
Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

# Chapter 2. Data Ingestion Design Patterns

Data engineering systems are rarely data generators. More often, their first stage is data acquisition from various data producers. Working with these producers is not easy; they can be different pipelines inside your team, different teams within your company, or even completely different organizations. Because each producer has dedicated constraints inherited from technical and business environments, interacting with them may be challenging for you.

But you have no choice. You have to adapt. Otherwise, you won't get any data, and as a result, you won't feed your data analytics or data science workloads. Or even worse, you will get some data, share it with your downstream consumers, and a few days later, you'll get some complaints. They may be about an incomplete dataset, inefficient data organization, or completely broken data requiring internal restoration processes and backfilling.

As you can see by now, bringing data to your system is a key task for making your life and your users' lives better. For that reason, this book has to start by covering data ingestion design patterns.

The patterns presented in this chapter address scenarios and challenges you may face while integrating data from external providers or from your other pipelines. It starts by discussing two common data loading scenarios: the full and incremental loads that you'll use to acquire all or part of the dataset, respectively. Next, it discusses a special type of data ingestion called data replication. You'll see there two other patterns to copy the data without and with transformation that may help you address data privacy issues.

Since ingesting the data also covers some topics not closely related to the moving data itself, you'll also learn more about technical parts of data ingestion. First, you need to know when to start the ingestion process, and here, the data readiness section will be helpful. Second, you must also know how to improve the user experience and address one of the biggest data engineering nightmares, the small files problem. That's where the data compaction section will come in handy. In the final section, you'll also learn that data

ingestion is not always a predictable process. Hopefully, the External Trigger pattern, which we discuss last in this chapter, will address this uncertainty.

With all the context set up, I can now let you discover the first data ingestion patterns for full and incremental load scenarios!

### Full Load

The full load design pattern refers to the data ingestion scenario that works on a complete dataset each time. It can be useful in many situations, including a database bootstrap or a reference dataset generation.

#### Pattern: Full Loader

The Full Loader implementation is one of the most straightforward patterns presented in this book. However, despite its simple two-step construction, it has some pitfalls.

##### Problem

You're setting up the Silver layer for our use case. One of the transformation jobs requires extra information about the device from an external data provider. This device's dataset changes only a few times a week. It's also a very slowly evolving entity with the total number of rows not exceeding one million. Unfortunately, the data provider doesn't define any attribute that could help you detect the rows that have changed since the last ingestion.

##### Solution

The lack of the last updated value in the dataset makes the Full Loader pattern an ideal solution to the problem.

The simplest implementation relies on two steps, extract and load (EL). It uses native data stores commands to export data from one database and import it to another. This EL approach is ideal for homogeneous data stores because it doesn't require any data transformation.

###### Passthrough Jobs

Extract and load jobs are also known as passthrough jobs because the data is simply passing through the pipeline, from the source to the destination.

Unfortunately, using EL pipelines will not always be possible. If you need to load the data between heterogeneous databases, you will need to adapt the input format to the output format with a thin transformation layer between the extract and load steps. Your pipeline then becomes an extract, transform, load (ETL) job, where you can leverage a data processing framework that often provides native interfaces for interacting with various data stores.

### Consequences

Despite this simple task of moving a dataset between two data stores, the Full Loader pattern comes with some challenges.

### Data volume

The Full Loader's implementations will often be batch jobs running on some regular schedule. If the data volume of the loaded dataset grows slowly, your data loading infrastructure will probably work without any issues for a long time thanks to these almost constant compute needs.

On the other hand, a more dynamically evolving dataset can lead to some issues if you use static compute resources to process it. For example, if the dataset doubles in size from one day to another, the ingestion process will be slower and can even fail due to static hardware limitations.

To reduce this data variability impact on your data loading process, you can leverage the auto-scaling capabilities of your data processing layer.

### Data consistency

The second risk related to the Full Loader pattern is the risk of losing data consistency. As the data must be completely overwritten, you may be tempted to fully replace it in each run with a drop-and-insert operation. If you opt for this strategy, you should be aware of its shortcomings.

First, think about data consistency from the consumer's perspective. What if your data ingestion process runs at the same time as the pipelines reading the dataset? Consumers might process partial data or even not see any data at all if the insert step doesn't complete. Transactions automatically manage data visibility, and they're the easiest mitigation of this concurrency issue. If your data store doesn't support transactions, you can rely on a single data exposition abstraction,1 such as view, and manipulate only the underlying technical and hidden structures. Figure 2-1 shows how to perform a switch between two technical tables and keep the data available.

Figure 2-1. Single data exposition abstraction example with a database view

Second, keep in mind that you may need to use the previous version of the dataset if unexpected issues arise. If you fully overwrite your dataset, you may not be able to perform this action, unless you use a format supporting the time travel feature, such as Delta Lake, Apache Iceberg, or Google Cloud Platform (GCP) BigQuery. Eventually, you can also implement the feature on your own by relying on the single data exposition abstraction concept presented in Figure 2-1.

### Not Only Ingestion

Although this chapter discusses data ingestion, remember that all the patterns presented here directly impact data analytics and data science workloads, as they load the data into the system.

### Examples

Let's see now how to implement the pattern in different technical contexts. First, if you have to ingest a dataset between two identical or compatible data stores, you can simply write a script and deploy it to your runtime service. Example 2-1 demonstrates how to load files from one Amazon Web Services (AWS) S3 bucket to another. The command automatically synchronizes the content of the buckets and removes all objects missing in the source but present in the destination (the --delete argument).

Example 2-1. Synchronization of buckets

```
aws s3 sync s3://input-bucket s3://output-bucket --delete
```

Commands like aws s3 sync are a great way to simply move datasets, but sometimes, the load operation may require some fine-tuning, like adding parallel or distributed processing. An example of such an implementation is Apache Spark.

Apache Spark, as a distributed data processing framework, can be seamlessly scaled so that even drastically changing volumes shouldn't negatively impact the data ingestion job as long as you scale your compute infrastructure. Besides, if you use it with a table file format like Delta Lake, you automatically address the consistency issues presented previously, thanks to the transactional and versioning capabilities. The cherry on the cake is that the code for the full data ingestion is pretty easy. Example 2-2 shows how to leverage the Apache Spark read and write API to write JSON records as a Delta Lake table.

Example 2-2. Extract load implementation with Apache Spark and Delta Lake

```
input_data = spark.read.schema(input_data_schema).json("s3://devices/list")
```

```
input_data.write.format("delta").save("s3://master/devices")
```

Finally, you can also implement the pattern for databases without the native versioning capability. Let's take an example of Apache Airflow and PostgreSQL. The implementation is more complex because it requires dedicated data ingestion and data exposition tasks.

