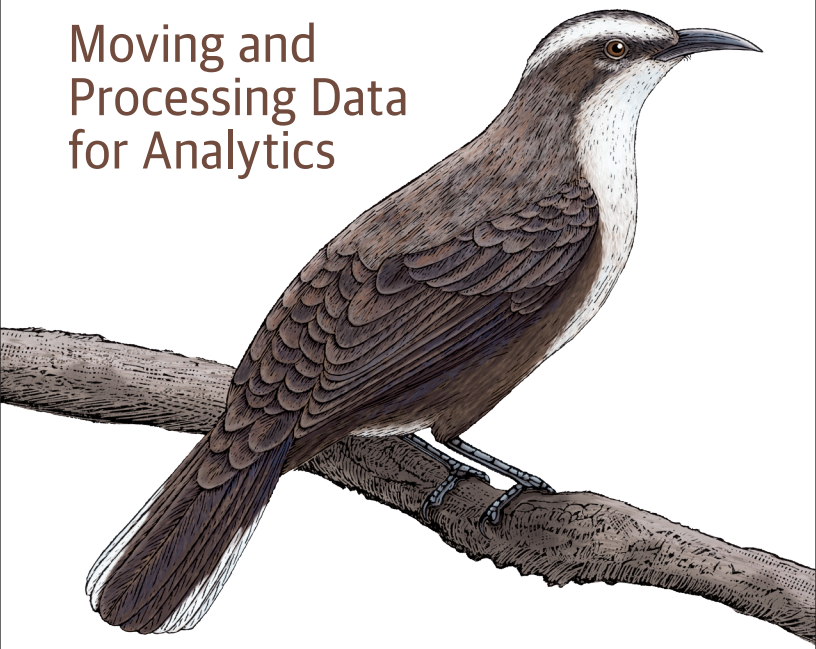
O'REILLY®

Data Pipelines Pocket Reference

Moving and Processing Data for Analytics



James Densmore

O'REILLY®

Data Pipeline Pocket Reference

Data pipelines are the foundation for success in data analytics. Moving data from numerous diverse sources and transforming it to provide context is the difference between having data and actually gaining value from it. This pocket reference defines data pipelines and explains how they work in today's modern data stack.

You'll learn common considerations and key decision points when implementing pipelines, such as batch versus streaming data ingestion and build versus buy. This book addresses the most common decisions made by data professionals and discusses foundational concepts that apply to open source frameworks, commercial products, and homegrown solutions.

- What a data pipeline is and how it works
- How data is moved and processed on modern data infrastructure, including cloud platforms
- Common tools and products used by data engineers to build pipelines
- How pipelines support analytics and reporting needs
- Considerations for pipeline maintenance, testing, and alerting

DATA

oreilly.com
Twitter: @oreillymedia

US $24.99 CAN $32.99
ISBN: 978-1-492-08783-0

Data Pipelines Pocket Reference

Moving and Processing Data for Analytics

James Densmore

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

TRS

Table of Contents

Preface

Data pipelines are the foundation for success in data analytics and machine learning. Moving data from numerous, diverse sources and processing it to provide context is the difference between having data and getting value from it.

I've worked as a data analyst, data engineer, and leader in the data analytics field for more than 10 years. In that time, I've seen rapid change and growth in the field. The emergence of cloud infrastructure, and cloud data warehouses in particular, has created an opportunity to rethink the way data pipelines are designed and implemented.

This book describes what I believe are the foundations and best practices of building data pipelines in the modern era. I base my opinions and observations on my own experience as well as those of industry leaders who I know and follow.

My goal is for this book to serve as a blueprint as well as a reference. While your needs are specific to your organization and the problems you've set out to solve, I've found success with variations of these foundations many times over. I hope you find it a valuable resource in your journey to building and maintaining data pipelines that power your data organization.

Who This Book Is For

This book's primary audience is current and aspiring data engineers as well as analytics team members who want to understand what data pipelines are and how they are implemented. Their job titles include data engineers, technical leads, data warehouse engineers, analytics engineers, business intelligence engineers, and director/VP-level analytics leaders.

I assume that you have a basic understanding of data warehousing concepts. To implement the examples discussed, you should be comfortable with SQL databases, REST APIs, and JSON. You should be proficient in a scripting language, such as Python. Basic knowledge of the Linux command line and at least one cloud computing platform is ideal as well.

All code samples are written in Python and SQL and make use of many open source libraries. I use Amazon Web Services (AWS) to demonstrate the techniques described in the book, and AWS services are used in many of the code samples. When possible, I note similar services on other major cloud providers such as Microsoft Azure and Google Cloud Platform (GCP). All code samples can be modified for the cloud provider of your choice, as well as for on-premises use.

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.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://oreil.ly/datapipelinescode*.

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

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 Pipelines Pocket Reference* by James Densmore (O'Reilly). Copyright 2021 James Densmore, 978-1-492-08783-0."

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

O'Reilly Online Learning

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 *http://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
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

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/data-pipelines-pocket-ref*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

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

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Thank you to everyone at O'Reilly who helped make this book possible, especially Jessica Haberman and Corbin Collins. The invaluable feedback of three amazing technical reviewers, Joy Payton, Gordon Wong, and Scott Haines led to critical improvements throughout. Finally, thank you to my wife Amanda for her encouragement from the moment this book was proposed, as well as my dog Izzy for sitting by my side during countless hours of writing.

