O'REILLY®

Data Engineering Design Patterns

Recipes for Solving the Most Common Data Engineering Problems

Bartosz Konieczny

"This book is the seminal work for the future of data engineering design patterns and should be required reading for any data professional. It is as important to the future of the profession as the Gang of Four's *Design Patterns* was for software design."

Scott Haines, coauthor, *Delta Lake: The Definitive Guide*

"Data engineering often feels like solving the same problems over and over. Bartosz Konieczny changes that with this book. Covering everything from idempotency to error handling and data observability, this is the definitive guide to building resilient data pipelines with reusable, proven design patterns."

Adi Polak, director, Confluent

Data Engineering Design Patterns

Data projects are an intrinsic part of an organization's technical ecosystem, but data engineers in many companies continue to work on problems that others have already solved. This hands-on guide shows you how to provide valuable data by focusing on various aspects of data engineering, including data ingestion, data quality, idempotency, and more.

Author Bartosz Konieczny guides you through the process of building reliable end-to-end data engineering projects, from data ingestion to data observability, focusing on data engineering design patterns that solve common business problems in a secure and storage-optimized manner. Each pattern includes a user-facing description of the problem, solutions, and consequences that place the pattern into the context of real-life scenarios.

Throughout this journey, you'll use open source data tools and public cloud services to apply each pattern. You'll learn:

- Challenges data engineers face and their impact on data systems
- How these challenges relate to data system components
- Useful applications of data engineering patterns
- How to identify and fix issues with your current data components
- Technology-agnostic solutions to new and existing data projects, with open source implementation examples

Bartosz Konieczny is a freelance data engineer who's been coding since 2010. He's held various senior hands-on positions that allowed him to work on many data engineering problems in batch and stream processing.

DATA

US $79.99 CAN $99.99
ISBN: 978-1-098-16581-9

O'REILLY®

Praise for *Data Engineering Design Patterns*

The book you now have in your hands is the seminal work for the future of data engineering design patterns. This should be required reading for any data professional, and is as important to the future of the profession as the Gang of Four's design patterns were for software design.

—*Scott Haines, coauthor,* Delta Lake: The Definitive Guide

Data engineering often feels like solving the same problems over and over—Bartosz Konieczny changes that with this book. Covering everything from idempotency to error handing and data observability, this is the definitive guide to building resilient data pipelines with reusable, proven design patterns.

—*Adi Polak, director at Confluent*

Bartosz has made a great contribution to drive data engineering forward! Data engineering is the technical backbone on which the leading tech companies build their dominance, and this knowledge needs to spread beyond the technical elite. *Data Engineering Design Patterns* is a great step in that direction. It captures years of experience crafting solutions to common data engineering challenges. It gives names and concise descriptions to recurring architectural patterns that are folklore among data engineering veterans from the Hadoop age, and can now spread to a wider audience.

—*Lars Albertsson, data engineering entrepreneur*

Stoked to see that some of the data engineering principles I've advocated for in the past—like immutability, deterministic transformations, and idempotency—are not only taking root but getting expanded upon and developed to a whole new level in this book. A great resource for data engineers looking to build reliable, scalable pipelines.

—*Maxime Beauchemin, original creator of Apache Airflow and Superset*

Data engineering suffers from a glut of complexity due to an ongoing proliferation of languages, frameworks and tools. This book provides clear roadmaps for solving data engineering problems regardless of the underlying technology being applied.

—Matt Housely, coauthor, Fundamentals of Data Engineering

A must-read for data engineers and architects, this book distills complex data engineering challenges into actionable, well-structured design patterns that empower teams to build scalable, maintainable data systems with confidence.

—Keith Mascarenhas, data engineering leader

A must-read for any data professional, this book transforms complex data engineering challenges into practical, actionable patterns. Offering a blueprint for mastering the art of building resilient data pipelines with ease, this is an indispensable guide that bridges the gap between theory and practice in data engineering.

—Lipi Patnaik, senior software developer, Zeta

This book is an invaluable resource for data engineers, offering practical patterns that simplify building robust, scalable data systems. An essential read for mastering the art of data pipeline design.

—William Jamir Silva, senior software engineer, Adjust GmbH

This book has to be one of the best data engineering design patterns books I've come across. The content is easy to understand and grasp, and the author has made every effort to cover the most critical and commonly used design patterns. Every design pattern problem and solution reminded me of challenges and solutions I came up with at work. This book is a must-read for data engineers looking to level up in their workplace.

—Rahul Arulkumaran, Foundry Digital

Data Engineering Design Patterns

Recipes for Solving the Most Common Data Engineering Problems

Bartosz Konieczny

O'REILLY®

Data Engineering Design Patterns
by Bartosz Konieczny

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Michele Cronin
Production Editor: Jonathon Owen
Copyeditor: Doug McNair
Proofreader: Piper Content Partners
Indexer: BIM Creatives, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

April 2025: First Edition

Revision History for the First Edition
2025-04-14: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098165819* for release details.

978-1-098-16581-9

[LSI]

Table of Contents

Preface

As a data engineer coming from the software engineering world, design patterns have always accompanied me on my journey. Adapter design pattern helped me write a backend with pluggable I/O abstractions, Template Method design pattern let me write an easily adaptable business logic, and thanks to the Builder design pattern, I could set up an easily maintainable unit tests layer. Having these great experiences in mind, I have been looking for similar standardized solutions since my first day in the data engineering space.

Over time, in each new project, I found something that was similar to previous projects. By connecting these dots I first completed a list of data engineering patterns for cloud services.[1] Meantime, I have been continuing to enrich my data engineering design patterns list, despite working on different business domains and with different technologies.

That's how, by summer 2023, I ended up with a quite solid list of data engineering design patterns that I included in the proposal for this book—which, since you are holding the book in your hands, was accepted. I hope the book will add a missing standardization piece each data engineer can rely on to identify a problem, its solution, and warning points, and I also hope it will help data engineers work with the data engineering tools of tomorrow.

1 I was very into the topic of cloud and data engineering between 2020 and 2022. Following this interest, I self-published the book back in 2022. The first chapter is available for free on my site.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

This element signifies a general note.

This element indicates a warning or caution.

The Structure of This Book

This book follows the workflow of a classical data engineering project that starts with data ingestion and ends with day-to-day monitoring. The steps of the project correspond to the main chapters, so you can easily identify the stage to which each pattern in a chapter applies.

Additionally, each chapter has a two-level structure, with the levels being design pattern categories and the design patterns themselves. Why this two-level organization? First, a given data engineering problem can have at least two possible solutions, and it wouldn't be possible to logically group them without having this first level of design pattern categories. Second, data engineering design patterns have their own names that sometimes sound mysterious, and design pattern categories provide extra application context that helps you know where to apply a particular pattern without requiring you to delve into details.

Finally, for each pattern, you'll find the following subsections:

Problem
: This subsection provides a real-world example of when you can use the pattern.

Solution
: This subsection describes the pattern in more technical detail. Usually, it starts with a high-level explanation followed by the technical implementation model.

Consequences
: Patterns have their trade-offs, and this section explains what you should look for before implementing them. Whenever possible, each gotcha is completed with a mitigation solution.

Examples
: In this final part, you'll find code snippets explaining how to use the pattern within the modern data engineering tools. Unfortunately, it's not technically possible to share the pattern's implementation in all existing data tools, so this book uses popular open source projects (Apache Spark, Apache Flink, Apache Airflow, PostgreSQL, and Delta Lake). Occasionally, the implementation extends the scope to the managed services in the public cloud. The code snippets are written in Python, SQL, and sometimes Scala or Java if the Python implementation is not available.

At the end of this book, you will find a table summarizing all the described patterns. Also, the book has a GitHub repository that includes a glossary of terms that should give you the definitions of the most frequently used acronyms in the book.

How to Use This Book

It depends on your experience. If you've just started your data engineering journey, you probably haven't seen most of the presented problems yet. In that case, reading this book cover to cover is a good approach.

On the other hand, if you already have some significant experience and some of the problems look familiar, reading the book start to finish may not be the best idea. Instead, you can start by picking the patterns you haven't heard about. To complete the picture, you can later go back to the patterns you know and see if you have implemented them in the same way.

Also, no matter what your level of experience may be, code snippets will help you put the theory into practice and better understand what the pattern implementations can look like in your projects.

Once you've read the book and played with the code snippets, you can consider it a reference book that's almost like a *cookbook* that you can consult whenever you're facing a new data problem to solve (and not because of the flan recipe from Chapter 1).

What Should I Know Prior to Reading This Book?

This book will not be a great resource if you have just started in data engineering and don't have any commercial experience. In my opinion, a minimum of six months of commercial experience with data engineering should help you grasp the ideas more easily. Other than that, the minimum technical knowledge you'll need to get the most out of this book is as follows:

- Familiarity with data engineering concepts, such as extract, transform, load (ETL), extract, load, transform (ELT), data warehousing, data ingestion, and data orchestration.
- Cloud awareness. Even though this book tends to favor open source technologies, there are places where cloud technology is more appropriate (e.g., data security). You don't have to be a cloud expert, but you should at least be able to understand the basics, such as what a managed service is.
- Hands-on experience with data processing logic in Java, Scala, Python, or SQL. Ideally, you have already deployed this logic in production.

If you feel like there are gaps in your knowledge of the required topics, you should be able to easily fill in the gaps by reading *Fundamentals of Data Engineering* by Joe Reis and Matt Housley (O'Reilly, 2022). The book provides a comprehensive overview of the data engineering space that will not only help you understand the content in this book but also better prepare you to deal with the challenges you will face in your day-to-day work.