The data ingestion task writes the dataset to an explicitly versioned table. It can be expressed as a COPY statement from Example 2-3, where ${version} is a parameter provided by Apache Airflow's operator.

Example 2-3. Loading data to a versioned table

```
COPY devices_${version} FROM '/data_to_load/dataset.csv' CSV  DELIMITER ';' HEADER;
```

Next, the exposition task changes the reference of the view exposed to the end users. For that, it can rely on the view update operation in Example 2-4.

Example 2-4. Exposing one versioned table publicly

```
CREATE OR REPLACE VIEW devices AS SELECT * FROM devices_${version}
```

The pipeline may require additional steps, such as retrieving the input dataset and creating versioned tables. I omit them here for brevity's sake, but you'll find the full example in the GitHub repo.

Incremental Load

Full load is an easy scenario, but it can be costly to implement for continuously growing datasets. Incremental load patterns are better candidates for them because they ingest smaller parts of a physically or logically divided dataset, often at a higher frequency.

Pattern: Incremental Loader

The first incremental design pattern processes new parts of the dataset, thus its name, the Incremental Loader.

Problem

In our blog analytics use case, most visit events come from a streaming broker in real time. But some of them are still being written to a transactional database by legacy producers.

You need to create a dedicated data ingestion process to bring these legacy visits to the Bronze layer. Due to the continuously increasing data volume, the process should only integrate the visits added since the last execution. Each visit event is immutable.

Solution

The continuously growing dataset is a good condition in which to use the Incremental Loader pattern. There are two possible implementations that depend on the input data structure:

The first implementation uses a so-called delta column to identify rows added since the last run. Typically, for event-driven data like immutable visits, this column will be ingestion time.

The second implementation relies on time-partitioned datasets. Here, the ingestion job uses time-based partitions to detect the whole new bunch of records to ingest. It greatly simplifies and optimizes the process as the data to ingest is already filtered out and logically organized in the storage layer. To ensure that a new partition can be ingested, you can use the Readiness Marker pattern.

Be Aware of Real-Time Issues

Using event time as a delta column is risky. Your ingestion process might miss records if your data producer emits late data (see “Late Data”) for the event time you already processed.

You can find both implementations in Figure 2-2. As you’ll notice, the delta column implementation needs to remember the last ingestion time value to incrementally process new rows. On the other hand, the partition-based implementation doesn’t have this requirement because it can implicitly resolve the partition to process from the execution date. For example, if the loader runs at 11:00, it can target the partition for the previous hour.

Figure 2-2. Two possible implementations of the Incremental Loader pattern

Consequences

The incremental quality is nice for reducing ingested data volume, but it can also be challenging.

Hard deletes

Using the pattern can be tricky for mutable data. Let’s say that instead of the append-only visits, you need to deal with the events that can be updated and deleted. If the ingestion

process relies on the delta column, it can identify the updated rows and copy their most recent version. Unfortunately, it's not that simple for deleted rows.

When a data provider deletes a row, the information physically disappears from the input dataset. However, it's still present in your version of the dataset because the delta column doesn't exist for a deleted row. To overcome this issue you can rely on soft deletes, where the producer, instead of physically removing the data, simply marks it as removed. Put differently, it uses the UPDATE operation instead of DELETE.

### Insert-Only Tables

Another answer to the mutability issue could be insert-only datasets. As the name suggests, they accept only new rows via an INSERT operation. They shift the data reconstruction responsibility onto consumers, who must correctly detect any deleted and modified entries. The insert-only tables are also known as append-only tables.

### Backfilling

Even these basic data ingestion tasks have a risk of backfilling. The pattern might have a surprisingly bad effect on your data ingestion pipelines in that scenario.

Let's imagine a pipeline relying on the delta column implementation. After processing two months of data, you were asked to start a backfill. Now, when you launch the ingestion process, you'll be doing the full load instead of the incremental one. Therefore, the job will need more resources to accommodate the extra rows.

Thankfully, you can mitigate the problem by limiting the ingestion window. For example, if your ingestion job runs hourly, you can limit the ingestion process to one hour only. In SQL, it can be expressed as delta_column BETWEEN ingestion_time AND ingestion_time + INTERVAL '1 HOUR'. This operation brings two things:

Better control over the data volume. Even in the case of backfilling, you won't be surprised by the compute needs.

Simultaneous ingestion. You can run multiple concurrent backfilling jobs at the same time, as long as the input data store supports them.

The dataset size problem doesn't happen in the partition-based implementation if your ingestion job works on one partition at a time.

### Examples

The script-based example from the Full Loader also applies to the incremental load. The operation from Example 2-5 simply moves all objects with the date=2024-01-01 prefix key to another bucket. Even though it looks straightforward, there's a catch. If you omit the date=2024-01-01 prefix from the right side of the command, the ingestion task will flatten the output storage layout.

Example 2-5. Synchronization of S3 buckets

```
aws s3 sync s3://input/date=2024-01-01 s3://output/date=2024-01-01 --delete
```

Sometimes the implementation can't be a simple script. As with the Full Loader, you can rely on your processing and orchestration layers. Example 2-6 is an example of the partition-based implementation with Apache Airflow and Apache Spark. The workflow in Apache Airflow, most commonly known as a DAG, starts with a FileSensor waiting for the next partition to be available. It's a required step to avoid loading partial data and propagating an invalid dataset. The snippet uses a simple verification on the next partition, but Airflow supports sensors for other backends and includes AWS Glue (AwsGlueCatalogPartition Sensor), GCP BigQuery (BigQueryTablePartitionExistenceSensor), or Databricks (DatabricksPartitionSensor). Once the partition is ready, the pipeline triggers the data ingestion job.

Example 2-6. Incremental Loader DAG example

```
next_partition_sensor = FileSensor(
 task_id='input_partition_sensor',
 filepath=get_data_location_base_dir() + '/{{ data_interval_end | ds }}',
 mode='reschedule',
)
load_job_trigger = SparkKubernetesOperator(application_file='load_job_spec.yaml',
 # ... omitted for brevity
)
load_job_sensor = SparkKubernetesSensor(
 # ... omitted for brevity
)

next_partition_sensor >> load_job_trigger >> load_job_sensor
```

The data ingestion job takes the arguments for the input and output locations defined in Example 2-7. The job definition relies on the immutable execution time expressed here as the {{ ds }} macro. If you use these immutable properties, you greatly simplify the backfilling

because they will never change. The EventsLoader job uses the specified time expression arguments alongside the Apache Spark API to read and write the dataset.

