Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 9. Data Quality Design Patterns

Trust is an important value of a dataset. Exchanging data is like a mutual transaction, in which you either provide or consume a service (dataset). The final goal is to make the producer and consumer happy about this dataset exchange. Unfortunately, you will rarely be excited about working with a dataset that cannot be trusted, as any insights drawn from it could be wrong at any moment.

One of the causes of lost trust is poor dataset quality, which means incompleteness, inaccuracy, and/or inconsistency issues. But the good news is that these issues are not new, and even though data engineers continue to fight against them, there are some design patterns to mitigate data quality issues.

In this chapter, we're going to address data quality issues with the help of design patterns organized into three different categories. In the first category, you will see how to enforce quality and thus avoid exposing data of poor quality to your downstream consumers.

In the next part, you'll see how to address data quality issues at the schema level. Oftentimes, your producers can generate data without any apparent issues, until the day they decide to modify the schema. Depending on the evolution type, this may lead to a fatal failure of your pipeline and thus a loss of trust in your data provider.

In the last part, we're going to see how to guarantee that our enforcement rules today will still be relevant for the data of tomorrow. That's why, in addition to controlling the data and its schema, it's important to observe the dataset and spot any new issues before your consumers do. These observation techniques will help you keep your enforcement rules up to date by providing the freshest overview of the processed datasets.

That's just the context for this chapter. If you are eager to see more concrete examples, I invite you to the first section on quality enforcement.

Quality Enforcement

Ensuring the quality of your dataset means that you will avoid sharing an incomplete, inconsistent, or inaccurate dataset. Quality enforcement is therefore the first category of data quality patterns you'll apply to your pipelines with the goal of sharing trustworthy data.

Pattern: Audit-Write-Audit-Publish

The first way to ensure good dataset quality is to add controls to the data flow. This approach is similar to assertions in unit tests that verify whether your code is performing correctly against an expected input. It's possible to transpose these assertions to data flows and consequently produce the kinds of data quality guards in the pipelines that might stop the whole execution if the dataset doesn't meet expectations.

Problem

Your daily batch ETL job generates statistics for the user visits presented back in Figure 1-1 from Chapter 1. The results have not been good for the past week. In fact, the number of unique visitors dropped by 50%, and the product team considers this to be an issue. As a result, it has started a new marketing campaign to bring visitors to the website.

Today, while you were working on a new feature of this job, you discovered that the unique visitors aggregation is not computed correctly. You informed the product team, which stopped the campaign but asked you to ensure that there will be no similar issue in the future.

Solution

Generated data volume that drops by 50% compared with previous days is a perfect use case where the Audit-Write-Audit-Publish pattern can shine.

Write-Audit-Publish Evolution

The Audit-Write-Audit-Publish (AWAP) pattern is an evolution of the Write-Audit-Publish (WAP) pattern shared by Michelle Ufford at the DataWorks Summit in 2017. The original talk introducing WAP to the world is still available on YouTube at "Whoops, the Numbers Are Wrong! Scaling Data Quality @ Netflix". Unlike WAP, AWAP completes the validation logic with the input data verification to perform some usually lightweight validation on the input dataset.

The idea behind AWAP is to add controls (aka *audit steps*) to ensure that both input and output datasets meet defined business and technical requirements, such as completeness and exactness. Figure 9-1 shows a pipeline with these extra audit steps.

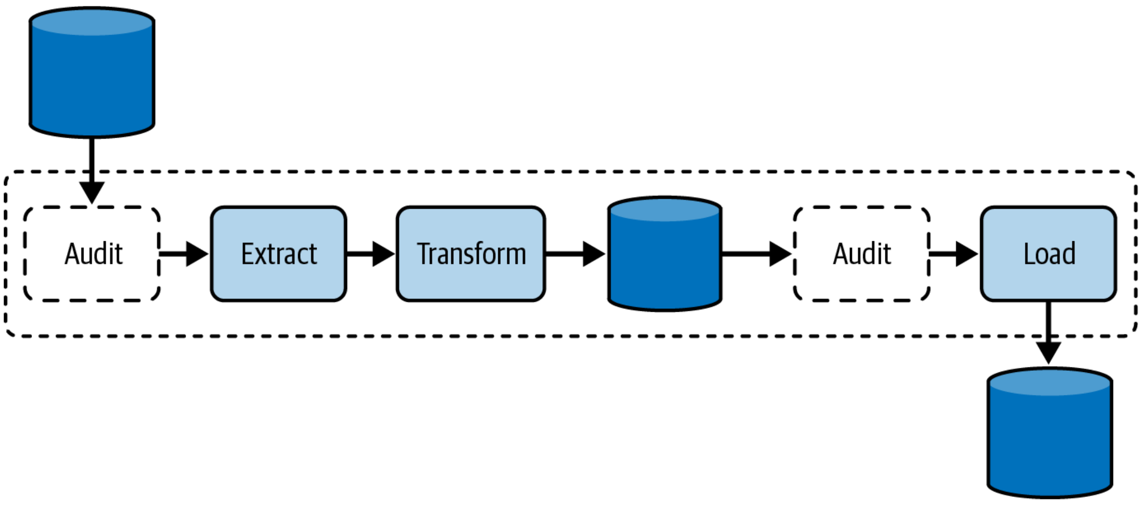


Figure 9-1. The AWAP pattern applied to a pipeline

As you can see in Figure 9-1, the main difference between the two audit tasks is the audited data store:

1. The first audit job is responsible for analyzing the input data source before you start transforming the dataset. Very often, you will limit the validation here to fast operations such as input file format validation, file or table size control, or schema checks. For example, let's imagine you're working on a table that usually has three columns called a, b, and c, and that you want to load a new CSV file to the table. The first audit step could validate whether the CSV file has these three fields present and defined correctly, just by analyzing the first line of the file. Although it's technically possible to validate the full dataset as well, keep in mind that you'll risk reading the dataset twice, once in the first audit step and once in the transformation step.
2. The second audit job validates the transformed data. You can consider it to be an extension of your local unit tests that is going to run on the real dataset. Therefore, the control functions will focus more on the data. For example, if you need to transform columns a, b, and c from the previous point and the transformation should never be NULL, you can add a validation function here.

With these two audit steps, you must be very careful when considering a validation function to be redundant. To help you understand this, let's take a look at an example of the nullability validation from the previous paragraph. If you added this validation on top of the input dataset, you would verify whether the dataset generated by your data provider meets your expectations. On the other hand, if you added the NULL validation to the second audit step, you would ensure that your transformation logic doesn't generate missing values. On the surface, both actions validate NULLs, but as you can see, their intent and scope are different.

A risk with the same validation function is processing the whole dataset many times. If you're concerned about that, maybe because of the data volume you're working on, you should always consider putting the validation action in the most exhaustive place. In the case of our NULL validation, that's the second audit step, where it could detect NULLs coming from both the input dataset and your transformation logic.

Unit Tests and AWAP

Unit tests are important for any system relying on software, including data engineering pipelines. However, with regard to the data, unit tests are static. You create them at some point in time that may reflect current reality but may not represent what will happen in the future. For that reason, I mentioned that the audit steps from the AWAP pattern extend unit tests on top of the real-world data. But don't get me wrong, unit tests (which are local) should always be the first line of defense against any data quality issues caused by incorrect business logic implementation.

Validation in audit steps can operate at the records level and thus validate attributes of particular records, or it can operate at the dataset level to verify overall properties, such as data volume, distinctiveness of particular columns, or even the proportion of NULLs in a column. You may be thinking that the outcome of the audit steps is always a failed pipeline, but it doesn't have to be. Apart from failure, other possible outcomes are as follows:

Data dispatching

If only part of the audited output dataset is invalid, you can still promote the valid portion to the downstream consumers and keep the invalid records in a separate storage. This sounds like the Dead-Letter pattern, but it's different because there are no unexpected runtime errors. Instead, the dead-lettering logic results from your explicit data control mechanism.