CHAPTER 1

Introduction to Data Pipelines

Behind every glossy dashboard, machine learning model, and business-changing insight is data. Not just raw data, but data collected from numerous sources that must be cleaned, processed, and combined to deliver value. The famous phrase "data is the new oil" has proven true. Just like oil, the value of data is in its potential after it's refined and delivered to the consumer. Also like oil, it takes efficient pipelines to deliver data through each stage of its value chain.

This Pocket Reference discusses what these data pipelines are and shows how they fit into a modern data ecosystem. It covers common considerations and key decision points when implementing pipelines, such as batch versus streaming data ingestion, building versus buying tooling, and more. Though it is not exclusive to a single language or platform, it addresses the most common decisions made by data professionals while discussing foundational concepts that apply to homegrown solutions, open source frameworks, and commercial products.

What Are Data Pipelines?

Data pipelines are sets of processes that move and transform data from various sources to a destination where new value can

be derived. They are the foundation of analytics, reporting, and machine learning capabilities.

The complexity of a data pipeline depends on the size, state, and structure of the source data as well as the needs of the analytics project. In their simplest form, pipelines may extract only data from one source such as a REST API and load to a destination such as a SQL table in a data warehouse. In practice, however, pipelines typically consist of multiple steps including data extraction, data preprocessing, data validation, and at times training or running a machine learning model before delivering data to its final destination. Pipelines often contain tasks from multiple systems and programming languages. What's more, data teams typically own and maintain numerous data pipelines that share dependencies and must be coordinated. Figure 1-1 illustrates a simple pipeline.

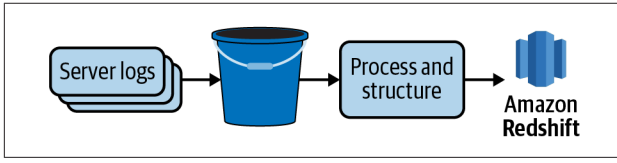


Figure 1-1. A simple pipeline that loads server log data into an S3 Bucket, does some basic processing and structuring, and loads the results into an Amazon Redshift database.

Who Builds Data Pipelines?

With the popularization of cloud computing and software as a service (SaaS), the number of data sources organizations need to make sense of has exploded. At the same time, the demand for data to feed machine learning models, data science research, and time-sensitive insights is higher than ever. To keep up, *data engineering* has emerged as a key role on analytics teams. *Data engineers* specialize in building and maintaining the data pipelines that underpin the analytics ecosystem.

A data engineer's purpose isn't simply to load data into a data warehouse. Data engineers work closely with data scientists

and analysts to understand what will be done with the data and help bring their needs into a scalable production state.

Data engineers take pride in ensuring the validity and timeliness of the data they deliver. That means testing, alerting, and creating contingency plans for when something goes wrong. And yes, something will eventually go wrong!

The specific skills of a data engineer depend somewhat on the tech stack their organization uses. However, there are some common skills that all good data engineers possess.

SQL and Data Warehousing Fundamentals

Data engineers need to know how to query databases, and SQL is the universal language to do so. Experienced data engineers know how to write high-performance SQL and understand the fundamentals of data warehousing and data modeling. Even if a data team includes data warehousing specialists, a data engineer with warehousing fundamentals is a better partner and can fill more complex technical gaps that arise.

Python and/or Java

The language in which a data engineer is proficient will depend on the tech stack of their team, but either way a data engineer isn't going to get the job done with "no code" tools even if they have some good ones in their arsenal. Python and Java currently dominate in data engineering, but newcomers like Go are emerging.

Distributed Computing

Solving a problem that involves high data volume and a desire to process data quickly has led data engineers to work with *distributed computing* platforms. Distributed computing combines the power of multiple systems to efficiently store, process, and analyze high volumes of data.

One popular example of distributed computing in analytics is the Hadoop ecosystem, which includes distributed file storage via Hadoop Distributed File System (HDFS), processing via MapReduce, data analysis via Pig, and more. Apache Spark is another popular distributed processing framework, which is quickly surpassing Hadoop in popularity.

Though not all data pipelines require the use of distributed computing, data engineers need to know how and when to utilize such a framework.

Basic System Administration

A data engineer is expected to be proficient on the Linux command line and be able to perform tasks such as analyze application logs, schedule cron jobs, and troubleshoot firewall and other security settings. Even when working fully on a cloud provider such as AWS, Azure, or Google Cloud, they'll end up using those skills to get cloud services working together and data pipelines deployed.

A Goal-Oriented Mentality

A good data engineer doesn't just possess technical skills. They may not interface with stakeholders on a regular basis, but the analysts and data scientists on the team certainly will. The data engineer will make better architectural decisions if they're aware of the reason they're building a pipeline in the first place.

Why Build Data Pipelines?

In the same way that the tip of the iceberg is all that can be seen by a passing ship, the end product of the analytics workflow is all that the majority of an organization sees. Executives see dashboards and pristine charts. Marketing shares cleanly packaged insights on social media. Customer support optimizes the call center staffing based on the output of a predictive demand model.

What most people outside of analytics often fail to appreciate is that to generate what is seen, there's a complex machinery that is unseen. For every dashboard and insight that a data analyst generates and for each predictive model developed by a data scientist, there are data pipelines working behind the scenes. It's not uncommon for a single dashboard, or even a single metric, to be derived from data originating in multiple source systems. In addition, data pipelines do more than just extract data from sources and load them into simple database tables or flat files for analysts to use. Raw data is refined along the way to clean, structure, normalize, combine, aggregate, and at times anonymize or otherwise secure it. In other words, there's a lot more going on below the water line.

