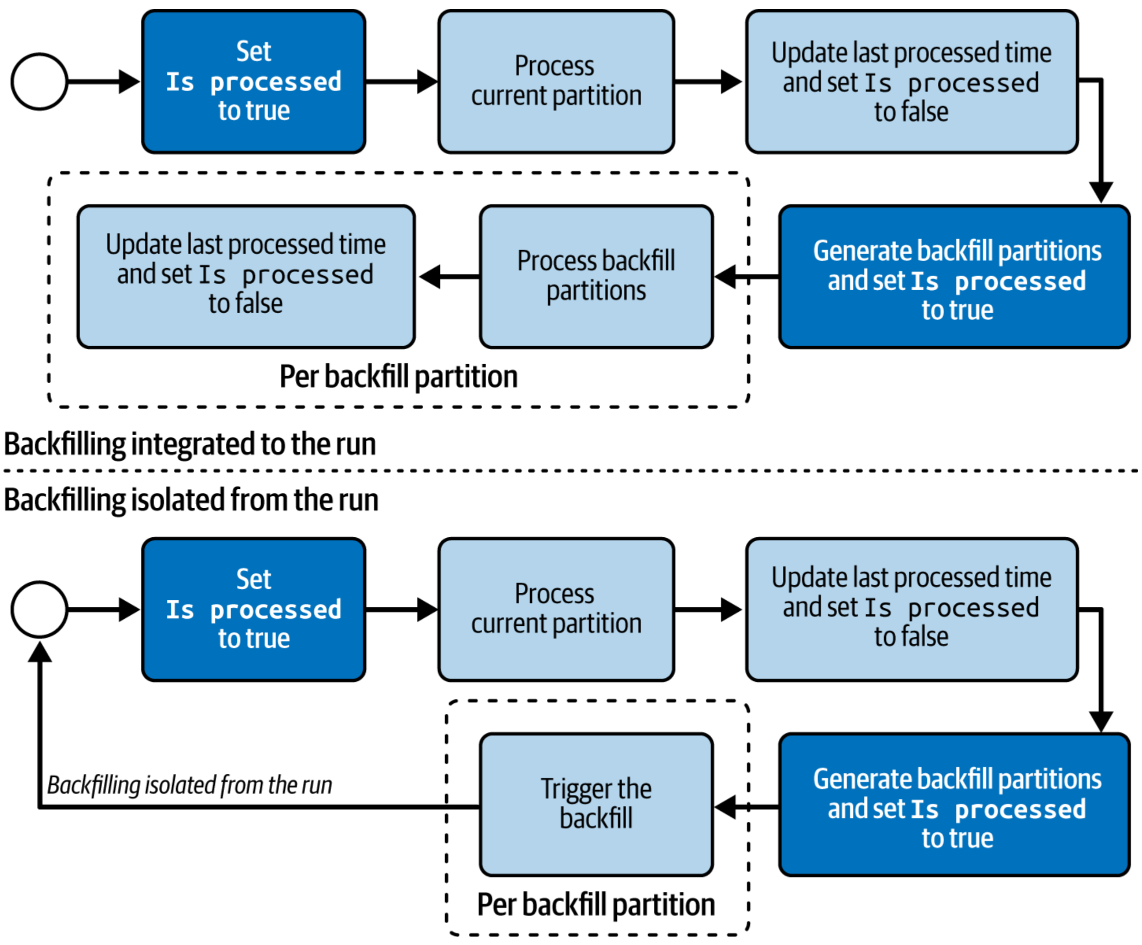

Set Is processed to true
Process current partition
Update last processed time and set Is processed to false
Update last processed time and set Is processed to false
Process backfill partitions
Generate backfill partitions and set Is processed to true
Per backfill partition
Backfilling integrated to the run
Backfilling isolated from the run
Set Is processed to true
Process current partition
Update last processed time and set Is processed to false
Backfilling isolated from the run
Trigger the backfill
Generate backfill partitions and set Is processed to true
Per backfill partition

Figure 3-10. Late Data Integrator for concurrent pipelines

It's worth noting that if the task that generates partitions to backfill fails, its future executions will not run due to the dependency on the previous run. Consequently, the pipeline will get stuck in a long in-progress state requiring your manual intervention to unblock it.