Example 2-7. Partitioned events loader

```
# ...
  mainClass: com.waitingforcode.EventsLoader
  mainApplicationFile: "local:///tmp/dedp-1.0-SNAPSHOT-jar-with-dependencies.jar"
  arguments:
    - "/data_for_demo/input/date={{ ds }}"
    - "/data_for_demo/output/date={{ ds }}"
```

Since you already learned how to leverage the Apache Spark API in the previous section, let's move directly to the Incremental Loader based on a delta column implementation. The delta column implementation slightly differs from the partitioned version as it removes the sensor step and executes the ingestion job directly (see Example 2-8).

Example 2-8. Incremental Loader for transactional (not partitioned) dataset

```
load_job_trigger = SparkKubernetesOperator(
  # ...
  application_file='load_job_spec_for_delta_column.yaml',
)
load_job_sensor = SparkKubernetesSensor(
  # ...
)

load_job_trigger >> load_job_sensor
```

This job includes an extra filtering operation on top of the delta column to get the rows corresponding to the time range configured for a given job execution. Thanks to the filter, if you need to run a job execution again, it will not take any extra records, and thus it will guarantee consistent data volume. Example 2-9 shows the job adapted to the delta column implementation.

Example 2-9. Data ingestion job with delta column and time boundaries

```
in_data = (spark_session.read.text(input_path).select('value',
    functions.from_json(functions.col('value'), 'ingestion_time TIMESTAMP')))
```

```
input_to_write = in_data.filter(
  f'ingestion_time BETWEEN "{date_from}" AND "{date_to}"'
)

input_to_write.mode('append').select('value').write.text(output_path)
```

Pattern: Change Data Capture

The Incremental Loader will not work for all use cases. When you need a lower ingestion latency or built-in support for the physical deletes, the next pattern will be a better choice.

Problem

Unfortunately, the legacy visit events you integrated with the Incremental Loader must evolve. Their ingestion rate is too slow, and downstream consumers have started to complain about too much time spent waiting for the data. To mitigate the issue, your product manager asked you to integrate these transactional records into your streaming broker as soon as possible. The ingestion job must capture each change from the table within 30 seconds and make it available to other consumers from a central topic.

Solution

The latency requirement makes it impossible to use the Incremental Loader. The pattern has some job scheduling and query execution overheads that could make the expected latency difficult to reach.

A better candidate is the Change Data Capture (CDC) pattern. Due to its internal ingestion mechanism, it guarantees lower latency. The pattern consists of continuously ingesting all modified rows directly from the internal database commit log. It allows lower-level and faster access to the records, compared to any high-level query or processing task.

A commit log is an append-only structure. It records any operations on the existing rows at the end of the logfile. The CDC consumer streams those changes and sends them to the streaming broker or any other configured output. From that point on, consumers can do whatever they want with the data, such as storing the whole history of changes or keeping the most recent value for each row.

Besides guaranteeing lower latency, CDC intercepts all types of data operations, including hard deletes. So there is no need to ask data producers to use soft deletes for data removal.

### Consequences

The latency promise is great, but like any engineering component, it also brings its own challenges.

#### Complexity

The CDC pattern is different from the two others covered so far as it requires different setup skills. The Full Loader and Incremental Loader can be implemented by a data engineer alone, as long as the required compute and orchestration layers exist. The CDC pattern may need some help from the operations team, for example, to enable the commit log on the servers.

#### Data scope

Be careful about the data scope you want to target with this pattern. Depending on the client's implementation, you may be able to get the changes made only after starting the client process. If you are interested in the previous changes too, you will need to combine CDC with other data ingestion patterns from this chapter.

#### Payload

Besides latency, another difference between CDC and the Incremental Loader is the payload. CDC will bring additional metadata with the records, such as the operation type (update, insert, delete), modification time, or column type. As a consumer of this data, you may need to adapt your processing logic to ignore irrelevant attributes.

#### Data semantics

Don't get this wrong; the pattern ingests data at rest. As a side effect, these static rows become data in motion. Why is this worth emphasizing? Data in motion has different processing semantics for many operations that appear to be trivial in the data-at-rest world.

Let's look at an example of the JOIN operation. If you perform it against static tables orchestrated from your data orchestration layer and you don't get a result, it's because there is no matching data. But if you run the query against two dynamic streaming sources and you don't get a match, it's because the data might not be there yet. One stream can simply be later than the other, and the JOIN operation may eventually succeed in the future. For that reason, you shouldn't consider the data ingested from the CDC consumer to be static data.

### Examples

There are many ways to implement the pattern. You can create your own commit log reader or rely on the available solutions. One of the most popular open source solutions is Debezium.2 The framework supports many relational and NoSQL databases and uses Kafka Connect as the bridge between the data-at-rest and data-in-motion worlds. The setup is mostly configuration driven (see Example 2-10).

Example 2-10. Debezium Kafka Connect configuration for PostgreSQL

```
{
 "name": "visits-connector",
 "config": {
   "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
   "database.hostname": "postgres", "database.port": "5432",
   "database.user": "postgres", "database.password": "postgres",
   "database.dbname" : "postgres", "database.server.name": "dbserver1",
   "schema.include.list": "dedp_schema",
   "topic.prefix": "dedp"
 }
}
```

This snippet shows the configuration file for a PostgreSQL database. It defines the connection parameters, all the schemas to include in the watching operation, and finally the prefix for the created topic for each synchronized table. As a result, if there is a dedp_schema.events table, the connector will write all the changes to the dedp.dedp_schema.events topic. Figure 2-3 illustrates this dependency.

Figure 2-3. Debezium architecture in a nutshell, showing a Kafka Connect consumer leveraging the database commit logs and writing all matching datasets to dedicated Apache Kafka topics

Besides creating a new Kafka Connect task, you need to prepare the database. The PostgreSQL expects the logical replication stream enabled with the pgoutput plug-in and a user with all necessary privileges. Now, you certainly understand better why the CDC pattern is more challenging in terms of setup than the Incremental Loader.

The good news is that lake-native formats do support CDC in a simpler way. Delta Lake has a built-in change data feed (CDF) feature to stream the changed rows that you can enable

either as a global session property or as a local table property with the readChangeFeed option. Example 2-11 demonstrates both configuration approaches.