Supplying Data to Analysts and Data Scientists

Don't rely on data analysts and data scientists hunting for and procuring data on their own for each project that comes their way. The risks of acting on stale data, multiple sources of truth, and bogging down analytics talent in data acquisition are too great. Data pipelines ensure that the proper data is delivered so the rest of the analytics organization can focus their time on what they do best: delivering insights.

How Are Pipelines Built?

Along with data engineers, numerous tools to build and support data pipelines have emerged in recent years. Some are open source, some commercial, and some are homegrown. Some pipelines are written in Python, some in Java, some in another language, and some with no code at all.

Throughout this Pocket Reference I explore some of the most popular products and frameworks for building pipelines, as well as discuss how to determine which to use based on your organization's needs and constraints.

Though I do not cover all such products in depth, I do provide examples and sample code for some. All code in this book is written in Python and SQL. These are the most common, and in my opinion, the most accessible, languages for building data pipelines.

In addition, pipelines are not just built—they are monitored, maintained, and extended. Data engineers are tasked with not just delivering data once, but building pipelines and supporting infrastructure that deliver and process it reliably, securely, and on time. It's no small feat, but when it's done well, the value of an organization's data can truly be unlocked.

CHAPTER 2

A Modern Data Infrastructure

Before deciding on products and design for building pipelines, it's worth understanding what makes up a modern data stack. As with most things in technology, there's no single right way to design your analytics ecosystem or choose products and vendors. Regardless, there are some key needs and concepts that have become industry standard and set the stage for best practices in implementing pipelines.

Let's take a look at the key components of such an infrastructure as displayed in Figure 2-1. Future chapters explore how each component factors into the design and implementation of data pipelines.

Diversity of Data Sources

The majority of organizations have dozens, if not hundreds, of data sources that feed their analytics endeavors. Data sources vary across many dimensions covered in this section.

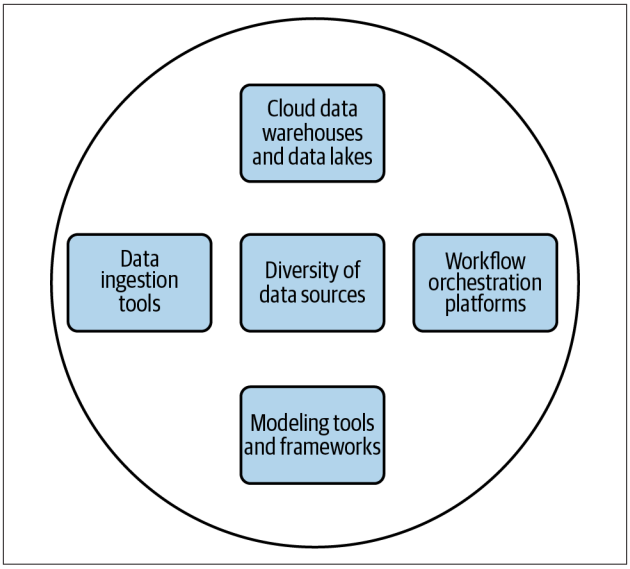


Figure 2-1. The key components of a modern data infrastructure.

Source System Ownership

It's typical for an analytics team to ingest data from source systems that are built and owned by the organization as well as from third-party tools and vendors. For example, an ecommerce company might store data from their shopping cart in a PostgreSQL (also known as Postgres) database behind their web app. They may also use a third-party web analytics tool such as Google Analytics to track usage on their website. The combination of the two data sources (illustrated in Figure 2-2) is required to get a full understanding of customer behavior leading up to a purchase. Thus, a data pipeline that ends with an analysis of such behavior starts with the ingestion of data from both sources.

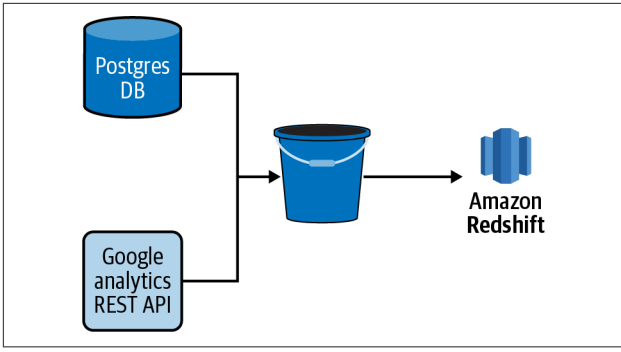


Figure 2-2. A simple pipeline with data from multiple sources loaded into an S3 bucket and then a Redshift database.

> **NOTE**
>
> The term *data ingestion* refers to extracting data from one source and loading it into another.

Understanding the ownership of source systems is important for several reasons. First, for third-party data sources you're likely limited as to what data you can access and how you can access it. Most vendors make a REST API available, but few will give you direct access to your data in the form of a SQL database. Even fewer will give you much in the way of customization of what data you can access and at what level of granularity.

Internally built systems present the analytics team with more opportunities to customize the data available as well as the method of access. However, they present other challenges as well. Were the systems built with consideration of data ingestion? Often the answer is no, which has implications ranging from the ingestion putting unintended load on the system to the inability to load data incrementally. If you're lucky, the

engineering team that owns the source system will have the time and willingness to work with you, but in the reality of resource constraints, you may find it's not dissimilar to working with an external vendor.