Nonblocking audit

If your processed dataset has some imperfections, you may want to promote it to the final storage despite the audit errors. In that case, it's good to annotate it as having some issues so that readers can evaluate trust and not process it if the data doesn't meet their expectations. For example, if one of your columns has an unexpected increase of NULL values but the overall dataset looks fine, consumers who don't rely on that column can still use the dataset. Consumers who use the column can also decide to process the dataset, if the proportion of the missing values is under their acceptable threshold. For the annotation, you can create a data summary entry in a table or in a file, where you would list all possible data quality issues.

The description so far makes it sound like the AWAP pattern works only for batch pipelines. That's not true because stream workloads can also use it with two different approaches depicted in Figure 9-2.

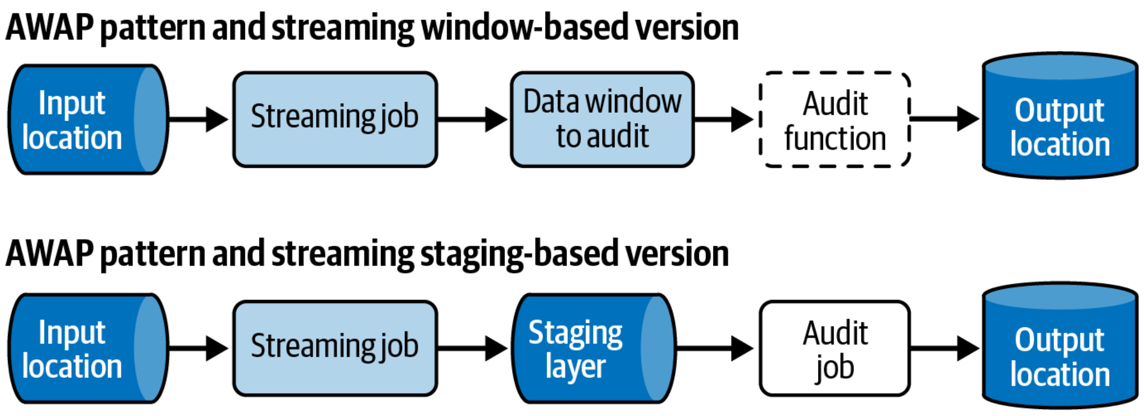


Figure 9-2. The AWAP pattern applied to streaming

The first implementation, which is called *window based* in the diagram, creates processing time windows directly in the streaming job. Once the window closes, the job runs data audit steps that can apply one of the strategies (fail/dispatch/ignore) to the buffered records. The second approach, which is called staging based, doesn't modify the data processing logic. Instead, it only changes the output to write transformed records to a staging layer where an audit job can run before eventually promoting the dataset to the final output location.

As you may have noticed, AWAP in a streaming context follows the more classical WAP approach. The first audit step is missing because data is continuously flowing to the system, and in most cases, it should be simpler to validate the records after the transformation step of the streaming job. This is a real-world example of the exhaustiveness rule for validation functions mentioned previously in this section.

Consequences

The AWAP pattern brings extra safety, but it incurs some extra costs.

Compute cost

Depending on the nature of your audit steps, you may have an additional compute cost. Metadata-based operations, such as validating file formats, will be cheap, but operations working on data, such as row-based validations, will be more expensive. But that's the price you have to pay to ensure the quality of generated data.

Rules coverage

Let's stay with the row validation example here. If you need to verify values for each incoming row, you'll define a set of business rules. Unfortunately, as datasets may evolve over time, the rules from today might not fully cover a dataset of tomorrow. For that reason, it's better not to consider the AWAP-controlled pipelines to be 100% reliable. There is still a risk of forgotten or out-of-date validations that, hopefully, you will spot with one of the patterns in "Quality Observation".

Streaming latency

AWAP in a streaming context may add some extra latency. For example, if you want to assert a NULL values distribution within a processing window, the data delivery will be delayed by the window accumulation period.

An issue may not be an issue

Keep in mind that an issue spotted by the audit steps may not be a real issue. This may sound surprising, but remember, data is dynamic, and something that appears wrong may turn out to be correct. An example here would be an audit step validating the data volume for our blogging platform introduced in "Case Study Used in This Book". If you encounter unexpected success, such as being quoted in social media, normally, you should notice an unexpectedly high number of visits and thus much more data volume to process. Consequently, it doesn't mean something was wrong on the data producer side.

For that reason, you don't need to consider all audit failures to be critical issues. Sometimes, an invalid outcome can only trigger an alert and require further investigation on your part.

Examples

Let's begin this section with an example from Apache Airflow and PostgreSQL, thus SQL-based validation. Example 9-1 shows possible tasks you can create as part of your batch pipeline. First, the pipeline starts by auditing the input dataset with the rules detailed later. If the validation is successful, the pipeline starts the transformation that in turn gets validated before being written to the final data store.

Example 9-1. The AWAP pattern in a batch pipeline

```
audit_file_to_load = PythonOperator(
  task_id='audit_file_to_load',
  python_callable=local_validate_the_file_before_processing
)
transform_file = PythonOperator(
  task_id='transform_file',
  python_callable=flatten_input_visits_to_csv
)
def local_validate_flatten_visits():
 validate_flatten_visits(get_current_context())
audit_transformed_file = PythonOperator(
  task_id='audit_transformed_file',
  python_callable=local_validate_flatten_visits
)
load_flattened_visits_to_final_table = PostgresOperator(
  task_id='load_flattened_visits_to_final_table',
  sql='/sql/load_file_to_visits_table.sql'
)

(next_partition_sensor >> audit_file_to_load >> transform_file
 >> audit_transformed_file >> load_flattened_visits_to_final_table)
```

The input data audit consists of asserting on JSON lines the correctness and overall file size. Example 9-2 shows a snippet of the validation function; you will find all the code in the GitHub repo.

Example 9-2. Input dataset validation (local_validate_the_file_before_processing **function)**

```
if f_size < min_size:
 validation_errors.append(
  f'File is to small. Expected at least {min_size} bytes but got {f_size}')
if lines < min_lines:
 validation_errors.append(
  f'File is too short. Expected at least {min_lines} lines but got {lines}')
if invalid_json_line:
 validation_errors.append(
  f'File contains some invalid JSON lines. The first error found was
  {invalid_json_line}, line {invalid_json_line_number}')

if validation_errors:
  raise Exception('Audit failed for the file:\n-'+"\n-".join(validation_errors))
```

As you can see, the final error message includes all the issues the input file can encounter. The same logic works for the processed dataset, in which the audit function relies on the pandas library to look for any NULL values. Remember, our job operates in a constraintless CSV format that requires these extra NULL checks in the audit step. Otherwise, you could rely on the Constraints Enforcer pattern. Example 9-3 shows this extra audit step for NULL values checks.

Example 9-3. Processed data validation example (validate_flatten_visits **function)**

```
required_columns = ['visit_id', 'event_time', 'user_id', 'page', 'ip', 'login',
 'browser', 'browser_version', 'network_type', 'device_type', 'device_version']
cols_w_nulls = []
visits = pandas.read_csv(partition_file(context, 'csv'), sep=';', header=0)
for validated_column in required_columns:
 if visits[validated_column].isnull().any():
  cols_w_nulls.append(validated_column)


if columns_with_nulls:
 raise Exception('Found nulls in not nullable columns:'+','.join(cols_w_nulls))
```

Let's complete the picture with a streaming example using Apache Spark Structured Streaming. One of the methods for running jobs uses triggers (i.e., time-based expressions defining the execution frequency for the data processing logic). It's a great alternative to processing windows because it's stateless and naturally does not involve state management overhead. First, Example 9-4 shows the code for writing the processed results to a staging table.

Example 9-4. Apache Spark Structured Streaming and a Delta Lake staging table

```
visits = (spark_session.readStream
  .option('kafka.bootstrap.servers', 'localhost:9094').option('subscribe', 'visits')
  .option('startingOffsets', 'EARLIEST').option('maxOffsetsPerTrigger', '50')
  .format('kafka').load()
  .selectExpr('CAST(value AS STRING)')
   .select(F.from_json("value", get_visit_event_schema()).alias("visit"), "value")
  .selectExpr('visit.*')
)
# ...
write_query = (visits.writeStream
  .trigger(processingTime='15 seconds')
  .option('checkpointLocation', checkpoint_dir)
  .foreachBatch(write_dataset_to_staging_table).start())
```

Next, the second job streams the staging table and performs the data quality controls. Depending on the evaluation outcome, it writes the results to the final destination (if there are no issues) or to the errors destination. This logic is omitted in Example 9-5 for the sake of brevity, but you can find it in the GitHub repo.

