Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 10. Data Observability Design Patterns

The data quality design patterns from the previous chapter are crucial to guaranteeing the relevance of your datasets. However, as they focus mainly on the data itself, relying only on data quality solutions won't be enough for you to have end-to-end control of your data engineering stack.

Let's take a look at an example to understand this better. The Audit-Write-Audit-Publish (AWAP) pattern is a great protection mechanism against processing data of poor quality. Unfortunately, even if your AWAP job perfectly detects all issues, you may still be in trouble. An example of this occurs when your AWAP job doesn't run because of an upstream flow interruption and you are not aware of it.

There is good news, though: the data observability design patterns from this chapter fill the gaps left by their data quality counterparts by adding monitoring and alerting capabilities to the system. To address these extra issues, the observability pattern solutions rely on two pillars: detection and tracking.

The detection design patterns spot any problems related to the data or time. They would be great candidates to handle the AWAP's data flow interruption issue mentioned previously. They will also be useful for notifying you whenever your batch job takes too much time to complete.

Tracking design patterns focus on understanding the relationships among datasets, columns, and the data processing layer. They will help you discover the data generation graph that in large organizations often spans across different teams. They are also helpful in understanding the transformation logic for individual columns, especially when it comes to columns created from multiple inputs.

This short introduction should be enough to convince you that observability design patterns, although they might sound like "things the operations team should do," are also important for us data engineers. And to drive this point home even more, let's now turn to the first observability design patterns: data detectors.

Data Detectors

Data engineers process data. Unsurprisingly, the first observability category helps analyze the health of our systems from the data standpoint.

Pattern: Flow Interruption Detector

The first serious data-related issue is dataset unavailability. This issue will have a strong impact on your systems because without any data, your data processing job will not run, leading to data unavailability in its downstream dependencies.

Problem

One of your streaming jobs is synchronizing data to an object store. The synchronized dataset is the data source for many batch jobs managed by different teams. It ran great for seven months until one day, it processed the input records without writing them to the object store.

Because the job didn't fail, you didn't notice the issue. You only realized something was wrong when one of your consumers complained about the lack of new data to process. Instead of relying on consumer feedback, which is not good for your reputation, you want to introduce a new observability mechanism to detect this data unavailability scenario.

Solution

To capture any data unavailability errors, and as a result, increase trust in your data, you can rely on the Flow Interruption Detector pattern.

The implementation of the pattern will vary depending on the technical context and processing mode. Let's start with the stream processing introduced in the problem statement. Basically, you can have two different data ingestion modes here:

Continuous data delivery

In this mode, you expect to get at least one record every unit of time, like a minute or a second. In that context, the Flow Interruption Detector consists of triggering an alert whenever there are no new data points registered for the specified unit of time, like no data coming in for one minute.

Irregular data delivery

Here, you expect to see some delivery interruptions that have nothing to do with errors. For example, you might assume that no data will come in for five consecutive minutes. The pattern's implementation in that context consists of analyzing time windows instead of data points and raising an alert whenever the period without the data is longer than the accepted no-data window duration. Since the data flow is irregular, using the continuous data delivery assumption would result in many false-positive alarms that might lead to alarm fatigue.[1]

Figure 10-1 compares the two approaches. As you'll notice, the only difference between them is the data evaluation period. Continuous data delivery analyzes a specific unit of time (a minute in the example), while for irregular data delivery it monitors multiple consecutive points in time.

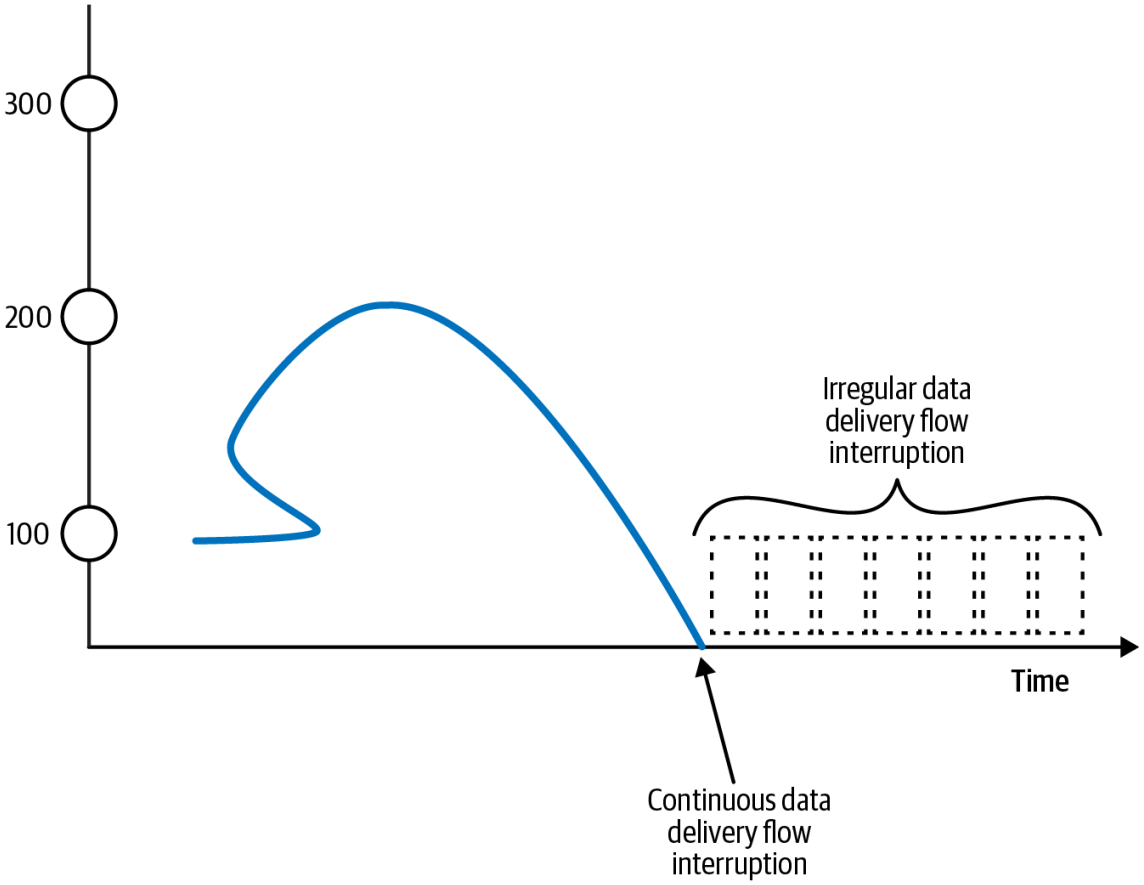

Figure 10-1. Flow interruption detection for continuously arriving and irregularly arriving data

However, data flow interruption can also occur in batch pipelines and data-at-rest databases. Spotting unavailability here consists of analyzing the data freshness of the metadata, data, or storage layer:

Metadata layer

This layer stores all additional information about the observed table, such as creation time and last modification time. A flow might be interrupted when the modification time hasn't been changed according to the configured threshold. For example, a table with new data ingested hourly could trigger an alert whenever no changes are observed for more than one hour.

