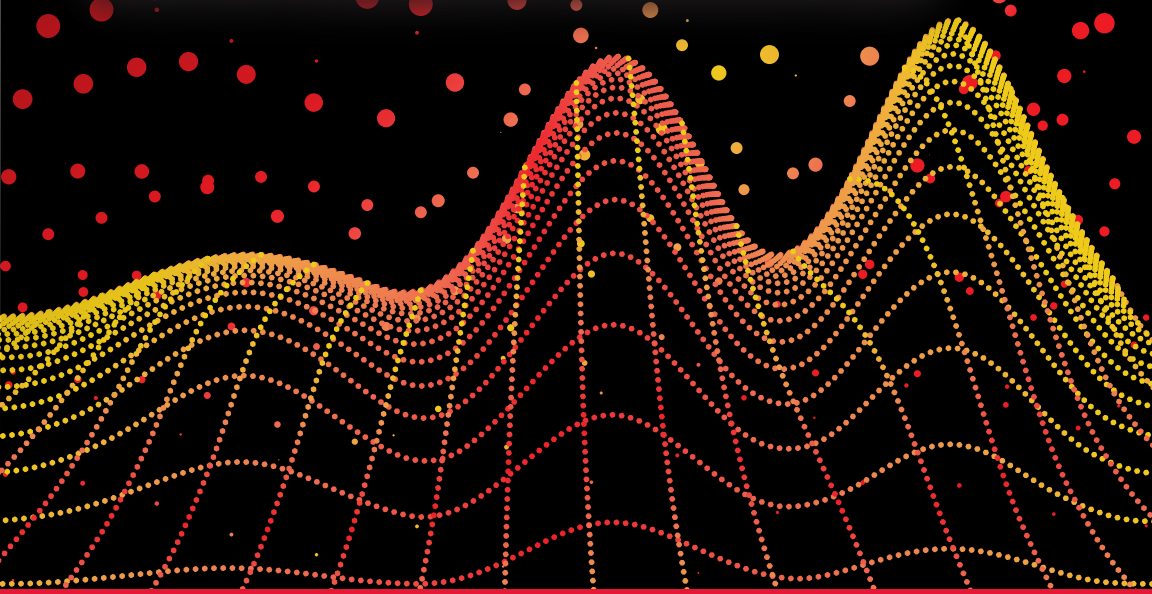
O'REILLY®

Compliments of unravel™

Rebuilding Reliable Data Pipelines Through Modern Tools

Ted Malaska
with the assistance of Shivnath Babu



REPORT

unravel™

RADICALLY SIMPLIFY YOUR DATA PIPELINES

Your Modern Data Applications, ETL, IoT, Machine Learning, Customer 360 and more, need to perform reliably. With Big Data, that's not always easy.

Unravel makes data work.

Unravel removes the blind spots in your data pipelines, providing AI-powered recommendations to drive more reliable performance in your modern data applications.

Greater Productivity

98% reduction in troubleshooting time

Guaranteed Reliability

100% of apps delivered on time

Lower Costs

60% reduction in cost

DON'T JUST MONITOR PERFORMANCE – OPTIMIZE IT.

UNRAVELDATA.COM

Rebuilding Reliable Data Pipelines Through Modern Tools

Ted Malaska
with the assistance of Shivnath Babu

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Rebuilding Reliable Data Pipelines Through Modern Tools
by Ted Malaska

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: Jonathan Hassell
Development Editor: Corbin Collins
Production Editor: Christopher Faucher
Copyeditor: Octal Publishing, LLC
Proofreader: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2019: First Edition

Revision History for the First Edition
2019-06-25: First Release

978-1-492-05814-4

[LSI]

Table of Contents

CHAPTER 1
Introduction

Back in my 20s, my wife and I started running in an attempt to fight our ever-slowing metabolism as we aged. We had never been very athletic growing up, which comes with the lifestyle of being computer and video game nerds.

We encountered many issues as we progressed, like injury, consistency, and running out of breath. We fumbled along making small gains and wins along the way, but there was a point when we decided to ask for external help to see if there was more to learn.

We began reading books, running with other people, and running in races. From these efforts we gained perspective on a number of areas that we didn't even know we should have been thinking about. The perspectives allowed us to understand and interpret the pains and feelings we were experiencing while we ran. This input became our internal monitoring and alerting system.