You'll find a complete example of a concurrent pipeline protected against duplicated executions in the GitHub repo.

Stateful pipelines and very late data

By addressing the issue of a fixed lookback window, the Dynamic Late Data Integrator pattern brings up another challenge which is exclusively related to stateful pipelines. Let's imagine a pipeline implementing the Incremental Sessionizer pattern with the last successful run having taken place on 2024-10-20. There hasn't been any late data so far, but the next day's execution spots late data ingested for the partition of 2024-09-21. Since your job is stateful, you will need to regenerate all executions from 2024-09-21 to 2024-10-20 to guarantee the correctness of your dataset.

As you can see, it might not be enough to consider a lookback window to be dynamic because this can lead to heavy backfills when you get very late records. If this is problematic, you will need to define the accepted lookback window even for dynamically created lookback windows.

Scheduling complexity

The dynamic lookback window version involves creating backfilling pipelines dynamically. Depending on your storage layer, getting the last modification time for each partition might not be easy. This step can involve dealing with the internal details of a storage technology or even implementing the update tracking table on your own.

Examples

As an example, we're going to use yet again the Dynamic Task Mapping of Apache Airflow. Since you discovered this feature in the Static Late Data Integrator pattern, I'll omit this part and directly cover the challenge you may face when the last update partition time is not natively available. Thankfully, in our example for Delta Lake, we can get the partition information after some coding effort to use exclusive Scala classes.

The Scala API for Delta Lake provides a DeltaLog class that has a getChanges method. The results are all the created or deleted files from the table version you specify. Once you get these files, you can simply get the latest Delta Lake version for each partition (see Example 3-17).

Example 3-17. Extracting modified partitions from a Delta Lake version

```
val deltaLog = DeltaLog.forTable(sparkSession, jobArguments.tableFullPath)

val partitionsChangeVersions: Iterator[(String, Long)] = deltaLog.getChanges(0)

 .flatMap {

 case (version, actions) => {

  val changedPartitionsInVersion: Set[String] = actions.map {

   case addFile: AddFile if addFile.dataChange =>

    Some(addFile.partitionValues.map(e => s"${e._1}=${e._2}").mkString("/"))

   case removeFile: RemoveFile if removeFile.dataChange =>

    Some(removeFile.partitionValues.map(e => s"${e._1}=${e._2}").mkString("/"))

   case _ => None}.filter(_.isDefined).map(_.get).toSet


  changedPartitionsInVersion.map(partition => (partition, version))

 }

}

val lastVersionForEachPartition: Map[String, Long] = partitionsChangeVersions

 .toSeq.groupBy(pair => pair._1).mapValues(pairs => pairs.map(_._2).max)
```

lastVersionForEachPartition

The next part consists of getting the partitions to backfill (see Example 3-18).

Example 3-18. Method to get partitions to backfill where the last processed version is lower than the last written version

```
val partitionsToBackfill = sparkSession.read.format("delta").load(TablePath)
 .select("partition", "isProcessed", "lastProcessedVersion").as[PartitionState]
 .filter(state => state.lastProcessedVersion.isDefined)
 .filter(state => {
  (!lastVersionPerPartition.contains(state.partition) ||
  lastVersionPerPartition(state.partition) > state.lastProcessedVersion.get) &&
  !state.isProcessed && state.partition < currentPartition
 })
 partitionsToBackfill.map(_.partition).collect()
```

As you'll notice, for Delta Lake, we reason in terms of table versions, which is a simpler concept than update time. Next, the job writes partitionsToBackfill as a timestamped file to avoid race conditions in case of concurrent executions. Consequently, each job execution will use its own late data integration file to create backfilling tasks dynamically via Dynamic Task Mapping and the function from Example 3-19.