Data layer

Generally, interactions with the metadata layer will be less expensive in most scenarios due to more direct access to the information and thus the lack of data processing needs. Unfortunately, your data store may not have the metadata layer available or the update information may be missing. In that situation, you may need to enrich the table with the modification time column to detect a flow interruption if the last update passed the configured threshold. If adding this column is not possible, you can count the number of rows in each evaluation time period and compare the results to see whether there is new data. For example, in our hourly job, if the count doesn't change in two consecutive hours, it'll be a sign of flow interruption. This count-based approach may also require some extra storage to preserve the count statistics for past runs.

Storage layer

The last flow interruption detection strategy is based on the storage layer. Typically, you could use it with any file format, including raw JSON files or more advanced table file formats. The implementation here consists of monitoring the time when the last file was written in a storage space and raising an interruption alert whenever there are no updates within the expected threshold.

Consequences

Flow interruption detection may look simple, but unfortunately, the implementation hides some traps such as the threshold, metadata, and false positives.

Threshold

Finding the perfect threshold for both per-minute and per-window implementations is not easy. Expecting at least one record per minute, as in our previous example, is an easy choice but may not be realistic. If you expect to process hundreds of events or more per minute, you will need to choose a different number.

Relying on the volume observed in the past may be a tempting way to define the threshold. It's often used as the solution to this threshold-finding problem, but it also has a gotcha. From time to time, it can generate false positives, for example, whenever a marketing operation generates more activity than usual.

Metadata

The solution based on the metadata is cheap but may not be perfect. First, this metadata layer with the last modification time or the number of rows may simply not be available for your database. Second, even if the layer is there, the modification can include not only data operations but also metadata changes, such as schema evolution, which obviously doesn't add any new records. You must be careful when evaluating it for the purpose of the Flow Interruption Detector pattern.

False positives for storage

If you rely on the storage layer for flow interruption detection, beware of all housekeeping operations, such as compaction. This operation does indeed create new files, but it doesn't produce new datasets. Instead, compaction simply merges existing data blocks. From the storage's perspective, there is activity, but it will not count as flow continuity since the dataset remains unchanged.

Examples

Let's begin this section with Apache Kafka, Prometheus, and Grafana. To detect the flow interruption, we'll use an expression that evaluates how many records are written every minute (kafka_server_brokertopicmetrics_messagesin_total). This is shown in Example 10-1.

Example 10-1. Evaluation of incoming messages per minute for a visits topic

```
sum without(instance)(rate(
kafka_server_brokertopicmetrics_messagesin_total{topic="visits"}[1m]))
```

The next step is to configure the alert to detect an interruption whenever the last values (five in our example) are equal to zero. If this condition is met, Grafana will raise a data interruption alert. Since this operation relies on the user interface and not the code, I'll let you read the screenshot in the GitHub repo.

The interruption can also be tracked for batch workloads, thanks to the last written message information. PostgreSQL supports this message tracking with a configuration parameter called track_commit_timestamp. This attribute enables a pg_ xact_commit_timestamp function that you can reference in the flow interruption query (see Example 10-2).

Example 10-2. Flow interruption with the last commit time function

```
SELECT

CAST(EXTRACT(EPOCH FROM NOW()) AS INT) AS "time",

CAST(EXTRACT(EPOCH FROM NOW() - MAX(pg_xact_commit_timestamp(xmin))) AS INT)
AS value

FROM dedp.visits_flattened
```

The query extracts the last commit timestamp from the current time that you can reference later in your notification layer to trigger an alert whenever the difference is bigger than the accepted flow interruption threshold.

The flow interruption mechanism can also rely on the data producer. Example 10-3 shows a Delta Lake producer that synchronizes data from Apache Kafka with a Delta Lake table. At the end of this action, it also sends a last update time metric to Prometheus. That way, you can use the same time evaluation conditions as shown previously to spot a data ingestion interruption.

Example 10-3. Ingesting the last writing time for a Delta Lake table

```
visits_to_write.write.format('delta').insertInto(get_valid_visits_table())

from prometheus_client import CollectorRegistry, Gauge, push_to_gateway

registry = CollectorRegistry()

metrics_gauge = Gauge('visits_last_update_time',

 'Update time for the visits Delta Lake table', registry=registry)

metrics_gauge.set_to_current_time()

metrics_gauge.set(1)

push_to_gateway('localhost:9091', job='visits_table_ingestor', registry=registry)
```

Pattern: Skew Detector

In addition to data interruption issues, you can face data skew problems that will directly impact your processing layer. An unbalanced and thus skewed dataset can increase processing time or, even worse, trigger a batch pipeline on an incomplete dataset.

Problem

After you put the Flow Interruption Detector in place, some consumers complained yet again. This time, they were unhappy because your batch job processed an incomplete dataset.

You found out that your job worked correctly, but the pipeline processed a half-empty dataset. After talking to your data provider, you learned that there were some data generation issues on its side. You're wondering how to overcome this problem in the future by always processing a complete dataset.

Solution

The situation described in the problem statement is a typical example of *data skew*. Although the term has become popular in data processing to describe a situation in which some tasks have more load than others, *skew* is also a valid word to describe a situation in which a pipeline processes different data volumes in two consecutive executions. Thankfully, the Skew Detector pattern brings you more control over this phenomenon.

This solution consists of three steps. The first step identifies the comparison window. Put differently, you must start by determining which time periods in the pipeline's runs to compare. For example, if you have a daily batch job, you could decide to compare the currently processed dataset with the previous day's dataset.

Once you identify the time periods, you need to set a tolerance threshold. This value determines how different the compared datasets can be. For example, if you set it to 50%, it means that your processing job tolerates working on 50% less or 50% more data than in the previous comparison window. To set up this tolerance threshold, you can analyze the data variations you observed in previous days or directly ask your business users about the expected differences.