Example 2-11. CDF setup in Delta Lake

```
spark_session_builder
 .config('spark.databricks.delta.properties.defaults.enableChangeDataFeed', 'true')

spark_session.sql('''
 CREATE TABLE events (
  visit_id STRING, event_time TIMESTAMP, user_id STRING, page STRING
 )
 TBLPROPERTIES (delta.enableChangeDataFeed = true)''')
```

Also, with the enableChangeDataFeed property, you can configure the throughput limits with the maxFilesPerTrigger or maxBytesPerTrigger. The tables also support time travel, so you can start reading from a particular version. Example 2-12 shows the most basic reader configuration processing four files each time from the very first table's version.

Example 2-12. CDF usage in Delta Lake

```
events = (spark_session.readStream.format('delta')
 .option('maxFilesPerTrigger', 4).option('readChangeFeed', 'true')
 .option('startingVersion', 0).table('events'))
query = events.writeStream.format('console').start()
```

Other than a different reader configuration, there is a difference in the returned dataset. Example 2-13 shows that a CDF table contains some extra columns compared to the classical table.

Example 2-13. CDF table output

```
+-------------+-------------------+------------+---------------+--------------------+
|    visit_id|       event_time|_change_type|_commit_version|  _commit_timestamp|
+-------------+-------------------+------------+---------------+--------------------+
| 1400800256_0|2023-11-24 01:44:00|     insert|           6|2023-12-03 13:28:...|
| 1400800256_1|2023-11-24 01:36:00|     insert|           6|2023-12-03 13:28:...|
| 1400800256_2|2023-11-24 01:44:00|     insert|           6|2023-12-03 13:28:...|
```

```
| 1400800256_3|2023-11-24 01:37:00|     insert|          6|2023-12-03 13:28:...|
+-------------+-------------------+------------+---------------+--------------------+
```

The extra columns in Example 2-13 begin with the _ and mean, respectively, how the row changed, at which version, and when. This example comes from an append-only table and may not be that interesting. However, if you apply some in-place operations like UPDATE, the changes feed will contain rows for both pre- and post-update versions. You can identify them with update_preimage and update_postimage types.

## Replication

Another family of data ingestion patterns is data replication, the main goal of which is to copy data as is from one location to another. But that's only in a perfect world. In the real world, you will often need to alter the input, for example, because of regulatory compliance.

### Data Loading Versus Replication

These two concepts look similar at first glance, but there is a slight difference. Replication is about moving data between the same type of storage and ideally preserving all its metadata attributes, such as primary keys in a database or event positions in a streaming broker. Loading is more flexible and doesn't have this homogeneous environment constraint.

### Pattern: Passthrough Replicator

As with data loading, the data replication area has a passthrough mode that you can use by default if you can accept the consequences.

#### Problem

Your deployment process consists of three separate environments: development, staging, and production. Many of your jobs use a reference dataset with device parameters that you load daily on production from an external API. For a better development experience and easier bug detection, you want to have this dataset in the remaining environments.

The reference dataset loading process uses a third-party API and is not idempotent. This means that it may return different results for the same API call throughout the day. That's why you can't simply copy and replay the loading pipeline in the development and staging environments. You need the same data as in production.

#### Solution

A data provider which is not idempotent, plus the required consistency across environments, is a great reason to use the Passthrough Replicator pattern. You can set it up either at the compute level or the infrastructure level.

The compute implementation relies on the EL job, which is a process with only two phases, read and write. Ideally, the EL job will copy files or rows from the input as is (i.e., without any data transformation). Otherwise, it could introduce data quality issues, such as type conversion from string to dates or rounding of floating numbers.

The infrastructure part is based on a replication policy document where you configure the input and output location and let your data storage provider replicate the records on your behalf.

### Consequences

The key learning here is to keep the implementation simple. However, even the simplest implementation possible may have some challenges to address.

### Keep it simple

Remember, you need the data as is. To reduce the interference risk in the replicated dataset, you should rely on the simplest replication job possible, which is ideally the data copy command available in the database.

However, when the command is not available and you must use a data processing framework for a text format like JSON, avoid relying on the JSON I/O API. Instead, use the simpler raw text API that will take and copy lines as they are, without any prior interpretation.

Additionally, if you do care about other aspects, such as the same number of files or even the same filenames, you should avoid using a distributed data processing framework if it doesn't let you customize those parameters.

### Security and isolation

Cross-environment communication is always tricky and can be error prone if the replication job has bugs. In that scenario, there is a risk of negatively impacting the target environment, even to the point of making it unstable. You certainly don't want to take that risk in production, so for that reason, you should implement the replication with the push approach instead of pull. This means that the environment owning the dataset will copy it to the others and thus control the process with its frequency and throughput.

Even though the push strategy greatly reduces the risk of instability, you can still encounter some issues. You can imagine a use case when you start a job on your cloud subscription and it takes the last available IP address in the data processing subnet. Other jobs will not run as long as the replicator doesn't complete.

### PII data

If the replicated dataset stores personally identifiable information (PII), or any kind of information that cannot be propagated from the production environment, use the Transformation Replicator pattern, which we discuss next. It adds an extra transformation step to get rid of any unexpected attributes.

### Latency

The infrastructure-based implementation often has some extra latency, and you should always check the service level agreement (SLA) of the cloud provider to see if it's acceptable as the solution. Even though the problem announcement discusses a development experience, you might want to implement it for other and more time-sensitive scenarios.

### Metadata

Do not ignore the metadata part because it could make the replicated dataset unusable. For example, replicating only the Apache Parquet files of a Delta Lake table will not be enough. The same applies to Apache Kafka, where you should care not only about the key and values but also about headers and the order of events within the partition.

### Examples

You can implement the pattern in two ways. The first implementation relies on the code, which can be either a distributed data processing framework or a data copy utility script running on your storage layer that you already saw in the Full Loader pattern. The code-based solution is more error prone, but you can leverage your framework's I/O layer (see Example 2-14).

Example 2-14. JSON data replication with Apache Spark

```
input_dataset = spark_session.read.text(f'{base_dir}/input/date=2023-11-01')

input_dataset.write.mode('overwrite').text(f'{base_dir}/output-raw/date=2023-11-01')
```

The code uses Apache Spark to synchronize semi-structured JSON files. It uses the simplest API possible to copy JSON lines data without any interference in the data itself. However, please notice that the snippet in the example doesn't preserve the files (i.e., the number of files in the source and destination can be different, even though they will both store the same data).