Example 9-5. Audit job on top of a Delta Lake staging table

```
visits = (spark_session.readStream.format('delta')
 .option('maxBytesPerTrigger', 20000000)
 .table(get_staging_visits_table())
 .withColumn('is_valid', row_validation_expression)
)
# ...
write_query = (visits.writeStream
 .trigger(processingTime='30 seconds')
 .option('checkpointLocation', checkpoint_dir)
 .foreachBatch(audit_dataset_and_write_to_output_table)
 .start())
```

Pattern: Constraints Enforcer

The AWAP pattern validates the data directly from your data processing pipeline. Put differently, the implementation effort is on your end. However, there is an easier way to create trustworthy datasets by delegating those quality controls to the database or the storage format, thus relying on a more declarative approach.

Problem

A batch pipeline processes visits from Figure 1-1 (back in Chapter 1) and writes the results back to a table. Even though it has been running without any issues for several months, you're now getting random NULL values for several required fields. The data processing job is already complex, and you want to avoid adding data validation complexity to it. You're looking for an alternative approach that will fail the loading process if there are any data quality errors, such as missing required fields.

Solution

Delegating responsibility for validation to the database is what the Constraints Enforcer pattern is responsible for.

The implementation starts by identifying the attributes that should have the constraint rules assigned to them. It's a very business-specific step, in which the rules can be driven by your product team or legislation. For example, an orders dataset will certainly require some attributes to be defined, such as the order amount and the buyer's billing address. Unfortunately, that's just an ecommerce-specific example, and there is no one-size-fits-all solution.

Once you identify the attributes, it's time for you to assign the constraints. They can be from different categories:

Type constraints

A type constraint ensures that all values for a given attribute will always be of the same type. It greatly simplifies processing since consumers know what kind of data they're dealing with. Type-based constraints are part of the dataset schema and the backbone of the Schema Consistency pattern.

Nullability constraints

These define an attribute as never missing or possibly missing. If an attribute is defined as not nullable, a nullability constraint will reject any rows with missing values. On the other hand, if an attribute is configured as nullable, the dataset will accept missing values. This setting also communicates to downstream consumers possible operations to add, such as filtering the column that can have null values to eliminate rows with missing values.

Value constraints

These rely on one value or a set of values or expressions that are allowed for an attribute, plus a comparison operator. A value constraint compares the value from the inserted property with the expected value. If the outcome is negative, the record is rejected with a failure. Examples include x <= NOW() for inserted value (x) to never be in the future, and x BETWEEN 1901 AND 2000 for x to be in the 20th century.

Integrity constraints

These are often part of transactional databases modeled with the Normalizer pattern. In that context, integrity constraints ensure that a value present in a table references a real value present in another table. For example, if a website visit references a page that doesn't exist in the pages table, the integrity constraint will be broken, and consequently, the website visit row won't be added to the visits table.

Although this implementation is commonly present in databases, you can encounter it while working with file formats. For example, Delta Lake includes a CHECK operator that will verify each value against the specified condition. Also, serialization formats such as Apache Avro and Apache Protobuf implement the Constraints Enforcer pattern. They natively cover type constraints, and if you install additional extensions, they may also cover value constraints. You'll see this in the Examples section.

The Constraints Enforcer pattern is informative for consumers, as it defines the dataset's shape and possible values, and interactive for producers, as it prevents them from adding records without passing the expected validation controls.

Consequences

Using the Constraints Enforcer pattern is a definitive way to ensure good data quality. It's simpler than writing data validation logic, but it also has some drawbacks.

All-or-nothing semantics

Most of the time, the constraints defined at the database level follow transactional all-or-nothing semantics. This means that if any of the input rows from the ingested dataset don't respect the validation rules, none of the rows will be accepted.

Also, databases often stop at the first encountered error. If you, as a data producer, generate a dataset with multiple issues, you'll need to go back and forth with the database several times to discover all the problems. To mitigate this problem and compile the full list of issues, you could implement the validation rules on the data producer side—but by doing so, you'd lose the informative and interactive advantages presented earlier in this chapter.

Data producer shift

The Constraints Enforcer pattern is data producer–oriented since it exposes the constraints to the data writer. However, different consumers may have different data expectations. For example, a nullable field in the database may be required for some consumers, and as a result, you, as a consumer, may still need to implement data validation or data filtering logic on top of an already constrained dataset.

Constraints coverage

It's not always possible to cover all validation rules. That's especially apparent with table file formats that, for example, may not cover integrity constraints. The constraints from the AWAP pattern are more flexible as the single limitation is your programming language. Consequently, you may need to complete the database constraints with the ones defined in your data processing job.

Examples

Delta Lake has been referenced in this section in a few places, so let's start with this table file format and see how to apply all three categories of constraints. Example 9-6 creates a table with type constraints and nullability constraints, and then, it defines the value constraint, ensuring that the values in the event_time column are always from the past.

Example 9-6. Delta Lake constraints

```
CREATE TABLE default.visits (
 visit_id STRING NOT NULL,
 event_time TIMESTAMP NOT NULL
) USING delta;


ALTER TABLE default.visits ADD CONSTRAINT
  event_time_not_in_the_future CHECK (event_time < NOW() + INTERVAL "1 SECOND")
```

From now on, if any of the inserted rows violates the specified rules, you will get a DELTA_VIOLATE_CONSTRAINT_WITH_VALUES or DELTA_NOT_NULL_CONSTRAINT_VIOLATED error. Consequently, none of the records added in the transaction will be written to the table.

Another, perhaps more surprising, place where you can use the Constraints Enforcer pattern is in serialization file formats, like Protobuf. The library implements the type constraint natively, and if you install protovalidate,[1] you can extend the scope with value constraints.

Example 9-7 shows a visits event annotated with extra validation properties. You can see that we control the minimum length of the visit_id field, the time with the lt_now (aka lower than now) expression, and even specify the page that cannot end with an HTML extension.

Example 9-7. Protobuf and constraints with protovalidate

```
message Visit {
 string visit_id = 1 [(buf.validate.field).string.min_len = 5];
 google.protobuf.Timestamp event_time = 2 [
   (buf.validate.field).timestamp.lt_now = true,
   (buf.validate.field).required = true];
 string user_id = 3 [(buf.validate.field).required = true];
 string page = 4 [(buf.validate.field).cel = {
   message: "Page cannot end with an html extension"
   expression: "this.endsWith('html') == false"
 }, (buf.validate.field).required = true];
}
```

Now, if you call validate(...) on any of the visit class instances and, intentionally or not, break one of the rules, you will get a ValidationError, as in Example 9-8.

Example 9-8. A ValidationError **for a broken protovalidate constraint**

```
Traceback (most recent call last):
 File "...visits_generator.py", line 39, in <module>
   validate(visit_to_send)
 File "...protovalidate/validator.py", line 61, in validate
   raise ValidationError(msg, violations)
protovalidate.validator.ValidationError: invalid Visit
```

Schema Consistency

The schema constraints you discovered with the Constraints Enforcer pattern solve the data consistency problem. However, schemas have a special place in data engineering that is much more complex than simply defining the field types of a table. In this section, you'll learn about other challenges and how to solve them with two design patterns.

Pattern: Schema Compatibility Enforcer

Datasets are dynamic because their values can change over time, and the Constraints Enforcer pattern validates these evolved entries against a set of predefined rules. But what if I told you that schemas can also have this validation? If that's something new to you, let's take a look at the next pattern.

Problem

You're running a sessionization job that you implemented with the Stateful Sessionizer pattern. It ran great for months, but then the team generating your input data made several changes, and as a result, the job has failed many times in the past month. It turns out, the new team removed the fields used by your application, thinking they were obsolete.

After discussing the issue with your new colleagues, you've asked them to build a solution to avoid any schema-breaking changes.