The final step consists of implementing the skew calculation. It can be either of the following:

- A window-to-window comparison (as mentioned in the previous paragraph), in which you calculate the percentage difference between two values. It applies to both batch jobs and continuous streaming applications where you might compare data with processing time or event time windows.
- A more complex calculation in which you use the ratio of a standard deviation. Here, you detect how each of the data points deviates from the mean of the dataset. This approach is particularly useful if you need to calculate data skew in a partitioned storage system, such as an Apache Kafka topic or a PostgreSQL table. Thankfully, many of these data stores have a standard deviation function (stddev) that you can simply call to get the ratio as part of a formula. For example, in STDDEV(x)/AVG(x), *x* is the observed metric, such as storage usage by a partition.

The Skew Detector pattern is a good candidate for use in the first Audit stage of the AWAP pattern explained in the previous chapter. It'll then act as a guard preventing the processing of partial datasets.

Consequences

Even though the solution sounds simple, practice reveals some tricky points related to the dataset itself.

Seasonality

Seasonality is probably the biggest challenge with the Skew Detector pattern. If you assume that each time, the comparison window will get 50% more or 50% less data than the previous window and that that is fine, then you have no issue. But what if that will be true only sometimes or, even worse, never?

The data often relies on your organization's activity. For example, when the marketing department is running ad campaigns, you might get 50% more records than usual. The same variance is valid for a seasonal business where summer may bring more data than the winter season, or the opposite.

Finding a rule under these variable circumstances is not straightforward. To find it, you'll rely on your business knowledge to build the comparison formulas and eventually add some exceptions to ignore or adapt the most variable periods.

Communication

Even though you're able to define the threshold in the changing environment presented in the previous paragraph, there is still some room left for false positives. An aggressive and successful marketing campaign is one of the examples that can bring much more data than the accepted threshold.

Mitigating the issue requires more communication skills than technical skills as you will need to synchronize with other departments in your organization so that you can adapt the alerts for the chosen time periods and be able to spot false positives.

Fatality loop

A window-to-window comparison can introduce some fatality loops in cases of skew. Let's get a better understanding of this by looking at an example. You're running a batch job daily, and one day, the skew validation step fails because the dataset is three times smaller than the dataset from the day before. If you don't fix the issue by the next day, the valid dataset from that day will be considered to be skewed since the comparison window is on a day-to-day basis. After all, compared to the failed day, you now have three times as much data.

The best way to fix the issue is to always have a valid and complete dataset for the comparison windows, so you need to ask your data producer to solve the issue. If the fix is not possible before the next pipeline's scheduled run, you can apply the volume comparison not to the day before but to the dataset used by the most recent successful pipeline run, which in our example might be the dataset processed two days ago.

Examples

You already know about the ratio of a standard deviation and how it relates to skew detection. To make this more real, let's see how to write a query to compute the skew in PostgreSQL partitioned tables. The query from Example 10-4 relies on the data catalog's (and hence, the metadata's) pg_stat_user_tables to get the number of rows from the partitioned visits_all_range_ tables and compute the ratio for each evaluation time.

Example 10-4. Computing the ratio of a standard deviation for PostgreSQL partitioned tables

```
SELECT
 NOW() AS "time", (STDDEV(n_live_tup) / AVG(n_live_tup)) * 100 AS value
 FROM pg_catalog.pg_stat_user_tables
WHERE relname != 'visits_all_range' AND relname LIKE 'visits_all_range_%';
```

Later, whenever the standard deviation ratio reaches the configured threshold, you can consider it to be a storage data skew. Again, you will find a full demo in the GitHub repo. The STDDEV and AVG functions are pretty common functions in data stores. They're also present in Prometheus, where you can use them to calculate the storage skew for another data store, which is Apache Kafka (see Example 10-5).

Example 10-5. Computing the ratio of a standard deviation for Apache Kafka partitions

```
stddev(sum(kafka_log_size{topic='visits'}) by (partition)) /
 avg(kafka_log_size{topic='visits'}) * 100
```

Now, let's see how to apply the Skew Detector pattern to a data pipeline with the window-to-window comparison mode. To demonstrate this scenario, we're going to use an Apache Airflow pipeline that loads JSON files into a PostgreSQL table. The pipeline's code is shown in Example 10-6. The workflow starts with a sensor (see "Pattern: Readiness Marker") that unlocks the pipeline whenever the next partition is created. Later, the compare_volumes function verifies whether the current partition is at most 50% smaller or 50% larger than the previous partition. That's our window-to-window skew validation. If the condition holds, the pipeline moves to the last step, which consists of loading data to a PostgreSQL table.

Example 10-6. Data skew in a data validation task

```
next_partition_sensor = FileSensor(...)


def compare_volumes():
 context = get_current_context()
 previous_dag_run = DagRun.get_previous_dagrun(context['dag_run'])
 if previous_dag_run:
  previous_execution_date = previous_dag_run.execution_date
  current_file_path = get_full_path(context['logical_date'], 'json')
  current_file_size = os.path.getsize(current_file_path)
  previous_file_path = get_full_path(previous_execution_date, 'json')
  previous_file_size = os.path.getsize(previous_file_path)
  size_ratio = current_file_size / previous_file_size
  if size_ratio > 1.5 or size_ratio < 0.5:
```

```
  raise Exception(f'Unexpected file size detected for the...')


volume_comparator = PythonOperator(task_id='compare_volumes',
                                   python_callable=compare_volumes)
transform_file = PythonOperator(...)
load_flattened_visits_to_final_table = PostgresOperator(...)


(next_partition_sensor >> volume_comparator >> transform_file
>> load_flattened_visits_to_final_table)
```

Time Detectors

In addition to data, time is an important metric for data pipelines. It helps you spot latency issues in your data processing layer.

Pattern: Lag Detector

Speaking of latency, the first pattern from this category defines how far a data consumer can be behind the data producer. This measure is often an indicator for upcoming data quality problems, such as data freshness and data unavailability.

Problem

It has been one week since one of your streaming jobs processed 30% more data than the previous job. Unfortunately, you missed the email announcing this increase, and now, one of your downstream consumers is complaining about slower data delivery. You've promised them that this is the last time. You want to put a scaling strategy in place, but the first required step is to monitor how fast your consumer processes input data.

Solution

A good way to measure how your consumer is doing is to use the Lag Detector pattern.

The first thing to do is to define the lag unit. This choice will depend on the data store. For an Apache Kafka topic, it can be the record position or the record append time. For a Delta Lake table, you can rely on the commit number, while for a time-partitioned data store, you can leverage the partition timestamp.

Upon identifying the unit, you can define the comparison expression that will verify the currently processed unit with the most recent one available in your storage. The difference between them will then represent the lag. Example 10-7 shows a high-level Python algorithm for the lag calculation.