We learned that shin splints were mostly because of old shoes landing wrong when our feet made contact with the ground. We learned to gauge our sugar levels to better inform our eating habits.

The result of understanding how to run and how to interpret the signals led us to quickly accelerate our progress in becoming better runners. Within a year we went from counting the blocks we could run before getting winded to finishing our first marathon.

It is this idea of understanding and signal reading that is core to this book, applied to data processing and data pipelines. The idea is to provide a high- to mid-level introduction to data processing so that

you can take your business intelligence, machine learning, near-real-time decision making, or analytical department to the next level.

Who Should Read This Book?

This book is for people running data organizations that require data processing. Although I dive into technical details, that dive is designed primarily to help higher-level viewpoints gain perspective on the problem at hand. The perspectives the book focuses on include data architecture, data engineering, data analysis, and data science. Product managers and data operations engineers can also gain insight from this book.

Data Architects

Data architects look at the big picture and define concepts and ideas around producers and consumers. They are visionaries for the data nervous system for a company or organization. Although I advise architects to code at least 50% of the time, this book does not require that. The goal is to give an architect enough background information to make strong calls, without going too much into the details of implementation. The ideas and patterns discussed in this book will outlive any one technical implementation.

Data Engineers

Data engineers are in the business of moving data—either getting it from one location to another or transforming the data in some manner. It is these hard workers who provide the digital grease that makes a data project a reality.

Although the content in this book can be an overview for data engineers, it should help you see parts of the picture you might have previously overlooked or give you fresh ideas for how to express problems to nondata engineers.

Data Analysts

Data analysis is normally performed by data workers at the tail end of a data journey. It is normally the data analyst who gets the opportunity to generate insightful perspectives on the data, giving companies and organizations better clarity to make decisions.

This book will hopefully give data analysts insight into all the complex work it takes to get the data to you. Also, I am hopeful it will give you some insight into how to ask for changes and adjustments to your existing processes.

Data Scientists

In a lot of ways, a data scientist is like a data analyst but is looking to create value in a different way. Where the analyst is normally about creating charts, graphs, rules, and logic for humans to see or execute, the data scientist is mostly in the business of training machines through data.

Data scientists should get the same out of this book as the data analyst. You need the data in a repeatable, consistent, and timely way. This book aims to provide insight into what might be preventing your data from getting to you in the level of service you expect.

Product Managers

Being a product manager over a business intelligence (BI) or data-processing organization is no easy task because of the highly technical aspect of the discipline. Traditionally, product managers work on products that have customers and produce customer experiences. These traditional markets are normally related to interfaces and user interfaces.

The problem with data organizations is that sometimes the customer's experience is difficult to see through all the details of workflows, streams, datasets, and transformations. One of the goals of this book with regard to product managers is to mark out boxes of customer experience like data products and then provide enough technical knowledge to know what is important to the customer experience and what are the details of how we get to that experience.

Additionally, for product managers this book drills down into a lot of cost benefit discussions that will add to your library of skills. These discussions should help you decide where to focus good resources and where to just buy more hardware.

Data Operations Engineers

Another part of this book focuses on signals and inputs, as mentioned in the running example earlier. If you haven't read Site Relia-

bility Engineering (O'Reilly), I highly recommend it. Two things you will find there are the passion and possibility for greatness that comes from listening to key metrics and learning how to automate responses to those metrics.

Outline and Goals of This Book

This book is broken up into eight chapters, each of which focuses on a set of topics. As you read the chapter titles and brief descriptions that follow, you will see a flow that looks something like this:

- The ten-thousand-foot view of the data processing landscape
- A slow descent into details of implementation value and issues you will confront
- A pull back up to higher-level terms for listening and reacting to signals

Chapter 2: How We Got Here

The mindset of an industry is very important to understand if you intend to lead or influence that industry. This chapter travels back to the time when data in an Excel spreadsheet was a huge deal and shows how those early times are still affecting us today. The chapter gives a brief overview of how we got to where we are today in the data processing ecosystem, hopefully providing you insight regarding the original drivers and expectations that still haunt the industry today.