Glossary and Code Examples

Code examples are available for download on my GitHub page. To run them, you'll need your favorite IDE and Docker installed with Docker Compose.

The GitHub repo has the same organizational structure as the book, meaning each chapter has its own directory where you'll find fully working examples of the patterns organized into subdirectories. Each directory has a dedicated *README.md* that will guide you through the demo.

Also, the GitHub repository includes a glossary with the most important terms referred to in the book. It's a complimentary resource that should help you recall some concepts if you haven't heard about them for a while.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Data Engineering Design Patterns* by Bartosz Konieczny (O'Reilly). Copyright 2025 Bartosz Konieczny, 978-1-098-16581-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

O'REILLY®

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/dataEngDesignPatterns*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Writing a book for O'Reilly has always been a dream. It turns out that I've made it come true, and although only my name is on the cover, my journey wouldn't have been possible without the support and inspiration I have received from my loved ones and my data community. That's why, before I let you discover the first patterns, I owe a few thank-yous!

First and foremost, thanks to my family. To Sylwia, my wife, who has always been supportive, even when I had serious doubts about writing, freelancing, and programming. To Maja and Arthur, my lovely children, who have reminded me there is a life beyond the screens. Thank you for making the whole book-writing process more enjoyable and less monotonous, even though you unexpectedly shortened some of my miracle morning routines. ;) Also a big thank-you to Jarek and Hania, my parents, whose trust helped me grow and try impossible things, like writing this book!

In addition to my family, I couldn't have written this book without the inspiring people I have met in the data community. To start with, thank you to Jacek Laskowski, who has been a mentor to me since I saw his StackOverflow answers and read his *The Internals of...* books while I was learning Apache Spark. I'm very glad I could meet you in person and learn even more!

Also, a big thank-you to Scott Haines, whose contributions to the data engineering space helped me grasp various stream processing and lakehouse aspects. And I probably never would have tried to write this book without the feedback you shared with me at DAIS 2022. Thank you for that chat and all your involvement in the data community that, I'm sure, has helped many more than just me!

If you're reading this book, it's also thanks to the involvement of tech reviewers. Keith Mascharenas, Laura Uzcategui, Lipi Deepaakshi Patnaik, Matthew Housley, Scott Haines, and William Jamir Silva, thank you so much for your questions and all the technical details that helped me grow as a writer and engineer. I'm sure the readers will now appreciate the book even more! At this time, I'd also like to thank Rahul Arulkumaran, Leszek Michalak, and Jonathan Roussot for their valuable feedback after the first Early Release versions of this book.

Besides the tech reviewers, Aaron Black and Michele Cronin from O'Reilly helped make this book come alive. Thank you, Aaron, for our discussions about the book and your guidance for the proposal writing. Michele, without your involvement, the book wouldn't be here, for sure. I'm very glad that we could collaborate, and I wish all other authors could be supported by a development editor like you!

Finally, I'd like to give a special mention to people who greatly contributed to my professional growth. To start with, thanks to Frank Pavageau, who taught me clean code principles before I knew there was a dedicated book on them! Also, a big thank-you to Jérôme Guibert, who taught me how to leverage automation to make my work easier and more enjoyable! And besides the project knowledge shared by Frank and Jérôme, I've been learning a lot from the data community. Big thank-yous go to Adi Polak, Holden Karau, Itai Yaffe, and Jungtaek Lim, who have inspired and taught me over the years through their content contributions to the data engineering world!

I wish that you, dear reader, can find such inspirational people around you.

CHAPTER 1

Introducing Data Engineering Design Patterns

Design patterns are well established in the software engineering space, but they have only recently begun getting traction in the data engineering world. Consequently, I owe you a few words of introduction and an explanation of what design patterns are in the context of data engineering.

What Are Design Patterns?

You may be surprised at how many times you rely on patterns in your daily life. Let's take a look at an example involving cooking and one of my favorite desserts, flan; if you like creamy desserts and haven't tried flan yet, I highly recommend it! When you want to prepare flan, you need to get all the ingredients and follow a list of preparation steps. As an outcome, you get a tasty dessert.

Why am I giving this cooking example as the introduction to a technical book about design patterns? It's because a recipe is a great representation of what a design pattern should be: a predefined and customizable template for solving a problem. How does this flan example apply to this definition?

- The ingredients and the list of preparation steps are the *predefined template*. They give you instructions but remain customizable, as you might decide to use brown sugar instead of white, for example.
- There can be a single use or many uses. The flan can be a dessert you'll share with family at teatime, or it can be a product that you'll sell to make a living. This is the *contextualization of a design pattern*. Design patterns always respond to a specific problem, which in this example is the problem of how to share a pleasant dessert with friends or how to produce the dessert to generate business revenue.

- You can decide to prepare this delicious dessert once or many times, if it happens to be your new favorite. For each new preparation, you won't reinvent the wheel. Chances are, you'll rely on the same successful recipe you tried before. That's the *reusability of the pattern*.
- But you must also be aware that preparing and eating flan has some implications for your life and health. If you prepare it every day, you'll maybe have less time for sports practice, and as a result, you might have some health issues in the long run. These are the *consequences of a pattern*.
- Finally, the recipe *saves you time* as it has been tested by many other people before. Additionally, it introduces a common dictionary that will make your life easier when discussing it with other people. Finding a recipe for flan is easier than finding one for caramel custard, which is a less popular name for flan.

Now, how does all this relate to data engineering? Again, let's use an example. You need to process a semi-structured dataset from a continuously running job. From time to time, you might be processing a record with a completely invalid format that will throw an exception and stop your job. But you don't want your whole job to fail because of that simple malformed record. This is our *contextualization*.

To solve this processing issue, you'll apply a set of best practices to your data processing logic, such as wrapping the risky transformation with a `try-catch` block to capture bad records and write them to another destination for analysis. That's the *predefined template*. These are the rules you can adapt to your specific use case. For example, you could decide not to send these bad records to another database and instead, simply count their occurrences.

Turns out that the example of handling erroneous records without breaking the pipeline has a specific name, *dead-lettering*. Now, if you encounter the same problem again, but in a slightly different context—maybe while working on an ELT pipeline and performing the transformations in a data warehouse directly—you can apply the same logic. That's the *reusability of the pattern*. The Dead-Letter pattern is one of the error management patterns detailed in Chapter 3.

However, you shouldn't follow the Dead-Letter pattern blindly. As with eating a flan every day, implementing the pattern has some *consequences* you should be aware of. Here, you add extra logic that adds some extra complexity to the codebase. You must be ready to accept this.

Finally, a data engineering design pattern represents a holistic picture of a solution for a given problem. It then *saves you time* and also introduces a common language that can greatly simplify discussions with your teammates or data engineers you have just met.

Yet More Design Patterns?

If you write software, you've heard about the Gang of Four's design patterns[1] and maybe even consider them as one of the clean code pillars. And now, you're probably asking yourself, aren't they enough for data engineering projects? Unfortunately, no.

Software design patterns are the recipes that you can use to keep an easily maintainable codebase. Since the patterns are standardized ways to represent a given concept, they're quickly understandable by any new person in the project.

For example, a pattern to avoid allocating unnecessary objects is *Singleton*. A newcomer who is aware of the design pattern can quickly identify it and understand its purpose in the code.

Writing maintainable code does indeed apply to data engineering projects, but it's not enough. Besides pure software aspects, you need to think about the data aspects, such as the aforementioned failure management, backfilling, idempotency, and data correctness aspects.

Common Data Engineering Patterns

The failed record management from the previous section is only one example of a data engineering design pattern. The others are part of the book, which follows a typical data flow from data ingestion to final data exposition with monitoring and alerting. Therefore, in the book you will find:

Chapter 2, "Data Ingestion Design Patterns"
: Bringing data to your system will always be the first technical step in your architecture. After all, it guarantees that you have data to work on!

Chapter 3, "Error Management Design Patterns"
: Errors, just like data, are an intrinsic part of data engineering. Errors may result from coding mistakes but may also come directly from data providers, who might not respect their initial engagements, for example, by sharing a dataset without required fields defined.

Chapter 4, "Idempotency Design Patterns"
: A natural consequence of errors is retries that are either automatic or manual. In case of an automatic retry, part or all of your data pipeline will rerun and probably try to rewrite already saved records. If you trigger a pipeline manually, you

1 The 23 software engineering design patterns introduced in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley Professional, 1994) are colloquially known as the Gang of Four design patterns because of the book's four authors.

start a backfill[2] that will execute one or more past pipelines. Thanks to idempotency, the multiple runs will generate unique outputs.

Chapter 5, "Data Value Design Patterns"
: Once you're able to deal with errors and retries, you can take care of the data and generate meaningful datasets for your business users. To do so, you may need to summarize the dataset or combine it with other data sources. All of this creates extra value that is important for your end users.

Chapter 6, "Data Flow Design Patterns"
: After providing a direct value to your consumers by exposing an enriched dataset, you can move to the next step and include the dataset generation as part of a data flow. The data flow defines how the pipeline that generates data value interacts with other data components in your organization.

Chapter 7, "Data Security Design Patterns", "Data Security Design Patterns"
: After the first six chapters, you should know how to bring the data to your system and how to enhance it to meet your business needs. However, you must also ensure that the dataset is securely stored and that it meets data privacy requirements.

Chapter 8, "Data Storage Design Patterns"
: Security is crucial, but leveraging data storage techniques to reduce the latency of processing the data is also important. That's why in this chapter you'll see how to leverage your data storage to improve the user experience.