Solution

To ensure that you as the data producer don't introduce any breaking changes, you can use the Schema Compatibility Enforcer pattern.

Depending on your data store, you'll use one of the three available schema compatibility enforcement modes:

Via an external service or library

This is the enforcement mode Apache Kafka's Schema Registry uses to expose an API the producers and consumers communicate with. Schema Registry versions each schema and validates schema changes against the configured compatibility rules. Alternatively, instead of a service, you could use a library. For example, Apache Avro has a SchemaValidator class that you could use to validate that a schema doesn't have incompatible changes. That said, this library doesn't allow you to set the compatibility rule.

Implicit with inserts

This is the enforcement mode for table file formats or relational databases. When you create a new table, you define the constraints, such as nullability, type, or accepted range of values. At the same time, you implicitly set the compatibility mode that prevents any record not respecting the constraints from being written. However, there is no way to define an explicit schema compatibility mode as the implementation relies on an external service or library.

Event driven for data definition language (DDL)

This approach extends the implicit mode. In some relational databases, such as PostgreSQL and SQL Server, you can add event triggers that will run SQL functions before committing any DDL operations such as DROP COLUMN or RENAME COLUMN. The function logic can include your schema enforcement rules and roll back the operation if a user tries to perform an incompatible change. On the other hand, if you don't need such fine-grained control, you can prevent all schema modifications by not granting the ALTER TABLE permission to a user.

All three modes are summarized in Figure 9-3.

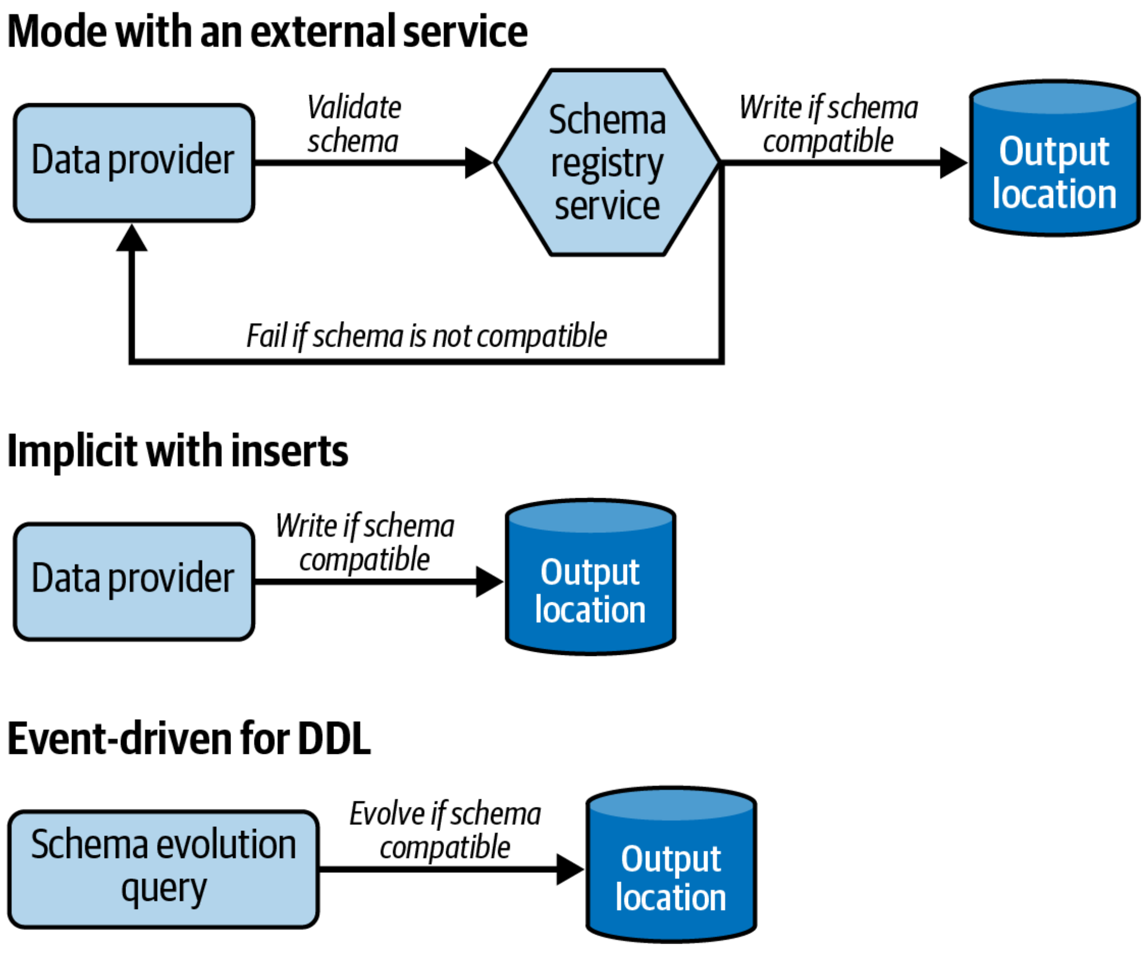


Figure 9-3. Schema compatibility mode workflows

Let's complete this implementation section with an analysis of various compatibility modes you can define for your schemas (if supported by your data store). The schema compatibility mode informs downstream consumers of what evolution they should be ready for. One of the most common compatibility scenarios involves nontransitive rules in which two consecutive schema versions (such as version and version+1 or version and version−1) must remain compatible. The available modes are as follows:2

Backward compatibility

Here, a consumer using a new schema can still read data generated with an old schema. For example, if a new schema has a new optional field, then since the consumer is using the new schema, the added field will simply be missing in the records generated with the old schema.

Forward compatibility

In this mode, a consumer with an old schema can read data generated with a new schema. For example, if an optional field that was in an old schema has been deleted from a new schema, then the removed optional attribute will be missing from the records produced with the new schema. The consumer will see that the field's value is empty, but the emptiness is already part of the contract as the field was marked as optional and thus possibly empty.

Full compatibility

This mode mixes backward and forward compatibilities, so consumers with a new schema can read data generated with a previous schema, and consumers using an old schema can still access data generated with a new schema.

The compatibilities can also be *transitive*. This means that the compatibility between all past (backward) and future (forward) schemas must be guaranteed. Table 9-1 summarizes these compatibility scenarios.

Compatibility modes	**Allowed actions**	**Semantics**
Backward nontransitive Backward transitive	Delete field Add optional field	Consumers with a newer version can read data produced with an older version.
Forward nontransitive Forward transitive	Add field Delete optional field	Consumers with an older version can read data produced with a newer version.
Full nontransitive Full transitive	Add optional field Delete optional field	Consumers with a newer version can read data produced with an older version. Consumers with an older version can read data produced with a newer version.

Table 9-1. Schema compatibility actions summary

As you can see in Table 9-1, the transitive and nontransitive allowed actions are the same. That may be confusing, so let's take a look at an example of backward compatibility to see the transitive mode broken. Our initial schema looks like the one in Example 9-9.

Example 9-9. Initial schema for transitive versus nontransitive allowed actions

```
Schema Order (v0):

  order_id LONG REQUIRED
```

Let's imagine now that we need to add an amount. After all, an order without an amount doesn't make any sense, does it? Since the schema is set to be backward compatible, the amount field must be optional (i.e., it must have a default value, as shown in Example 9-10).

Example 9-10. Backward-compatible version of the Order **schema**

```
Schema Order (v1):

  order_id LONG REQUIRED
```

amount DOUBLE DEFAULT 0.0

Turns out, our product team asked us to remove the default as it might lead to erroneous insights on the data analytics side. We can therefore create a new version with the amount field set as required, as shown in Example 9-11.

Example 9-11. Final version of the Order **schema**

Schema Order (v2):

order_id LONG REQUIRED

amount DOUBLE REQUIRED

For a transitive backward dependency, the last change is not compatible between v0 and v2 since from that standpoint, a consumer using the newest version (v2) can't read the data produced by the very first version (v0). However, the evolution looks fine from the nontransitivity standpoint. Put differently, the consumer using the most recent version (v2) can read a schema produced in the previous version (v1). In that case, even if a data producer omitted the amount field, the schema definition would put a default value in it. As a result, each order would have an amount.