Example 3-19. Reading a configuration file and triggering past executions

```
@task
def generate_backfilling_arguments():
  context = get_current_context()
  current_partition_date = context['ds']
  dag_run: DagRun = context['dag_run']
  dag_run_start_time: str = dag_run.start_date.isoformat()

  def _run_id_for_event_time(event_time: str) -> str:
   return f'backfill_{event_time}_from_{current_partition_date}_{dag_run_start_time}'

  configuration = read_backfilling_configuration(current_partition_date)
  return list(map(lambda partition: {
   'execution_date': partition['event_time'],
   'trigger_run_id': _run_id_for_event_time(partition['event_time'])
```

```
 }, configuration['partitions']))
```

If your job can run concurrently, you'll need to add dependency constraints on the previous execution. Therefore, some tasks in the pipeline, despite overall concurrency, will run sequentially.

Apache Airflow achieves this sequential execution with the task-level depends_on_past attribute. If it's set, the orchestrator blocks any execution for a given task as long as its previous execution doesn't succeed. In our case, it guarantees that there will only be one job generating partitions to backfill. Example 3-20 shows our pipeline with the default allowed concurrency and sequential execution enforced on the key tasks for the Dynamic Late Data Integrator pattern.

Example 3-20. A pipeline configured with the default concurrency and custom sequentiality on some tasks

```
with DAG('devices_loader', max_active_runs=5,
 default_args={'depends_on_past': False, ...},
) as dag:
 processing_marker = SparkKubernetesOperator(
  task_id='mark_partition_as_being_processed', depends_on_past=True # ...
 )
 backfill_creation_job = SparkKubernetesOperator(
  task_id='get_late_partitions_and_mark_them_as_being_processed',# ...
  depends_on_past=True
 )
 # ...
```

Filtering

An error in data engineering tasks does not always mean a technical failure. Errors also include human mistakes, like incorrectly implementing filters, which might lead to partial or bad data being exposed to end users.

Pattern: Filter Interceptor

One of the most common data operations is filtering. It lets you select only the records that are relevant to a given business use case. Despite this popularity, it's often hard to know what particular condition has filtered out most of the rows, which, as a result, would give you the ability to detect errors due to aggressive and possibly buggy filtering conditions. The pattern presented here provides more insight.

Problem

One of your batch jobs uses a distributed data processing framework. Recently, you released a new version to production and noticed a sudden spike of filtered data volume to 90% from 15%. You're wondering if the change comes from the data or from software regression, and you can’t find this out by simply looking at the execution plan. The framework performs many optimizations, and one of them collapses the filtering expressions into a single one.

Solution

Ideally, you should address this issue with physical query execution plan analysis. However, this might lack precision, especially when the engine performs some optimizations such as combining all filter conditions into one execution step. In that case, the plan will contain the number of filtered rows for all filters and not the statistics for each condition. The Filter Interceptor pattern overcomes that.

The implementation is relatively straightforward in data processing frameworks with a programmatic API. Instead of simply expressing your filtering condition, you must wrap it with a counter logic that you increment if the condition evaluates to true. At the end of the job execution, after completing your business logic, you must explicitly gather all filter counters.

The implementation requires a bit more effort in declarative languages like SQL. The solution here consists of using a subquery or a temporary table that exposes filtering conditions as columns. Let’s take an example of a query with the two filter expressions a IS NOT NULL and b != "abc". The implementation would include them as new columns storing the validation results (*a_is_not_null*, *b_is_not_abc*) in a subquery table. These columns would later be used in the main query as filtering predicates. As you can see, it’s not as easy as just wrapping the filtering logic with the programmatic API. If this is confusing, you’ll find an example in the Examples section.

Stay Pragmatic

Remember to use the right tool for the job. If the programmatic API is better for what you’re trying to do, use it, even though you have been writing only SQL queries so far! The opposite is true as well.

Consequences

The pattern gives you some extra insight into job execution but doesn’t do it for free.

Runtime impact

Wrapping the filtering condition will impact the job execution time and resources. However, the impact should be small. Counters from the implementation are rather simple data structures living locally in each task, and exchanging them across the network to get the final result shouldn’t be costly. The impact can be greater for the SQL example, where you might need to create a temporary table before executing the queries to correctly extract the filtering stats and really filter out the data before writing to another table.