Example 10-7. High-level lag equation; getters will depend on your data store

```
last_available_unit = get_last_available_unit()
last_processed_unit = get_last_processed_unit()
```

lag = last_available_unit - last_processed_unit

In cases of partitioned data stores, like the aforementioned Apache Kafka, you will get a lag measure per partition. That's why the next step will define the strategy for the partitioned results, which can be one of the following:

- If you want to discover the biggest lag, you should call the MAX function on the partitioned results. That way, you will be able to detect the worst-case scenario, even if only one of the partitions lags behind.
- If you want to know how most of the partitions perform, you should use a *percentile* function, such as P90 or P95. That way, you will be sure that 90% or 95% (percentile value) of your partitions are within one lag value. That said, it doesn't mean all of them have the same lag. For example, a lag of 10 seconds for P90 means that 90% of the partitions have a lag of 10 seconds at worst, and some may have less than that.

You can also combine the two approaches to follow an overall latency with a percentile function and the worst-case scenario with the MAX aggregation.

The Average Trap

Averaging is a popular statistical function, but in the context of observability, it hides some traps. Let's suppose that the lag metrics for seven partitions are 10, 5, 30, 2, 3, 5, and 3 seconds. The average lag will be 8 seconds, while the P90 lag will be 18 seconds. Put differently, 90% of the data is processed within 18 seconds, for real! If you just looked at the average of 8 seconds, you could wrongly conclude that the job is performing pretty well. For that reason, percentiles are more relevant for lag detection.

Consequences

Skewed data is a serious consequence for many things in data engineering. The Lag Detector pattern is not an exception here, as you will read next.

Data skew

If you decide to represent the lag as a single unit with the MAX(...) function, beware, because a poor result may not be directly related to your consumer. If, for whatever reason, one partition gets more load than the others, your consumer will naturally process it slower. That doesn't mean the consumer has an issue, though. Instead, you should think of better distributing the data during the writing step so that consumers can work on even partitions.

Examples

Let's see first how to analyze the lag for an Apache Spark Structured Streaming job processing data continuously from an Apache Kafka topic. Apache Spark has a listener's mechanism that you can leverage for lag detection. Our listener will trigger after processing each window of records and compare the offsets of previously processed rows with the most recent ones present in each partition. The snippet from Example 10-8 shows this step (the full code is available in the GitHub repository).

Example 10-8. Microbatch offsets reader

```
class BatchCompletionSlaListener(StreamingQueryListener):
```

```
 def onQueryProgress(self, event: "QueryProgressEvent") -> None:
  latest_offsets_per_partition = self._read_last_available_offsets()
  visits_end_offsets = json.loads(event.progress.sources[0].endOffset)
  visits_offsets_per_partition: Dict[str,int] = visits_end_offsets['visits']
```

After retrieving the offsets, the listener initializes a connection to the Prometheus instance, computes the difference (lag) between the processed and most recent offsets, and sends the results to Prometheus. The second part of the onQueryProgress snippet is in Example 10-9.

Example 10-9. Lag calculation

```
registry = CollectorRegistry()
metrics_gauge = Gauge('visits_reader_lag', '...', registry=registry,
 labelnames=['partition'])
for partition, value in visits_offsets_per_partition.items():
 lag = latest_offsets_per_partition[partition] - value
 metrics_gauge.labels(partition=partition).set(lag)

push_to_gateway('localhost:9091', job='...', registry=registry)
```

Implementing the Lag Detector pattern is also possible for table file formats like Delta Lake. This time, we're going to use a different method of running our Apache Spark Structured Streaming job. Instead of executing continuously, the job will run on schedule, process all data available at a given moment, and stop. That way, you can leverage Apache Spark's checkpoint mechanism and avoid managing the progress manually. For the batch schedule, the job uses the availableNow trigger (see Example 10-10).

Example 10-10. Structured Streaming job with the availableNow **trigger**

```
visits_stream = spark_session.readStream.table('default.visits')

console_printer = (visits_stream.writeStream.trigger(availableNow=True)
 .option('checkpointLocation', checkpoint_dir)
 .option('truncate', False).format('console'))

console_printer.start().awaitTermination()
```

The query variable includes the progress information made by the job. That's where you will find the last processed version of the Delta Lake table. You can later read it and send it to

the monitoring endpoint (Prometheus Gateway, in our case), as demonstrated in Example 10-11.

Example 10-11. Submitting the last processed version of the visits table

```
last_version = query.lastProgress["sources"][0]["endOffset"]["reservoirVersion"]


registry = CollectorRegistry()
metrics_gauge = Gauge('visits_reader_version',
 'Last read version of the visits table', registry=registry)
metrics_gauge.set(last_version)
push_to_gateway('localhost:9091', job='visits_reader_version', registry=registry)
```

The previous snippet was for the data consumer. But to determine the last written version, the data producer should also emit a metric. Here, the last version can be found directly by running the DESCRIBE HISTORY query after generating the data (see Example 10-12).

Example 10-12. Getting the last version of the table

```
# ... data generation step, transaction commit
last_written_version = (spark_session.sql('DESCRIBE HISTORY default.visits')
 .selectExpr('MAX(version) AS last_version').collect()[0].last_version)
```

That way, you can configure your alert as a difference between the last written version and the last read version. If this difference is bigger than the accepted threshold, it can be a sign of increasing consumer lag.

Pattern: SLA Misses Detector

Using the Lag Detector pattern from the previous section is a great way to measure the processing pace of a data consumer. However, it's not a single time-based measure that you can add to your data processing jobs. The lag-based approach can be completed with a solution based on an SLA that directly asserts the execution time of a given workflow.

Problem

Your task is to complete a batch job scheduled at 6:00 a.m. within 40 minutes. Your downstream consumers are critical data pipelines. They must generate various business statistics before 8:00 a.m. You did your best to optimize the job to respect the 40 minute SLA, but you know that unpredictable things happen and the SLA may be broken one day.

For that reason, you want to implement an observability mechanism that will notify you and your downstream consumers whenever the job is taking more than 40 minutes.

Solution

To ensure that your consumers are notified about any latency problems with your batch job, you can use the SLA Misses Detector pattern.

This pattern consists of measuring the processing time and comparing it to the maximum allowed execution time. The implementation depends on the processing mode:

Batch job

This is the simplest scenario because you will measure the difference between the end time and the start time. If this difference is greater than the defined SLA threshold, then the run will be marked as an SLA miss and an SLA miss notification will be sent.

Streaming job

If the processing framework works in a microbatch or event time–windowed mode,[2] (i.e., when each stream execution operates on a bunch of data), you can use the same technique as for the batch jobs, to subtract the start time from the end time for each iteration.

