Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 4. Idempotency Design Patterns

Each data engineering activity eventually leads to errors—you already know that from the previous chapter. Thankfully, correctly implemented error management design patterns address most of the issues. Yes, you read that correctly: most, not all. But why?

Let's take a look at an example of an automatic recovery from a temporary failure. From the engineering standpoint, that's a great feature as you don't have anything to do besides configuring the number of attempts to retry. However, from the data perspective, this great feature brings a serious challenge for consistency. A retried task or job might replay already successful write operations in the target data store, leading to duplication in the best-case scenario. You read that right: duplication is the best-case scenario because duplicates can be removed on the consumer's side. But let's imagine the contrary. The retried item generates duplicates that cannot be removed because you can't even tell they represent the same data! Welcome to your nightmare and bad publicity for your dataset.

Hopefully, you can mitigate these issues with the idempotency design patterns presented in this chapter. But before you see how they apply to data engineering, let's recall the idempotency definition. The best example to explain it is the absolute function. You know, it's the simple method that returns a positive number even if the input argument is a negative number. Why is it idempotent? Because no matter how many times you invoke the function, you always get the same result. In other words, absolute(-1) == absolute(absolute(absolute(-1))).

Idempotency in a data engineering context has the same purpose. It's a way to ensure that no matter how many times you run a data processing job, you'll always get consistent output without duplicates or with clearly identifiable duplicates. By the way, avoiding duplicates will not always be possible. If you generate the data to a messaging system that doesn't support transactional producers, retries can still generate duplicated entries. However, thanks to idempotent processing, your consumers will be able to identify those records as such.

In this chapter, you'll discover various idempotency approaches in data engineering. You'll learn what to do if you can fully overwrite the dataset or when you only have its subset available. You'll also learn how to leverage databases to implement an idempotency strategy. Finally, you'll see a design pattern to keep the dataset immutable but idempotent.

And one last thing before you see the patterns: I'd like to leave here a special mention of Maxime Beauchemin, who made idempotency popular in 2018 with his state-of-the-art article "Functional Data Engineering: A Modern Paradigm for Batch Data Processing".

Overwriting

The first idempotency family covers the data removal scenario. Removing existing data before writing new data is the easiest approach. However, running it on big datasets can be compute intensive. For that reason, to handle the removal, you can use data- or metadata-based solutions.

Pattern: Fast Metadata Cleaner

Metadata operations are often the fastest since they don't need to interact with the data files. Instead, they operate on a much smaller layer that describes these data files. Because of that, we often say that the metadata part operates on the logical level instead of the physical one. The next pattern you're going to see leverages metadata to enable fast data cleaning.

Problem

Your daily batch job processes between 500 GB and 1.5 TB of visits data events. To guarantee idempotency, you define two steps. The first action removes all rows added to the table by the previous run with a DELETE operation. The second task inserts processed rows with an INSERT operation.

The workflow ran fine for three weeks, but then it started suffering from latency issues. Over the past several weeks, the table has grown a lot and the DELETE task's performance has degraded considerably. You're looking now for a more scalable and idempotent pipeline design for this continuously growing table and the daily batch job.

Solution

DELETE is probably the first operation that comes to mind for data removal. Unfortunately, it may perform poorly on big volumes of data as it's often a two-step action. A DELETE has to first identify the rows to delete and later overwrite the all identified data files. Thankfully, faster alternatives relying on the metadata operations exist. DROP TABLE and TRUNCATE TABLE are two such operations and are the building blocks of the Fast Metadata Cleaner pattern.

TRUNCATE TABLE table_a = DELETE FROM table_a

Semantically, the TRUNCATE TABLE command does the same thing as DELETE FROM without conditions. In both cases you'll get all records removed. However, under the hood, TRUNCATE is different as it doesn't do the table scan. For that reason, it's classified as a *metadata operation*.

But how can truncating or dropping a table replace physical deletes? It all boils down to changing your perceptions. Instead of considering the dataset as a single monolithic unit, you can think about it as multiple physically divided datasets that together form a whole logical data unit. Put differently, you can store the dataset in multiple tables and expose it from a single place, like a view. Figure 4-1 shows a high-level example of such an incremental workload in which weekly tables compose the final yearly dataset.

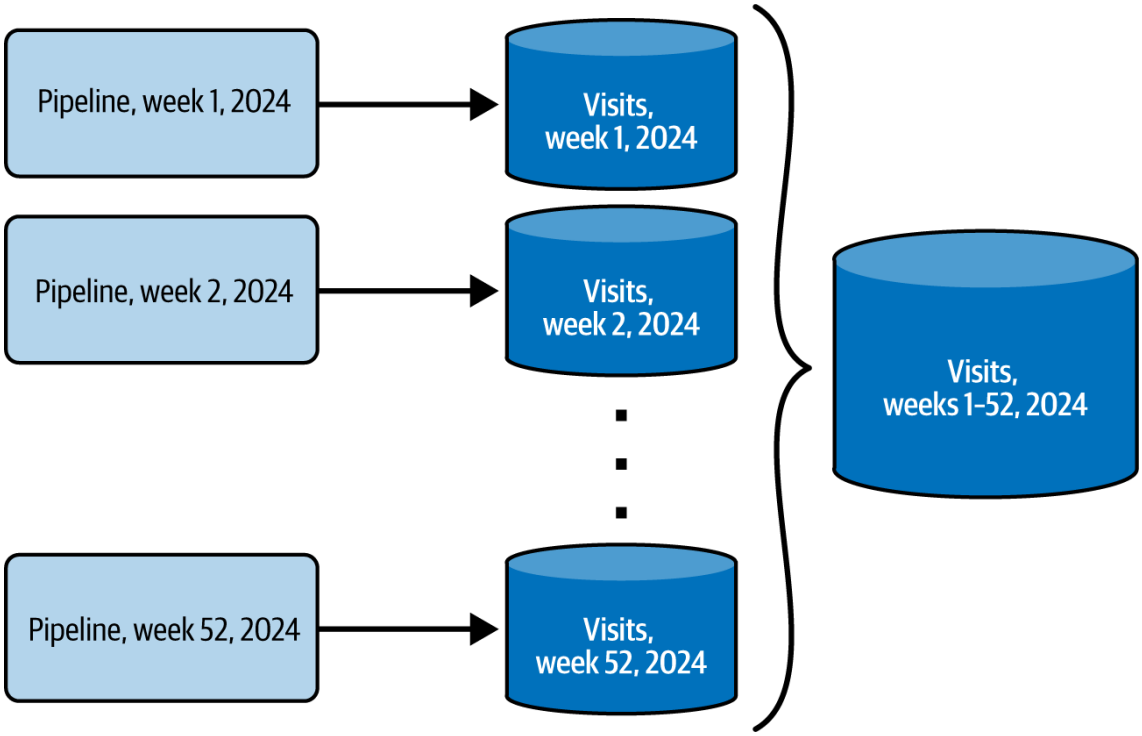

Figure 4-1. Physically isolated dataset in weekly tables and a common data exposition view for all the weeks in each year

To achieve idempotency, the Fast Metadata Cleaner pattern relies on dataset partitioning and data orchestration. You need to define the partitioning carefully since it directly impacts the *idempotency granularity*. What does that mean? As you saw in Figure 4-1, the whole visits dataset is composed of 52 weekly (aka partitioned) tables. The idempotency granularity is one week. In other words, this granularity defines at the same time the units on top of which you can apply the metadata operations to clean the table. It has an important consequence for backfilling, but I'll let you discover it in the next section.

Next, you have to adapt the data orchestration to the idempotency granularity. The adaptation consists of adding these extra steps:

- Analyze the execution date and decide whether the pipeline should start a new idempotency granularity or continue with the previous one. For example, you can use the Exclusive Choice pattern to analyze current execution context and decide what to do next. If you deal with weekly tables and the analysis finds that the pipeline's execution day is Monday, the pipeline will follow the initialization branch. Otherwise, it can go directly to the data insertion part.
- Create the idempotency environment. Here, you'll leverage two metadata operations, namely, TRUNCATE TABLE or DROP TABLE. The solution using TRUNCATE will often be preceded with a task to create the idempotency context table, whereas the approach leveraging DROP will be followed by the table's creation. Using DROP has another implication, but we need to move on to understand it better.
- Update the single abstraction exposing the idempotency context tables. It could be, for example, a view built as a union of the weekly tables. However, the DROP-based approach may result in an error if a user tries to access the view while you are

dropping one of the tables. If this is an issue, you can mitigate it with an optional step that removes the table from the view before dropping it from the database.

Overall, a pipeline using TRUNCATE or DROP for our weekly visits idempotency tables could look like the one in Figure 4-2.