Declarative languages

As you already know, sometimes it’s better to code even though you get used to working only with SQL. The Filter Interceptor is one of the examples where declarative languages are less powerful than the programmatic API. The programmatic API, besides providing some flexibility, helps you write code that is easier to grasp and maintain over time.

Streaming

The implementation is not only challenging for declarative languages but also for streaming jobs. Although it’s easier than for SQL, it may require transforming your stateless job into a stateful one, hence adding some additional state management overhead to count the number of filters applied so far for each input record.

Also, since streaming data is continuously arriving, you should define some time boundaries for the interceptor statistics. Otherwise, you may not be able to relate the statistics to the current time and may not know what filters are currently the most active in the queries. For example, the filtering statistics could rely on time-based processing windows so that you can get trends over time and have a single view of the whole job history.

Examples

The Solution section already gave you an idea of how to implement the pattern with the programmatic API of a data processing framework. Let’s now learn what it means to concretely implement it with Apache Spark SQL. Depending on your language, you can implement it with either filter programmatic functions (Scala API) or mapInPandas (PySpark). The former, because it’s easier, is present in the GitHub repo. Let’s then focus here on a more complex implementation.

To start with, we’re defining a class that wraps a filtering condition and an Apache Spark accumulator together as a FilterWithAccumulator class. The accumulator will increment at each false evaluation of the filter. You can find both declarations in Example 3-21.

Example 3-21. Filter Interceptor in PySpark: accumulators and filtering functions

```
@dataclasses.dataclass

class FilterWithAccumulator:

 name: str

 filter: Callable[[Any], bool]

 accumulator: Accumulator[int]


filters_with_accumulators = {

 'type': [

  FilterWithAccumulator('type is null', lambda device: device['type'] is not None,

   spark_context.accumulator(0)),

  FilterWithAccumulator('type is too short (1 chars or less)',

    lambda device: len(device['type']) > 1, spark_context.accumulator(0))

  ],

  # ...

}
```

Next comes Example 3-22 with the data processing step. As you can see, it relies on the mapInPandas transformation that calls a filter_null_type function. This custom function iterates all previously declared filters and evaluates each of them, and in the case of a false result, it increases the associated accumulator.

Example 3-22. Filter Interceptor in PySpark: the job

```
def filter_null_type(devices_iterator: Iterator[pandas.DataFrame]):
 def filter_row_with_accumulator(device_row):
  for device_row_attribute in device_row.keys():
   for filter_with_accumulator in filters_with_accumulators[device_row_attribute]:
    if not filter_with_accumulator.filter(device_row):
     filter_with_accumulator.accumulator.add(1)
     return False
  return True

 for devices_df in devices_iterator:
  yield devices_df[devices_df.apply(lambda device:
   filter_row_with_accumulator(device), axis=1) == True]

valid_devices = input_dataset.mapInPandas(filter_null_type, input_dataset.schema)
valid_devices.write.mode('append').format('delta').save(output_dir)
```

Finally, to get the filtering statistics, you need to check the accumulators' value by iterating the list (see Example 3-23).

Example 3-23. Filter Interceptor in PySpark: getting the values

```
for key, accumulators in filters_with_accumulators.items():
 for accumulator_with_filter in accumulators:
  print(f'{key} // {accumulator_with_filter.name} //
      {accumulator_with_filter.accumulator.value}')
```

That was for the programmatic API. As you saw, the pattern relies on a wrapper for the filter function. The same solution applies to the SQL version, but this time, with additional aliases (see Example 3-24).