Chapter 9, "Data Quality Design Patterns"
: The bad news is that even though you have implemented all the previous chapters, your data may still be irrelevant to your consumers if you don't get rid of data quality issues, or worse, if you're not even aware of them.

Chapter 10, "Data Observability Design Patterns"
: This is the last step in your journey, where you'll define various monitoring metrics that will be important to the data you work on. Alongside the data quality design patterns, the data observability design patterns help make your data trustworthy by alerting you whenever something bad is happening or is about to happen.

2 This is known as reprocessing. So as not to confuse you, from this point forward, we'll refer to any task processing past data as *backfilling*, whether the data has already been processed or not. Technically, there is a small difference between reprocessing and backfilling, which you can learn about in the glossary available on GitHub.

Case Study Used in This Book

The design patterns in this book are not tied to one specific business domain. However, understanding them without any business context would be hard, especially for less experienced readers. For that reason, you'll see each pattern introduced in the context of our case study project, which is a blog data analytics platform.

Our project follows common data practices and is divided into the layers presented in Figure 1-1.

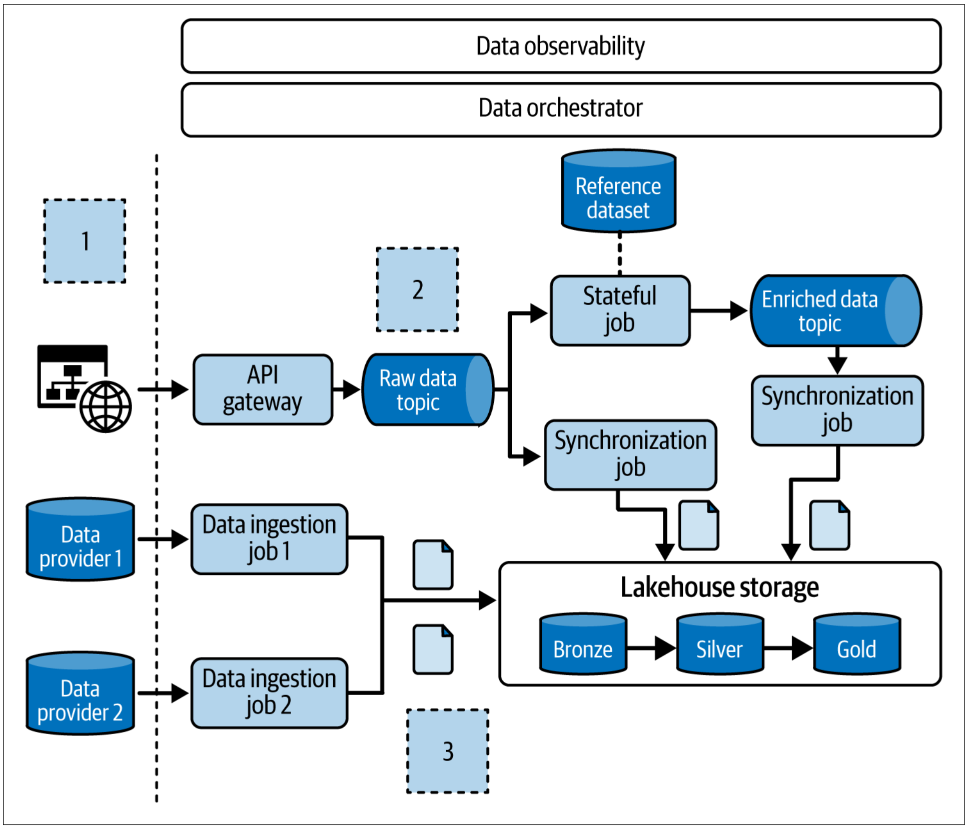


Figure 1-1. Blog analytics platform used in the case study

Figure 1-1 highlights the three most important parts of the project:

Online and offline data ingestion components
: The online part applies to the data generated by users interacting with the blogs hosted on our platform. The offline part, marked in the figure as "Data provider," applies to the static external or internal datasets such as referential datasets,

which are produced on a less regular schedule than the visit events (for example, once an hour).

The real-time layer
: This is where you can find streaming job processing events data from a streaming broker. The jobs here may be one of two types. The first is a business-facing job that generates data for the stakeholders, such as a real-time session aggregation. The second type is a technical job that is often a technical enabler for other business use cases. An example here would be data synchronization with the data at-rest storage for ad hoc querying.

The data organization layer
: This layer follows a now-common dataset structure that's based on the Medallion architecture[3] principle, in which a dataset may live in one of three different layers: Bronze, Silver, and Gold. Each layer applies to a different data maturity level. The Bronze layer stores data in its raw format, unaltered and probably with serious data quality issues. The Silver layer is responsible for cleansed and enriched datasets. Finally, the Gold layer exposes data in the format expected by the final users, such as data marts or reference datasets.

  Why are these three storage layers interesting in the context of this book? Each layer represents a different data maturity level, exactly like the design patterns presented here. The patterns impacting business value will mostly expose the data in the Gold layer, while the others will remain behind, in the Bronze or Silver layer. Problem statement sections for the patterns may reference those layers to help you better understand any issues you encounter.

The schema doesn't present any implementation details on purpose. Focusing on them could shift your focus to the technology instead of the universal pattern-based solutions that are the main topic of the book. But it doesn't mean you won't see any technical details in the next chapters. On the contrary! Each pattern has a dedicated Examples section where you will see different implementations of the presented pattern.

Summary

Now that you've read this chapter, you should understand not only that flan is a great creamy dessert but also that its recipe is a great analogy for the data engineering design patterns that you will discover in the next nine chapters. I know it's a lot, but with a cup of coffee or tea and your favorite dessert (why not flan!), it'll be an exciting learning journey!

3 You can learn more about the Medallion architecture in Chapter 4 of *Delta Lake: The Definitive Guide* by Denny Lee et al. (O'Reilly, 2024).

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.

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

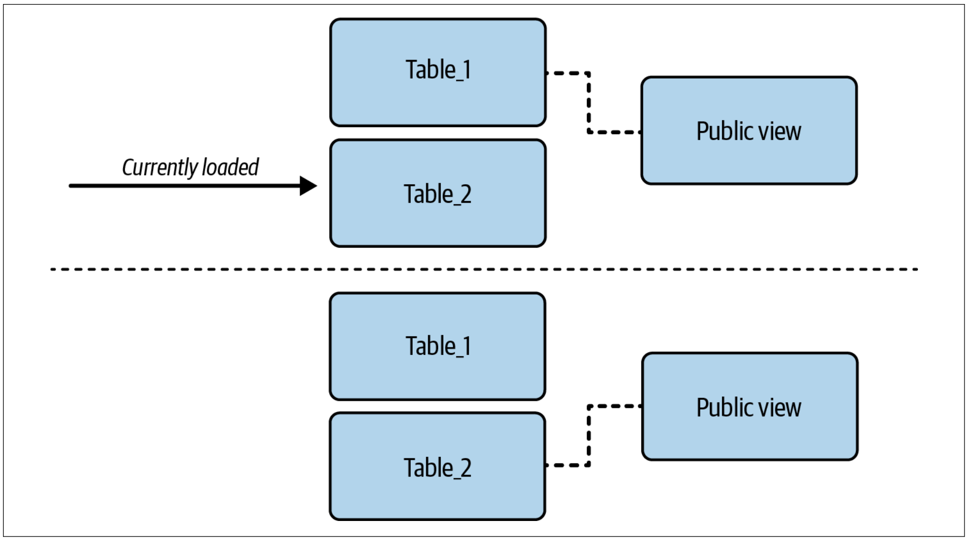


Figure 2-1. Single data exposition abstraction example with a database view

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" on page 51) 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.

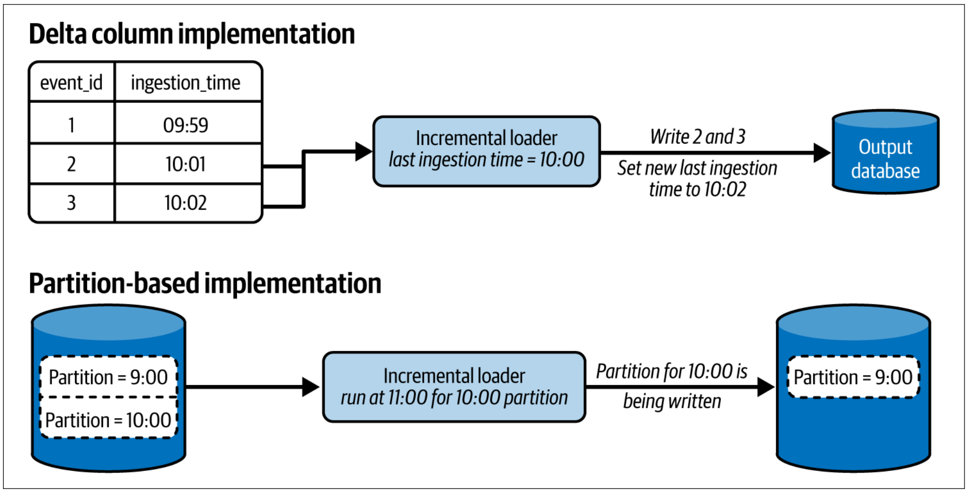

Delta column implementation
event_id
ingestion_time
1
09:59
2
10:01
3
10:02
Incremental loader
last ingestion time = 10:00
Write 2 and 3
Set new last ingestion time to 10:02
Output database
Partition-based implementation
Partition = 9:00
Partition = 10:00
Incremental loader
run at 11:00 for 10:00 partition
Partition for 10:00 is being written
Partition = 9:00

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 (`AwsGlueCatalogPartitionSensor`), 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.