Chapter 3: The Data Ecosystem Landscape

This chapter talks about data ecosystems in companies, how they are separated, and how these different pieces interact. From that perspective, I focus on processing because this book is about processing and pipelines. Without a good understanding of the processing role in the ecosystem, you might find yourself solving the wrong problems.

Chapter 4: Data Process at its Core

This is where we descend from ten thousand feet in the air to about one thousand feet. Here we take a deep dive into data processing

and what makes up a normal data processing job. The goal is not to go into details of code, but I get detailed enough to help an architect or a product manager be able to understand and speak to an engineer writing that detailed code.

Then we jump back up a bit and talk about processing in terms of data pipelines. By now you should understand that there is no magic processing engine or storage system to rule them all. Therefore, understanding the role of a pipeline and the nature of pipelines will be key to the perspectives on which we will build.

Chapter 5: Identifying Job Issues

This chapter looks at all of the things that can go wrong with data processing on a single job. It covers the sources of these problems, how to find them, and some common paths to resolve them.

Chapter 6: Identifying Workflow and Pipeline Issues

This chapter builds on ideas expressed in Chapter 5 but from the perspective of how they relate to groups of jobs. While making one job work is enough effort on its own, now we throw in hundreds or thousands of jobs at the same time. How do you handle isolation, concurrency, and dependencies?

Chapter 7: Watching and Learning from Your Jobs

Now that we know tons of things can go wrong with your jobs and data pipelines, this chapter talks about what data we want to collect to be able to learn how to improve our operations.

After we have collected all the data on our data processing operations, this chapter talks about all the things we can do with that data, looking from a high level at possible insights and approaches to give you the biggest bang for your buck.

Chapter 8: Closing Thoughts

This chapter gives a concise look at where we are and where we are going as an industry with all of the context of this book in place. The goal of these closing thoughts is to give you hints to where the future might lie and where fill-in-the-gaps solutions will likely be short lived.

CHAPTER 2
How We Got Here

Let's begin by looking back and gaining a little understanding of the data processing landscape. The goal here will be to get to know some of the expectations, players, and tools in the industry.

I'll first run through a brief history of the tools used throughout the past 20 years of data processing. Then, we look at producer and consumer use cases, followed by a discussion of the issue of scale.

Excel Spreadsheets

Yes, we're talking about Excel spreadsheets—the software that ran on 386 Intel computers, which had nearly zero computing power compared to even our cell phones of today.

So why are Excel spreadsheets so important? Because of expectations. Spreadsheets were and still are the first introduction into data organization, visualization, and processing for a lot of people. These first impressions leave lasting expectations on what working with data is like. Let's dig into some of these aspects:

Visualization
: We take it for granted, but spreadsheets allowed us to see the data and its format and get a sense of its scale.

Functions
: Group By, Sum, and Avg functions were easy to add and returned in real time.

Graphics
: Getting data into graphs and charts was not only easy but provided quick iteration between changes to the query or the displays.

Decision making
: Advanced Excel users could make functions that would flag cells of different colors based on different rule conditions.

In short, everything we have today and everything discussed here is meant to echo this spreadsheet experience as the data becomes bigger and bigger.

Databases

After the spreadsheet came *database generation*, which included consumer technology like Microsoft Access database as well as big corporate winners like Oracle, SQL Server, and Db2, and their market disruptors such as MySQL and PostgreSQL.

These databases allowed spreadsheet functionality to scale to new levels, allowing for SQL, which gives an access pattern for users and applications, and transactions to handle concurrency issues.

For a time, the database world was magical and was a big part of why the first dot-com revolution happened. However, like all good things, databases became overused and overcomplicated. One of the complications was the idea of *third normal form*, which led to storing different entities in their own tables. For example, if a person owned a car, the person and the car would be in different tables, along with a third table just to represent the ownership relationship. This arrangement would allow a person to own zero or more than one car and a car to be owned by zero or more than one person, as shown in Figure 2-1.