Example 3-24. SQL queries for Filter Interceptor

```
spark_session.sql('''SELECT * FROM (
  SELECT
  CASE
   WHEN (type IS NOT NULL) IS FALSE THEN 'null_type'
   WHEN (LEN(type) > 2) IS FALSE THEN 'short_type'
   WHEN (full_name IS NOT NULL) IS FALSE THEN 'null_full_name'
   WHEN (version IS NOT NULL) IS FALSE THEN 'null_version'
   ELSE NULL
  END AS status_flag,
  type, full_name, version
 FROM input)''').createTempView('input_with_flags')



spark_session.sql('''SELECT COUNT(*), status_flag FROM input_with_flags WHERE
status_flag IS NOT NULL GROUP BY status_flag''').createTempView('grouped_filters')


(spark_session.sql('SELECT type, full_name, version FROM input_with_flags
 WHERE status_flag IS NULL')
 .write.mode('append').format('delta').save(f'{base_dir}/devices-valid-sql-table'))
```

Example 3-24 starts with a table creation query, which returns all input columns plus a computed filter alias based on an if-elseif-elseif-...else statement. The job later uses the results of that table to count the number of filtered rows for each condition and to create a new user-facing table with all valid records.

Fault Tolerance

Let's finish this chapter with a form of protection that ensures recoverability for continuous data processing workflows, such as streaming ones. The challenge with these workflows is to know when to start after stopping the job. Without a proper progress tracking mechanism, you'll end up reprocessing already processed data.

Pattern: Checkpointer

The fatal error is particularly critical in stream processing. Remember, these applications are working on continuously arriving events that are often stored in an append-only log. Put differently, you can't simply restart them as batch pipelines since the dataset doesn't have any particular organizational structure, such as partitions, that could help you figure out what to process next.

Problem

You're processing the visit events in streaming. The job counts the number of unique visits in 10-minute windows. You're worried that any fatal failure will stop the job and make it reprocess the data from the beginning. To mitigate that risk, you're looking for a solution that will persist the results as the query moves on.

Solution

To avoid reprocessing past data, your job must keep track of the most recent position in the consumed data source, as well as the computed state. The Checkpointer pattern implements this tracking mechanism.

Checkpointing consists of recording the data processing process in a more persistent storage than the job's environment, which may change when you restart it. Two approaches exist here, depending on your consumer's logic.

Data processing framework based

If you rely on a data processing framework, the progress information may be recorded in the environment managed by the framework itself. Apache Spark Structured Streaming and Apache Flink are great examples here as they store the progress metadata in a resilient object store with full progress tracking management.

Data store based

On the other hand, if you are using the data store SDK, you may be interacting with the data store layer for the checkpoint information. An example here is Apache Kafka SDK, which persists the checkpoint data to an Apache Kafka topic (__consumer_offsets), or Amazon Kinesis Client Library (KCL), which writes checkpoints to an Amazon DynamoDB table.

Two implementations also exist for the checkpointing operation itself. The first one is configuration driven, where you only configure the checkpointing frequency and delegate the execution to your library. That's how Apache Spark Structured Streaming and Apache Flink work.

The second implementation relies on an intentional checkpointing action from your code. Here, after reading and processing the records, you'll be responsible for confirming this operation to avoid getting the same data in the next execution. An example here is an Apache Kafka custom consumer with the commit methods.

Consequences

Although the pattern provides an extra fault tolerance mechanism, it doesn't do it for free. Latency is the biggest drawback here.

Delivery guarantee versus latency trade-off

Position tracking is not an expensive operation in terms of latency. It only accumulates some numbers for each input partition in memory and persists them once in a while to a persistent storage. However, the pattern also applies to the state in cases of stateful applications such as user sessions (cf. the Stateful Sessionizer pattern). Tracking the state may have a more significant latency impact as the state will probably be many times bigger than those numeric positions.

For that reason, once again, you'll need to balance the latency requirements and the processing guarantee. The more frequent the checkpoints are, the slower the job will be due to checkpoint creation overhead. When you opt for less frequent checkpoints, the job will spend less time dealing with the metadata, but on the other hand, in the case of failure, you may have more data to reprocess.

Exactly-once feeling

The pattern gives you the exactly-once delivery feeling, but it's just an impression. The first reason for this is the distributed character of the job. There could be multiple tasks working in parallel and in an asynchronous manner. If one of them fails in the middle of the work before triggering the checkpoint, the restart will involve retries and reprocessing of the already successful records.

To achieve exactly-once delivery, you'll need to apply one of the idempotency patterns presented in the next chapter. The checkpointing alone will not be enough.