Ingestion Interface and Data Structure

Regardless of who owns the source data, how you get it and in what form is the first thing a data engineer will examine when building a new data ingestion. First, what is the interface to the data? Some of the most common include the following:

- A database behind an application, such as a Postgres or MySQL database
- A layer of abstraction on top of a system such as a REST API
- A stream processing platform such as Apache Kafka
- A shared network file system or cloud storage bucket containing logs, comma-separated value (CSV) files, and other flat files
- A data warehouse or data lake
- Data in HDFS or HBase database

In addition to the interface, the structure of the data will vary. Here are some common examples:

- JSON from a REST API
- Well-structured data from a MySQL database
- JSON within columns of a MySQL database table
- Semistructured log data
- CSV, fixed-width format (FWF), and other flat file formats
- JSON in flat files
- Stream output from Kafka

Each interface and data structure presents its own challenges and opportunities. Well-structured data is often easiest to work with, but it's usually structured in the interest of an application or website. Beyond the ingestion of the data, further steps in the pipeline will likely be necessary to clean and transform into a structure better suited for an analytics project.

Semistructured data such as JSON is increasingly common and has the advantage of the structure of attribute-value pairs and nesting of objects. However, unlike a relational database, there is no guarantee that each object in the same dataset will have the same structure. As you'll see later in this book, how one deals with missing or incomplete data in a pipeline is context dependent and increasingly necessary as the rigidity of the structure in data is reduced.

Unstructured data is common for some analytics endeavors. For example, Natural Language Processing (NLP) models require vast amounts of free text data to train and validate. Computer Vision (CV) projects require images and video content. Even less daunting projects such as scraping data from web pages have a need for free text data from the web in addition to the semistructured HTML markup of a web page.

Data Volume

Though data engineers and hiring managers alike enjoy bragging about petabyte-scale datasets, the reality is that most organizations value small datasets as much as large ones. In addition, it's common to ingest and model small and large datasets in tandem. Though the design decisions at each step in a pipeline must take data volume into consideration, high volume does not mean high value.

All that said, most organizations have at least one dataset that is key to both analytical needs as well as high volume. What's *high volume*? There's no easy definition, but as it pertains to pipelines, it's best to think in terms of a spectrum rather than a binary definition of *high-* and *low-* volume datasets.

NOTE

As you'll see throughout this book, there's as much danger in oversimplifying data ingestion and processing—with the result being long and inefficient runs—as there is in over-engineering pipeline tasks when the volume of data or complexity of the task is low.

Data Cleanliness and Validity

Just as there is great diversity in data sources, the quality of source data varies greatly. As the old saying goes, "garbage in, garbage out." It's important to understand the limitations and deficiencies of source data and address them in the appropriate sections of your pipelines.

There are many common characteristics of "messy data," including, but not limited to, the following:

- Duplicate or ambiguous records
- Orphaned records
- Incomplete or missing records
- Text encoding errors
- Inconsistent formats (for example, phone numbers with or without dashes)
- Mislabeled or unlabeled data

Of course, there are numerous others, as well as data validity issues specific to the context of the source system.

There's no magic bullet for ensuring data cleanliness and validity, but in a modern data ecosystem, there are key characteristics and approaches that we'll see throughout this book:

Assume the worst, expect the best
: Pristine datasets only exist in academic literature. Assume your input datasets will contain numerous validity and

consistency issues, but build pipelines that identify and cleanse data in the interest of clean output.

Clean and validate data in the system best suited to do so
: There are times when it's better to wait to clean data until later in a pipeline. For example, modern pipelines tend to follow an extract-load-transform (ELT) rather than extract-transform-load (ETL) approach for data warehousing (more in Chapter 3). It's sometimes optimal to load data into a data lake in a fairly raw form and to worry about structuring and cleaning later in the pipeline. In other words, use the right tool for the right job rather than rushing the cleaning and validation processes.

Validate often
: Even if you don't clean up data early in a pipeline, don't wait until the end of the pipeline to validate it. You'll have a much harder time determining where things went wrong. Conversely, don't validate once early in a pipeline and assume all will go well in subsequent steps. Chapter 8 digs deeper into validation.

Latency and Bandwidth of the Source System

The need to frequently extract high volumes of data from source systems is a common use case in a modern data stack. Doing so presents challenges, however. Data extraction steps in pipelines must contend with API rate limits, connection timeouts, slow downloads, and source system owners who are unhappy due to strain placed on their systems.

NOTE

As I'll discuss in Chapters 4 and 5 in more detail, data ingestion is the first step in most data pipelines. Understanding the characteristics of source systems and their data is thus the first step in designing pipelines and making decisions regarding infrastructure further downstream.

Cloud Data Warehouses and Data Lakes

Three things transformed the landscape of analytics and data warehousing over the last 10 years, and they're all related to the emergence of the major public cloud providers (Amazon, Google, and Microsoft):

- The ease of building and deploying data pipelines, data lakes, warehouses, and analytics processing in the cloud. No more waiting on IT departments and budget approval for large up-front costs. Managed services—databases in particular—have become mainstream.
- Continued drop-in storage costs in the cloud.
- The emergence of highly scalable, columnar databases, such as Amazon Redshift, Snowflake, and Google Big Query.