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

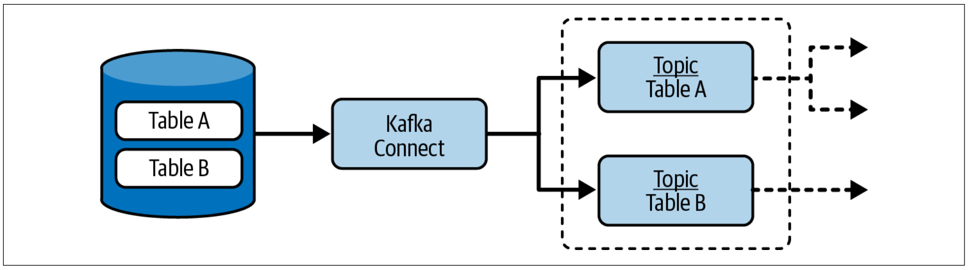


Figure 2-3. Debezium architecture in a nutshell, showing a Kafka Connect consumer leveraging the database commit logs and writing all matching datasets to dedicated Apache Kafka topics

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"
    }
  }
}
```

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.

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

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).

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" on page 51.

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 `File Sensor` (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:

1. 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.
2. 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.
3. 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.

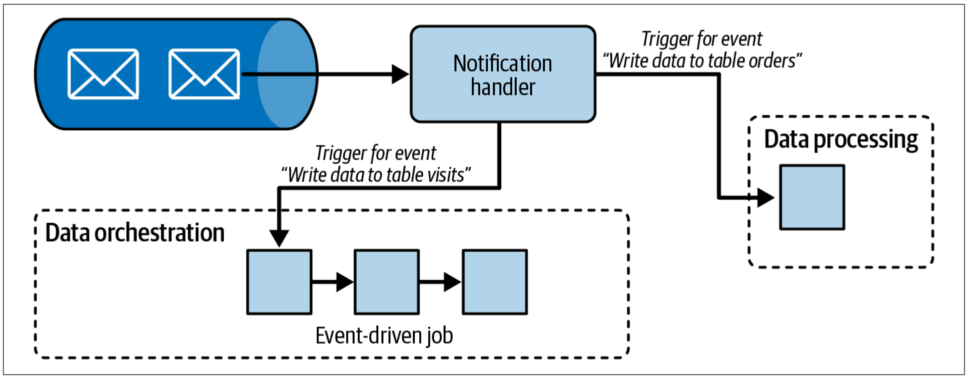

Notification handler
Trigger for event "Write data to table orders"
Data processing
Trigger for event "Write data to table visits"
Data orchestration
Event-driven job

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!

CHAPTER 3

Error Management Design Patterns

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

—Benjamin Franklin

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

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

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

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

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

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

Unprocessable Records

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

Pattern: Dead-Letter

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

Problem

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

Transient Versus Nontransient Errors

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

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

Solution

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

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

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

- Resiliency, so that you don't need to think about a dead-letter strategy for your dead-letter storage.
- Monitoring ease, because it's the key success factor of this implementation. After all, you want to know when your job starts dealing with erroneous records and, especially, how many of them have been written recently. By analyzing these two metrics you should be able to better understand whether the errors are only occasional issues or whether the whole system is going down.
- Writing performance, since writing the unprocessed records to an extra place will incur some cost in the overall job execution time.

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

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

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

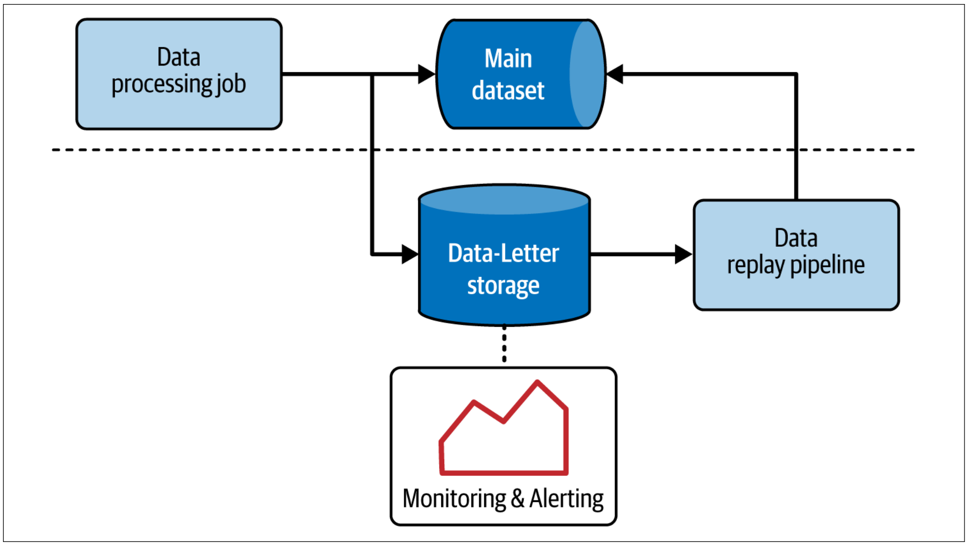


Figure 3-1. Components involved in the Dead-Letter pattern

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

Consequences

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

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

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

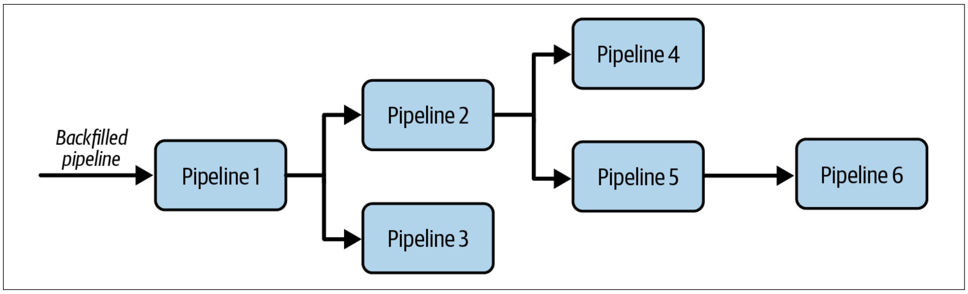


Figure 3-2. Snowball backfilling effect where backfilling Pipeline 1 triggers the same process for all downstream consumers

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

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

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

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

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

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

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

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

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

Examples

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

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

```
# source omitted for brevity

invalid_data_output: OutputTag = OutputTag('invalid_visits', Types.STRING())
visits: DataStream = data_source.map(MapJsonToReducedVisit(invalid_data_output),
  Types.STRING())
```

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

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

```
# MapJsonToReducedVisit snippet w/ reference to the side output
def map_rows(self, json_payload: str) -> str:
try:
  evt = json.loads(json_payload)
  evt_time = int(datetime.datetime.fromisoformat(evt['event_time']).
  yield json.dumps({'visit_id': evt['visit_id'], 'event_time': evt_time,
                    'page': evt['page']})
except Exception as e:
  yield self.invalid_data_output, _wrap_input_with_error(json_payload, e)

kafka_sink_valid_data: KafkaSink = ...
kafka_sink_invalid_data: KafkaSink = ...

visits.get_side_output(invalid_data_output).sink_to(kafka_sink_invalid_data)
visits.sink_to(kafka_sink_valid_data)
```

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

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

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

```
spark_session.sql('''
 SELECT type, full_name, version, name_with_version,
  WHEN (full_name IS NOT NULL OR version IS NOT NULL)
    AND name_with_version IS NULL THEN false ELSE true
  END AS is_valid
 FROM (SELECT type, full_name, version, CONCAT(full_name, version) AS name_w_version
FROM devices_to_load)''')
```

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

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

```
devices_to_load_with_validity_flag.persist()

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

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

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

Duplicated Records

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

Pattern: Windowed Deduplicator

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

Automatic Retries

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

Problem

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

Solution

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

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

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

Windows in Batch

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

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

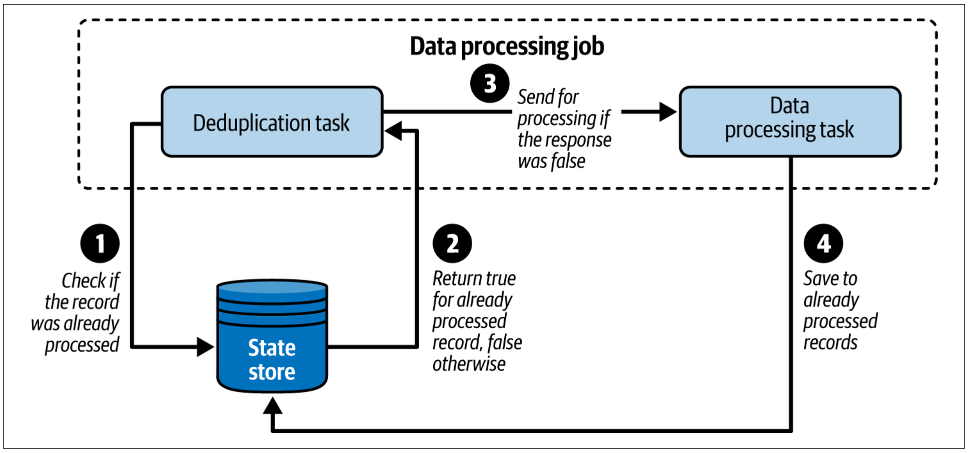

Data processing job
3
Send for processing if the response was false
Deduplication task
Data processing task
1
Check if the record was already processed
2
Return true for already processed record, false otherwise
4
Save to already processed records
State store