On the other hand, if the windowed mode is not supported, the solution consists of measuring the difference between reading and writing each record to the output data store. The gathered metrics can be later aggregated to a MAX or a percentile function to determine, respectively, the longest delay and the overall delay. To gather the metrics, you can use the Online Observer pattern or Offline Observer pattern.

Even though an SLA miss can be related to the lag introduced in the previous pattern, the two detection patterns are complementary but not always interchangeable. A clear example explaining why is a skewed partition in an input data store. If a consumer implements a throughput limitations mechanism to always process the same volume of data, the job will respect the SLA. However, the lag will be continuously increasing as the data on the skewed partition will not be processed as it comes in.

The same dependency is valid in the opposite direction. Let's take a batch job processing data generated daily by an upstream pipeline. At the moment of starting the processing, the batch job won't have any lag due to the daily data processing schedule, but it can still miss the SLA if it takes too much time to complete.

Consequences

The SLA Misses Detector pattern is relatively straightforward when it comes to batch pipelines. However, it's more challenging for streaming pipelines, especially due to the data arrival semantics.

Late data and event time

Although processing time is a common unit of measure for calculating SLA misses and the simplest one as well, it's not the only one. You can also use event time and get an idea of the end-to-end time between data generation and processing. It sounds simple, right? In theory, it is, but in practice, whenever you interact with event time, you risk dealing with late data, which from the SLA's standpoint may not be your fault.

Simply speaking, if a producer loses its network connectivity and delivers locally buffered data a few minutes later, your event time–based SLA may not be respected because of the delivery interruption and not because of any data processing issues on your end. But at the same time, late-arriving data may not break your processing time SLA, as it's closely related to how fast the job processes input data, no matter the event time. Put differently, it's based

on the current time. The processing time and event time aspects don't cover the same ground, and it's good to separate them when it comes to observing SLA.

Consequently, in this scenario, you will monitor the processing time SLA as the difference between the reading time and the writing time for a record, plus the event time SLA if relevant. The event time SLA will subtract the record generation time from the record writing time. Figure 10-2 depicts this time difference.

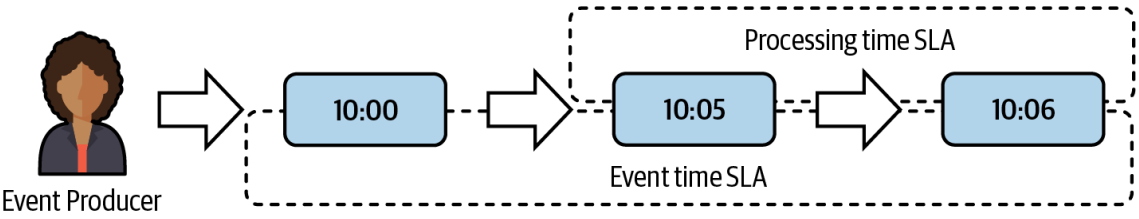

Figure 10-2. Processing time and event time SLAs illustrated

Examples

An easy way to demonstrate SLA monitoring is in Apache Airflow. This data orchestration framework supports an sla time parameter that you can define in each workflow's task to be sure it completes in time. Example 10-13 defines an SLA of 10 seconds on processing_task_2.

Example 10-13. SLA definition in Apache Airflow

```
@task(sla=datetime.timedelta(seconds=10))

def processing_task_2():
```

Despite this simplicity, the SLA mechanism in the current version of Apache Airflow (2.10.2)[3] exhibits a peculiar behavior. The framework computes the SLA from the execution start time of the pipeline and not the start time of the task. Put differently, for an execution that's scheduled for eight o'clock, if processing_task_2 starts later than 08:00:10, it will already be considered to be late.

Besides batch systems, SLA monitoring can work for streaming workflows. Our next example uses two Apache Flink jobs. The first of them is a data processing job writing records continuously to another Apache Kafka topic. The output format includes a start_processing_time_unix_ms attribute that we're going to use in a second job to calculate the SLA metrics. Example 10-14 shows how this important metric is generated.

Example 10-14. Decorating a record with the processing time attribute

```
def map_json_to_reduced_visit(json_payload: str) -> str:

 # ...

 return json.dumps(ReducedVisitWrapper(

  start_processing_time_unix_ms=time.time_ns() // 1_000_000, ...).to_dict())
```

Now, the second job reads this attribute alongside the writing time to the output topic and computes the difference between the two columns (see Example 10-15).

Example 10-15. Processing execution time calculation for Apache Kafka

```
CREATE TEMPORARY TABLE reduced_visits (
  `start_processing_time_ms` BIGINT,
  `append_time` TIMESTAMP METADATA FROM 'timestamp' VIRTUAL
) WITH ('connector' = 'kafka', ...)""")

sla_query: Table = table_environment.sql_query("""
SELECT
 append_time,
 ((1000 * UNIX_TIMESTAMP(CAST(append_time AS STRING)) +
  EXTRACT(MILLISECOND FROM append_time)) -
  start_processing_time_ms) AS time_difference,
  FLOOR(append_time TO MINUTE) AS visit_time_minute
FROM reduced_visits""")
```

Next, the SLA monitoring job generates percentiles per one-minute event time windows and emits them to the monitoring stack; from there they can be accessed to create SLA misses alerts. The aggregation step is in Example 10-16.

Example 10-16. Aggregation step of the SLA job

```
sla_query_datastream # ...
 .key_by(extract_grouping_key)
 .window(TumblingEventTimeWindows.of(Time.minutes(1)))
 .aggregate(aggregate_function=PercentilesAggregateFunction(),
   window_function=PercentilesOutputWindowFormatter()
# ...
)
```

The full code and an example for Apache Spark Structured Streaming are both in the GitHub repo.

Data Lineage

So far, you have seen solutions for observing the data you're processing. But what happens if the detected issues are not your fault? You saw this in the SLA Misses Detector pattern, where late events can break the event time–based SLA. In this case, you need to know who to ask for help in better understanding the reasons for latency. A great way to get this knowledge is by getting to know the dependencies of your datasets by building a kind of family tree for them.

Pattern: Dataset Tracker

Data lineage patterns operate at the dataset and the data entry levels. Let's start by discovering the dataset tracking that, as the name implies, applies to various data containers, such as tables, folders, topics, and queues. In the end, it creates a dependency tree among these containers to clearly represent data providers and data consumers.

Problem

You're consuming a dataset of poor quality. Your batch job is regularly failing because the schema is not consistent. After investigating, you find out that one of the fields has had different data types over time.