These changes breathed new life into data warehouses and introduced the concept of a data lake. Though Chapter 5 covers data warehouses and data lakes in more detail, it's worth briefly defining both now, in order to clarify their place in a modern data ecosystem.

A *data warehouse* is a database where data from different systems is stored and modeled to support analysis and other activities related to answering questions with it. Data in a data warehouse is structured and optimized for reporting and analysis queries.

A *data lake* is where data is stored, but without the structure or query optimization of a data warehouse. It will likely contain a high volume of data as well as a variety of data types. For example, a single data lake might contain a collection of blog posts stored as text files, flat file extracts from a relational database, and JSON objects containing events generated by sensors in an industrial system. It can even store structured data like a standard database, though it's not optimized for querying such data in the interest of reporting and analysis.

There is a place for both data warehouses and data lakes in the same data ecosystem, and data pipelines often move data between both.

Data Ingestion Tools

The need to ingest data from one system to another is common to nearly all data pipelines. As previously discussed in this chapter, data teams must contend with a diversity of data sources from which to ingest data from. Thankfully, a number of commercial and open source tools are available in a modern data infrastructure.

In this Pocket Reference, I discuss some of the most common of these tools and frameworks, including:

- Singer
- Stitch
- Fivetran

Despite the prevalence of these tools, some teams decide to build custom code to ingest data. Some even develop their own frameworks. The reasons vary by organization but are often related to cost, a culture of building over buying, and concerns about the legal and security risks of trusting an external vendor. In Chapter 5, I discuss the build versus buy trade-offs that are unique to data ingestion tools. Of particular interest is whether the value of a commercial solution is to make it easier for data engineers to build data ingestions into their pipelines or to enable nondata engineers (such as data analysts) to build ingestions themselves.

As Chapters 4 and 5 discuss, data ingestion is traditionally both the *extract* and *load* steps of an ETL or ELT process. Some tools focus on just these steps, while others provide the user with some *transform* capabilities as well. In practice, I find most data teams choose to limit the number of transformations they make during data ingestion and thus stick to ingestion tools

that are good at two things: extracting data from a source and loading it into a destination.

Data Transformation and Modeling Tools

Though the bulk of this chapter has focused on moving data between sources and destinations (data ingestion), there is much more to data pipelines and the movement of data. Pipelines are also made up of tasks that transform and model data for new purposes, such as machine learning, analysis, and reporting.

The terms *data modeling* and *data transformation* are often used interchangeably; however, for the purposes of this text, I will differentiate between them:

Data transformation
: Transforming data is a broad term that is signified by the *T* in an ETL or ELT process. A transformation can be something as simple as converting a timestamp stored in a table from one time zone to another. It can also be a more complex operation that creates a new metric from multiple source columns that are aggregated and filtered through some business logic.

Data modeling
: Data modeling is a more specific type of data transformation. A data model structures and defines data in a format that is understood and optimized for data analysis. A data model is usually represented as one or more tables in a data warehouse. The process of creating data models is discussed in more detail in Chapter 6.

Like data ingestion, there are a number of methodologies and tools that are present in a modern data infrastructure. As previously noted, some data ingestion tools provide some level of data transformation capabilities, but these are often quite simple. For example, for the sake of protecting *personally identifiable information* (*PII*) it may be desirable to turn an email address into a hashed value that is stored in the final

destination. Such a transformation is usually performed during the ingestion process.

For more complex data transformations and data modeling, I find it desirable to seek out tools and frameworks specifically designed for the task, such as dbt (see Chapter 9). In addition, data transformation is often context-specific and can be written in a language familiar to data engineers and data analysts, such as SQL or Python.

Data models that will be used for analysis and reporting are typically defined and written in SQL or via point-and-click user interfaces. Just like build-versus-buy trade-offs, there are considerations in choosing to build models using SQL versus a *no-code* tool. SQL is a highly accessible language that is common to both data engineers and analysts. It empowers the analyst to work directly with the data and optimize the design of models for their needs. It's also used in nearly every organization, thus providing a familiar entry point for new hires to a team. In most cases, choosing a transformation framework that supports building data models in SQL rather than via a point-and-click user interface is desirable. You'll get far more customizability and own your development process from end to end.

Chapter 6 discusses transforming and modeling data at length.

Workflow Orchestration Platforms

As the complexity and number of data pipelines in an organization grows, it's important to introduce a *workflow orchestration platform* to your data infrastructure. These platforms manage the scheduling and flow of tasks in a pipeline. Imagine a pipeline with a dozen tasks ranging from data ingestions written in Python to data transformations written in SQL that must run in a particular sequence throughout the day. It's not a simple challenge to schedule and manage dependencies between each task. Every data team faces this challenge, but thankfully there are numerous workflow orchestration platforms available to alleviate the pain.

NOTE

Workflow orchestration platforms are also referred to as *workflow management systems* (WMSs), *orchestration platforms*, or *orchestration frameworks*. I use these terms interchangeably in this text.

Some platforms, such as Apache Airflow, Luigi, and AWS Glue, are designed for more general use cases and are thus used for a wide variety of data pipelines. Others, such as Kubeflow Pipelines, are designed for more specific use cases and platforms (machine learning workflows built on Docker containers in the case of Kubeflow Pipelines).

Directed Acyclic Graphs

Nearly all modern orchestration frameworks represent the flow and dependencies of tasks in a pipeline as a graph. However, pipeline graphs have some specific constraints.