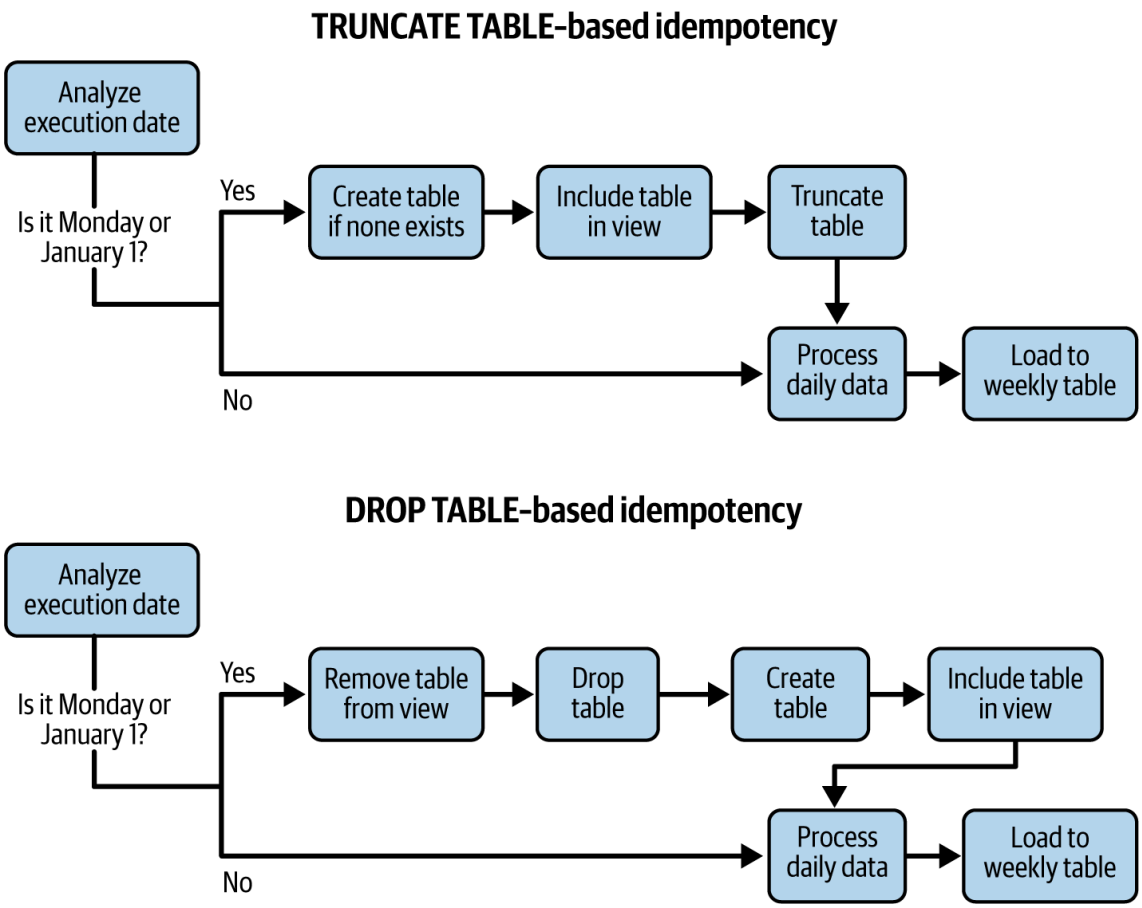

Figure 4-2. Fast Metadata Cleaner pattern example on top of weekly tables for TRUNCATE TABLE and DROP TABLE commands

Besides incremental and partitioned datasets, the Fast Metadata Cleaner pattern applies to the full datasets. In that case, you can simplify the workflow and run the table's re-creation step at each load or use the alternative Data Overwrite pattern presented in the next section. Figure 4-3 shows an example of the Fast Metadata Cleaner pattern adapted to a fully loaded table.

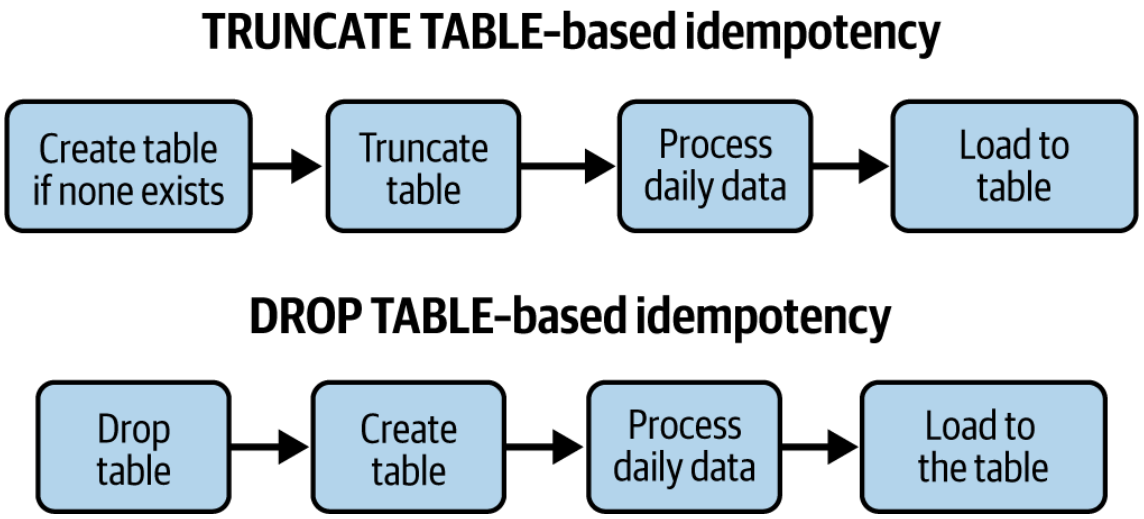

Figure 4-3. The Fast Metadata Cleaner and a fully loaded table

Consequences

These *fast* and *data removal* keywords sound fantastic, but despite its bright side, the pattern has some gotchas you should be aware of.[1]

Granularity and backfilling boundary

The pattern defines an idempotency granularity that is also a *backfilling granularity*. In other words, if you replay the pipeline, you have to do it from the task that creates a partitioned table. Otherwise, you'll end up with an inconsistent dataset.

For example, if you partition the data on a weekly basis and you need to backfill for only one day, you have no choice but to rerun the whole week. This doesn't mean you'll have to reprocess full pipelines for other days, though. If only one day generated an invalid dataset, it's enough to replay only the data loading step for the remaining days.

Another limitation related to granularity is the issue of fine-grained backfills, for example, for one data provider, user, or customer. The Fast Metadata Cleaner pattern will not help here because the metadata operations always work on whole tables.

Metadata limits

Also be aware of the limits of your data store. The pattern relies on creating dedicated partitions or tables, but unfortunately, it often won't be possible to create them indefinitely. For example, modern data warehouses like GCP BigQuery and AWS Redshift have, respectively, limits of 4,000 partitions and 200,000 tables. Both numbers are huge, but if you need to apply this pattern in multiple pipelines operating on different tables, you can reach these high quotas very quickly.

To overcome these limitation issues, you can add a *freezing* step to transform the mutable idempotent tables into immutable ones, thus reducing the partition scope. For example, weekly tables could turn into monthly or yearly tables if there are no possible changes after a freezing period.

Also, the Fast Metadata Cleaner pattern works only on the databases supporting metadata operations. Among them, you will find data warehouses, lakehouses, and relational

databases. On the other hand, they may be difficult to implement on top of object stores, where you will rely on the Data Overwrite pattern.

Data exposition layer

The final point is about access. The dataset is not living in a single place anymore, and your end users may not want to know the internal details of the design and may instead prefer to access the data from a single point of entry. To overcome that issue, you can use a solution similar to a database view, such as a logical structure grouping multiple tables and exposing them as a single unit.

Schema evolution

Another challenge is schema evolution. If your idempotency tables get a new optional field, you'll need a separate pipeline to update the schema of already existing tables. Doing that in the Fast Metadata Cleaner pattern would automatically involve reprocessing the data, which is less effective.

However, there is another scenario in which you evolve the schema and add a new required field. In that case, you can include the new field in the Fast Metadata Cleaner pattern because replaying past runs will automatically trigger processing and thus add the new field.

Examples

The pattern heavily relies on a data orchestration layer. It's not surprising that you're going to see an example with Apache Airflow, this time coordinating a pipeline writing data to a PostgreSQL table. The key part of the pipeline from Example 4-1 is the BranchPythonOperator. This task verifies the execution date and, depending on the outcome, goes to the data processing or follows the weekly table management.

Example 4-1. Idempotency router with BranchPythonOperator

```
def retrieve_path_for_table_creation(**context):
  ex_date = context['execution_date']
  should_create_table = ex_date.day_of_week == 1 or ex_date.day_of_year == 1
  return 'create_weekly_table' if should_create_table else "dummy_task"



check_if_monday_or_first_january_at_midnight = BranchPythonOperator(
  task_id='check_if_monday_or_first_january_at_midnight',
  provide_context=True,
  python_callable=retrieve_path_for_table_creation
)
```

Example 4-2 shows the weekly table management part. The workflow starts by creating a weekly table suffixed with the week number retrieved from Apache Airflow's execution

context. The next task uses PostgresViewManagerOperator, which is a custom operator that refreshes the visits view with the new weekly table.

Example 4-2. Table management branch

```
create_weekly_table = PostgresOperator(# ...
  sql='/sql/create_weekly_table.sql'
)
recreate_view = PostgresViewManagerOperator(# ...
  view_name='visits',
  sql='/sql/recreate_view.sql'
)
```

The next tasks in the pipeline are common for both branches and consist of loading the input dataset to the weekly table. We're omitting them here for brevity, but you can find the full example in the GitHub repo.

Pattern: Data Overwrite

If using a metadata operation is not an option (for example, because you work on an object store that doesn't have the TRUNCATE and DROP commands), you have no other choice but to apply a data operation. Thankfully, there is also a dedicated pattern for this category.

Problem

One of your batch jobs runs daily. It works on the visits dataset stored in event time–partitioned locations in an object store. The pipeline is still missing a proper idempotency strategy because each backfilling action generates duplicated records. You've heard about the Fast Metadata Cleaner pattern, but you can't use it because of the lack of a proper metadata layer. That's why you're looking for an alternative solution.

Solution

When the metadata layer is unavailable or using it involves a lot of effort, you can rely on the data layer and the Data Overwrite pattern.