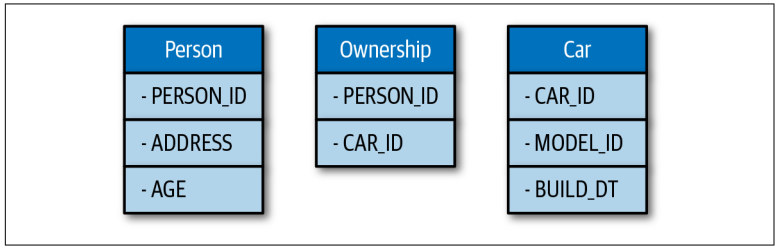


Figure 2-1. Owning a car required three tables

Although third normal form still does have a lot of merit, its design comes with a huge impact on performance and design. This impact is a result of having to join the tables together to gain a higher level of meaning. Although SQL did help with this joining complexity, it also enabled more functionality that would later prove to cause problems.

The problems that SQL caused were not in the functionality itself. It was making complex distributed functionality accessible by people who didn't understand the details of how the function would be executed. This resulted in functionally correct code that would perform poorly. Simple examples of functionality that caused trouble were *joins* and *windowing*. If poorly designed, they both would result in more issues as the data grew and the number of involved tables increased.

More entities resulted in more tables, which led to more complex SQL, which led to multiple thousand-line SQL code queries, which led to slower performance, which led to the birth of the *appliance*.

Appliances

Oh, the memories that pop up when I think about the *appliance database*. Those were fun and interesting times. The big idea of an appliance was to take a database, distribute it across many nodes on many racks, charge a bunch of money for it, and then everything would be great!

However, there were several problems with this plan:

Distribution experience
: The industry was still young in its understanding of how to build a distributed system, so a number of the implementations were less than great.

High-quality hardware
: One side effect of the poor distribution experience was the fact that node failure was highly disruptive. That required processing systems with extra backups and redundant components like power supplies—in short, very tuned, tailored, and pricey hardware.

Place and scale of bad SQL
: Even the additional nodes with all the processing power they offered could not overcome the rate at which SQL was being abused. It became a race to add more money to the problem, which would bring short-term performance benefits. The benefits were short lived, though, because the moment you had more processing power, the door was open for more abusive SQL. The cycle would continue until the cost became a problem.

Data sizes increasing
: Although in the beginning the increasing data sizes meant more money for vendors, at some point the size of the data outpaced the technology. The outpacing mainly came from the advent of the internet and all that came along with it.

Double down on SQL
: The once-simple SQL language would grow more and more complex, with advanced functions like windowing functions and logical operations like PL/SQL.

All of these problems together led to disillusionment with the appliance. Often the experience was great to begin with, but then became expensive and slow and costly as the years went on.

Extract, Transform, and Load Platforms

One attempt to fix the problem with appliances was to redefine the role of the appliance. The argument was that appliances were not the problem. Instead, the problem was SQL, and data became so complex and big that it required a special tool for transforming it.

The theory was that this would save the appliance for the analysts and give complex processing operations to something else. This approach had three main goals:

- Give analysts a better experience on the appliance
- Give the data engineers building the transformational code a new toy to play with
- Allow vendors to define a new category of product to sell

The Processing Pipeline

Although it most likely existed before the advent of the Extract, Transform, and Load (ETL) platforms, it was the ETL platforms that pushed pipeline engineering into the forefront. The idea with a *pipeline* is now you had to have many jobs that could run on different systems or use different tools to solve a single goal, as illustrated in Figure 2-2.