Pipeline steps are always *directed*, meaning they start with a general task or multiple tasks and end with a specific task or tasks. This is required to guarantee a path of execution. In other words, it ensures that tasks do not run before all their dependent tasks are completed successfully.

Pipeline graphs must also be *acyclic*, meaning that a task cannot point back to a previously completed task. In other words, it cannot cycle back. If it could, then a pipeline could run endlessly!

With these two constraints in mind, orchestration pipelines produce graphs called directed acyclic graphs (DaGs). Figure 2-3 illustrates a simple DAG. In this example, Task A must complete before Tasks B and C can start. Once they are both completed, then Task D can start. Once Task D is complete, the pipeline is completed as well.

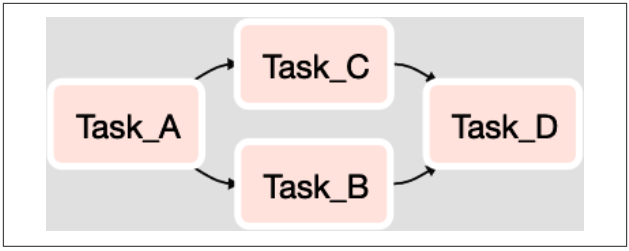


Figure 2-3. A DAG with four tasks. After Task A completes, Task B and Task C run. When they both complete, Task D runs.

DAGs are a representation of a set of tasks and not where the logic of the tasks is defined. An orchestration platform is capable of running tasks of all sorts.

For example, consider a data pipeline with three tasks. It is represented as a DAG in Figure 2-4.

- The first executes a SQL script that queries data from a relational database and stores the result in a CSV file.
- The second runs a Python script that loads the CSV file, cleans, and then reshapes the data before saving a new version of the file.
- Finally, a third task, which runs the COPY command in SQL, loads the CSV created by the second task into a Snowflake data warehouse.

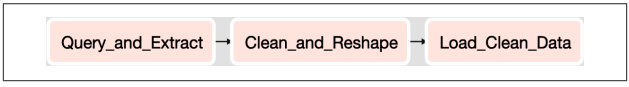


Figure 2-4. A DAG with three tasks that run in sequence to extract data from a SQL database, clean and reshape the data using a Python script, and then load the resulting data into a data warehouse.

The orchestration platform executes each task, but the logic of the tasks exists as SQL and Python code, which runs on different systems across the data infrastructure.

Chapter 7 discusses workflow orchestration platforms in more detail and provides hands-on examples of orchestrating a pipeline in Apache Airflow.

Customizing Your Data Infrastructure

It's rare to find two organizations with exactly the same data infrastructure. Most pick and choose tools and vendors that meet their specific needs and build the rest on their own. Though I talk in detail about some of the most popular tools and products throughout this book, many more come to market each year.

As previously noted, depending on the culture and resources in your organization, you may be encouraged to build most of your data infrastructure on your own, or to rely on SaaS vendors instead. Regardless of which way you lean on the build-versus-buy scale, you can build the high-quality data infrastructure necessary to build high-quality data pipelines.

What's important is understanding your constraints (dollars, engineering resources, security, and legal risk tolerance) and the resulting trade-offs. I speak to these throughout the text and call out key decision points in selecting a product or tool.

CHAPTER 3

Common Data Pipeline Patterns

Even for seasoned data engineers, designing a new data pipeline is a new journey each time. As discussed in Chapter 2, differing data sources and infrastructure present both challenges and opportunities. In addition, pipelines are built with different goals and constraints. Must the data be processed in near real time? Can it be updated daily? Will it be modeled for use in a dashboard or as input to a machine learning model?

Thankfully, there are some common patterns in data pipelines that have proven successful and are extensible to many use cases. In this chapter, I will define these patterns. Subsequent chapters implement pipelines built on them.

ETL and ELT

There is perhaps no pattern more well known than ETL and its more modern sibling, ELT. Both are patterns widely used in data warehousing and business intelligence. In more recent years, they've inspired pipeline patterns for data science and machine learning models running in production. They are so well known that many people use these terms synonymously with data pipelines rather than patterns that many pipelines follow.

Given their roots in data warehousing, it's easiest to describe them in that context, which is what this section does. Later sections in this chapter describe how they are used for particular use cases.

Both patterns are approaches to data processing used to feed data into a data warehouse and make it useful to analysts and reporting tools. The difference between the two is the order of their final two steps (transform and load), but the design implications in choosing between them are substantial, as I'll explain throughout this chapter. First, let's explore the steps of ETL and ELT.

The *extract* step gathers data from various sources in preparation for loading and transforming. Chapter 2 discussed the diversity of these sources and methods of extraction.

The *load* step brings either the raw data (in the case of ELT) or the fully transformed data (in the case of ETL) into the final destination. Either way, the end result is loading data into the data warehouse, data lake, or other destination.

The *transform* step is where the raw data from each source system is combined and formatted in a such a way that it's useful to analysts, visualization tools, or whatever use case your pipeline is serving. There's a lot to this step, regardless of whether you design your process as ETL or ELT, all of which is explored in detail in Chapter 6.

Separation of Extract and Load

The combination of the extraction and loading steps is often referred to as *data ingestion*. Especially in ELT and the EtLT subpattern (note the lowercase *t*), which is defined later in this chapter, extraction and loading capabilities are often tightly coupled and packaged together in software frameworks. When designing pipelines, however, it is still best to consider the two steps as separate due to the complexity of coordinating extracts and loads across different systems and infrastructure.