The implementation depends on the technology, but typically, it relies on a native dataset replacement command. Your technical stack will drive the available solutions here, and the following will also apply:

- If you use a data processing framework, you may simply need to set an option while configuring your data writer. For this, Apache Spark uses a save mode and Apache Flink uses the write mode properties. Once you've configured your data writer, the data processing framework will do the rest (i.e., cleaning the existing files before writing). This configuration-driven solution can be extended to a selective overwriting if the output data store supports the conditions. That's the case with Delta Lake, where you can overwrite only a part of the dataset that matches the filtering condition specified in a replaceWhere option.
- If you work directly with SQL, you have multiple choices:

- First, you can use a combination of DELETE FROM and INSERT INTO operations. It's a simple approach known by many engineers working with databases.
- A more concise alternative to DELETE and INSERT leverages the INSERT OVERWRITE command. This alternative overwrites the whole table with the records from the INSERT part of the statement. However, besides the conciseness, there is also a semantical difference. INSERT OVERWRITE doesn't support selecting rows to overwrite, whereas the combination of DELETE and INSERT operations does.
- Finally, for the SQL part, you can also use the data loading commands available in your data store. Some of them, such as LOAD DATA OVERWRITE in BigQuery, support data overwriting natively. The others should be preceded with a TRUNCATE TABLE command.

Not Only List of Columns

Although it's a commonly shared belief, the INSERT command doesn't need explicitly defined values. You can also insert records from another table by issuing a SELECT statement that will match the list of columns to insert. For example, INSERT INTO visits (id, v_time) SELECT visit_id, visit_time FROM visits_raw would add all visits present in the visits_raw table, without having to declare them explicitly.

Running the overwriting command doesn't guarantee your data will disappear, though. If you use a data store–supporting time travel feature, thus making it possible to restore the dataset to one of its past versions, the data blocks will still be there after you execute the overwrite. They will only be deleted after the configured retention period or after running the vacuum operation to reclaim unused space if the command is supported. Among the examples of data stores not deleting data on the way, you'll find table file formats, GCP BigQuery, and Snowflake.

Consequences

Even though the pattern operates on the data layer directly and has wider support than the Fast Metadata Cleaner, it has some drawbacks.

Data overhead

Since there is a data operation involved, the pattern can perform poorly if the overwritten dataset is big and not partitioned. In that case, the overwrite will be slower over the course of days, as there will be more and more data to process.

You can try to mitigate this overhead by applying some storage optimizations, like partitioning. They should reduce the volume of data to overwrite and hence make the replacement action faster. Storage strategies are also a design patterns family you'll discover later in this book.

Vacuum need

A DELETE operation might not remove the data immediately from the disk. This happens with table file formats and relational databases, where deleted data blocks, albeit not

accessible by users with SELECT queries, still exist on disk. To reclaim the space occupied by these dead rows, you will need to run a vacuum process that will remove them for real.

Examples

Many modern data engineering solutions, including Databricks and Snowflake, provide a native implementation of the pattern with the INSERT OVERWRITE operation. The command truncates the content of the table or partition(s) before inserting new data. Example 4-3 replaces all rows in the devices table with the rows from the devices_staging table. Typically, this implementation is very flexible as you can extend this simple SELECT statement to more complex expressions involving joins or aggregations.

Example 4-3. INSERT OVERWRITE **example**

```
INSERT OVERWRITE INTO devices SELECT * FROM devices_staging WHERE state = 'valid';
```

Besides this pure SQL capability, the implementations of the pattern might rely on a separate data loading component. That's the case with BigQuery, which supports a writeDisposition in the jobs feature. The load job from Example 4-4 ingests devices data from the CSV file into the devices table. It sets the --replace=true flag to remove all existing data before writing the new data to save. This attribute is a shortcut for the WRITE_TRUNCATE distribution introduced previously.

Example 4-4. Loading data with a prior table truncation in BigQuery

```
bq load dedp.devices gs://devices/in_20240101.csv ./info_schema.json --replace=true
```

But if you're not using any of these tools, no worries because Apache Spark also implements the pattern. It's as configuration based as BigQuery because it lets you set a *save mode* option. The overwrite mode from Example 4-5 behaves exactly like BigQuery's replace flag (i.e., it drops all existing data before writing). Although it looks simple, you must be aware of one thing: the save mode by itself is not transactional (i.e., everything depends on the target data format). Thankfully, the modern table file format addresses that issue because the delete is a new commit in the log and the data files remain untouched.

Example 4-5. Overwriting data in PySpark

```
input_data.write.mode('overwrite').text(job_arguments.output_dir)
```

Updates

Removing a complete dataset to guarantee idempotency is an easy approach. Unfortunately, some types of datasets are not good candidates for full replacement. This is the case with updated incremental datasets, in which each new version generated by your data provider contains only a subset of modified or updated data. If you try to rewrite the whole dataset, you'll have to do some preparation work to keep only the most recent version of each entity. Thankfully, an easier approach exists, and you'll discover it in this section.

Pattern: Merger

If your dataset identity is static (i.e., there is no risk of modifying the identity of the rows), and your dataset only supports updates or inserts, then the best approach is to merge

changes with the existing dataset. But that's only a theory because in real life, there are some extra considerations you should take into account.

Problem

You're writing a pipeline to manage a stream of changes synchronized from your Apache Kafka topic via the Change Data Capture pattern. Your new batch pipeline must replicate all changes to the existing dataset stored as a Delta Lake table. The table must fully reflect the data present at a given moment in the data source, so it cannot contain duplicates.

Solution

If you don't have the complete dataset available—for example, if you're working with the incremental changes streamed from a database in our problem statement—you need to consider combining changes with an existing dataset. In a nutshell, that's what the Merger pattern does.

Simpler Overwrite

Idempotent processing for fully available datasets is easier with one of the overwriting patterns (see "Overwriting") because they simply delete and replace a dataset. The Merger pattern, on the other hand, requires you to interact with the data to combine new and existing rows. To keep things simple, the Merger pattern in this section is presented in the context of incremental datasets that can't be easily managed with a delete-and-replace approach.

The most important part of the implementation is the first step, when you define the attributes you're going to use to combine the new dataset with the old one. You can use a single property—such as the user ID—if it guarantees uniqueness across the dataset. If that's not the case, you can use multiple attributes, such as the visit ID and visit time for a website visit event.

Next, you need to find a way to combine datasets in your processing layer. Nowadays, most widely used solutions—including data processing frameworks, table file formats, and data warehouses—support the MERGE (aka UPSERT) command, which is the best way to implement the Merger pattern.

Once you find the right execution method, you need to define the behavior for each of the possible scenarios, which are as follows:

Insert

In this mode, the entry from the new dataset doesn't exist in your current dataset. Therefore, it's a new record you have to add.

Update

Here, both datasets store a given record, but it's very likely that the new dataset will provide an updated version of the record.

Delete

This is the trickiest case because the Merger pattern doesn't support deletes. As you saw before, if a record is missing from the dataset you want to merge, nothing will happen. For that reason, deletes are only possible if they're expressed as soft deletes (i.e., updates with

an attribute marking a given record as removed). That way, you can detect the change and apply a hard or soft delete to your data. If you need a refresher, review our discussion of soft deletes in the Incremental Loader pattern.

Overall, the MERGE statement covering all three scenarios could look like the statement in Example 4-6.

Example 4-6. Implementation of soft deletes for the Merger pattern

```
MERGE INTO dedp.devices_output AS target

USING dedp.devices_input AS input

ON target.type = input.type AND target.version = input.version

WHEN MATCHED AND input.is_deleted = true THEN

DELETE

WHEN MATCHED AND input.is_deleted = false THEN

UPDATE SET full_name = input.full_name

WHEN NOT MATCHED AND input.is_deleted = false THEN

INSERT (full_name, version, type) VALUES (input.full_name, input.version, input.type)
```

You might be surprised to see the is_deleted flag used for the INSERT statement. However, it's important to use it here because otherwise, you could insert removed records during the first execution of the Merger pattern and consequently never get rid of them. Figure 4-4 shows what happens if you remove this flag for the first run of a job relying on the Merger pattern.