Your upstream data provider is not aware of these changes because it's not the one that generates the dataset. Your data provider is processing data generated by yet another team, so you want to understand the dataset dependency to better detect which team introduces the type inconsistency issue.

Solution

Analyzing dataset composition is a good fit for the Dataset Tracker pattern. The solution creates a family tree of datasets within your organization that you can use to easily discover the dependencies between the datasets and thus also between the teams. Figure 10-3 shows an example for an order dataset combined with two other tables that in their turn are built upon an Apache Kafka topic. Also, each dataset is annotated with the team responsible for maintaining it.

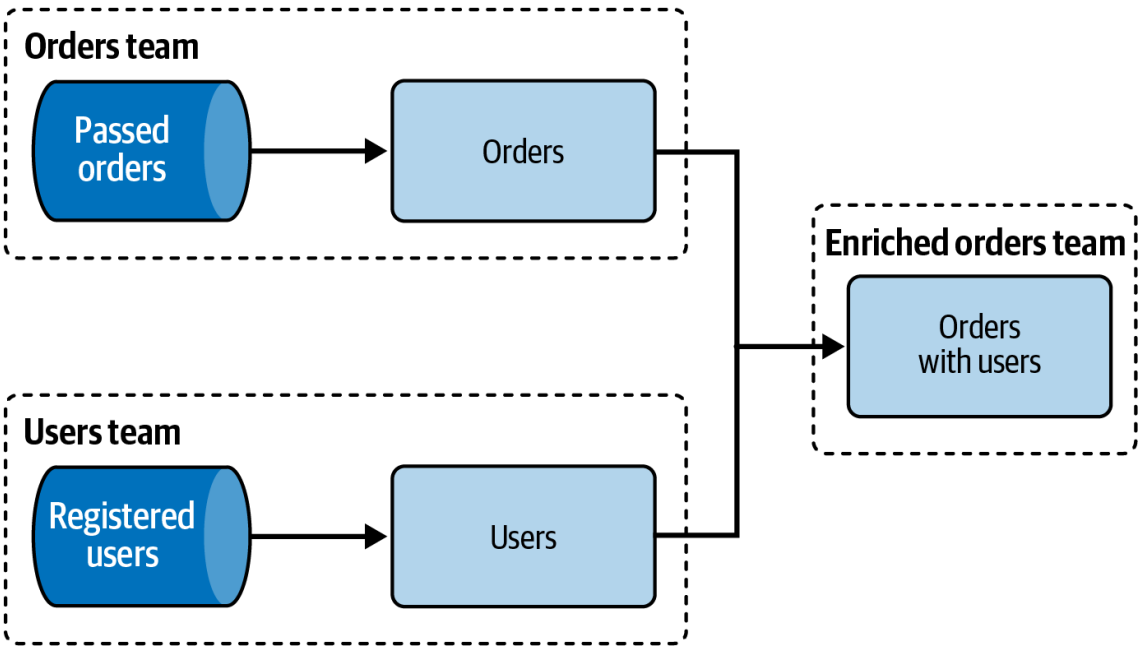

Figure 10-3. An example of datasets tracking in which all three teams are responsible for a dedicated domain and all related data objects (tables, topics, etc.) 4

When it comes to the implementation, there are two possible solutions. The first uses fully managed cloud services and will be transparent to you, as you will have nothing to do. In this implementation, the dependency tree construction is delegated to a service or framework that analyzes the dependencies among jobs, tables, and dashboards interacting

with each dataset. That's how Databricks captures the lineage of its tables registered in Unity Catalog, and it's also the way GCP's Dataplex service automatically records tracking information for other GCP data services, such as BigQuery and Dataproc.

However, the automated version is always limited to specific types of jobs or data stores. For example, even though Dataplex supports lineage from BigQuery data loading jobs, it doesn't support data loading from BigQuery Data Transfer Service. For that reason, there is a second implementation in which you implement the Dataset lineage on your own. The implementation starts with identifying the inputs and outputs for each query, task, or pipeline. This step can happen at two levels:

The data orchestration layer

Each pipeline that generates a dataset must include the inputs and outputs that will be reported to an external data lineage service. Some of the existing tools, such as Apache Airflow for OpenLineage, can detect dependencies between datasets automatically.[5]

The database layer

Here, your lineage job analyzes the executed queries to identify the dependencies between datasets and saves them in the lineage service. For example, you could build the orders with the users table from Figure 10-3 as SELECT ... FROM orders o JOIN users u ON u.id = o.user_id. The database layer identification transforms this query into a tree and extracts all reference tables.

The lineage implementation for the manual version doesn't stop at input/output extraction, though. In addition to this declarative or algorithmic solution, you need a layer that will interpret the identified dependencies and represent them visually. Figure 10-4 shows the full picture of this manual approach. As you can see, this approach brings some flexibility but also requires more implementation effort due to the need to monitor the number of deployed services and interactions among them.

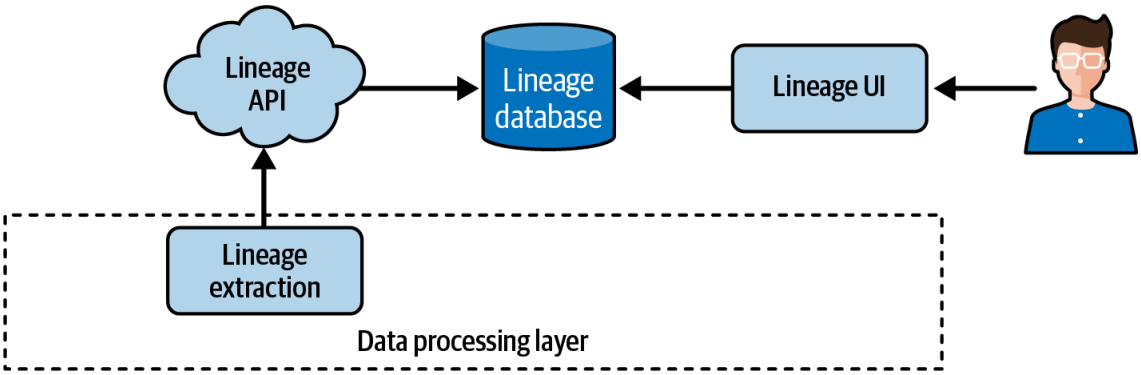

Figure 10-4. The flow for a manual lineage implementation. The fully managed approach follows the same diagram, but the lineage extraction is performed out of the box.

Consequences

Even though data lineage is becoming less and less obscure nowadays thanks to the most recent advances in cloud and open source technologies, it also has some gotchas.