Chapters 4 and 5 describe data ingestion techniques in more detail and provide implementation examples using common frameworks.

The Emergence of ELT over ETL

ETL was the gold standard of data pipeline patterns for decades. Though it's still used, more recently ELT has emerged as the pattern of choice. Why? Prior to the modern breed of data warehouses, primarily in the cloud (see Chapter 2), data teams didn't have access to data warehouses with the storage or compute necessary to handle loading vast amounts of raw data and transforming it into usable data models all in the same place. In addition, data warehouses at the time were row-based databases that worked well for transactional use cases, but not for the high-volume, bulk queries that are commonplace in analytics. Thus, data was first extracted from source systems and then transformed on a separate system before being loaded into a warehouse for any final data modeling and querying by analysts and visualization tools.

The majority of today's data warehouses are built on highly scalable, columnar databases that can both store and run bulk transforms on large datasets in a cost-effective manner. Thanks to the I/O efficiency of a columnar database, data compression, and the ability to distribute data and queries across many nodes that can work together to process data, things have changed. It's now better to focus on extracting data and loading it into a data warehouse where you can then perform the necessary transformations to complete the pipeline.

The impact of the difference between row-based and column-based data warehouses cannot be overstated. Figure 3-1 illustrates an example of how records are stored on disk in a row-based database, such as MySQL or Postgres. Each row of the database is stored together on disk, in one or more blocks depending on the size of each record. If a record is smaller than

a single block or not cleanly divisible by the block size, it leaves some disk space unused.

OrderId	CustomerId	ShippingCountry	OrderTotal
1	1258	US	55.25
2	5698	AUS	125.36
3	2265	US	776.95
4	8954	CA	32.16

Block 1	1, 1258, US, 55.25
Block 2	2, 5698, AUS, 125.36
Block 3	3, 2265, US, 776.95
Block 4	4, 8954, CA, 32.16

Figure 3-1. A table stored in a row-based storage database. Each block contains a record (row) from the table.

Consider an online transaction processing (OLTP) database use case such as an e-commerce web application that leverages a MySQL database for storage. The web app requests reads and writes from and to the MySQL database, often involving multiple values from each record, such as the details of an order on an order confirmation page. It's also likely to query or update only one order at a time. Therefore, row-based storage is optimal since the data the application needs is stored in close proximity on disk, and the amount of data queried at one time is small.

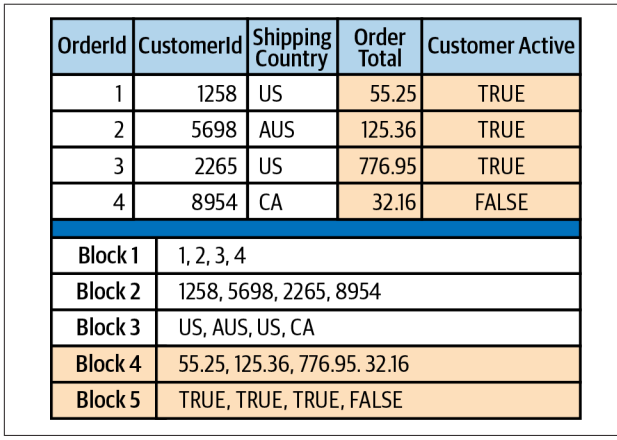
The inefficient use of disk space due to records leaving empty space in blocks is a reasonable trade-off in this case, as the speed to reading and writing single records frequently is what's most important. However, in analytics the situation is reversed. Instead of the need to read and write small amounts of data frequently, we often read and write a large amount of data infrequently. In addition, it's less likely that an analytical query requires many, or all, of the columns in a table but rather a single column of a table with many columns.

For example, consider the order table in our fictional e-commerce application. Among other things, it contains the dollar amount of the order as well as the country it's shipping to. Unlike the web application, which works with orders one at a time, an analyst using the data warehouse will want to analyze orders in bulk. In addition, the table containing order data in the data warehouse has additional columns that contain values from multiple tables in our MySQL database. For example, it might contain the information about the customer who placed the order. Perhaps the analyst wants to sum up all orders placed by customers with currently active accounts. Such a query might involve millions of records, but only read from two columns, OrderTotal and CustomerActive. After all, analytics is not about creating or changing data (like in OLTP) but rather the derivation of metrics and the understanding of data.

As illustrated in Figure 3-2, a columnar database, such as Snowflake or Amazon Redshift, stores data in disk blocks by column rather than row. In our use case, the query written by the analyst only needs to access blocks that store OrderTotal and CustomerActive values rather than blocks that store the row-based records such as the MySQL database. Thus, there's less disk I/O as well as less data to load into memory to perform the filtering and summing required by the analyst's query. A final benefit is reduction in storage, thanks to the fact that blocks can be fully utilized and optimally compressed since the same data type is stored in each block rather than multiple types that tend to occur in a single row-based record.

All in all, the emergence of columnar databases means that storing, transforming, and querying large datasets is efficient within a data warehouse. Data engineers can use that to their advantage by building pipeline steps that specialize in extracting and loading data into warehouses where it can be transformed, modeled, and queried by analysts and data scientists who are more comfortable within the confines of a database. As such, ELT has taken over as the ideal pattern for data

warehouse pipelines as well as other use cases in machine learning and data product development.