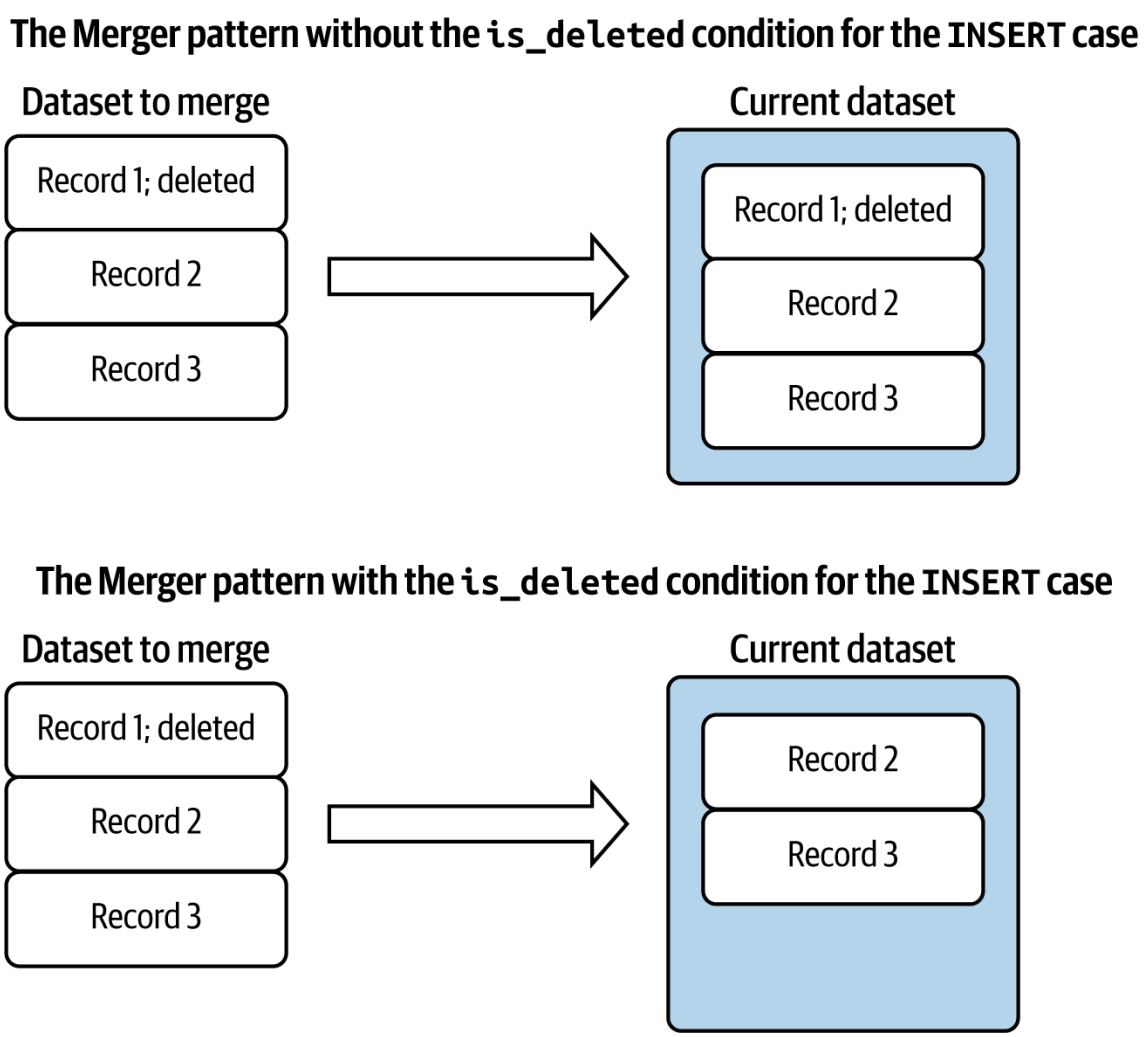

Figure 4-4. What happens during the first run of the Merger pattern with different is_deleted conditions for the INSERT case

Consequences

Despite its apparent simplicity, the pattern hides some gotchas and trade-offs, especially due to the character of the dataset (incremental or full).

Uniqueness

This is the first and most important requirement. Your data provider, or your data generation job, must define some immutable attributes you can use to safely identify each record. Otherwise, the merge logic will simply not work because instead of updating a row in case of backfilling, it might insert a new one, leading to inconsistent duplicates.

I/O

Unlike the Fast Metadata Cleaner, Merger is a data-based pattern. It works directly at the data blocks level, which makes it more compute intensive. However, modern databases and table file formats optimize this reading part by searching for the impacted records in the metadata layer first. The optimization helps to skip processing irrelevant files.

Incremental datasets with backfilling

You need to be aware of a shortcoming of the Merger pattern in the context of backfilling. Let's take a look at an example to help us better understand this issue. Table 4-1 shows how a job implementing the Merger pattern changed a dataset over time. As you can see, the job correctly integrated the updated and softly deleted rows, and at this point in time, the dataset is consistent.

Ingestion time	New rows	Output table rows
07:00	A	A
08:00	A–U, B	A–U, B
09:00	B–D, C	A–U, C
10:00	M, N, O	A–U, C, M, N, O

Table 4-1. Incremental dataset loading with the Merger pattern (U stands for update, and D stands for delete)

Now, let's imagine that you need to replay the pipeline from 08:00. Since the dataset is incremental, the backfill will start from the most recent version, which is the one containing the following rows: A–U, C, M, N, and O. As you can see, some of them are missing in the parts of the table written after 08:00. Consequently, during backfilling, your consumers won't see the same data as during the normal run. Table 4-2 shows the output rows available to consumers while the backfilled table slowly returns to normal after backfilling the last period.

Ingestion time	New rows	Current rows	Output table rows
08:00	A–U, B	A–U, C, M, N, O	A–U, B, C, M, N, O
09:00	B–D, C	A–U, B, C, M, N, O	A–U, C, M, N, O
10:00	M, N, O	A–U, C, M, N, O	A–U, C, M, N, O

Table 4-2. Incremental dataset after backfilling with the Merger pattern

To mitigate this issue, you may need to implement a restore mechanism outside the pipeline that will roll back the table to the first replayed execution. It's relatively easy to do if the database natively supports this versioning capability, for example, via a time travel feature that's available in table file formats. But since it transforms the stateless Merger pattern into a stateful one, you'll learn more about this solution in the Stateful Merger pattern.

Backfilling for Data Provider's Mistakes

If your data provider has introduced some errors into the dataset, you don't need to replay your pipeline. Instead, simply ask for a dataset with the fixed errors so that you can process it as a new dataset increment. The MERGE operation will apply the correct values for invalid rows. This solution works as long as the identity of the rows doesn't change (i.e., you can still match the rows from the invalid version with the rows from the new valid dataset).

Examples

Let's see the pattern in action with Apache Airflow and a SQL query loading new devices. For simplicity's sake, the pipeline consists of only one task executing a SQL query. The query starts with the operations present in Example 4-7.

Example 4-7. The first part of the Merger pattern query

```
CREATE TEMPORARY TABLE changed_devices (LIKE dedp.devices);

COPY changed_devices FROM '/data_to_load/dataset.csv' CSV  DELIMITER ';' HEADER;

# ...
```

The part from Example 4-7 is responsible for loading the new file into a temporary table that will be automatically destroyed at the end of the transaction. An important thing here is the table creation statement based on the LIKE operator. It avoids declaring all attributes of the target table here, which might potentially lead to metadata desynchronization if you managed the two schemas in different places.

Next, the query declares the MERGE operation relevant to the pattern itself (see Example 4-8).

Example 4-8. The second part of the Merger pattern query

```
# ...

MERGE INTO dedp.devices AS d USING changed_devices AS c_d

 ON c_d.type = d.type AND c_d.version = d.version

 WHEN MATCHED THEN

   UPDATE SET full_name = c_d.full_name

 WHEN NOT MATCHED THEN

   INSERT (type, full_name, version) VALUES (c_d.type, c_d.full_name, c_d.version)
```

Example 4-8 demonstrates the expected changes for our dataset. First, the query manages new rows with the WHEN NOT MATCHED THEN section, followed by an INSERT statement. Second, if the file has some updates, then the WHEN MATCHED THEN branch is

responsible for applying the changes. The query doesn't handle the deletes because the input file is incremental (i.e., it only brings new records or changed attributes for existing ones). Deletes are not expected for this query.

If you are interested in the code using soft deletes, you can check out the the GitHub repo.

Pattern: Stateful Merger

As you learned in the previous section, the Merger pattern lacks some consistency for datasets during the backfillings. If consistency is important to you, you should try the alternative pattern presented in this section.

Problem

You managed to synchronize changes between two Delta Lake tables with the help of the Merger pattern. Unfortunately, one week later, you detected an issue in the merged dataset and your business users asked you to backfill the dataset. As they care about consistency, they want you to restore the dataset to the last valid version before triggering any backfilling. The Merger pattern is not adapted for that, which is why you are looking for a way to extend it and support your response to these kinds of demands in the future.

Solution

Whenever you need to restore a dataset, the Merger pattern won't be enough because it focuses only on the merge action. But there is an alternative called a Stateful Merger pattern that provides data restoration capability via an extra state table.

This extra state table involves some changes in the pipeline. The workflow now has an additional step in the beginning to restore the merged table if needed and another at the end to update the state table. Figure 4-5 illustrates what the Stateful Merger pattern looks like with these additional tasks.

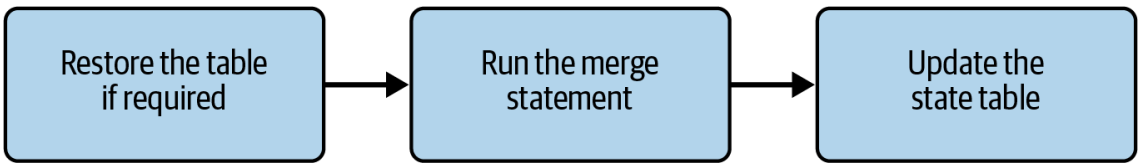

Figure 4-5. The workflow of the Stateful Merger pattern

To better understand the logic, let's start with the last task. Once the merge operation from the middle completes, it creates a new version of the merged table. The completion also triggers another task that retrieves the created table version and associates it with the pipeline's execution time. For example, if the execution at 09:00 creates version 5 and the execution at 10:00 writes version 6, the state table will look like Table 4-3.

Execution time	Table version
08:00	4

Execution time	Table version
09:00	5
10:00	6

Table 4-3. State table after running the pipeline at 09:00 and 10:00

Knowing what the table looks like, we can now better grasp the role of the restoration step from the beginning of the workflow. The first thing to keep in mind is that the restore process will happen only when the pipeline runs in the backfilling mode. Otherwise, it will do nothing.

How do you implement this backfilling detection logic? If your data orchestrator provides a context for the execution, and from this context, you can learn about the execution mode (backfilling or normal run), you can simply analyze this context metadata. If that's not the case, you need to implement some logic leveraging the state table. The high-level logic consists of the following:

1. Getting the version of the table created by the previous pipeline's run. If this version is missing, it means you'll run the pipeline for the first time or backfill the first pipeline's execution. In that case, you can clean the table with the help of the TRUNCATE TABLE command and move directly to the merge operation.
2. Comparing the current dataset version with the dataset version created by the previous pipeline's execution. Here, two things can happen:
    1. If the two versions are the same, there is nothing to restore as the pipeline is running in the normal mode. If we stay with our example from Table 4-3, the new run for 11:00 would detect the same version for the most recent execution and the previous execution at 10:00. In both cases, the version will be 6, which means the pipeline is performing the normal run scenario.
    2. If the two versions are different, it means the pipeline has entered into the backfilling scenario. If we stay with our example from Table 4-3, the run at 09:00 would detect a difference between the previous and the most recent version (4 versus 6), meaning that the pipeline should restore the table before applying the merge.

Does this work? Let's assume our state table looks like Table 4-4.