Vendor lock

Fully managed solutions like the ones presented with Databricks and GCP often work within the service scope itself. Put differently, if you use them with open source data stores or databases available in other clouds, you may get only a partial view of the lineage.

Custom work

Sometimes, data orchestration frameworks abstract the input/output declaration as they can deduce them from the task configurations. However, that works only if you use the built-in task types. For any custom task type, you may also need to define the input/output resolution logic.

Examples

To demonstrate the Dataset Tracker pattern, let's use the OpenLineage API and Marquez UI.[6] Both are open source solutions that cover existing popular data engineering tools, such as Apache Airflow and Apache Spark.

To connect OpenLineage with Apache Airflow, we need to set the OpenLineage URL (for example, as the environmental variable OPENLINEAGE_URL=http://localhost:5000) and install the apache-airflow-providers-openlineage package. Thanks to the native OpenLineage support for various operators like PostgresOperator, the lineage setup doesn't require any extra steps on your part. All of the work will be done by OpenLineage extractors, giving you a clear view of how pipelines manipulate your datasets (see Figure 10-5).

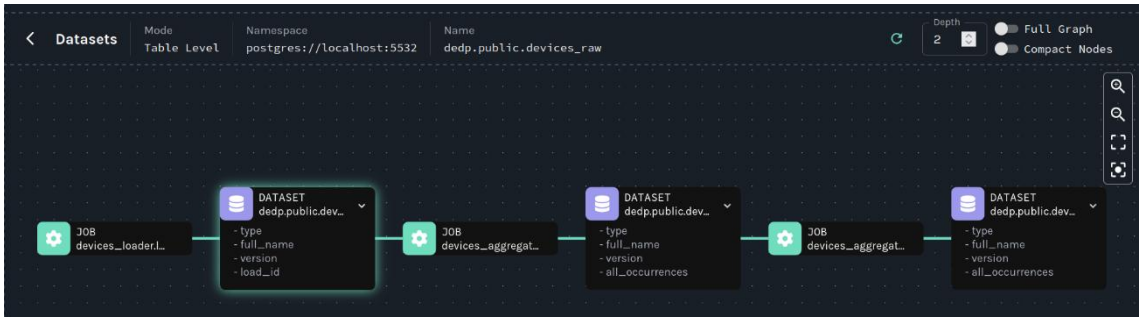

Figure 10-5. Apache Airflow DAG's lineage visualization in the Marquez UI

OpenLineage also works for Apache Spark. All you need to do is include the required OpenLineage dependencies while creating SparkSession. Example 10-17 shows how to do this.

Example 10-17. SparkSession with OpenLineage enabled

```
def create_spark_session_with_open_lineage(app_name: str) -> SparkSession:

 return (SparkSession.builder.master('local[*]')

  .appName(app_name)

  .config('spark.extraListeners',

   'io.openlineage.spark.agent.OpenLineageSparkListener')

  .config('spark.openlineage.transport.type', 'http')

  .config('spark.openlineage.transport.url', 'http://localhost:5000')
```

```
  .config('spark.openlineage.namespace', 'visits')
  .config('spark.jars.packages', 'io.openlineage:openlineage-spark_2.12:1.21.1')
.getOrCreate())
```

Pattern: Fine-Grained Tracker

Dataset tracking solves the dataset dependency problem, but it automatically brings up another question: how do we track the column(s)? Put differently, how do we determine which input columns compose each output column?

Problem

You implemented the Denormalizer pattern to avoid costly joins in a table. The table has grown to more than 30 columns in three years. Your team's composition changes pretty often, and each time, the new members ask you questions about the table dependencies.

You can answer most of the questions with the Dataset Tracker pattern, but one question remains unresolved. Your new colleagues want to know which columns from the upstream tables use each column from your denormalized table.

Solution

This column-level tracking detail is an ideal scenario for using the Fine-Grained Tracker pattern. It provides low-level details at the column or row level about the data origin.

Let's start with the column level. Some solutions support the Dataset Tracker natively, under constraints. That's still the case with Databricks with the Unity Catalog with the system.access.column_lineage tracking table. This feature is also natively present on Azure with the Purview service.

But you can also implement the lineage on your own. In that case, the implementation consists of analyzing the query execution plan and tracking all downstream dependencies for each column. For example, if your query looks like SELECT CONCAT(u.first_name, d.delivery_address) AS user_with_address FROM users u JOIN addresses d ON d.user_id = u.id, the downstream dependencies for user_with_address will be the first_name column from the users table and the delivery_address column from the addresses table.

Figure 10-6 shows how such a column lineage works for the implementations leveraging SQL. As you'll notice, the lineage code analyzes the execution tree to discover all dependencies for the output columns.