Delivery Modes

Exactly once is a delivery mode where the producer delivers records only one time to the data store. It's the perfect scenario that you can achieve with the idempotency patterns from Chapter 4. Two other available modes are impacted by checkpointing:

At least once

This happens when you perform the checkpoint after processing (and thus writing) the data; you can generate duplicates in case of retries.

At most once

This occurs when you create the checkpoint before processing; it involves losing the data in the case of processing failures.

Examples

Because this pattern works for streaming pipelines, let's focus on two slightly different implementations. The first solution comes from Apache Spark Structured Streaming and is present in Example 3-25. As you can see, the code defines the checkpoint storage in the checkpointLocation attribute. After starting the job, Apache Spark will write all processed offsets to the metadata files located under {base_dir}/checkpoint.

Example 3-25. Checkpoints in Apache Spark Structured Streaming

```
write_query = (input_stream_data.writeStream.outputMode('update')
  .option('checkpointLocation', f'{base_dir}/checkpoint')
  .foreachBatch(synchronize_visits_to_files).start())
```

The written files are named after each executed job's iteration. Example 3-26 shows an example of the type of file you can find under the checkpoint location.

Example 3-26. Checkpointed metadata for fault tolerance

```
$ cat /tmp/dedp/ch03/fault-tolerance/micro-batch/checkpoint/offsets/18
# omitted two irrelevant lines
{"visits":{"1":1276,"0":1224}}
```

Checkpointer in Apache Spark writes offsets at a job's iteration. This regularity adds overhead, but it provides a stronger guarantee and incurs less risk of processing duplicates in case of restart. Apache Flink, which is another stream processing framework, works a bit differently. It doesn't follow the job iteration's mode and instead is based on time.

Example 3-27 shows checkpoint configuration in Apache Flink.

Example 3-27. Time-based checkpointing with Apache Flink

```
checkpoint_interval_30_sec = 30000

env.enable_checkpointing(checkpoint_interval_30_sec, mode=EXACTLY_ONCE)


(env.get_checkpoint_config().enable_externalized_checkpoints(RETAIN_ON_CANCELLATI
ON))
```

Even though the snippet is short, it may be confusing because of various configuration parameters. The easiest to explain is the RETAIN_ON_CANCELLATION mode. This property simply asks Flink to keep the checkpointed files after a job's failure. By default, the checkpoint location is tied to the job instance and Flink removes it whenever the job restarts. When it comes to the EXACTLY_ONCE checkpoint mode, it impacts stateful operations, such as windowed counters. The configuration from the snippet guarantees that each input record reflects the state once. In the case of our counter, it means that any restart will not lead to counting an element twice.

Asynchronous Progress Tracking

Apache Spark 3.4.0 has introduced support for asynchronous checkpoints that are not synchronized with microbatches. At the time of writing this book (2024), the feature is still experimental, and the open source version doesn't support state store.

Summary

Errors are inevitable. They may come from buggy code, poor quality of the ingested data, or just temporary hardware issues. Error management design patterns are there to help you deal with the inevitable.

At the beginning of the chapter, you discovered three patterns that are adapted to data quality issues. You learned about using the Dead-Letter pattern to handle unprocessable records gracefully, the Windowed Deduplicator pattern to reduce the risk of duplicates, and the Late Data Detector pattern with Integrator to identify and process late data.

Next, you discovered the Filter Interceptor that can help you better understand how your code behaves in a filtering operation. Finally, you saw that failures can be critical for long-running applications and that thankfully, modern data processing services manage fault tolerance on your behalf with the Checkpointer pattern.

But as we've already mentioned a few times, error management doesn't guarantee exposing perfectly valid data. Even though they may give a feeling of an exactly-once delivery, which is the holy grail in delivery semantics, processing retries or backfills can still have a negative impact. But it's less serious when your pipelines are idempotent—and if you don't know what that means, the next chapter will shed some light on it!

1 The last update time is already stored at the partition level. However, having it in a single place alongside the last processed time simplifies understanding of the pattern and may be helpful in day-to-day maintenance.