Upon defining the compatibility mode and schema, the data producer interacts with each schema change with this external schema enforcement component. That way, if the produced data contains an evolution not supported by the compatibility mode, it'll be rejected.

Consequences

Even though the benefits outweigh the risks, there are some points to keep in mind.

Interaction overhead

Schema management, particularly via an external schema registry component, adds extra overhead to data generation. The producer must validate records against the most recent schema version.

Schema evolution

Schema evolution will be harder with the Schema Compatibility Enforcer pattern. Any schema change must agree with the schema compatibility level defined for the dataset. This may lead to a situation where renaming a field means adding a new field and deprecating the previous one. However, that's the price you pay for having more reliable data. You'll learn more about this aspect in "Pattern: Schema Migrator".

Examples

Apache Kafka's Schema Registry is the tool that made schema compatibility enforcement popular among data engineers, so naturally, it's part of the first example. To start, you need to define the schema alongside its compatibility mode. In our case, we're setting the schema from Example 9-12 to be forward compatible.

Example 9-12. Schema to register in the Schema Registry

```
{"type": "record", "namespace": "com.waitingforcode.model","name": "Visit",
"fields": [
 {"name": "visit_id", "type": "string"},
 {"name": "event_time",  "type": "int", "logicalType": "time"}
]}
```

Let's say that a new producer wants to generate a record without the visit_id field. Consequently, since writing a record now involves validating its schema against the Schema Registry, the operation will fail with the exception from Example 9-13.

Example 9-13. Schema compatibility error message

```
confluent_kafka.avro.error.ClientError: Incompatible Avro schema:409 message:
 {'error_code': 409, 'message': 'Schema being registered is incompatible with
 an earlier schema for subject "visits_forward-value",
 details: [{errorType:\'READER_FIELD_MISSING_DEFAULT_VALUE\',
 description:\'The field \'visit_id\' at path \'/fields/0\' in
 the old schema has no default value and is missing in the new schema\',
 ...
```

Regarding implicit enforcement, let's see how Delta Lake solves the issue. The table we're going to work with has been created with the columns present in Example 9-14.

Example 9-14. Initial schema for a Delta Lake schema enforcement example

```
root
|-- visit_id: string (nullable = true)
|-- page: string (nullable = true)
|-- event_time: long (nullable = true)
```

Now, let's imagine a producer who adds an extra column called ad_id. Consequently, since Delta Lake doesn't modify the schema without your permission, it will detect this change as incompatible with the current schema and respond with the exception from Example 9-15.

Example 9-15. Implicit schema enforcement in Delta Lake

```
pyspark.errors.exceptions.captured.AnalysisException: A schema mismatch detected when
writing to the Delta table
...

Table schema:
root
```

-- visit_id: string (nullable = true)

-- page: string (nullable = true)

Data schema:

root

-- visit_id: string (nullable = true)

-- page: string (nullable = true)

-- ad_id: string (nullable = true)

Pattern: Schema Migrator

Ensuring schema correctness prevents producers from making incompatible changes and prevents consumers from being interrupted if those incompatible modifications are possible. However, one schema-related problem still remains: how to keep consumers safe while giving them the ability to perform breaking schema changes, such as field type evolution and renaming?

Problem

You're looking to improve the structure of the visit events your jobs are generating downstream. From day one, you wanted to be user friendly, and for that reason you have been adding new fields without bothering your consumers. As a result, some of your domain-related fields are dispersed across an entire message that sometimes has up to 60 attributes, which is too many for most uses and makes understanding the domain very challenging.

Many of your consumers are complaining about difficulties related to processing and understanding the complicated domain. Ideally, they would like to have related attributes grouped in the same entity. For example, user-related attributes like login, email address, and age should be part of a single attribute called *user*.

You don't want to radically change the existing schema because that would break its compatibility. However, you do want to improve the organization of the attributes while giving your consumers some time to migrate to the new format.

Solution

You can't solve the problem with the Schema Compatibility Enforcer pattern as it only controls the types of changes that can be made. The solution relies on the Schema Migrator design pattern that enables schema evolution.

Transitive Compatibility

The Schema Migrator requires the schema compatibility to not be transitive. Otherwise, no field removal or renaming would be possible, as the transitive compatibility level guarantees consistency across all versions.

The first step consists of identifying the evolution. Three scenarios are possible here:

Rename

This will happen whenever you or your consumers find a given attribute name to be wrong or difficult to understand.

Type change

This is the scenario from our problem statement. Here, we either want to organize a schema better (for example, by simplifying it after multiple changes), or we simply want to optimize it for processing (for example, by adapting a heterogeneous date time text attribute to an epoch timestamp).

Removal

This is easy if you have a 100% guarantee that there are no downstream consumers processing the attribute you want to remove. If you don't, then you will need to find a substitution for them or even cancel the removal action.

Let's focus first on the most challenging scenarios, the rename and the type change. In both cases, you need to start by creating the new field with the renamed or retyped attribute. Next, you need to agree with your consumers on the transition time. Then, during that period, they will receive the previous and the new attributes at the same time. Only after reaching the deadline can you create a new version of the schema that contains only the modified version of the attribute.

Data Lineage

To detect whether an attribute is used by your consumers or not, you can rely on the Fine-Grained Tracker pattern in Chapter 10.

The removal scenario is slightly different as it requires agreeing with consumers on the field removal period. Again, once the deadline passes, you can create a new schema version, this time without the deleted property.

Consequences

The Schema Migrator pattern relies on a grace period for schema migration. During that time, the old schema is still valid and can be processed by consumers. As you may have deduced, this impacts the data size.

Size impact

This is a natural consequence of the Schema Migrator, which provides some safety mechanisms for performing schema migration but also incurs costs in the form of storage space, network transfer, and I/O as there is more data to save.

For some data formats, having a lot of fields is even officially discouraged. For example, Protobuf, in its "Proto Best Practices", warns against using hundreds of fields because each of them, even the unpopulated ones, takes up at least 65 bytes. The overall size of the Protobuf-generated builders can therefore reach the compilation limits of some languages like Java.

Size also has an impact on the metadata and statistics layer. At the time of this writing, Delta Lake collects statistics on the first 32 columns by default. Although you can change that, it may impact the writing time.

Impossible removal

The Schema Migrator pattern has some implementation limits in the field removal scenario. If a field is used by one of your consumers, removing it will not be possible if you cannot provide an alternative attribute.

Examples

Since the schema migration workflow is the same for various technologies, let's focus here on one data format and understand what happens if the schema migration doesn't follow the Schema Migrator pattern.

The consumer in our example extracts the visits of connected users to a dedicated table with the query from Example 9-16.

Example 9-16. Reading visits of connected users

```
INSERT INTO dedp.connected_users_visits
 SELECT visit_id, event_time, user_id, page, ip, login, from_page FROM dedp.visits
 WHERE is_connected = true AND from_page IS NOT NULL;
```

Now, let's suppose we just realized that the from_page column is poorly named and a better name would be referral. The worst thing we could do, because it would break consumers' workload, would be to run the rename operation directly with ALTER TABLE dedp.visits RENAME COLUMN from_page TO referral. The consumer would not be able to see the new data, as its query will fail first (see Example 9-17).

Example 9-17. Error that might happen without the Schema Migrator pattern

```
ERROR:  column "from_page" does not exist
LINE 2:     SELECT visit_id, event_time, user_id,  ..
```

To avoid this issue, you should migrate the rename by creating a new column first:

Example 9-18. Rename in the sense of Schema Migrator

```
ALTER TABLE dedp.visits ADD COLUMN referral VARCHAR(25) NOT NULL
```

You can remove the previous column only once your consumers adapt their workloads.

Since a similar workflow exists for Protobuf and Delta Lake,[3] I'll omit them from this section, but if you are interested, you can find out about them in the GitHub repo.

Quality Observation

Remember, datasets are dynamic. They change, and the constraint rules you define today may not be valid tomorrow. That's why it's important to observe what's going on with datasets and be ready to adapt the existing rules or add new constraints.