Despite that, Example 2-14 looks simple. Unfortunately, it'll not always be that straightforward for other data stores. Let's see the same extract and load operation for an Apache Kafka topic that requires an extra ordering guarantee within the partitions. Example 2-15 would be a simple extract and load job if it didn't have the write_sorted_events function in the middle. The function is crucial to guaranteeing that the replicated records include the metadata (.option('includeHeaders'...) and keep the same order as the input records (sortWithinPartitions('offset', ascending=True)).

Example 2-15. Passthrough Replicator with an ordering semantic

```
events_to_replicate = (input_data_stream
 .selectExpr('key', 'value', 'partition', 'headers', 'offset'))

def write_sorted_events(events: DataFrame, batch_number: int):
 (events.sortWithinPartitions('offset', ascending=True).drop('offset').write
 .format('kafka').option('kafka.bootstrap.servers', 'localhost:9094')
  .option('topic', 'events-replicated').option('includeHeaders', 'true').save())

write_data_stream = (events_to_replicate.writeStream
  .option('checkpointLocation', f'{get_base_dir()}/checkpoint-kafka-replicator')
  .foreachBatch(write_sorted_events))
```

Apart from the code-based implementations, there are the infrastructure-based ones. For Apache Kafka topic replication, you could use the MirrorMaker utility,3 while for the files, it could be the replication mechanism of your cloud provider. In Example 2-16, you can see an example of an S3 bucket replication with Terraform.

Example 2-16. AWS S3 bucket replication

```
resource "aws_s3_bucket_replication_configuration" "replication" {
 role   = aws_iam_role.replication.arn
 bucket = aws_s3_bucket.devices_production.id

 rule {
  id = "devices"
  status = "Enabled"
```

```
  destination {
    bucket       = aws_s3_bucket.devices_staging.arn
    storage_class = "STANDARD"
  }
 }
}
```

Pattern: Transformation Replicator

Even though the Kafka example looks complex, your replication scenario can require even more code effort. This is the case when you use the Transformation Replicator pattern.

Problem

Before releasing a new version of your data processing job, you want to perform tests against real data to avoid surprises during production. You can't use a synthetic data generator because your data provider often has data quality issues and it's impossible to simulate them with any tool. You have to replicate the data from production to the staging environment. Unfortunately, the replicated dataset contains PII data that is not accessible outside the production environment. As a result, you can't use a simple Passthrough Replicator job.

Solution

One big problem of testing data systems is...the data itself. If the data provider cannot guarantee consistent schema and values, using production data is unavoidable. Unfortunately, the production data very often has some sensitive attributes that can't move to other environments, where possibly more people can access it due to less strict access policies.

In that scenario, you should implement the Transformation Replicator pattern, which, in addition to the classical read and write parts from the Passthrough Replicator pattern, has a transformation layer in between.

Transformation is a generic term, but depending on your technical stack, it can be implemented as either of the following:

A custom mapping function if you use a data processing framework like Apache Spark or Apache Flink

A SQL SELECT statement if your processing logic can be easily run and expressed in SQL

The transformation consists of either replacing the attributes that shouldn't be replicated (for example, with the Anonymizer pattern) or simply removing them if they are not required for processing.

### Consequences

Since you'll be writing some custom logic, the risk of breaking the dataset is higher than with the Passthrough Replicator. And that's not the only drawback of the pattern!

#### Transformation risk for text file formats

Let's look at a rather innocent transformation example on top of a text file format such as JSON or CSV. You defined a schema for the replicated dataset but didn't notice that the datetime format is different from the standard used by your data processing framework. As a result, the replicated dataset doesn't contain all the timestamp columns and your job of staging fails because of that. Although you should be able to fix the issue very quickly, it causes unnecessary work in the release process.

That's why you should still apply the "keep it simple" approach here. In our example, instead of defining the timestamp columns as is, you can simply configure them as strings and not worry about any silent transformations.

#### Desynchronization

You need to take special care that the replication jobs implement this pattern to avoid any privacy-related issues. Data is continuously evolving, and nothing guarantees that the privacy fields you have today will still be valid in the future. Maybe new ones will appear or attributes that are not currently considered PII will be reclassified as PII.

To avoid these kinds of issues, if possible, you should rely on a data governance tool, such as a data catalog or a data contract in which the sensitive fields are tagged. With such a tool, you can automatize the transformation logic. Otherwise, you'll need to implement the rules on your own.

### Examples

As mentioned previously, there are two possible implementation approaches. The first of them is a data reduction approach that eliminates unnecessary fields. It's relatively easy to express. Some databases and compute layers, such as Databricks and BigQuery, support an EXCEPT operator. Example 2-17 shows this operator in action by selecting all rows but ip, latitude, and longitude.

Example 2-17. Dataset reduction with EXCEPT operator

```
SELECT * EXCEPT (ip, latitude, longitude)
```

Additionally, you can leverage your data processing framework to remove irrelevant columns. Example 2-18 shows how to use the PySpark drop function to remove the ip, latitude, and logitude columns from the dataset.

Example 2-18. Dataset reduction with drop function

```
input_delta_dataset = spark_session.read.format('delta').load(users_table_path)

users_no_pii = input_delta_dataset.drop('ip', 'latitude', 'longitude')
```

An alternative way to transform the dataset consists of controlling access to it. For example, your data provider can expose your user to only a subset of permitted columns, such as the ones in Example 2-19.

Example 2-19. Column-level access for user_a on table visits

```
GRANT SELECT (visit_id, event_time, user_id) ON TABLE visits TO user_a
```

Example 2-19 shows how to grant permissions on a subset of fields in AWS Redshift. The user_a will only be able to access the three columns mentioned after the SELECT operation. Although this is more verbose than the EXCEPT-based solution, it adds an extra layer of protection for accessing the private attributes. You'll learn more about this approach in the Fine-Grained Accessor pattern.

However, removing the rows may not always be an option, especially if they are important for the tested data processing job. For these use cases, you will define a column-based transformation to alter the sensitive fields (see Example 2-20).

Example 2-20. Column-based transformation

```
devices_trunc_full_name = (input_delta_dataset
  .withColumn('full_name',
    functions.expr('SUBSTRING(full_name, 2, LENGTH(full_name))'))
)
```

Column-based transformations work great for column-targeted operations. If you need to operate at the row level, or if the modification rule is complex, you may need a mapping function like the one in Example 2-21. The code leverages the Scala API for Apache Spark because it's more concise than the Python one. The code first calls the as function to convert input rows to a specific type. Later, it applies the transformation logic and exposes an eventually modified row from the transformed attribute.