Execution time	Version
2024-10-05	1

Execution time	Version
2024-10-06	2
2024-10-07	3
2024-10-08	4

Table 4-4. A state table after four executions of a daily job

Let's see what happens in each scenario:

- The next pipeline runs. The execution time is 2024-10-09, and the version created by the previous run (2024-10-08) is the same as the most recent version of the table. The pipeline doesn't need to restore the table and can move directly to the merge operation.
- The pipeline runs 2024-10-05 for the second time. There is no version created for 2024-10-04, so before proceeding to the merge operation, the restore task needs to truncate the table.
- The pipeline runs 2024-10-07 for the second time. The version created by the run from 2024-10-06 is different from the most recent version of the table, so the restore task needs to roll back the table to version 2. Once the pipeline for 2024-10-07 completes, it will update its version and the state table will look like Table 4-5.

Execution time	Version
2024-10-05	1
2024-10-06	2
2024-10-07	5
2024-10-08	4

Table 4-5. State table after backfilling 2024-10-07

- After backfilling 2024-10-07, your data orchestrator will also backfill 2024-10-08. However, in that context, the most recent version is equal to the version created by the previous (already backfilled) run. Consequently, the workflow falls back into the normal run scenario.

Consequences

Even though the Stateful Merger pattern addresses the backfilling issue of the Merger pattern, it also brings its own gotchas.

Versioned data stores

The presented implementation of the Stateful Merger pattern requires your data store to be versioned (i.e., each write should create a new version of the table). That's the only way you can track the state and restore the table to a prior version.

If you don't work on a database with versioning capabilities, such as table file formats, you should slightly adapt the implementation to your use case. In that scenario, the pipeline will be composed of the steps in Figure 4-6.

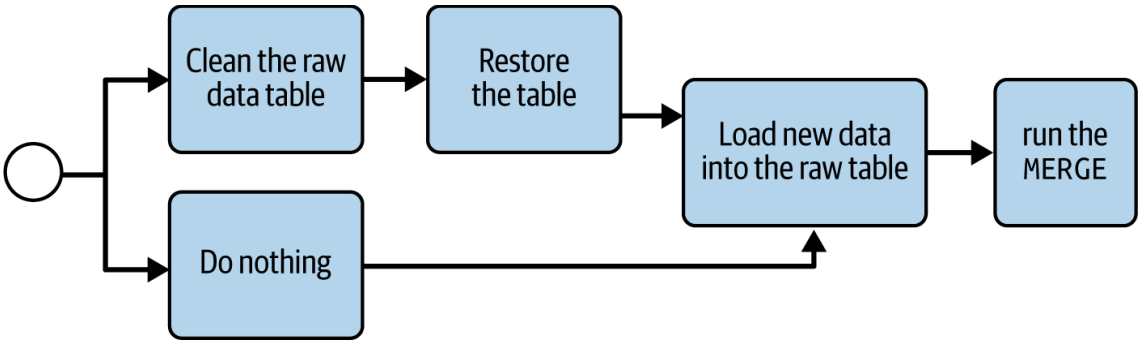

Figure 4-6. The Stateful Merger pattern adapted to a data store without versioning capabilities

Instead of versioning the table, the pipeline loads all raw data into a dedicated raw data table with a column storing the execution time. The backfilling detection logic verifies whether the raw data table has some records for the execution times in the future (see Example 4-9).

Example 4-9. Query verifying the pipeline's mode

```
SELECT CASE WHEN COUNT(*) > 0 THEN true ELSE false END

FROM dedp.devices_history WHERE execution_time > '{{ ts }}'
```

When the query returns true, it means the pipeline should first remove all rows matching the WHERE execution_time >= '{{ ts }}' condition. That way, the raw data stores all rows corresponding to the last valid version. Next, the workflow rebuilds the table by querying the devices_history table with the help of the Windowed Deduplicator pattern. In the end, the input data for the current execution time is loaded to the devices_history table and merged with the restored main table.

You can find a full example of PostgreSQL relying on this alternative approach in the GitHub repo.

Vacuum operations

Even though versioned datasets, such as tables in Delta Lake or Apache Iceberg, enable implementing the state table, they also hide a trap. After the configured retention duration, they remove files that are not used anymore by the dataset. Consequently, some of the prior versions will become unavailable at that moment.

You can mitigate the issue a bit by increasing the retention period, but that would increase your storage costs as well. Or you can accept the fact that the pipeline cannot be backfilled beyond the retention period.

Metadata operations

Besides the vacuum, there are other operations that can run against your table. One of them is compaction, which you discovered while you were learning about the Compactor pattern.

Compaction doesn't overwrite the data but only combines smaller files into bigger ones. But despite this no-data action, it also creates a new version of the table. As a result, if you always use the previous version from the state table in the restore action, you will miss the operations made between two merge runs.

To overcome this issue, assuming the versions increase by 1, you could change the logic and, instead of reading the previous version, read the version corresponding to the current execution time and subtract 1 (see Example 4-10).

Example 4-10. Changed logic for retrieving the version to restore

```
version_to_restore = version_for_current_execution_time - 1
```

To better understand this change, let's suppose we have the state table and dataset history in Figure 4-7.

State table

Execution time	Table version
2024-10-05	5
2024-10-06	7
2024-10-07	9
2024-10-08	10

Table history

Table version	Operation
5	MERGE
6	COMPACTION
7	MERGE
8	COMPACTION
9	MERGE
10	MERGE

Figure 4-7. State table for a table with no-data operations, such as compaction

If you want to backfill the pipeline executed on 2024-10-07, the version for this execution time is 9, so the version to restore will be 8. As you can see in the table history, this version corresponds to the compacted table between the runs of 2024-10-07 and 2024-10-08. The same logic will also work for the pipeline executed on 2024-10-08, where the restored version will be 9, meaning the one created by the previously executed pipeline on 2024-10-07.

Examples

Let's see how to implement the Stateful Merger pattern on top of Delta Lake and orchestrated from Apache Airflow. The first snippet shows the declaration of the state table. The table from Example 4-11 has two fields, one for the job's execution time and another for the corresponding Delta table version created.

Example 4-11. State table definition

```
CREATE TABLE IF NOT EXISTS `default`.`versions`
(execution_time STRING NOT NULL, delta_table_version INT NOT NULL)
```

Next comes the data restoration task. It implements the logic presented in the Solution section where, depending on the current and previous version, the table was either backfilled or not. The restoration task starting with Example 4-12 first retrieves the last table version created by the MERGE operation alongside the table version written at the previous execution time.

Example 4-12. Reading of current and past versions

```
last_merge_version = (spark.sql('DESCRIBE HISTORY default.devices')
    .filter('operation = "MERGE"')
    .selectExpr('MAX(version) AS last_version').collect()[0].last_version)


maybe_previous_job_version = spark.sql(f'''SELECT delta_table_version FROM versions
 WHERE execution_time = "{previous_execution_time}"''').collect()
```

After the restoration task comes the part that will evaluate the retrieved versions and, depending on the outcome, trigger table truncation or table restoration. Example 4-13 shows both steps in the exact same order.

Example 4-13. Data restoration action

```
if not maybe_previous_job_version:
 spark.sql('TRUNCATE TABLE default.devices')
else:
 previous_job_version = maybe_previous_job_version[0].delta_table_version
 if previous_job_version < last_merge_version:
  current_run_version = (spark_session.sql(f'''SELECT delta_table_version FROM
   versions WHERE execution_time = "{currently_processed_version}"''')
   .collect()[0].delta_table_version)
  version_to_restore = current_run_version - 1
  (DeltaTable.forName(spark, 'devices').restoreToVersion(previous_job_version ))
```

After eventually restoring the table, the pipeline executes the MERGE operation. The outcome of this operation creates a new commit version in the table, but this version is retrieved only in the next task and written to the state table. It's worth noting that there is a MERGE too because in the case of backfilling, the writer updates the previous value, and in the case of the normal run, it inserts it. Example 4-14 summarizes this logic.

Example 4-14. State table update after successful MERGE

```
last_version = (spark.sql('DESCRIBE HISTORY default.devices')
  .selectExpr('MAX(version) AS last_version').collect()[0].last_version)
new_version_df = (spark.createDataFrame([
 Row(execution_time=current_execution_time, delta_table_version=last_version)]))


(DeltaTable.forName(spark_session, 'versions').alias('old_versions')
 .merge(new_version.alias('new_version'),
```

```
 'old_versions.execution_time = new_version.execution_time')

 .whenMatchedUpdateAll().whenNotMatchedInsertAll().execute())
```

You can find the full snippet alongside the data orchestration part in the GitHub repo.

Database

The previous patterns discussed in this chapter require some extra work on your part. You need to either adapt the orchestration layer or use a well-thought-out writing operation. If this sounds like a lot of work, sometimes you can take shortcuts and rely on the databases to guarantee idempotency.

Pattern: Keyed Idempotency

The first pattern in this section uses key-based data stores and an idempotent key generation strategy. This mix results in writing data exactly once, no matter how many times you try to save a record.

Problem

Your streaming pipeline processes visit events to generate user sessions. The logic buffers all messages for a dedicated time window per user and writes an updated session to a key-value data store. As for other pipelines, you want to make sure this one is idempotent to avoid duplicates in case a task retries.

Solution

In the context of a key-based database, idempotency applies to the key generation logic on the data processing side. In our problem, it'll result in generating the same session ID for all visits events of a given user, thus writing it only once. By the way, that's a simple explanation of what the Keyed Idempotency pattern does.

When it comes to the actual implementation, you should start by finding immutable properties for the key generation. Depending on how lucky you are, your input dataset may already have unique attributes for your use case. For example, if you need to get the most recent activity for a user, you'll simply use the user ID as the key.