Figure 3-3. Windowed Deduplicator for a streaming job

State Stores

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

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

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

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

Consequences

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

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

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

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

Examples

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

Example 3-5. Deduplication with `dropDuplicates`

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

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

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

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

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

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

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

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

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

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

```
# ...part 2
.withWatermark("visit_time", "10 minutes")
.dropDuplicates(["visit_id", "visit_time"])
.drop("visit_time", "visit_id"))
```

Late Data

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

Pattern: Late Data Detector

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

Problem

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

Solution

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

Event and Processing Time

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

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

After defining the partition-based logic, you need to decide on an additional aggregation strategy that will calculate a single event time for all partitions to represent overall progress. Unlike in the previous step, where the MAX function is recommended to ensure monotonicity, for this global event time, you can opt to use the following:

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

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

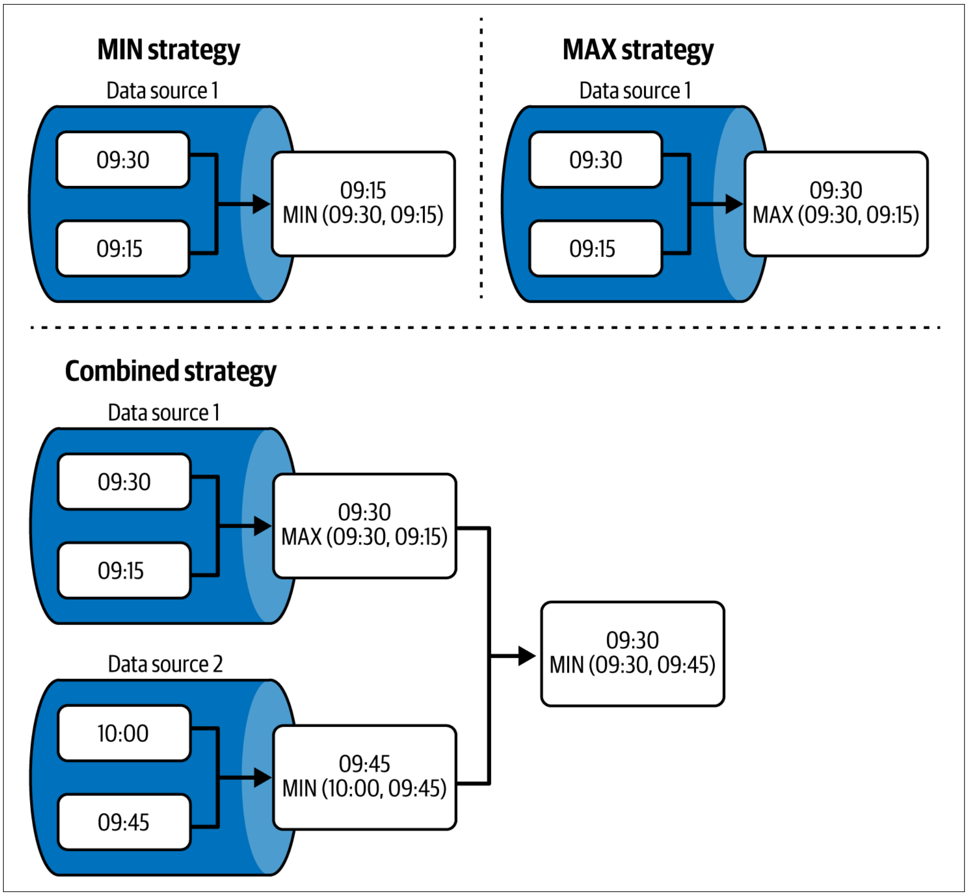


Figure 3-4. Different aggregation strategies applied to a job processing with a single partitioned data source and two partitioned data sources

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

To understand the watermark and the overall Late Data Detection pattern better, let's look at an example. Table 3-1 illustrates data flowing to a streaming system. As you'll notice, in the first row, there is no prior observation to define the watermark.

Consequently, the pipeline accepts all the records and defines a new output watermark. The output watermark, by the way, is the same as the watermark candidate here.

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

Table 3-1. Watermark of 30 minutes

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

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

Consequences

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

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

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

A stateful job accumulates events for a streaming job in a state store, as you discovered in the "Pattern: Windowed Deduplicator" on page 46. For example, it could count the number of visits for a user visiting a website. The challenge with this stateful accumulation is to know when to stop. Put differently, you should define when the accumulated state is complete. Often, you'll use the current watermark for that. Thus, by emitting each counter, you'll say, "For this user, there shouldn't be more new data."

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

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

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

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

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

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

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

Examples

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

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

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

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

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

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

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

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

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

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

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

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

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

class VisitLateDataProcessor(ProcessFunction):

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

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

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

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

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

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

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

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

Pattern: Static Late Data Integrator

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

Problem

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

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

Solution

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

Easy Solution for You, but Not for Others

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

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

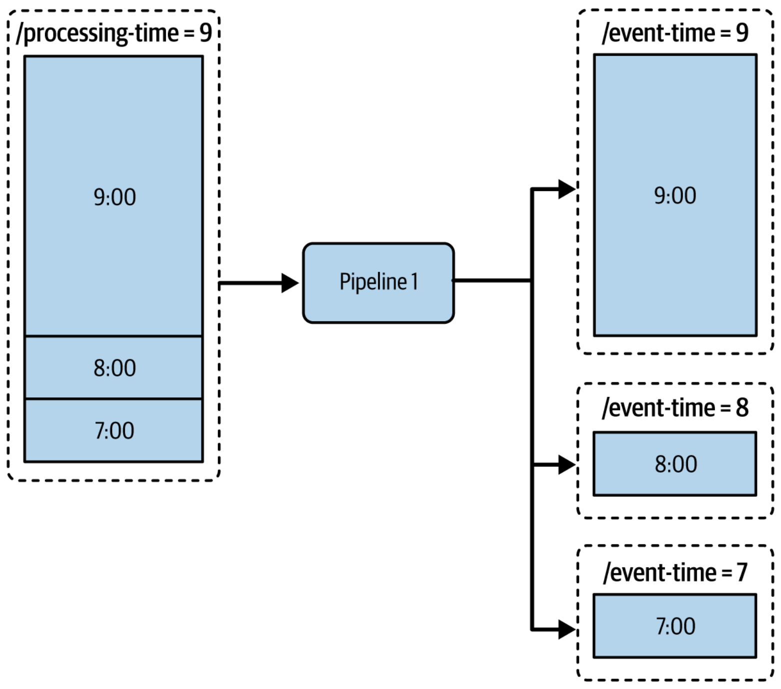


Figure 3-5. Shifting the late data problem

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

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

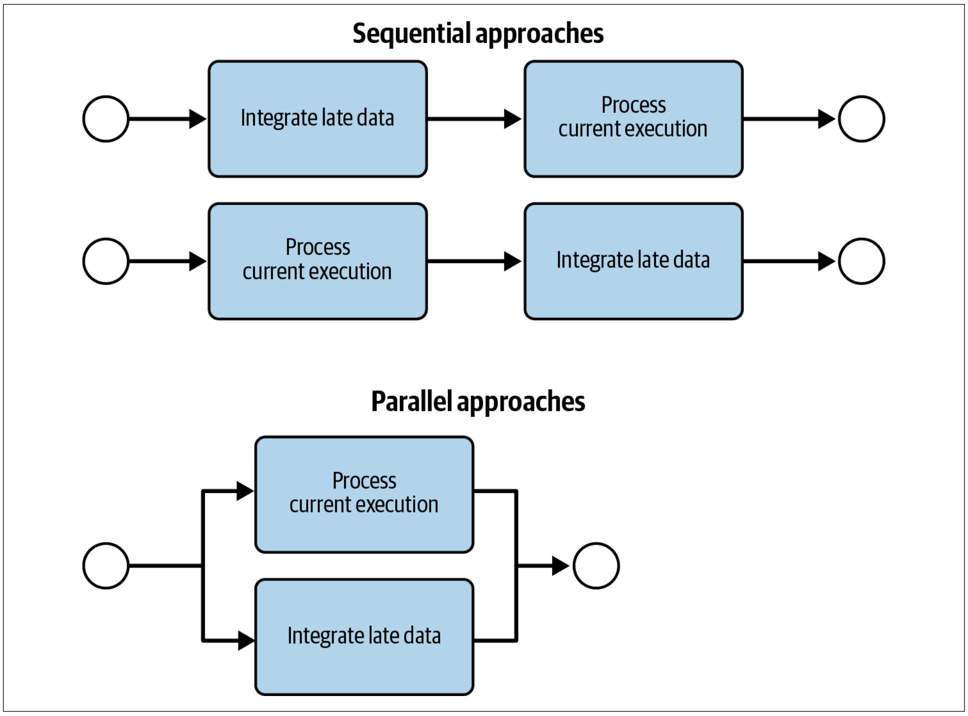


Figure 3-6. Strategies to include late data integration in pipelines

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

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

Consequences

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

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

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

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

Table 3-3. Overlapped backfilling examples

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

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

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

Figure 3-7 depicts both valid and invalid approaches.