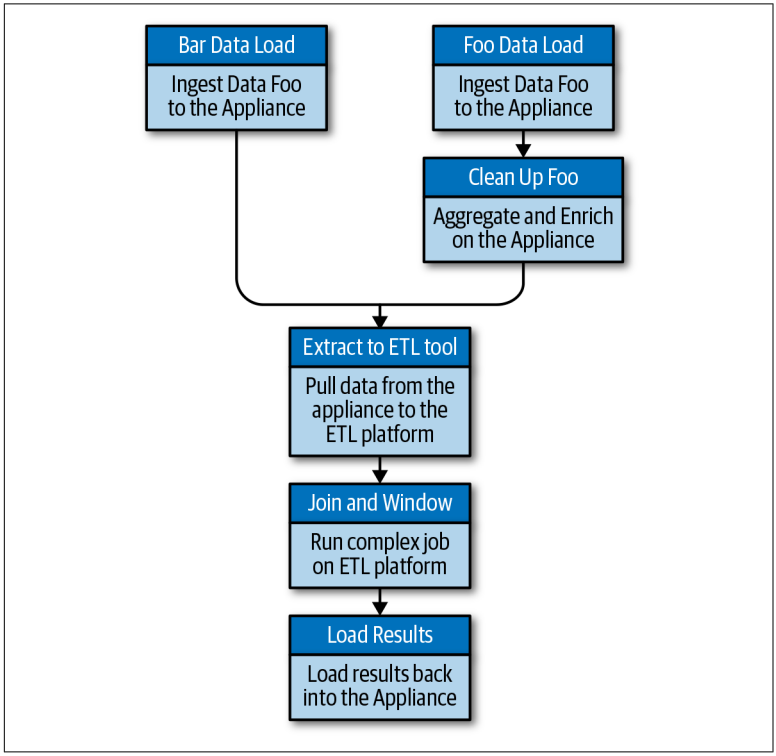


Figure 2-2. Pipeline example

The idea of the pipeline added multiple levels of complexity into the process, like the following:

Which system to use
: Figuring out which system did which operation the best.

Transfer cost
: Understanding the extraction and load costs.

Scheduling resources
: Dealing with schedules from many different systems, creating a quest to avoid bottlenecks and log jams.

Access rights
: Whenever you leave a system and interact with external systems, access control always becomes an interesting topic.

However, for all its complexity, the pipeline really opened the door for everything that followed. No more was there an expectation that one system could solve everything. There was now a global understanding that different systems could optimize for different operations. It was this idea that exploded in the 2000s into the open source big data revolution, which we dig into next.

Kafka, Spark, Hadoop, SQL, and NoSQL platforms

With the advent of the idea that the appliance wasn't going to solve all problems, the door was open for new ideas. In the 2000s, internet companies took this idea to heart and began developing systems that were highly tuned for a subset of use cases. These inventions sparked an open source movement that created a lot of the foundations we have today in data processing and storage. They flipped everything on its head:

Less complexity
: If many tables caused trouble, let's drop them all and go to one table with nested types (NoSQL).

Embrace failure
: Drop the high-cost hardware for commodity hardware. Build the system to expect failure and just recover.

Separate storage from compute logically
: Before, if you had an appliance, you used its SQL engine on its data store. Now the store and the engine could be made separately, allowing for more options for processing and future proofing.

Beyond SQL
: Where the world of the corporate databases and appliances was based on SQL, this new system allowed a mix of code and SQL.

> For better or worse, it raised the bar of the level of engineer that could contribute.

However, this whole exciting world was built on optimizing for given use cases, which just doubled down on the need for data processing through pipelines. Even today, figuring out how to get data to the right systems for storage and processing is one of the most difficult problems to solve.

Apart from more complex pipelines, this open source era was great and powerful. Companies now had little limits of what was technically possible with data. For 95% of the companies in the world, their data would never reach a level that would ever stress these new breeds of systems if used correctly.

It is that last point that was the issue and the opportunity: *if they were used correctly*. The startups that built this new world designed for a skill level that was not common in corporations. In the low-skill, high-number-of-consultants culture, this resulted in a host of big data failures and many dreams lost.

This underscores a major part of why this book is needed. If we can understand our systems and use them correctly, our data processing and pipeline problems can be resolved.

It's fair to say that after 2010 the problem with data in companies is not a lack of tools or systems, but a lack of coordination, auditing, vision, and understanding.

Cloud, On-Premises, and Hybrid Environments

As the world was just starting to understand these new tools for big data, the cloud changed everything. I remember when it happened. There was a time when no one would give an online merchant company their most valuable data. Then *boom*, the CIA made a groundbreaking decision and picked Amazon to be its cloud provider over the likes of AT&T, IBM, and Oracle. The CIA was followed by FINRA, a giant regulator of US stock transitions, and then came Capital One, and then everything changed. No one would question the cloud again.

The core technology really didn't change much in the data world, but the cost model and the deployment model did, with the result of doubling down on the need for more high-quality engineers. The

better the system, the less it would cost, and the more it would be up. In a lot of cases, this metric could differ by 10 to 100 times.

Machine Learning, Artificial Intelligence, Advanced Business Intelligence, Internet of Things

That brings us to today. With the advent of machine learning and artificial intelligence (AI), we have even more specialized systems, which means more pipelines and more data processing.