Example 2-21. Mapping function from a strongly typed Scala Spark API

```
case class Device(`type`: String, full_name: String, version: String) {
 lazy val transformed = {
  if (version.startsWith("1.")) {
   this.copy(full_name = full_name.substring(1), version = "invalid")
  } else {
    this
  }
 }
}
inputDataset.as[Device].map(device => device.transformed)
```

Data Compaction

Even a perfect dataset can become a bottleneck, especially when it grows over time because of new data. As a result, at some point, metadata-related operations like listing files can take even longer than data processing transformations.

Pattern: Compactor

The easiest way to address this issue of a growing dataset is to reduce the storage footprint of the underlying files. The Compactor pattern helps in this task.

Problem

Your real-time data ingestion pipeline synchronizes events from a streaming broker to an object store. The main goal is to make the data available for batch jobs within at most 10 minutes. Since it's a simple passthrough job, the pipeline is running without any apparent issues. However, after three months, all the batch jobs are suffering from the metadata overhead problem due to too many small files composing the dataset. As a result, they spend 70% of their execution time on listing files to process and only the remaining 30% on processing the data. This has a serious latency and cost impact as you use pay-as-you-go services.

Solution

Having small files is a well-known problem in the data engineering space.4 It has been there since the Hadoop era and is still present even in modern, virtually unlimited, object store–based lakehouses. Storing many small files involves longer listing operations and heavier I/O for opening and closing files. A natural solution to this issue is to store fewer files.

That's what the Compactor pattern does. It addresses the problem by combining multiple smaller files into bigger ones, thus reducing the overall I/O overhead on reading.

The implementation varies with the technology. Open table file formats have their dedicated compaction command that often runs a transactional distributed data processing job under the hood to merge smaller files into bigger ones as a part of the new commit. Apache Iceberg performs this via a rewrite data file action, while Delta Lake employs the OPTIMIZE command.

The compaction works differently on Apache Hudi, which is the third open table file format. A Hudi table can be configured as a merge-on-read (MoR) table where the dataset is written in columnar format and any subsequent changes are written in row format. As this approach favors faster writes, to optimize the read process, the compaction operation in Hudi merges the changes from the row storage with the columnar storage. This approach differs from Delta Lake and Apache Iceberg as they operate in the homogeneous columnar format only.

The Compactor also works for data stores other than lake-related storage. One of them is Apache Kafka, which is an append-only key-based logs system. In this configuration-based implementation, you only need to configure the compaction frequency. The whole compaction process is later managed, according to the frequency, by the data store. However, in key-based systems, the compaction consists of optimizing the storage by keeping only the most recent entry for a given record key. It still improves the storage footprint, but unlike compaction in table file formats, the operation overwrites the present data.

### Consequences

Despite its apparent harmlessness and native support in many data stores, the Compactor can require some significant design effort.

### Cost versus performance trade-offs

The compaction job is just a regular data processing job that can be compute intensive on big tables. If you consider only this aspect, you should execute it rarely, such as once a day, ideally outside working hours, and outside the pipeline generating the dataset.

On the other hand, rare execution can be problematic for jobs that work on the not yet compacted data as they will simply not take advantage of this optimization technique. You'll then need to choose your strategy and accept that it may not be perfect from both the cost and performance perspectives.

There is no one-size-fits-all solution, unfortunately. Sometimes, running it once a day will be fine as the not compacted dataset may be small enough to not impact consumers. On the other hand, sometimes it won't be acceptable and you'll even prefer to include compaction in the data ingestion process since penalizing consumers will have a bigger impact than impacting the ingestion throughput.

### Consistency

Remember, compaction simply rewrites already existing data. Consequently, consumers may have difficulties distinguishing the data to use from the data being compacted. For that reason, compaction is much simpler and safer to implement in modern, open table file formats with ACID properties (such as Delta Lake and Apache Iceberg) than in raw file formats (such as JSON and CSV).

### Cleaning

The compaction job may preserve source files. Consequently, the small files will still be there and will continue impacting metadata actions. For that reason, the compaction job alone won't be enough. You'll have to complete it with a cleaning job to reclaim the space taken up by the already compacted files.

To overcome that side effect, you will need to reclaim this occupied but not used space with commands like VACUUM, which are available in modern data storage technologies like Delta Lake, Apache Iceberg, PostgreSQL, and Redshift. But choose your cleaning strategy wisely, because you may not recover your dataset to the version based on these deleted, already compacted files.

### Example

Let's start by seeing how the pattern applies to data lakes and lakehouses using open table file formats like Delta Lake. The format provides a native compaction capability that's available from the programmatic API or as a SQL command. In both cases, you should look for the optimize keyword, as shown in Example 2-22. The code initializes the path-based Delta table object and calls the data compaction operation. As it only reorganizes the files that are available when the job starts, it's a safe and nonconflicting operation for readers and writers.

Example 2-22. Compaction job with Delta Lake

```
devices_table = DeltaTable.forPath(spark_session, table_dir)

devices_table.optimize().executeCompaction()
```

However, the compaction may require an extra VACUUM step to clean all irrelevant (because already compacted) files. Example 2-23 again leverages the Delta table abstraction but this time calls the vacuum() function. The cleaning process applies only to the files that are older than the configured retention threshold. Otherwise, it could lead to a corrupted table state because the vacuum could remove the files that are being written and are not yet committed.

Example 2-23. VACUUM in Delta Lake

```
devices_table = DeltaTable.forPath(spark_session, table_dir)

devices_table.vacuum()
```

Compaction is also available in other data stores, but compared to the Delta Lake example, it can be a bit misleading. Let's take a look at a transactional PostgreSQL database. Compaction there uses only the VACUUM command that reclaims space taken by dead tuples, which are the deleted rows not physically removed from the storage layer. You can trigger it with an explicit command.

This pattern is also present in Apache Kafka. When you create a topic, you can set the log compaction configuration, which is particularly useful when you write key-based records and each new append is a state update. With the compaction enabled, Apache Kafka runs a cleaning process that removes all but the latest versions of each key. The configuration supports properties like log.cleanup.policy (compaction strategy) and log.cleaner.min.compaction.lag.ms with log.cleaner.max.compaction.lag.ms (compaction frequency). Unlike in the Delta Lake example, Kafka's compaction is nondeterministic. You can't expect it to run on a regular schedule, and as a result, it doesn't guarantee that you'll always see a unique record for each key.

## Data Readiness

The different data ingestion semantics covered so far in this chapter are not the only problems you can encounter in this apparently easy data ingestion task. The next problematic question you'll certainly ask yourself is, "When should I start the ingestion process?"