Pattern: Offline Observer

Observation patterns can be organized according to their place in the data pipeline. The first type of pattern lives as a separate observation component that doesn't interfere with the data processing workflow.

Problem

You started a new data pipeline this month, and you haven't encountered a lot of data quality issues. The dataset is fully structured, and all business rules are correctly enforced by quality enforcement patterns. However, from your previous project, you know this won't last as the upstream dataset will evolve in the coming months. For that reason, you want to monitor the properties of the dataset, such as the distribution of values and the number of nulls per column. Since everything is fine at the moment, this monitoring layer shouldn't block your main pipeline.

Solution

In a scenario when the monitoring shouldn't block the processing workflow, it's best to opt for the Offline Observer pattern.

The implementation consists of creating a data observability job that will analyze the processed records and enhance the existing monitoring layer with extra insight. The insight will depend on the business context, but it can include properties like distribution of values, number of nulls in nullable fields, new but not processed fields in the input dataset, etc. That way, you can store these parameters and spot any data quality issues over time.

The data observability job doesn't impact the data generation process. It runs independently and could even be executed on a completely different schedule. For example, assuming all your data generators run throughout the day, you may want to schedule all observability jobs to run at night to avoid resource concurrency issues.

Observability Versus Auditing

Observability is not the same as auditing. An audit validates the dataset and is a blocking operation (i.e., whenever it detects some issues, the pipeline will block the pipeline). Observability is a nonblocking approach that monitors the datasets (i.e., it helps detect any issues but will not prevent the pipeline from moving on).

Consequences

Decorrelating data generation from data observation is good as it doesn't impact production resources. Unfortunately, there is another side of the coin.

Time accuracy

Since an offline observation job can run on any schedule, including much later than the data generator, it may not happen on a timely basis. In other words, the insight may come too late since all downstream consumers could already have processed a dataset with new data quality issues.

Compute resources

As the data observation job will be running on the side, you may be tempted to schedule it less frequently than the data generation job. For example, for hourly batch processing, you may execute the data observation job only once every 24 hours. Although this approach is valid, you need to be aware that it may require more compute resources as, instead of dealing with hourly data changes, you'll have to process 24 hours at once.

Eventually, you could consider sampling the observed dataset and therefore using only parts of it. Unfortunately, by extracting a subset to observe, you may miss some interesting observations.

Examples

The first example of the Offline Observer pattern is an Apache Airflow pipeline that runs on a different schedule than the data generation pipeline, asserts the quality of the generated dataset so far, and writes the statistics to a monitoring layer. The overall workflow includes the steps defined in Example 9-19.

Example 9-19. Tasks in an Offline Observer pipeline

```
wait_for_new_data = SqlSensor(...)

record_new_observation_state = PostgresOperator(...)

insert_new_observations = PostgresOperator(...)

wait_for_new_data >> record_new_observation_state >> insert_new_observations
```

Whenever there is new data to process, the observation job records a new observation state that includes the IDs of the first and last processed rows. This operation is required for idempotency to guarantee that in case of any row changes in the observed table, the analysis scope will be the same and thus consistent. Example 9-20 shows the data observation state recording query that's executed as part of the record_new_observation_state task.

Example 9-20. State recording query

```
INSERT INTO dedp.visits_monitoring_state (execution_time, first_row_id, last_row_id)

 SELECT

  '{{ execution_date }}' AS execution_time,

  MIN(id) AS first_row_id, MAX(id) AS last_row_id

 FROM dedp.visits_output

 WHERE id > COALESCE(

   (SELECT last_row_id FROM dedp.visits_monitoring_state WHERE

    execution_time = '{{ prev_execution_date }}'::TIMESTAMP),

   0

 )
```

Later, the Offline Observer pipeline generates the observation by performing aggregations on top of the selected first and last row IDs. The query is present in Example 9-21.

Example 9-21. Data observation query

```
INSERT INTO dedp.visits_monitoring(execution_time, all_rows, invalid_event_time,

 invalid_user_id, invalid_page, invalid_context)
```

```
SELECT
 '{{ execution_date }}' AS execution_time,
 COUNT(*) AS all_rows,
 ...
 SUM(CASE WHEN context IS NULL THEN 1 ELSE 0 END) AS invalid_context
FROM dedp.visits_output
 WHERE id BETWEEN
 (SELECT first_row_id FROM dedp.visits_monitoring_state WHERE
  execution_time = '{{ execution_date }}')
 AND
  (SELECT last_row_id FROM dedp.visits_monitoring_state WHERE
  execution_time = '{{ execution_date }}');
```

Implementing the Offline Observer is also possible for streaming pipelines. As with the batch pipelines, there is a separate job running on top of the processed data and generating observations. In the next example, we're going to analyze the data processing lag of the data producer and some data quality metrics. Example 9-22 shows the metrics for missing rows in an Apache Spark job.

Example 9-22. Offline Observer with Apache Spark Structured Streaming

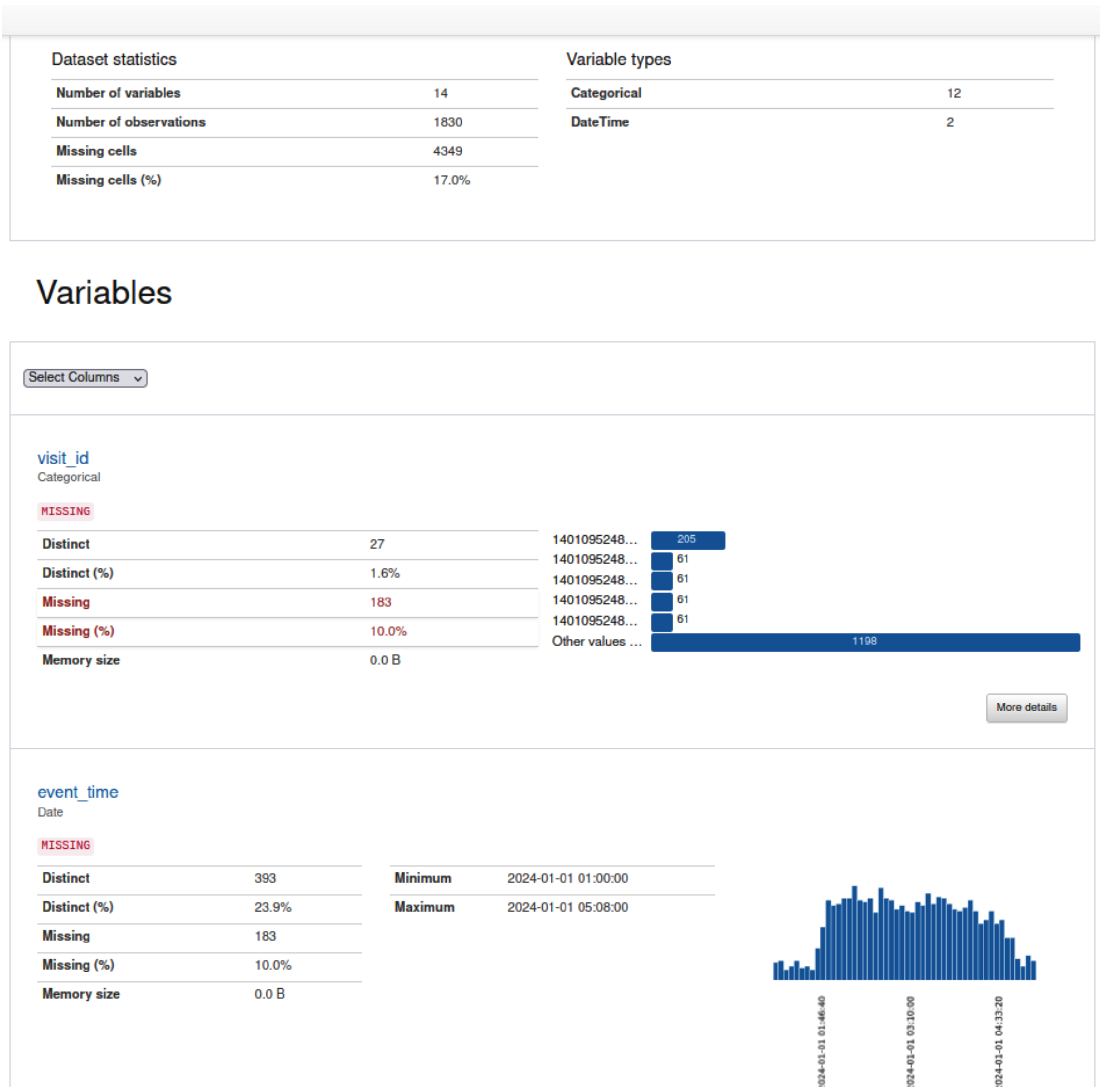
```
visits_to_observe = (input_data_stream
 .selectExpr('CAST(value AS STRING)')
 .select(functions.from_json(functions.col('value'), visit_schema).alias('visit'))
 .selectExpr('visit.*')
 .select('visit_id', 'event_time', 'user_id', 'page', 'context.referral',...)
 )
query = (visits_to_observe.writeStream.foreachBatch(generate_and_write_observations)
.option('checkpointLocation', checkpoint_location).start())
```