However, the key may not always be available. To understand this, let's take a look at an example of user session activity from website visits, which we first introduced in "Case Study Used in This Book". The input dataset contains only the user ID and visit time, so you can't rely on the user ID to create a session key as each new session would always replace the previous sessions. Instead, you could use the combination of the user ID and the first visit time to generate an idempotent key. Although this is a valid solution, it hides a trap depicted in Figure 4-8. As you can see, our job stopped because of an unexpected runtime error, and after the restart, the session ID changed because of late data written to the input data store.

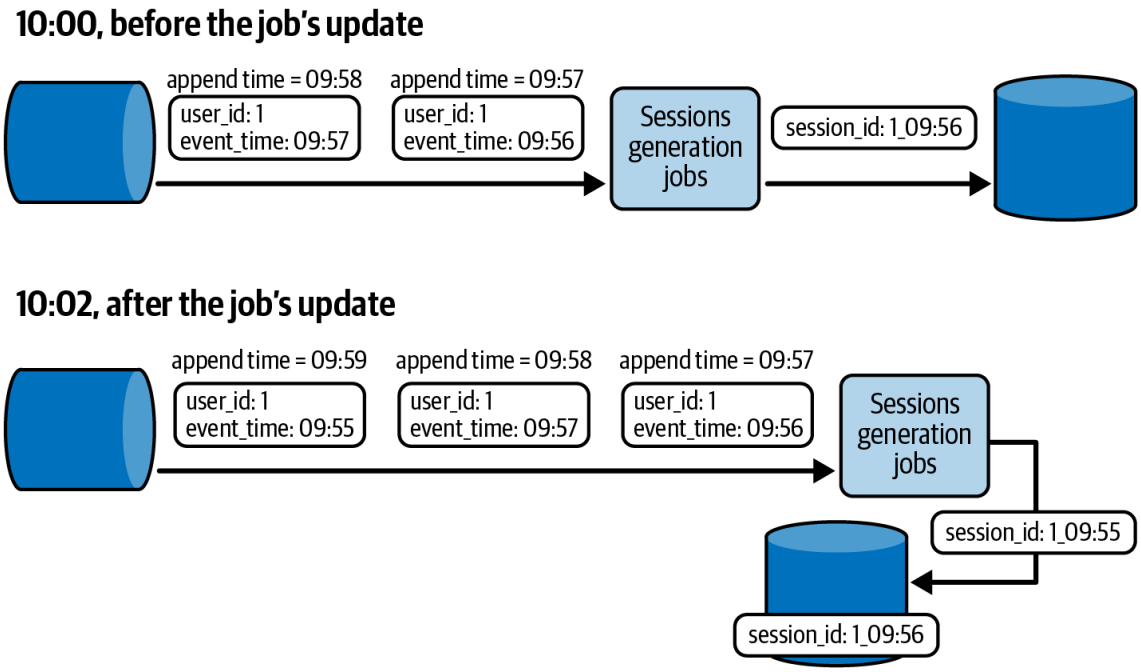

Figure 4-8. Late data impact on key generation; the late record for 09:55 creates a new session for the job restarted after an update

In the context of idempotent key generation for a user session, the event time attribute is mutable (i.e., the value may change between runs). For that reason, it's safer to use an immutable value, like an *append time*, which is the time a given entry was physically written to the streaming broker. Then, even if there are late events, the key won't be impacted by them and will remain the same. You'll find an example of this in Apache Kafka, where the property is indeed called append time. Other streaming brokers have the same attribute but call it by a different name. For example, Amazon Kinesis Data Streams uses the term *approximate arrival timestamp* for that property.

The same attribute, albeit named differently, exists in data-at-rest stores. It is often referred to as *added time*, *ingestion time*, or *insertion time* and is often implemented with a default value corresponding to the current time. If you apply this to our example, you'll see that for both batch and streaming cases, you're going to use the data from 10:09 to generate the session key. That way, you keep the key consistent across runs, even when late data arrives. Moreover, you'll be able to emit partial and consistent results as they'll share the same ID in the final state. Example 4-15 shows a WINDOW expression that retrieves the first recorded user activity to get attributes for the idempotent session key. The sorting is ascending, meaning that it will not be impacted by new data added to the table.

Example 4-15. Window operation using ingestion time

```
SELECT ... OVER (PARTITION BY user_id ORDER BY ingestion_time ASC, visit_time ASC)
```

The examples from this part have covered a key-based data store since it's the easiest one to explain. However, the key generation strategy also works for data containers, like files. For example, if you need to generate a new file from a daily batch job, you can name the file with the execution time. Consequently, a job running on 20/11/2024 would write a file named *20_11_2024*, and a job running the next day would write a file named *21_11_2024*, and so forth. Replaying a pipeline would always create one file. By the way, the example

applies to partitions or even tables, if you can afford to create a table every day. The one requirement here is to use immutable attributes, exactly like for key generation in the context of a key-value store.

Consequences

Despite its simplicity, this pattern has some gotchas mostly related to the databases.

Database dependent

Even though your job generates the same keys every time, it doesn't mean the pattern will apply everywhere. You might already deduce that it works well for databases with key-based support, such as NoSQL solutions (Apache Cassandra, ScyllaDB, HBase, etc.).

For other scenarios, the pattern is either not applicable or applicable with some extra effort or trade-offs. The first, more demanding implementation is a relational database. If you try to insert a row with the same primary key, you'll get an error instead of overwritting it as for a key-value store. That's why here, the writing operation becomes a MERGE instead of INSERT, which adds some extra complexity on the expression itself, exactly like in the Merger pattern presented previously.

Regarding the trade-offs, the best illustration here is Apache Kafka. It does support keys to uniquely identify each record, but as an append-only log, so it does it without deduplicating the events at insertion time. Instead, uses an asynchronous compaction mechanism that runs after writing the data. As a result, for some time, your consumers can see duplicated entries. They share the same keys, though, so it should be easier to distinguish them from new records.

Mutable data source

The compaction from the last example introduces the second gotcha. Besides duplicated entries, compaction can be configured to remove events that are too old. In that context, if you restart the job and the compaction deleted the first event used for the key creation, you'll take the next record from the log and logically break the idempotency guarantee. On the other hand, since the data has changed, using a different key does make sense as the record's shape will be different.

Examples

Now, let's see the Keyed Idempotency pattern in action with an Apache Spark Structured Streaming job transforming Apache Kafka visit events into sessions and writing them to a ScyllaDB table. The output table from Example 4-16 defines a unique key composed of the session_id and user_id fields. They are a guarantee for our idempotent session generation based on the following data source definition.

Example 4-16. ScyllaDB table for the sessions

```
CREATE TABLE sessions (
 session_id BIGINT,
 user_id BIGINT,
 pages LIST<TEXT>,
```

```
 ingestion_time TIMESTAMP,
PRIMARY KEY(session_id, user_id));
```

Next comes the logic for grouping the visits. Example 4-17 first extracts the value and timestamp attributes from each Kafka record. Next, the job builds the visit structure from the value's JSON and uses some of the attributes in the watermark definition. The timestamp column corresponds to the append time and naturally is a part of the key generation logic presented in the snippet.

Example 4-17. Visits grouping with append time (timestamp column)

```
(input_data.selectExpr('CAST(value AS STRING)', 'timestamp').select(F.from_json(
 F.col('value'), 'user_id LONG, page STRING, event_time TIMESTAMP')
 .alias('visit'), F.col('timestamp'))
.selectExpr('visit.*', 'UNIX_TIMESTAMP(timestamp) AS append_time')
.withWatermark('event_time', '10 seconds').groupBy(F.col('user_id')))
```

Finally, there is the idempotent key generation logic based on the append time. The first part, depicted in Example 4-18, handles the expired state and generates the final output with respect to the idempotent key.

Example 4-18. Idempotent ID generation logic

```
def map_visit_to_session(user_tuple: Any,
  input_rows: Iterable[pandas.DataFrame],
  current_state: GroupState) -> Iterable[pandas.DataFrame]:
 session_expiration_time_50_seconds_as_ms = 50 * 1000
 user_id = user_tuple[0]
 # ommitted for brevity
 if current_state.hasTimedOut:
  min_append_time, pages, = current_state.get
  session_to_return = {
   'user_id': [user_id],
   'session_id': [hash(str(min_append_time))],
   'pages': [pages]
 }
 else:
  # ...
  # accumulation logic explained below
```

The output is a session identified by the user, and session IDs get it directly from the accumulated state. The state accumulates in the second part of the code, presented in Example 4-19. The mapping function reads all records in each window and gets the earliest append time among them. It later sets this value to the first version of the session state. Whenever there are other visits for the same entity, the logic follows the if current_state.exists branch. However, as the append time in our Apache Kafka topic is guaranteed to be increasing, we can simply take the same append time as the one computed in the first iteration.

Example 4-19. Append time accumulation in the state

```
else:
 # ...
 data_min_append_time = 0
 for input_df_for_group in input_rows:
  # ...
  data_min_append_time = int(input_df_for_group['append_time'].min()) * 1000
  if current_state.exists:
   min_append_time, current_pages, = current_state.get
   visited_pages = current_pages + pages
   current_state.update((
    min_append_time, visited_pages
   ,))
  else:
   current_state.update((data_min_append_time, pages,))
```

The same solution can be implemented on top of other streaming brokers, and the only difference is the attribute name. If we take a look at the previously mentioned Amazon Kinesis Data Streams, it's enough to adapt the reading part as depicted in Example 4-20, where the approximate arrival timestamp column gets renamed to the append_time from the previous example. You could also avoid the renaming and use the approximate arrival timestamp in Example 4-18.

Example 4-20. Input part adapted to Amazon Kinesis Data Streams

```
(spark_session.readStream.format("kinesis") # ...
 .load().selectExpr("CAST(data AS STRING)",
     "approximateArrivalTimestamp AS append_time"))
```

Apache Kafka and a Timestamp Attribute

You can define the append time externally with an Apache Kafka producer. However, in the context of the pattern, it's a bit riskier and less reliable than using the mechanism fully

controlled by the broker. To see what strategy is set on your topic, you can verify the log.message.timestamp.type attribute.