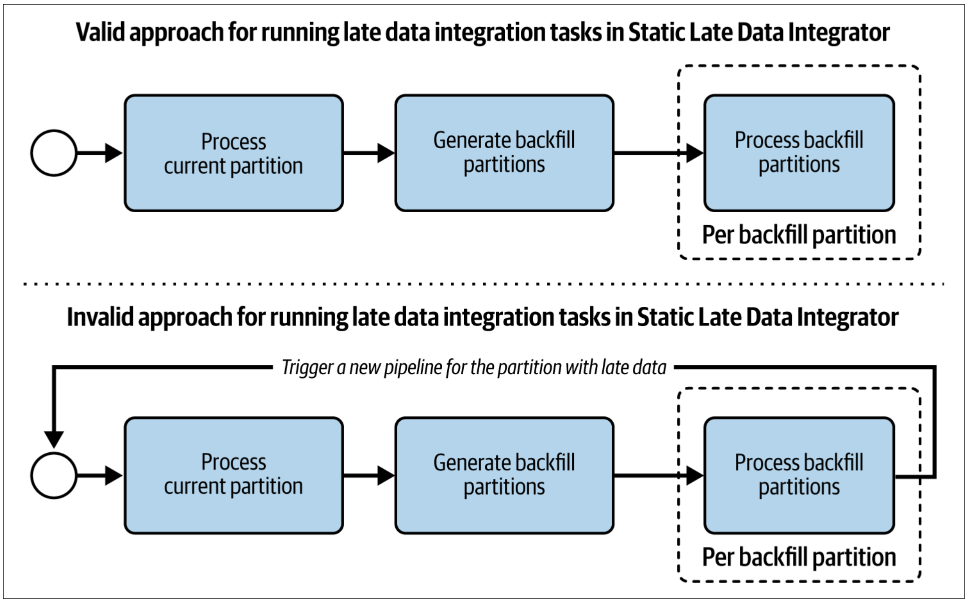

Valid approach for running late data integration tasks in Static Late Data Integrator
Process current partition
Generate backfill partitions
Process backfill partitions
Per backfill partition
Invalid approach for running late data integration tasks in Static Late Data Integrator
Trigger a new pipeline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

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

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

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

Examples

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

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

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

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

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

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

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

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

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

Example 3-14. Workflow with tasks created dynamically

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

Pattern: Dynamic Late Data Integrator

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

Problem

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

Solution

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

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

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

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

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

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

```sql
SELECT partition FROM state_table WHERE
`Last update time` > `Last processed time` AND `Partition` < `Processed partition`
```

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

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

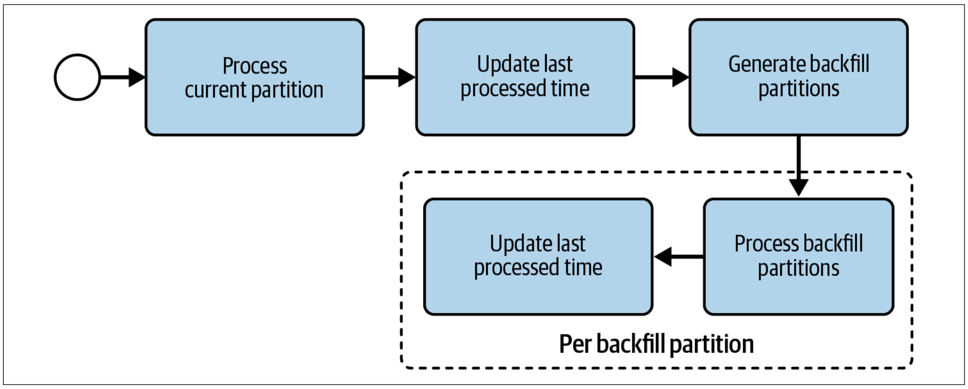

Process current partition
Update last processed time
Generate backfill partitions
Process backfill partitions
Update last processed time
Per backfill partition

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

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

Not Whole Partitions

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

Consequences

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

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

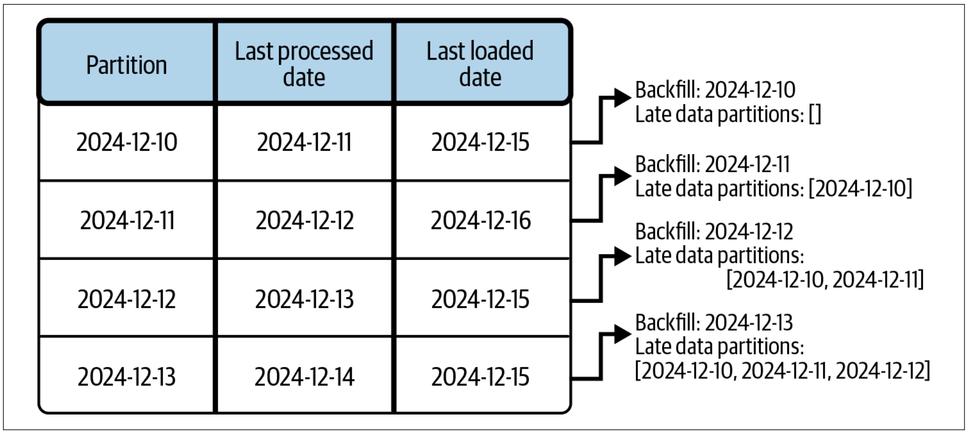


Figure 3-9. Late data integration concurrency problem

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

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

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

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

- Each pipeline needs to start with the task that updates the `Is processed` column of the currently processed partition. That way, you can avoid having the next execution generate the current partition as the one to backfill. Also, this task should run only if the execution of the previous run succeeded. In our example, this task for 2024-12-11 would start only if the same task successfully completed for the run of 2024-12-10.
- The task that generates the partitions to backfill should now also update all retrieved partitions as having been processed. In addition, it should run only if its previous execution succeeded. This dependency on the past runs helps avoid race conditions and triggering the same partitions in two different runs.

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

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

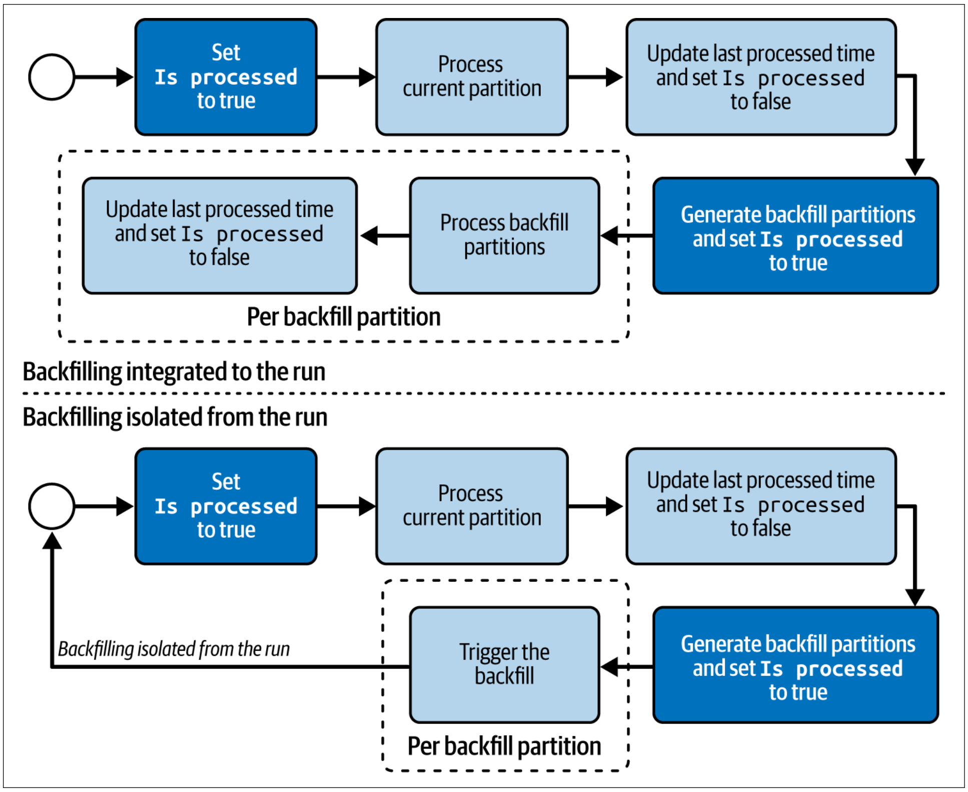

Set Is processed to true
Process current partition
Update last processed time and set Is processed to false
Generate backfill partitions and set Is processed to true
Process backfill partitions
Update last processed time and set Is processed to false
Per backfill partition
Backfilling integrated to the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

Figure 3-10. Late Data Integrator for concurrent pipelines

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

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

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

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

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

Examples

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

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

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

```scala
val deltaLog = DeltaLog.forTable(sparkSession, jobArguments.tableFullPath)
val partitionsChangeVersions: Iterator[(String, Long)] = deltaLog.getChanges(0)
 .flatMap {
 case (version, actions) => {
  val changedPartitionsInVersion: Set[String] = actions.map {
   case addFile: AddFile if addFile.dataChange =>
     Some(addFile.partitionValues.map(e => s"${e._1}=${e._2}").mkString("/"))
   case removeFile: RemoveFile if removeFile.dataChange =>
     Some(removeFile.partitionValues.map(e => s"${e._1}=${e._2}").mkString("/"))
   case _ => None}.filter(_.isDefined).map(_.get).toSet

  changedPartitionsInVersion.map(partition => (partition, version))
 }
}
val lastVersionForEachPartition: Map[String, Long] = partitionsChangeVersions
 .toSeq.groupBy(pair => pair._1).mapValues(pairs => pairs.map(_._2).max)
lastVersionForEachPartition
```

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

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

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

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

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

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

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

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

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

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

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

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

Filtering

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

Pattern: Filter Interceptor

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

Problem

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

Solution

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

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

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

Stay Pragmatic

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

Consequences

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

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

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

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

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

Examples

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

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

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

```
@dataclasses.dataclass
class FilterWithAccumulator:
 name: str
 filter: Callable[[Any], bool]
 accumulator: Accumulator[int]

filters_with_accumulators = {
 'type': [
  FilterWithAccumulator('type is null', lambda device: device['type'] is not None,
   spark_context.accumulator(0)),
  FilterWithAccumulator('type is too short (1 chars or less)',
    lambda device: len(device['type']) > 1, spark_context.accumulator(0))
  ],
  # ...
}
```

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

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

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

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

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

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

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

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

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

Example 3-24. SQL queries for Filter Interceptor

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


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

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

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

Fault Tolerance

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

Pattern: Checkpointer

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

Problem

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

Solution

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

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

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

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

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

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

Consequences

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

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

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

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

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

Delivery Modes

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

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

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

Examples

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

Example 3-25. Checkpoints in Apache Spark Structured Streaming

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

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

Example 3-26. Checkpointed metadata for fault tolerance

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

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

Example 3-27 shows checkpoint configuration in Apache Flink.