All of the observation logic is present in the generate_and_write_observations function. In the first step, it executes the same data observation query as in the Apache Airflow version presented previously. Then, it performs an extra operation and generates an HTML data profile report with the help of the ydata-profiling library (see Example 9-23).4

Example 9-23. Data profiling for the processed dataset

```
def generate_profile_html_report(visits_dataframe: DataFrame, batch_version: int):
 profile = ProfileReport(visits_dataframe, minimal=True)
 profile.to_file(f'{base_dir}/profile_{batch_version}.html')
```

The generated HTML page describes the characteristics of the observed dataset that you can use later to add, modify, or delete existing data quality rules in the enforcement step (see Figure 9-4).



Dataset statistics

Number of variables	14
Number of observations	1830
Missing cells	4349
Missing cells (%)	17.0%

Variable types

Categorical	12
DateTime	2

Variables

visit_id
Categorical
MISSING

Distinct	27
Distinct (%)	1.6%
Missing	183
Missing (%)	10.0%
Memory size	0.0 B

1401095248...	205
1401095248...	61
1401095248...	61
1401095248...	61
1401095248...	61
Other values ...	1198

event_time
Date
MISSING

Distinct	393
Distinct (%)	23.9%
Missing	183
Missing (%)	10.0%
Memory size	0.0 B

Minimum	2024-01-01 01:00:00
Maximum	2024-01-01 05:08:00

Figure 9-4. A data profile for the observed dataset

When it comes to lag detection, the function compares the most recently committed offset by the data generation job from the checkpoint location with the most recent offset present in the input topic. Example 9-24 shows this detection logic.

Example 9-24. Lag detection function

```
def get_last_offsets_per_partition(self) -> Dict[str, int]:
 last_processed_offsets = self._read_last_processed_offsets()
 last_available_offsets = self._read_last_available_offsets()


 offsets_lag = {}
 for partition, offset in last_available_offsets.items():
  lag = offset - last_processed_offsets[partition]
  offsets_lag[partition] = lag
 return offsets_lag
```

For the sake of concision, that was only a code snippet. The whole example is available in the GitHub repo.

Pattern: Online Observer

If latency between data processing and data observation via the Offline Observer pattern is an issue, you can opt for a more real-time alternative pattern, which is the Online Observer.

Problem

Last week, your data analytics colleagues complained about an unexpected format in the zip code field. It turns out that there is some data regression in the upstream dataset, and you couldn't prevent it with the data trust rules in place. Your Offline Observer did indeed discover the issue, but since it runs once per week, you couldn't detect the problem before your users did. In the future, you would like to avoid this kind of problem, so you want to be able to keep your consumers from finding out about data quality issues and fix them sooner than after one week.

Solution

The problem we've presented is a perfect example of the Offline Observer's limitations. Thankfully, overcoming it is relatively simple with the opposite pattern, which is called the Online Observer pattern.

The Online Observer still relies on a data observation job to generate all observation metrics. However, the difference between this pattern and the offline approach is the time at which it executes. The Online Observer's job is an intrinsic part of the data generation pipeline, and as a result, the produced insight is available just after the data generation. That approach can help avoid many communication and technical issues with the downstream consumers.

But a question arises: where should we put this observation job? If we reason about our data generator in terms of ETL or ELT steps, the most popular place to put it is after the Transform stage, where you can orchestrate the observation job in the Parallel Split pattern or Local Sequencer pattern (see Figure 9-5).

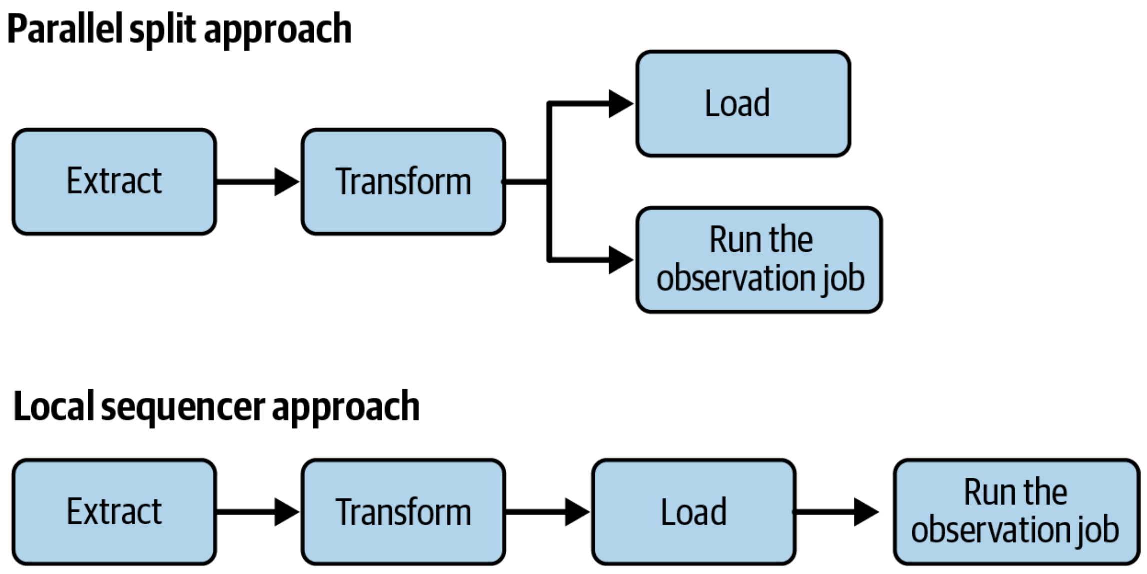


Figure 9-5. Approaches to implementing the Online Observer in a batch pipeline

When it comes to streaming pipelines, you need to integrate the observation logic into the data generation job. Even though this sounds like the batch implementation, there is a significant difference as you will not be able to run the observation step as a separate pipeline. Consequently, any issues with the data observability part, such as unexpected error or memory issues, may impact the whole job. You can try to mitigate this risk by sampling the dataset to perform data observation and then accepting the fact that you'll miss out on some insights during the process. Figure 9-6 shows this dependency between data processing and data observation generation.