Pattern: Transactional Writer

In addition to the key uniqueness you saw in the previous pattern, transactions are another powerful database capability that can help you implement idempotent data producers. Transactions provide all-or-nothing semantics, where changes are fully visible to consumers only when the writer confirms them. This confirmation step is more commonly known as *commit*, but with all that said, we're already covering the implementation a bit too much. Before delving into details, let's see where transactions can help.

Problem

One of your batch jobs leverages the unused compute capacity of your cloud provider to reduce the total cost of ownership (TCO). Thanks to this special runtime environment, you have managed to save 60% on the infrastructure costs. However, your downstream consumers start complaining about data quality.

Whenever the cloud provider takes a node off of your cluster, all the running tasks fail and retry on a different node. Because of this rescheduling, the tasks write the data again and your consumers see duplicates and incomplete records. You need to fix this issue and ensure that your job never exposes partial data.

Solution

The best way to protect your consumers from the incomplete data issue is to leverage the transactions with the Transactional Writer pattern. It relies on the native database transactional capacity so that any of the in-progress but not committed changes will not be visible to downstream readers.

From a broader perspective, the implementation consists of three main steps. In the first step, the producer initializes the transaction. Depending on your processing layer, this step can be explicit or implicit. In the explicit mode, you need to call a transaction initialization instruction, such as START TRANSACTION or BEGIN. In the implicit mode, your data processing layer handles the transaction opening on your behalf.

After the initialization, you write the data. While you're producing new records, the changes are added to the database but remain private to your transaction scope. Only in the end, when you have finished writing the data, do you need to change the new records' visibility to make them publicly available to consumers. That's where the *commit* step happens. If there is an issue, instead of publishing the data, you need to discard it by calling the action that is the opposite of the commit step, which is *rollback*. Figure 4-9 summarizes all of these actions.

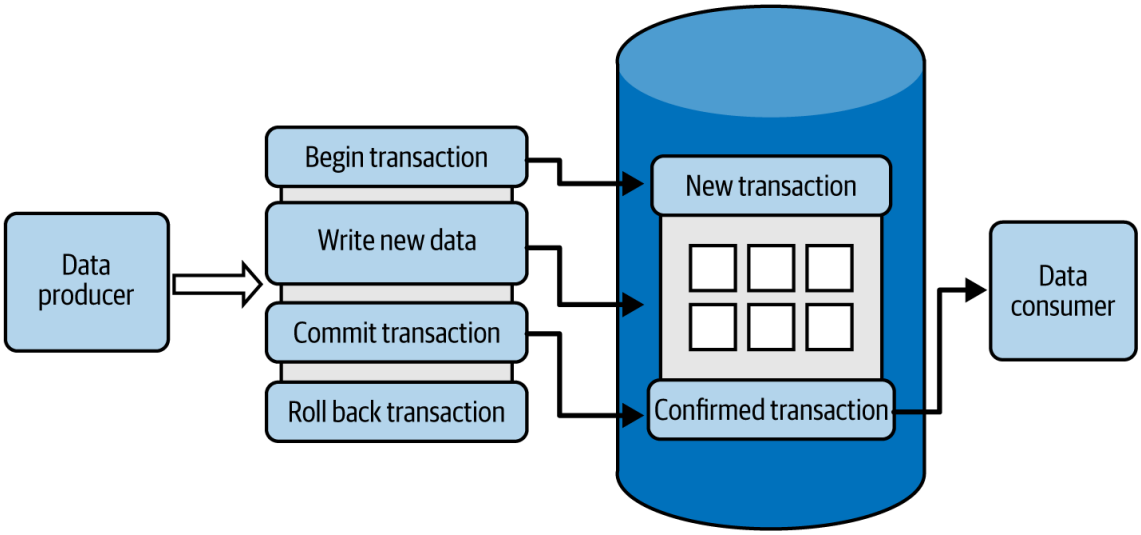

Figure 4-9. High-level view of a producer using the Transactional Writer pattern, in which the data is available to the consumer only after the commit step

However, life is not that rosy. From a low-level point of view, there are two implementations that you will use, depending on your processing model. The first one is for standalone jobs or ELT workloads processing datasets at the data storage layer directly, for example, in a data warehouse like BigQuery, Redshift, or Snowflake. Here, the transaction is usually declarative and fully managed by the data store, and so, despite the fact that it's under the hood, the processing can be distributed.

A different implementation applies to distributed data processing jobs, which are often implemented with the ETL paradigm. In this mode, multiple tasks work in parallel to write a dataset to the same output. Knowing that, you can deduce two possible implementations of the Transactional Writer pattern:

- The transaction is local (i.e., task based). Each task performs an isolated transaction. This works well as long as you don't encounter any job retries, but we'll let you discover this point in "Consequences".
- The whole job is transactional. In this mode, the job initializes the transaction before it starts running the tasks, and it commits the transaction once all the tasks complete their work. This provides a stronger guarantee than the local transaction but is also more challenging to achieve. For example, with Apache Spark and Delta Lake, the transaction is committed when the writer creates a new entry in the commit log directory. But if this step fails, data files will still be there and will need to be moved aside.

The idempotency comes from the all-or-nothing transactions semantics. In case of any error, the producer doesn't commit the transaction, which leads to either an automatic rollback or orphan records in the data storage layer that are not visible to the readers. However, if you backfill the data producer, the writing job will initialize a new transaction and thus insert the processed records again. Idempotency is then guaranteed only for the current run.

Read Uncommitted Isolation Level

Despite transactions on the writer's side, a reader might still see records from uncommitted transactions if it sets its transaction isolation level to read uncommitted. In the database world, this side effect is known as *dirty reads* because the records from uncommitted transactions might be rolled back.

Among the client libraries and data stores supporting transactions, you'll find modern table file formats (Delta Lake, Apache Iceberg, and Apache Hudi), streaming brokers (Apache Kafka), data warehouses (AWS Redshift and GCP BigQuery), and even relational database management systems (PostgreSQL, MySQL, Oracle, and SQL Server).

As you have seen, not all of the aforementioned technologies integrate perfectly with the distributed data processing tools. Table file formats are pretty well covered by major tools (Apache Flink and Apache Spark), whereas Apache Kafka transactional producers are only available for Apache Flink.

Consequences

A transactional producer is easier to implement than all of the patterns in this section as it's often just a matter of calling appropriate commands in the processing logic. Despite that, there are some pitfalls.

Commit step

Unlike a nontransactional write, a transactional one involves two extra steps, which are opening and committing the transaction, alongside resolving data conflicts at both stages.

The steps may have an impact on the overall data availability latency. For example, each file produced in a raw data format like JSON or CSV is immediately visible to consumers. On the other hand, the files generated in a transactional file format like Delta Lake become visible once the producer generates a corresponding commit logfile. Put differently, consumers will have to wait for the slowest task to complete before being able to access the transactional data.

But this coordination overhead is necessary to provide transactional capability and therefore to expose only complete datasets.

Distributed processing

Distributed data processing frameworks' support for transactions is not global. For example, the already mentioned Apache Kafka is not supported in Apache Spark, despite its popularity among data engineers. This greatly reduces the application of the pattern, unfortunately.

Idempotency scope

Remember, the idempotency is limited to the transaction itself! For example, if a distributed data processing framework uses local (i.e., task-based) transactions without any further coordination to store already committed tasks, any job restart will rewrite the data from committed transactions.

The same side effect applies to the backfilling scenarios where reprocessing the data will result in a new transaction eventually adding the same records.

Examples

First, let's take a look at an example of the Transactional Writer for a batch pipeline. The pipeline needs to load data from two datasets and apply each of them individually on the target table (see Example 4-21).

Example 4-21. Two visually separated operations within the same transaction

```
CREATE TEMPORARY TABLE changed_devices_file1 (LIKE dedp.devices);

COPY changed_devices_file1 FROM '/data_to_load/dataset_1.csv'

 CSV  DELIMITER ';' HEADER;

MERGE INTO dedp.devices AS d USING changed_devices_file1 AS c_d

-- ... ommitted for brevity


CREATE TEMPORARY TABLE changed_devices_file2 (LIKE dedp.devices);

COPY changed_devices_file2 FROM '/data_to_load/dataset_too_long_type.csv'

 CSV  DELIMITER ';' HEADER;

MERGE INTO dedp.devices AS d USING changed_devices_file1 AS c_d

-- ... ommitted for brevity


COMMIT;
```

As you can see in Example 4-21, one of the files stores rows with values that are too long for some columns. Because these two merge operations are visually separated, you may be thinking the first one will succeed while the second will fail. And you're right! That's the result, but we must add an important aspect here: none of them will commit. Put differently, the database won't accept the partial success because the SQL runner doesn't reach the commit stage. The same logic works for table file formats where the written files are not considered to be readable as long as there is no corresponding commit file.

And you may be surprised to hear that Apache Kafka, which is more often quoted in a stream processing context, also works that way for the transactions. The producer initializes the transaction by sending a special message to the partition, and once it reaches the commit step, it generates a new metadata event confirming the pending transaction. You can then implement the Transactional Writer pattern natively with the Kafka producers or from a distributed data processing framework like Apache Flink (see Example 4-22).

Example 4-22. Transactional data producer with Apache Flink

```
kafka_sink_valid_data = (KafkaSink.builder().set_bootstrap_servers("localhost:9094")

 .set_record_serializer(KafkaRecordSerializationSchema.builder()

   .set_topic('reduced_visits')

   .set_value_serialization_schema(SimpleStringSchema())

   .build())
```

```
.set_delivery_guarantee(DeliveryGuarantee.EXACTLY_ONCE)
.set_property('transaction.timeout.ms', str(1 * 60 * 1000))
.build())
```

Example 4-22 shows a configuration for a transactional Kafka writer. Two important attributes here are the delivery guarantee and the transaction timeout. The delivery guarantee is quite obvious as it involves using transactions for data delivery. The timeout parameter, although harder to understand at first glance, is also important. The exactly-once delivery relies on Apache Flink's checkpointing mechanism, which can take some time. If the timeout is too short and the checkpoint takes longer than the timeout parameter, Flink will be unable to commit the transaction due to its expiration.