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

```
checkpoint_interval_30_sec = 30000
env.enable_checkpointing(checkpoint_interval_30_sec, mode=EXACTLY_ONCE)

(env.get_checkpoint_config().enable_externalized_checkpoints(RETAIN_ON_CANCELLATION))
```

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

Asynchronous Progress Tracking

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

Summary

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

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

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

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

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.

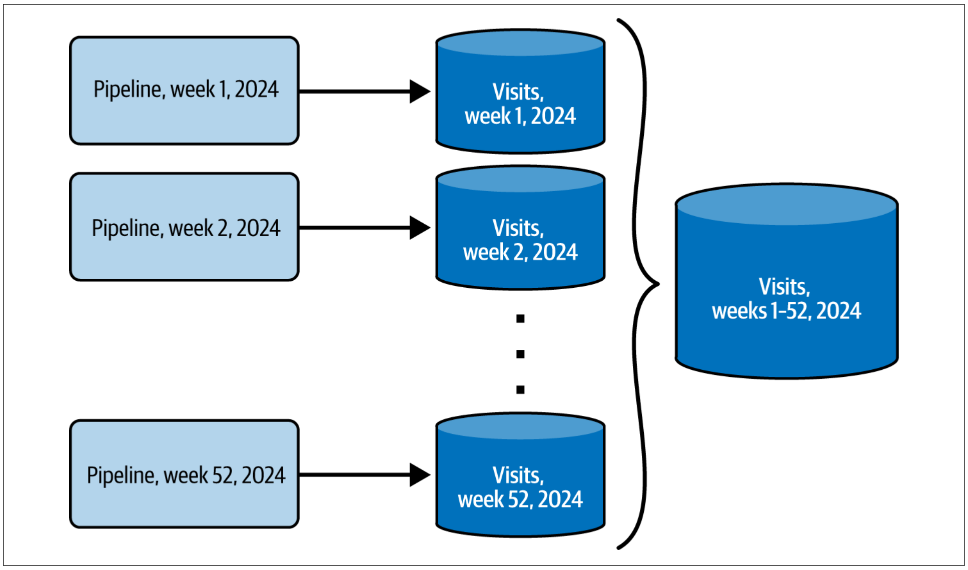


Figure 4-1. Physically isolated dataset in weekly tables and a common data exposition view for all the weeks in each year

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.

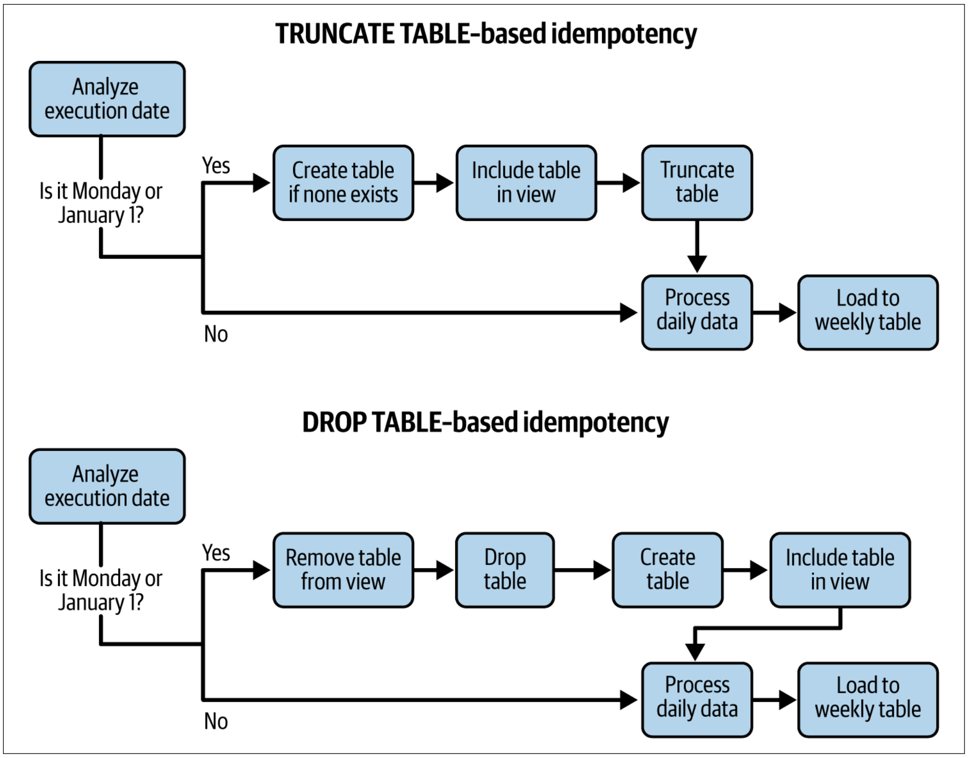


Figure 4-2. Fast Metadata Cleaner pattern example on top of weekly tables for `TRUNCATE TABLE` and `DROP TABLE` commands

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.

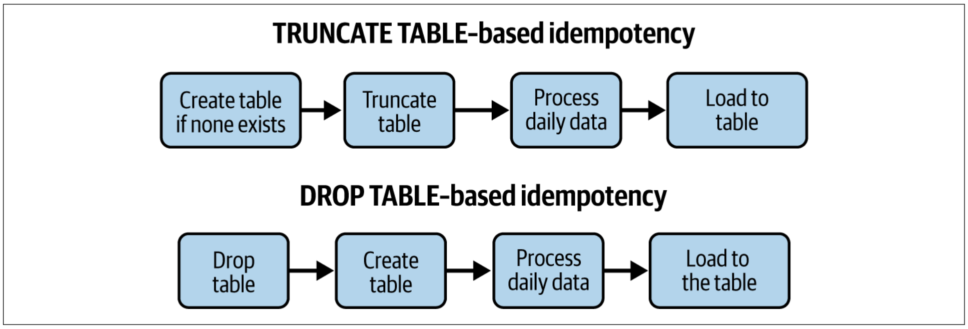


Figure 4-3. The Fast Metadata Cleaner and a fully loaded table

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

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.

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" on page 80) 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.

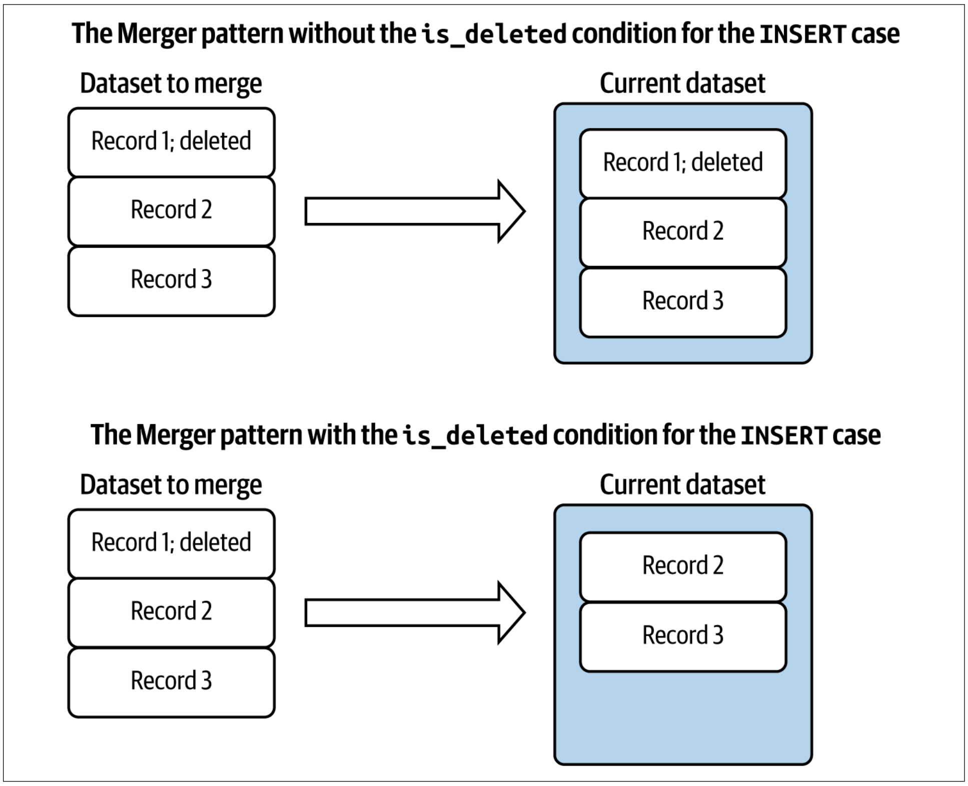


Figure 4-4. What happens during the first run of the Merger pattern with different `is_deleted` conditions for the `INSERT` case

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.

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

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

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.

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

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

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.