OrderId	CustomerId	Shipping Country	Order Total	Customer Active
1	1258	US	55.25	TRUE
2	5698	AUS	125.36	TRUE
3	2265	US	776.95	TRUE
4	8954	CA	32.16	FALSE

Block 1	1, 2, 3, 4
Block 2	1258, 5698, 2265, 8954
Block 3	US, AUS, US, CA
Block 4	55.25, 125.36, 776.95. 32.16
Block 5	TRUE, TRUE, TRUE, FALSE

Figure 3-2. A table stored in a column-based storage database. Each disk block contains data from the same column. The two columns involved in our example query are highlighted. Only these blocks must be accessed to run the query. Each block contains data of the same type, making compression optimal.

EtLT Subpattern

When ELT emerged as the dominant pattern, it became clear that doing some transformation after extraction, but before loading, was still beneficial. However, instead of transformation involving business logic or data modeling, this type of transformation is more limited in scope. I refer to this as *lowercase t* transformation, or *EtLT*.

Some examples of the type of transformation that fits into the EtLT subpattern include the following:

- Deduplicate records in a table
- Parse URL parameters into individual components

- Mask or otherwise obfuscate sensitive data

These types of transforms are either fully disconnected from business logic or, in the case of something like masking sensitive data, at times required as early in a pipeline as possible for legal or security reasons. In addition, there is value in using the right tool for the right job. As Chapters 4 and 5 illustrate in greater detail, most modern data warehouses load data most efficiently if it's prepared well. In pipelines moving a high volume of data, or where latency is key, performing some basic transforms between the extract and load steps is worth the effort.

You can assume that the remaining ELT-related patterns are designed to include the EtLT subpattern as well.

ELT for Data Analysis

ELT has become the most common and, in my opinion, most optimal pattern for pipelines built for data analysis. As already discussed, columnar databases are well suited to handling high volumes of data. They are also designed to handle wide tables, meaning tables with many columns, thanks to the fact that only data in columns used in a given query are scanned on disk and loaded into memory.

Beyond technical considerations, data analysts are typically fluent in SQL. With ELT, data engineers can focus on the extract and load steps in a pipeline (data ingestion), while analysts can utilize SQL to transform the data that's been ingested as needed for reporting and analysis. Such a clean separation is not possible with an ETL pattern, as data engineers are needed across the entire pipeline. As shown in Figure 3-3, ELT allows data team members to focus on their strengths with less interdependencies and coordination.

In addition, the ELT pattern reduces the need to predict exactly what analysts will do with the data at the time of building extract and load processes. Though understanding the general

use case is required to extract and load the proper data, saving the transform step for later gives analysts more options and flexibility.

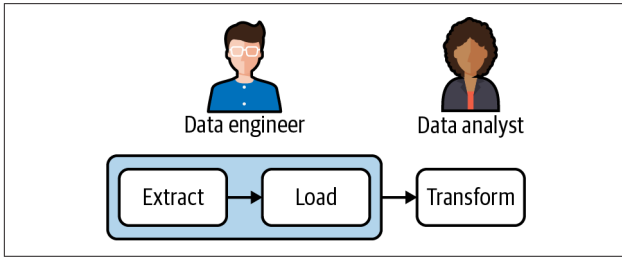


Figure 3-3. The ELT pattern allows for a clean split of responsibilities between data engineers and data analysts (or data scientists). Each role can work autonomously with the tools and languages they are comfortable in.

> **NOTE**
>
> With the emergence of ELT, data analysts have become more autonomous and empowered to deliver value from data without being "blocked" by data engineers. Data engineers can focus on data ingestion and supporting infrastructure that enables analysts to write and deploy their own transform code written as SQL. With that empowerment have come new job titles such as the *analytics engineer*. Chapter 6 discusses how these data analysts and analytics engineers transform data to build data models.

ELT for Data Science

Data pipelines built for data science teams are similar to those built for data analysis in a data warehouse. Like the analysis use case, data engineers are focused on ingesting data into a data warehouse or data lake. However, data scientists have different needs from the data than data analysts do.

Though data science is a broad field, in general, data scientists will need access to more granular—and at times raw—data than data analysts do. While data analysts build data models that produce metrics and power dashboards, data scientists spend their days exploring data and building predictive models. While the details of the role of a data scientist are out of the scope of this book, this high-level distinction matters to the design of pipelines serving data scientists.

If you're building pipelines to support data scientists, you'll find that the extract and load steps of the ELT pattern will remain pretty much the same as they will for supporting analytics. Chapters 4 and 5 outline those steps in technical detail. Data scientists might also benefit from working with some of the data models built for analysts in the transform step of an ELT pipeline (Chapter 6), but they'll likely branch off and use much of the data acquired during extract-load.

ELT for Data Products and Machine Learning

Data is used for more than analysis, reporting, and predictive models. It's also used for powering *data products*. Some common examples of data products include the following:

- A content recommendation engine that powers a video streaming home screen
- A personalized search engine on an e-commerce website
- An application that performs sentiment analysis on user-generated restaurant reviews

Each of those data products is likely powered by one or more machine learning (ML) models, which are hungry for training and validation data. Such data may come from a variety of source systems and undergo some level of transformation to prepare it for use in the model. An ELT-like pattern is well suited for such needs, though there are a number of specific challenges in all steps of a pipeline that's designed for a data product.