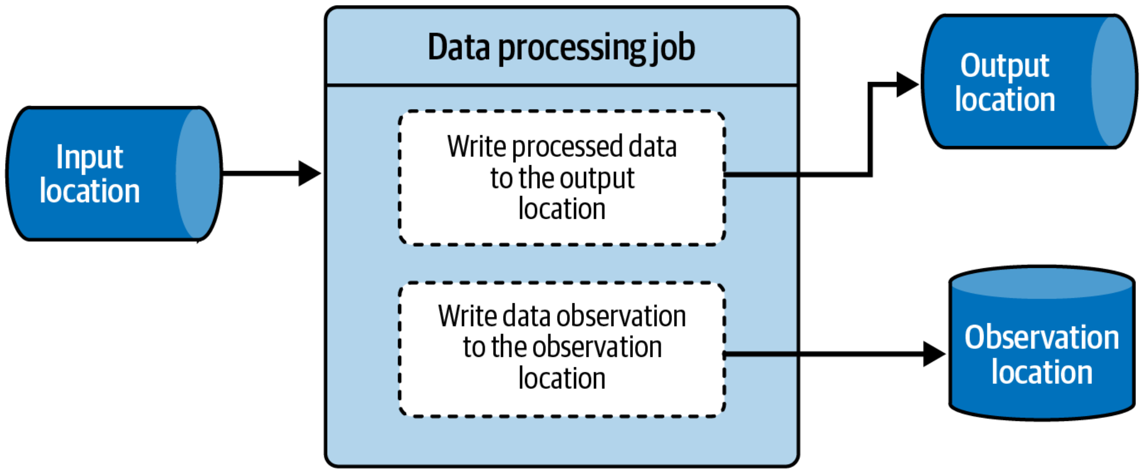

Data processing job
Input location
Write processed data to the output location
Write data observation to the observation location
Output location
Observation location

Figure 9-6. Online data observer for a streaming job

Not Only the Data

Even though this section discusses observability related to the processed data, observability covers a wider scope. It also includes technical metadata, such as the CPU, memory, or disk usage of your data engineering tools. Most of the time, they'll be near real-time measurements and therefore available in the Online Observer pattern.

Consequences

Even though the Online Observer pattern addresses the time accuracy issue of the Offline Observer, it has some gotchas.

Extra delays

If you use the Local Sequencer approach to integrate the data observation job, you'll add it as an extra step at the end of the pipeline. Naturally, this will delay the pipeline's completion. It's good to keep in mind that adding this extra monitoring step to the main workflow doesn't come for free.

Parallel splits

The Parallel Split approach adds an extra parallelism to the pipeline by running the observation and data loading steps at the same time. However, it also brings a danger of observing a partially valid dataset. To help you understand what this means, let's look at an example of a dataset with date time attributes loaded into a database. If the date time format is different from the one expected by the database, the database will be missing the date time values. However, the data observation step, since it runs on the loaded dataset directly, will not see this issue.

To mitigate this problem, you should use the Local Sequencer approach that observes the dataset that's exposed to consumers. Eventually, you can decide to apply your data observation scope to the processed and not exposed dataset. In this logic, the observation focuses on the transformation instead of the data loading task. Consequently, even though you might not face any data loading issues at the moment, this strategy may not be relevant throughout the whole lifespan of the pipeline.

Examples

In this section, I'm going to omit the observation code because it's the same as for the Offline Observer pattern and you can find it in the GitHub repo. Instead, let's see how to transform the offline observation code into a more reactive online one. The batch pipeline now integrates the data observation step into the data processing pipeline, and overall, the tasks look like those in Example 9-25.

Example 9-25. Online Observer for an Apache Airflow batch pipeline

```
wait_for_new_data = SqlSensor(...)

record_new_synchronization_state = PostgresOperator(...)

clean_previously_added_visits = PostgresOperator(...)

copy_new_visits = PostgresOperator(...)

record_new_observation_state = PostgresOperator(...)

insert_new_observations = PostgresOperator(...)


wait_for_new_data >> record_new_synchronization_state

 >> clean_previously_added_visits >> copy_new_visits

copy_new_visits >> record_new_observation_state >> insert_new_observations
```

As you'll notice, the observation pipeline from Example 9-19 is now part of the same data generation pipeline. That does indeed involve the potential risk of the pipeline's failure if there are any issues at the data observation level. To overcome this risk, you add a final task that does nothing and that will be triggered independently on the observation job. Consequently, the execution of this task will mark the pipeline as successful even in cases of observation failure. In Apache Airflow, you can achieve this with a trigger rule set to all_done.

What about streaming pipelines? Here, the code is also merged but the operation involves two major changes: lag detection and accumulators. The data reader now includes two extra columns, which are the partition number and the offset position for each retrieved row. Consequently, the lag detector code relies on them to get the most recently processed record in the microbatch. Example 9-26 shows these initial changes with a new class added to store the partition-offset pairs, plus a new accumulator object.

Example 9-26. Online Observer and lag detection adaptation

```
@dataclasses.dataclass
class PartitionWithOffset:
 partition: int
 offset: int



class PartitionToMaxOffsetAccumulatorParam(AccumulatorParam):
 def zero(self, default_max: PartitionWithOffset):
  return []

 def addInPlace(self, partitions_with_offsets: List[PartitionWithOffset],
   new_max_candidate: PartitionWithOffset):
  partitions_with_offsets.append(new_max_candidate)
  return partitions_with_offsets

def write_to_kafka_with_observer(visits: DataFrame, batch_number: int):
 ctx = visits_to_analyze.sparkSession.sparkContext
 max_offsets_tracker = ctx.accumulator([], PartitionToMaxOffsetAccumulatorParam())

 def analyze_generated_records(visits_iterator: Iterator[Row]):
  for visit_record in visits_iterator:
```

```
  # ...
  if visit_record.offset > max_local_offset:
   max_local_offset = visit_record.offset
  current_partition = visit_record.partition

  max_offsets_tracker.add(PartitionWithOffset(partition=current_partition,
   offset=max_local_offset))

visits_to_analyze.foreachPartition(analyze_generated_records)
```

Another modification is the use of accumulators to avoid complex SQL queries generating both numbers of invalid rows and max offsets per partition. As a result, the part summarizing invalid rows is also adapted to this new logic (see Example 9-27).

Example 9-27. Online Observer and invalid records summary

```
accumulators = {'event_time': spark_context.accumulator(0),
 'user_id': spark_context.accumulator(0), 'page': spark_context.accumulator(0)}
all_events_accumulator = spark_context.accumulator(0)

def analyze_generated_records(visits_iterator: Iterator[Row]):
 for visit_record in visits_iterator:
  if not visit_record.event_time:
   accumulators['event_time'].add(1)
  if not visit_record.user_id:
   accumulators['user_id'].add(1)
  if not visit_record.page:
   accumulators['page'].add(1)
# ...

observation_dump = {
 '@timestamp': datetime.utcnow().isoformat(),
 'invalid_event_time': accumulators['event_time'].value,
 'invalid_user_id': accumulators['user_id'].value,
 'invalid_page': accumulators['page'].value,
 'all_events': all_events_accumulator.value,

# ...
```

As a side note, *accumulators* are Apache Spark–specific components that run locally on each executor as long as you don't invoke the value method. When the value function is effectively called, the executors send their local accumulators to the main node of the cluster, which performs the results aggregation. In our data observation example, using accumulators is a great way to avoid querying the input dataset twice (once for the lag and once for the invalid columns).

Summary

In this chapter, you learned about three important components you can use to build trustworthy datasets. The first section covered design patterns related to quality enforcement. You saw there how to improve quality in different layers. The first layer was the pipeline, where the AWAP pattern helps you avoid processing and exposing poor-quality data. More protection comes from the databases, where you can define conditions on the inserted fields with the Constraints Enforcer pattern. Finally, you learned about an external component that helps keep the schema consistent within the Schema Compatibility Enforcer pattern.

Although enforcing constraints and controls prevents the publication of poor-quality datasets, it doesn't guarantee that there will be no issues. In fact, it only guarantees that there will be no issues with the rules you defined. But unfortunately, you may miss defining some of them or may simply need to adapt them to the evolved dataset. To overcome the issue of staying up-to-date, you will rely on the patterns from "Quality Observation". There, you learned about the Offline Observer and Online Observer patterns, which work on different levels. The former is a detached component that runs independently on the data generation pipeline, while the latter is the opposite. Which approach is right for you depends on your willingness to trade time for accuracy.

With that, we're coming slowly but surely to the end of our data engineering design patterns journey. However, there's one topic left to cover: the data observability that will help you detect issues in your data processing jobs and datasets. That's what the next chapter is all about!

1 The protovalidate GitHub repository covers the capacities of protovalidate more extensively.

2 The list omits the "None" mode on purpose as it doesn't enforce anything and shouldn't be used as part of the Schema Compatibility Enforcer pattern.

3 Protobuf's rename operation is safe because the encoded version doesn't store the field's name but only stores its tag. However, changing the type might not be safe, as per the Protobuf official documentation.

4 If you need to set up ydata-profiling, the official documentation provides all necessary information.