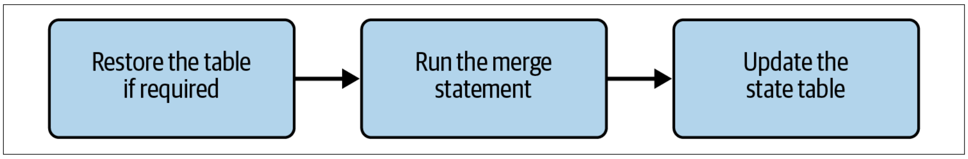


Figure 4-5. The workflow of the Stateful Merger pattern

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.

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

Execution time	Table version
08:00	4
09:00	5
10:00	6

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:
    a. 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.
    b. 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.

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

Execution time	Version
2024-10-05	1
2024-10-06	2
2024-10-07	3

Execution time	Version
2024-10-08	4

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.

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

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

  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.

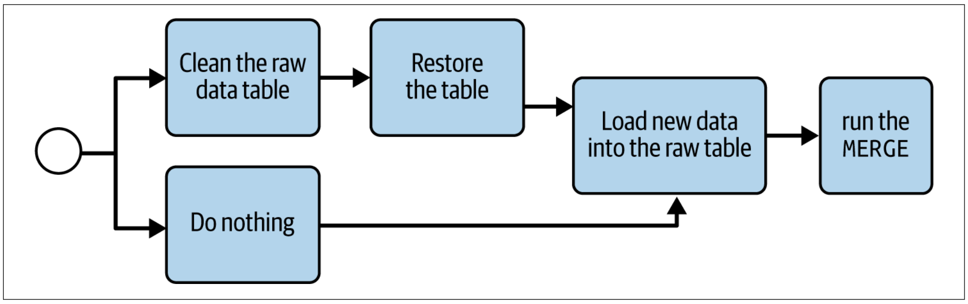


Figure 4-6. The Stateful Merger pattern adapted to a data store without versioning capabilities

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.

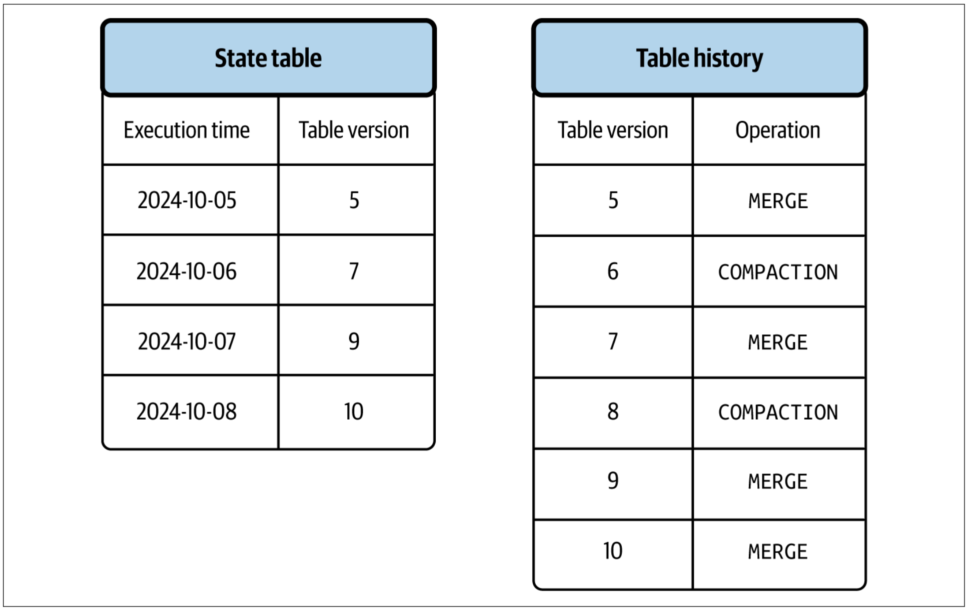
Compaction doesn't overwrite the data but only combines smaller files into bigger ones. But despite this no-data action, it also creates a new version of the table. As a result, if you always use the previous version from the state table in the restore action, you will miss the operations made between 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" on page 5. 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.

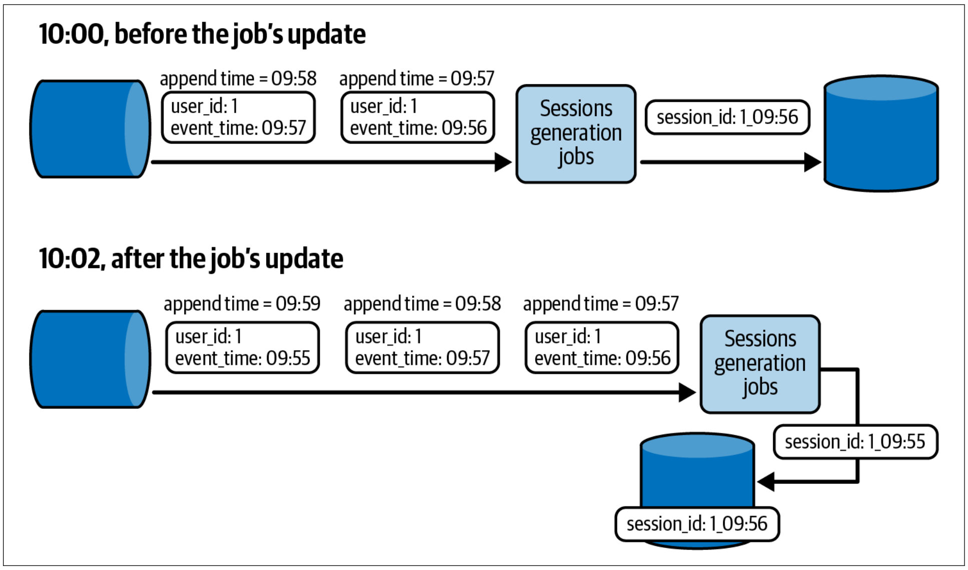


Figure 4-8. Late data impact on key generation; the late record for 09:55 creates a new session for the job restarted after an update

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.

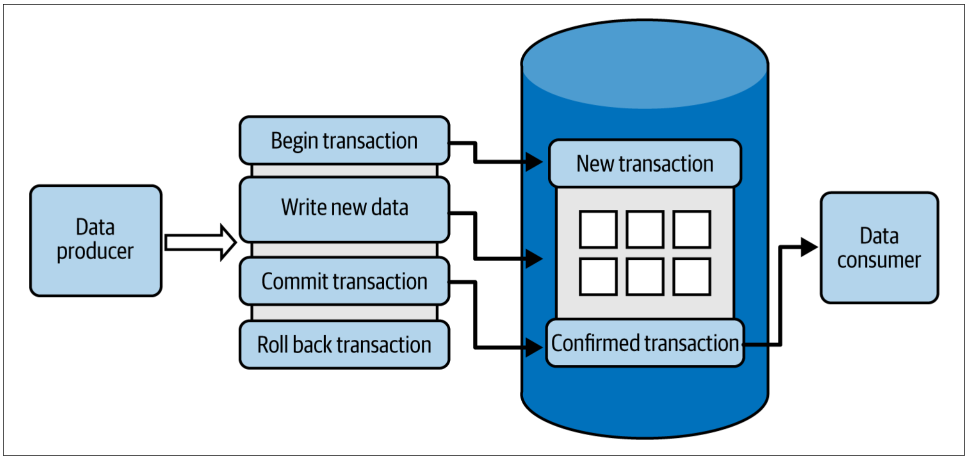


Figure 4-9. High-level view of a producer using the Transactional Writer pattern, in which the data is available to the consumer only after the commit step

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" on page 108.
- 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.

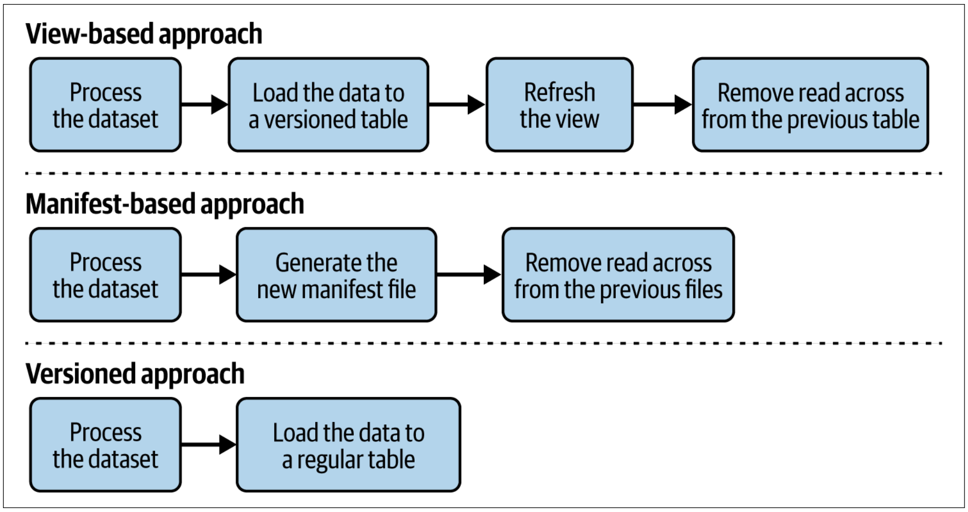

View-based approach
Process the dataset
Load the data to a versioned table
Refresh the view
Remove read across from the previous table
Manifest-based approach
Process the dataset
Generate the new manifest file
Remove read across from the previous files
Versioned approach
Process the dataset
Load the data to a regular table

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" on page 89. 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" on page 110 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!