### Pattern: Readiness Marker

The Readiness Marker is a pattern that helps trigger the ingestion process at the most appropriate moment. Its goal is to guarantee the ingestion of the complete dataset. Let's see how it achieves this.

#### Problem

Every hour, you're running a batch job that prepares data in the Silver layer of your Medallion architecture. The dataset has all known data quality issues fixed and is enriched with the extra context loaded from your user's database and from an external API service. For these reasons, other teams rely on it to generate ML models and BI dashboards. But there is one big problem: they often complain about incomplete datasets, and they've asked you to implement a mechanism that will notify them—directly or indirectly—when they can start consuming your data.

## Solution

The issue is particularly visible in the logically dependent but physically isolated pipelines maintained by different teams. Because of these isolated workloads, it's not possible for your job to directly trigger downstream pipelines. Instead, you can mark your dataset as ready for processing with the Readiness Marker pattern. Its implementation will depend on the file format and storage organization.

The first implementation uses an event to signal the dataset's completeness. Due to the popularity of object stores and distributed file systems in modern data architectures, this approach can be easily implemented with a flag file created after a successful data generation. This feature may be natively available in your data processing layer, as with Apache Spark (which writes a _SUCCESS file for raw file formats) or a new commit log for Delta Lake. If you can't leverage the data processing layer, you can implement the flag file from the data orchestration layer as a separate task executed after successful data processing. You'll see how to implement it in the Examples section.

A different implementation applies to partitioned data sources. If you're generating data for time-based tables or locations, the Readiness Marker can be conventional. Are you perplexed? Let's use an example to better understand the convention. Your job runs hourly and writes data to hourly based partitions. As a result, the run for 10:00 writes data to partition 10, the run for 11:00 to 11, and so on. Now, if your consumer wants to process partition 10, they must simply wait for your job to work on partition 11.

## Consequences

The Readiness Marker relies on the pull approach, in which the readers control the data retrieval process. As the implementations are implicit, there are some points to be aware of.

### Lack of enforcement

There is no easy way to enforce conventional readiness based on the flag file or the next partition detection. Either way, a consumer may start to consume the dataset while you're generating it.

Because of this implicitness, it's very important to communicate with your consumers and agree upon the conditions that may trigger processing on their side. Additionally, you should clearly explain the risks of not respecting the readiness conventions.

Reliability for late data

If the partitions are based on the event time, the partition-based implementation will suffer from late data issues. To understand this better, let's imagine that you closed the partition from eight, nine, and ten o'clock. This means your consumers have already processed the partitions from eight, nine, and ten o'clock. Unfortunately, you've just detected late data for nine o'clock. As your partitions are based on event time, you decide to integrate this data into the partition from nine o'clock. Very probably, your consumers won't be able to do the same as they may consider the partition to be closed.

That's why you should either consider partitions as immutable parts that will never change once closed or clearly define and share the mutability conditions with your consumers. Besides sharing the partition updates with consumers, you should notify them about new data to eventually process. We're going to address this issue in "Late Data".

Examples

For raw files, Apache Spark creates the flag file called _SUCCESS out of the box. Whenever you generate Apache Parquet, Apache Avro, or any other supported format, the job will always write the data files first, and only when this operation completes will it generate the marker file. Let's take a look at this in Example 2-24, where the job converts JSON files into Apache Parquet files and creates a _SUCCESS file under the hood.

Example 2-24. PySpark code generating the _SUCCESS file

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

dataset.write.mode('overwrite').format('parquet').save('devices-parquet')
```

As a consumer, you rely on the created _SUCCESS file to implement the Readiness Marker. If you use Apache Airflow, you can define this file as a condition in the FileSensor (see Example 2-25).

Example 2-25. FileSensor waiting for the _SUCCESS file

```
FileSensor(filepath=f'{input_data_file_path}/_SUCCESS, mode='reschedule'

# ...
```

Example 2-25 defines the filepath and also configures an important mode property to reschedule. Thanks to it, the sensor task will not occupy the worker slot without interruption. Instead, it will wake up and verify whether the configured file exists. This is an important point to consider if you want to avoid keeping your orchestration layer busy by only waiting for the data while other tasks may be ready to process it.

By the way, Apache Airflow also simplifies creating a Readiness Marker file if the latter is not natively available from the data processing job. The code from Example 2-26 generates the Readiness Marker file called COMPLETED in the last task called created_readiness_file.

Example 2-26. Creating a Readiness Marker file as a part of the data orchestration process

```
@task
def delete_dataset():
 shutil.rmtree(dataset_dir, ignore_errors=True)

@task
def generate_dataset():
 # processing part, omitted for brevity but available on GitHub

@task
def create_readiness_file():
 with open(f'{dataset_dir}/COMPLETED', 'w') as marker_file:
  marker_file.write('')