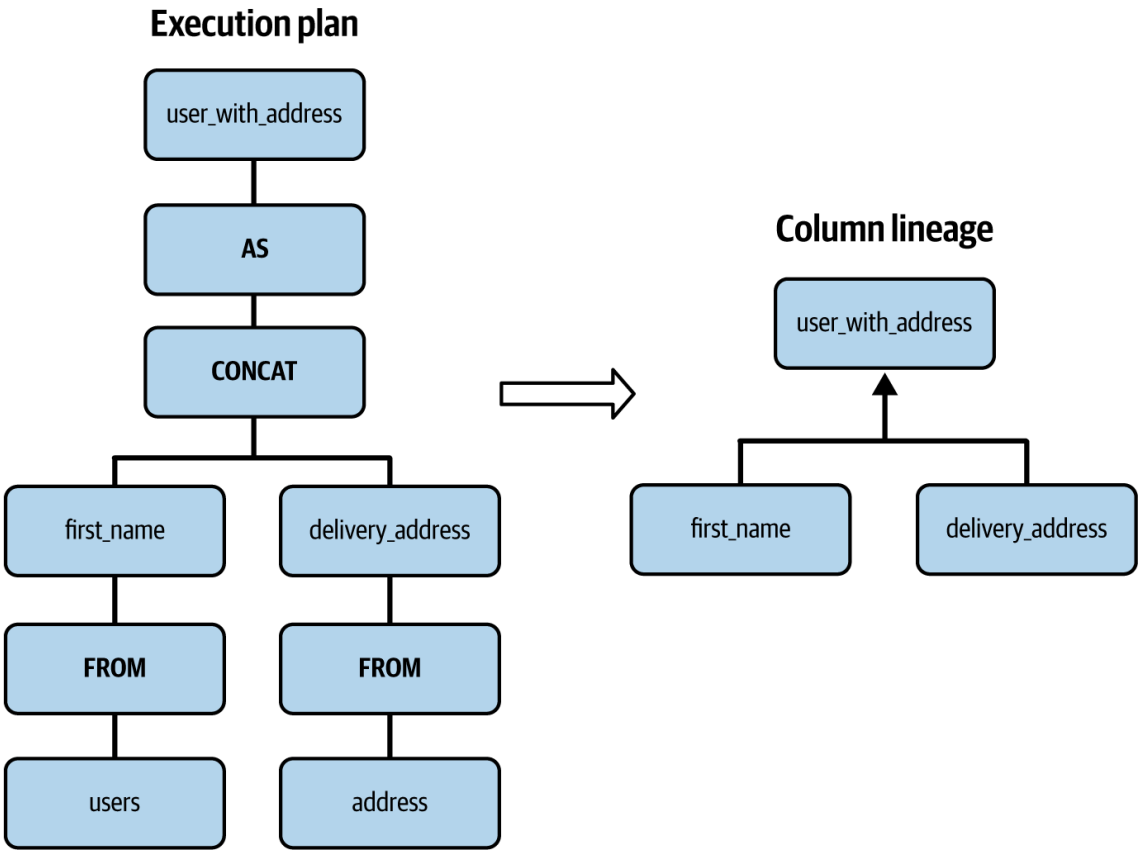

Figure 10-6. Column lineage dependencies for a query concatenating two columns (simplified)

The good news here is that popular data processing solutions like Apache Spark often have native support in data lineage tools, such as the OpenLineage framework.

The low-level tracking might also concern the rows. Here, you would like to know which job produced each row. The implementation consists of adding this information either as an extra attribute or as an additional column, for example, by leveraging the data decoration patterns.

Consequences

The Fine-Grained Tracker provides some extra insight but also has some limits.

Custom code

The column-level lineage analyzes execution plan nodes to detect dependencies. This approach works perfectly well when you use the standard mechanisms of your data processing tool, such as native SQL functions. However, when you transform your data with custom logic, such as programmatic mapping functions, getting the lineage will not be easy. These functions are opaque boxes, where the lineage framework can only see the output but oftentimes doesn't have enough capacity to interpret and understand the coded transformation logic.

Row-level visualization

Dataset and column-level lineage are two common categories in the data lineage world that are widely supported by both the extraction and visualization parts. However, row-level

information, although useful, doesn't belong there. It's more of an additional value that can be useful for debugging data quality issues and identifying jobs that write erroneous rows. Unfortunately, the row-level lineage won't integrate with the classical data lineage visualization tools. As a result, you'll need a separate query layer for the row-level lineage.

Evolution management

The transformations that build your column today may change in the future. Consequently, you must ensure that your column lineage solution, either fully managed or customized, supports this evolution tracking. Otherwise, you might get an incorrect view when it comes to defining the origins of a column after the upstream columns have changed.

Examples

Since the Fine-Grained Tracker at the column level for Apache Spark and OpenLineage requires the same SparkSession function as in the Dataset Tracker example, let's omit it for brevity's sake. If you are interested, you can find the example in the GitHub repo. Instead, let's focus on the row-level lineage, which requires some extra coding effort.

In this example, we're going to use two jobs. The visits_decorator_job will write data to a visits-decorated topic, while the visits_reducer_job will process this data and write it to a visits-reduced topic. The processing step is irrelevant to our example. That's why we'll focus on the tracking details included in the Kafka record's header.

Example 10-18 shows the row-level lineage for our jobs. As you will notice, both jobs set the same attributes, namely the job_version, job_name, and batch_version. The only difference is the parent_lineage attribute present in the visits_reducer_job, where you can find the lineage information from the visits_decorator_job and thus the data source. Having these upstream attributes makes it easier to understand the workflow and automatically makes any debugging discussions with the data producers easier, as you can provide them with extra debugging details.

Example 10-18. Row-level lineage for Apache Spark Structured Streaming

```
# visits_decorator_job

(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('job_name').alias('key'), F.lit(job_name).alias('value')),

 F.struct(F.lit('batch_version').alias('key'), F.lit(str(batch_number)

  .encode('UTF-8')).alias('value'))

)))


# visits_reducer_job

(visits_to_save.withColumn('headers', F.array(

 # same as for visits_decorator_job, plus the parent's lineage
```

```
 F.struct(F.lit('parent_lineage').alias('key'), F.to_json(F.col('headers'))
  .cast('binary').alias('value'))
)))
```

Summary

In this chapter, you learned how to add extra protection to your data workloads by using detectors and trackers. The first section presented two data detector patterns. The Flow Interruption Detector was the first of them. If you have any doubts about the continuity of the data you're processing, the Flow Interruption Detector is an observability solution. Also in this section, you learned about the Skew Detector pattern that you can employ to spot unbalanced datasets and partitions. Using the Skew Detector is a good way to control the completeness of the input dataset, especially for batch workloads working on homogeneous data volumes.

In the next section, you learned about time detectors. They operate in the time space and help spot any latency issues. The first of the described patterns was the Lag Detector. It's particularly interesting to use with streaming pipelines to detect whenever a streaming job falls behind the data producer. Lag detection can be extended with the SLA Misses Detector pattern, which ensures that each data processing task completes within a specified time period.

Finally, in the last section, you learned about the tracking portion of data observability with two data lineage patterns. The first of them focuses on datasets, hence its name, the Dataset Tracker. You will find it useful in big organizations where various teams exchange datasets among themselves and where in the end, you may not know the scope of each team's responsibilities. The Dataset Tracker will provide a global dependency graph among those teams and their datasets. If you need more details, you can leverage the second pattern, the Fine-Grained Tracker. It helps you see not only the big picture but also low-level details of the transformations applied to each column or the jobs producing particular rows.

If you haven't done it already, this chapter should convince you to add the data observability patterns to your system. And by the way, this is the last chapter of this book, but before I let you go—hopefully, to implement some patterns in the real world—I owe you some closing words.

1 Alarm fatigue is definitely something you want to avoid in order to preserve your resources and remain responsive to issues.

2 The processing time window will never be late as it'll always follow the real clock.

3 SLA refactoring is a planned Apache Airflow 3.1 feature detailed in "AIP-57 Refactor SLA Feature", viewed on September 23, 2024.

4 This will look familar to a data mesh–driven organization. If you are curious to learn more about this topic, in which dataset tracking is one of the key elements, you can find more information in two O'Reilly books: *Data Mesh* by Zhamak Dehghani (2022) and *Implementing Data Mesh* by Jean-Georges Perrin and Eric Broda (2024).

5 You can find more details on the integration in the Introduction and the "Supported operators" section of the Apache Airflow documentation.

6 The projects are available at the OpenLineage website and the Marquez Project GitHub.