Immutable Dataset

So far, you have seen patterns working on mutable datasets. This means that you can alter the datasets in any way, including total data removal. But what do you do if you cannot delete or update the existing data? A dedicated pattern exists for that category too.

Pattern: Proxy

This pattern is inspired by one of the best-known engineering sayings: "We can solve any problem by introducing an extra level of indirection." Hence its name, the Proxy.

Problem

One of your batch jobs generates a full dataset each time. Since you only need the most recent version of the data, you have been overwriting the previous dataset so far. However, your legal department has asked you to keep copies of all past versions, and consequently, your current mutable approach doesn't hold anymore. You need to rework the pipeline to keep each copy but expose only the most recent table from a single place.

Solution

The requirement expects the dataset to be immutable and thus written only once. To achieve this, you can implement the Proxy pattern. As the proxy in network engineering, it's an intermediate component between the end users and the real physical storage.

How does it work? First, you must guarantee the immutability by loading the new data into a different location each time. A good and easy solution is to use timestamped or versioned tables. They're like regular tables, except that their names are suffixed with a version or a timestamp to distinguish them. An important point is that all writing permissions should be removed from these tables after creating them. Consequently, they will be writable only once.

You can achieve the writable-once semantics more easily if your storage layer sits on top of an object store. You can additionally enhance the access controls with a locking mechanism. The locking approach is also known as *write once read many* (WORM) and is supported by all major object store services. For AWS S3, you'll use Object Lock; for Azure Blob , you'll use immutability policies; and for GCP, you'll rely on object holds or bucket locks.

After implementing the writable-once semantics, you need to create a single data access point, which is the proxy. Most of the time, it'll be a passthrough view that exposes the most recent table without any data transformations in the SELECT statement. This approach works for most data warehouses and relational databases, plus some NoSQL stores, such as OpenSearch (via aliases), Apache Cassandra, ScyllaDB, and even MongoDB.

If your data store doesn't support a specific view, you'll have to create a similar structure on your own. It can be a manifest file referencing the location or explicitly listing the files that should be processed by consumers. From a responsibility standpoint, it's better to isolate the manifest creation from the data processing job. That way, if for whatever reason the manifest creation fails, you won't have to reprocess the data, which most of the time will be a much slower operation.

Creating manual immutable tables and defining manifest files are only two ways to implement the Proxy pattern. The last alternative strategy applies to table file formats like Delta Lake and Apache Iceberg, plus some data warehouses like GCP BigQuery. Even though you can create one table per write for these data stores, a simpler implementation is possible. Remember, when you overwrite the table, the data is still there to guarantee time travel capability. Consequently, each write produces a new version of the table under the hood and keeps old data on disk available for querying or restoration if needed. However, sometimes, this solution may have limited and not configurable retention capabilities, like seven days for BigQuery at the moment of writing.

It's worth noting that it's not possible to remove write permissions for the natively versioned data stores presented in the previous paragraph, but thanks to the underlying storage system, permissions management is not required for this solution. As you can see, the Proxy pattern heavily relies on your database capabilities. Figure 4-10 summarizes the three possible implementations covered in this section.

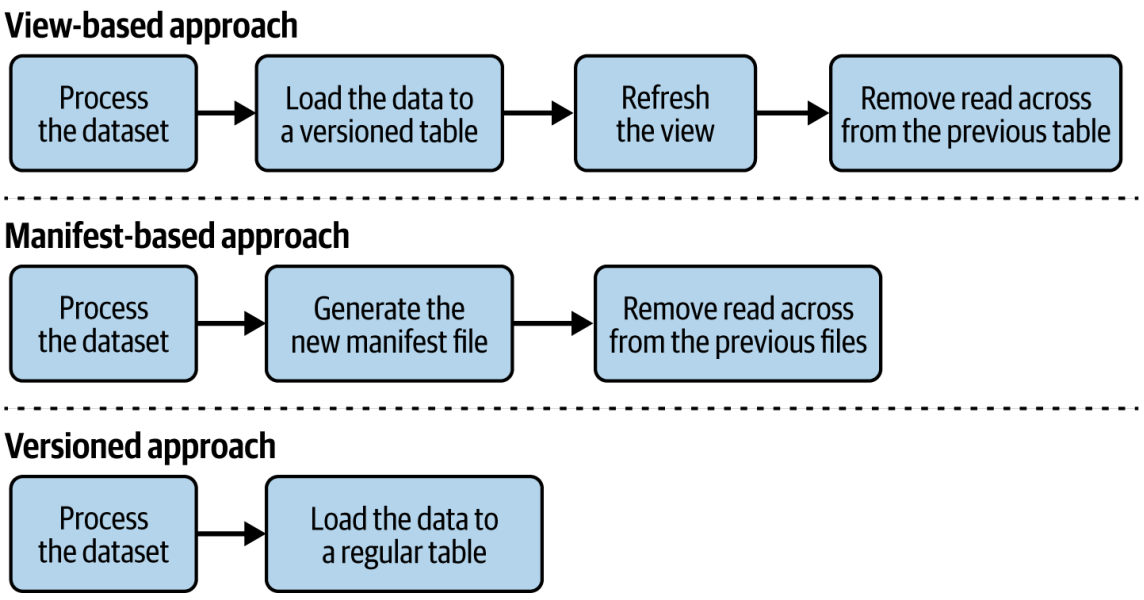

Figure 4-10. Implementation scenarios for the Proxy pattern

Consequences

At first glance, the pattern looks simple and familiar. However, it hides some important consequences.

Database support

Not all databases have this great view feature, which will be an immutable access point to expose underlying changing datasets. Although it can be replaced with a manifest file, that makes the reading process more cumbersome.

Immutability configuration

You can enforce immutability at the data orchestration level by configuring the output of the triggered writing task. But that won't be enough. You'll need some help, maybe from the infrastructure team, to enforce immutability on the data store too. You can do this by creating locks on object stores and removing writing permissions from the table, once it gets created.

Examples

Let's see how to implement the Proxy pattern with Apache Airflow and PostgreSQL. The pipeline is composed of the two steps that are defined in Example 4-23.

Example 4-23. The Proxy pattern's pipeline

```
load_data_to_internal_table = PostgresOperator(
  sql='/sql/load_devices_to_weekly_table.sql'
)
refresh_view = PostgresOperator(# ...
  sql='/sql/refresh_view.sql'
)


load_data_to_internal_table >> refresh_view
```

Example 4-23 starts by loading the data into a hidden internal table. Since this step is a simple COPY command you saw previously in this chapter, let's move directly to the refresh_view.sql query in Example 4-24.

Example 4-24. View refresh

```
{% set devices_internal_table = get_devices_table_name() %}

CREATE OR REPLACE VIEW dedp.devices AS
  SELECT * FROM {{ devices_internal_table }};
```

The view refresh is also based on a SQL operation that is simple but hides an important detail, which is the generation of the internal table name. Remember, if the pipeline's instance reruns, it can't rewrite the previous dataset. Instead, it must write the new dataset to a different table. That's where the get_devices_table_name function comes into play. Example 4-25 shows how the function leverages Apache Airflow's context to create unique table names.

Example 4-25. Generation of a unique table name

```
def get_devices_table_name() -> str:
 context = get_current_context()
 dag_run: DagRun = context['dag_run']
 table_suffix = dag_run.start_date.strftime('%Y%m%d_%H%M%S')
 return f'dedp.devices_internal_{table_suffix}'
```

The function in Example 4-25 uses the pipeline start time to compute a unique suffix for the internal table and guarantee that each load goes to its dedicated storage space.

Other aspects to keep in mind for the Proxy pattern are the permissions. The implementation should also ensure the user doing the manipulation can only create tables. Otherwise, the user could, even accidentally, delete previously created internal tables and, as a consequence, break the immutability guarantee provided by the Proxy pattern.

Summary

Always expecting the worst is probably not the best way to go through life, but it's definitely one of the best approaches you can take to your data engineering projects. As you know from the previous chapter, errors are inevitable and it's better to be prepared. The backbone of this preparation consists of the error management design patterns. However, they mitigate the impact of failure on the processing layer only.

To complete the cycle of handling error management, you need idempotency and typically the design patterns described in this chapter. To start, you saw data overwriting patterns that automatically replace the dataset, either by leveraging fast metadata operations like TRUNCATE or DROP or simply by physically replacing the dataset files.

The overwriting patterns are good if you have the whole dataset available in each operation. If that's not the case and your input is an incremental version, you can use the Merger pattern detailed in "Updates". Even though the combination operation looks costly at first glance, modern data storage solutions optimize it by leveraging the metadata (statistics) too!

These overwriting and update pattern categories mostly rely on the data orchestration layer. If you don't have one, maybe because your job is a streaming job, no worries as you can always rely on the database itself for idempotency. That's the next category, where you can use either idempotent row key generation or transactions to ensure unique record delivery, even under retries.

Finally, sometimes, your data must remain immutable (i.e., you must be able to write it only once). This scenario isn't supported by the patterns presented so far. Instead, you should opt for the Proxy pattern shown in "Immutable Dataset" and use an intermediary layer to expose the data.

These last two chapters on error management and idempotency talked mostly about a technical aspect of data engineering. Even though they help to improve business value, error management and idempotency don't generate meaningful datasets alone. Instead, you should leverage the data value design patterns that we cover in the next chapter!

[1] You may think that the name Fast Metadata Cleaner implies the availability of the metadata operations, and you'd be right. We don't need to go too far into detail here, but this is a potential consequence.