delete_dataset() >> generate_dataset() >> create_readiness_file()
```

This code snippet uses the @task decorator, which is a convenient way to declare data processing functions in Apache Airflow. The most important thing to keep in mind here is that the readiness marker should always be generated as the last step in a pipeline (i.e., after performing the last transformation of the dataset).

Event Driven

In ideal data ingestion scenarios, new datasets are available on a regular schedule. You can trigger your pipeline and use the Readiness Marker pattern before you start processing. In less-than-ideal scenarios, it's hard to predict the incoming frequency. Consequently, you must shift your mindset from static ingestion to event-driven ingestion.

## Pattern: External Trigger

The Readiness Marker pattern from the previous section relies on pull semantics, in which the consumer is responsible for checking whether there is new data to process. But the event-driven nature of a dataset favors push semantics, in which the producer is in charge of notifying consumers about data availability.

### Problem

Let's say the backend team of your organization releases new features at most once a week, between Monday and Thursday. Each release enriches your reference datasets, where you keep all the features available on your website at any given moment. So far, the refresh job for this dataset has been scheduled for once a day. It has been reloading data even when there were no changes, which led to some wastage of compute resources.

With the goal of reducing costs, you want to change the scheduling mechanism and run the pipeline only when there is something new to process. As the backend team sends a notification event to a central message bus whenever it publishes a new feature, you think about implementing a more event-driven approach.

### Solution

Unpredictable data generation is often caused by the event-driven nature of the data. This problem can be solved with a time-scheduled job that copies the whole dataset or runs very often to check whether there is something new to process. However, this wastes compute resources and adds an unnecessary operational burden. A better approach is to address this issue with an event-driven External Trigger pattern.

#### Not Only Trigger

If for whatever reason you don't have a job to trigger, you can run the ingestion process directly in the notification handler. We'll therefore talk about event-driven data ingestion.

The pattern consists of three main actions:

Subscribing to a notification channel. The first step sets up the connection between the external world (event-driven producers) and your pipelines. From now on, whenever something new happens, you'll have the ability to handle it.

Reacting to the notifications. This is the events handler, and the role of this step is to analyze the event and decide whether it should result in triggering a pipeline in the data orchestration layer or starting a job in the data processing layer. Depending on the message

bus technology, your handler can subscribe to particular events, such as data creation in a given table. If this filtering feature is not possible, the handler will need to implement it on its own.

Triggering the ingestion pipeline in the data orchestration or data processing layer. Here, the handler starts data ingestion either by triggering a workflow to orchestrate or directly triggering a job. For the sake of simplicity, one event should trigger one ingestion pipeline. However, it's also possible to start multiple ones, for example, when the same dataset is the input data source for various workloads.

You'll find these three main actions in Figure 2-4, where the trigger arrow is the reaction to the subscribed notification channel.

Figure 2-4. External trigger for a data orchestration layer and a data processing layer

Consequences

Even though this event-driven character sounds appealing because it reduces resource waste, it also has some consequences in your data stack.

Push versus pull

The External Trigger component can implement pull or push semantics. The difference is the key in understanding the pattern's impact on your system. The pull-based trigger continuously checks whether there are new events to process, while the push-based trigger does nothing as long as the event producer doesn't notify it about something new to process.

The pull-based trigger is a long-running job, so it's a process that stays up and checks at short, regular intervals whether there is new data to process. Although it's a technically valid implementation, it's not the most optimized since the job may spend most of its time pulling zero messages from the notification source.

A better alternative for this pattern is the push-based trigger, where the data source informs the endpoint(s) about new messages present in the bus. Each notification message starts a new consumer instance which finishes after reacting to the event.

Execution context

There is a risk that the external trigger may become just a ping mechanism that calls a data orchestrator endpoint. Although this simplistic approach is tempting, you should keep in mind that the triggered data ingestion pipeline will need to be maintained like any other. Hence, if you simply trigger it, you may not have enough context to understand why it has been triggered and what it's supposed to process.

For these reasons, it's important to enrich the triggering call with any appropriate metadata information, including the version of the trigger job, the notification envelope, the processing time, and the event time. They will be useful in day-to-day monitoring, when you will need to investigate the reasons for any eventual failures.

Error management

The events are the key elements here, and without them, you won't be able to trigger any work. For that reason, you should design the trigger for failure with the goal in mind to keep the events whatever happens. Typically, you'll rely here on the patterns described in the next chapter, such as the Dead-Letter pattern.

Examples

The pattern is easy to implement in the cloud. AWS, Azure, and GCP provide serverless function services that enable this event-driven capability. Therefore, they're great candidates for the triggers. Besides, most of the data orchestrators expose an API that you can use to start a pipeline. Let's see how to connect an AWS Lambda function with Apache Airflow. First, take a look at the pipeline definition in Example 2-27.

Example 2-27. Externally triggered DAG definition

```
with DAG('devices-loader', max_active_runs=5, schedule_interval=None,
 default_args={'depends_on_past': False,}) as dag:
# the pipeline just copies the file from the trigger,
# I'm omitting the content for brevity, it's available on GitHub
```

Compared to the previous DAGs, the one in Example 2-27 has the schedule_interval set to None. This means the Airflow scheduler will ignore it in the planning stage, and the only way to start the pipeline is with an explicit trigger action. It'll be the responsibility of the Lambda function that runs whenever a new object appears in the monitored S3 bucket (see Example 2-28).

Example 2-28. AWS Lambda handler to trigger the DAG

```
def lambda_handler(event, ctx):
 payload = {
  'event': json.dumps(event),
  'trigger': {
   'function_name': ctx.function_name, 'function_version': ctx.function_version,
```

```
    'lambda_request_id': ctx.aws_request_id
  },
  'file_to_load': (urllib.parse.
  unquote_plus(event['Records'][0]['s3']['object']['key'],encoding='utf-8')),
  'dag_run_id': f'External-{ctx.aws_request_id}'
}

trigger_response = requests
  .post('http://localhost:8080/api/v1/dags/devices-loader/dagRuns',
  data=json.dumps({'conf': payload}), auth=('dedp', 'dedp'), headers=headers)

if trigger_response.status_code != 200:
   raise Exception(f"""Couldn't trigger the `devices-loader` DAG.
    {trigger_response} for {payload}""")
else:
   return True
```

As you can see, the function is relatively straightforward. You may be wondering, where is the resiliency part? AWS Lambda implements it at the infrastructure level with failed-event destinations, where the service sends any records from the failed function's invocations. You can also configure the batch size for stream data sources or concurrency level.

### Explicitness of Code Snippets

Code snippets, unless specified otherwise, are written with readability in mind. That's one of the reasons why in Example 2-28 you see hardcoded credentials. As a general rule, hardcoding credentials directly in your code is a bad practice, as they may easily leak. To mitigate this issue, you can use one of the data security design patterns from Chapter 7.

## Summary

When you started reading this chapter, you probably considered data ingestion to be a necessary but not technically challenging step. Hopefully, this chapter has proven to you that the opposite is correct.

You've learned that even this simple operation of moving data from one place to another comes with some challenges. Without a Readiness Marker, you may ingest incomplete data as a customer or get bad press among users if you are the provider. Without the Compactor

pattern, your virtually unlimited lakehouse will become a performance bottleneck pretty fast just because of API calls.

Also, even though you've learned about data ingestion relatively early in this book, remember this step is not reserved for the front doors of your system or for simple EL pipelines. Most of the patterns discussed here are great candidates for the extract step in the ETL and ELT pipelines, so you may even use them in more business-oriented jobs. That's another reason to not underestimate their importance.

Finally, I have good news and bad news. The good news is that you now have a list of templates you can apply to ingest the data in both pure technical and business contexts. The bad news is that it's just the beginning. After ingesting the data, you will process it, and there, too, things can go wrong. Hopefully, the next chapter will give you some other recipes for building more efficient data engineering systems!

1 You can learn more about it in the section on the Proxy pattern in Chapter 4.

2 Besides Kafka Connect, Debezium has two other implementations called Debezium Embedded Engine and Debezium Server.

3 Explaining MirrorMaker in depth is beyond the scope of this book. If you are interested in more details, the official documentation covers it extensively.

4 There's a detailed explanation of the problem of having small files in Chapters 4 and 5 of The Cloud Data Lake by Rukmani Gopalan (O'Reilly, 2023).