We have all the power, logic, and technology in the world at our fingertips, but it is still difficult to get to the goals of value. Additionally, as the tools ecosystem has been changing, so have the goals and the rewards.

Today, we can get real-time information for every part of our business, and we can train machines to react to that data. There is a clear understanding that the companies that master such things are going to be the ones that live to see tomorrow.

However, the majority of problems are not solved by more PhDs or pretty charts. They are solved better by improving the speed of development, speed of execution, cost of execution, and freedom to iterate.

Today, it still takes a high-quality engineer to implement these solutions, but in the future, there will be tools that aim to remove the complexity of optimizing your data pipelines. If you don't have the background to understand the problems, how will you be able to find these tools that can fix these pains correctly?

Producers and Considerations

For producers, a lot has changed from the days of manually entering data into spreadsheets. Here are a number of ways in which you can assume your organization needs to take in data:

File dropping
: This is the act of sending data in units of files. It's very common for moving data between organizations. Even though streaming is the cool, shiny option, the vast majority of today's data is still sent in batch file submission over intervals greater than an hour.

Streaming
: Although increasing in popularity within companies, streaming is still not super common between companies. Streaming offers near-real-time (NRT) delivery of data and the opportunity to make decisions on information sooner.

Internet of Things (IoT)
: A subset of streaming, IoT is data created from devices, applications, and microservices. This data normally is linked to high-volume data from many sources.

Email
: Believe it or not, a large amount of data between groups and companies is still submitted over good old-fashioned email as attachments.

Database Change Data Capture (CDC)
: Either through querying or reading off a database's edit logs, the mutation records produced by database activity can be an important input source for your data processing needs.

Enrichment
: This is the act of mutating original data to make new datasets. There are several ways in which data can be enriched:

  - **Data processing:** Through transformation workflow and jobs
  - **Data tagging/labeling:** Normally human or AI labeling to enrich data so that it can be used for structured machine learning
  - **Data tracing:** Adding lineage metadata to the underlying data

The preceding list is not exhaustive. There are many more ways to generate new or enriched datasets. The main goal is to figure out how to represent that data. Normally it will be in a data structure governed by a schema, and it will be data processing workflows that get your data into this highly structured format. Hence, if these workflows are the gateway to making your data clean and readable, you need these jobs to work without fail and at a reasonable cost profile.

What About Unstructured Data and Schemas?

Some will say, "Unstructured data doesn't need a schema." And they are partly right. At a minimum, an unstructured dataset would have one field: a string or blob field called *body* or *content*.

However, unstructured data is normally not alone. It can come with the following metadata that makes sense to store alongside the body/content data:

Event time
: The time the data was created.

Source
: Where the data came from. Sometimes, this is an IP, region, or maybe an application ID.

Process time
: The time the data was saved or received.

Consider the balloon theory of data processing work: there is *N* amount of work to do, and you can either say I'm not going to do it when we bring data in or you can say I'm not going to do it when I read the data.

The only option you don't have is to make the work go away. This leaves two more points to address: the number of writers versus readers, and the number of times you write versus the number of times you read.

In both cases you have more readers, and readers read more often. So, if you move the work of formatting to the readers, there are more chances for error and waste of execution resources.

Consumers and Considerations

Whereas our producers have become more complex over the years, our consumers are not far behind. No more is the single consumer of an Excel spreadsheet going to make the cut. There are more tools and options for our consumers to use and demand. Let's briefly look at the types of consumers we have today:

SQL users
: This group makes up the majority of SQL consumers. They live and breathe SQL, normally through Java Database Connectivity

(JDBC)/Open Database Connectivity (ODBC) on desktop development environments called Integrated Development Environments (IDEs). Although these users can produce group analytical data products at high speeds, they also are known to write code that is less than optimal, leading to a number of the data processing concerns that we discuss later in this book.

Advanced users
: This a smaller but growing group of consumers. They are separated from their SQL-only counterparts because they are empowered to use code alongside SQL. Normally, this code is generated using tools like R, Python, Apache Spark, and more. Although these users are normally more technical than their SQL counterparts, they too will produce jobs that perform suboptimally. The difference here is that the code is normally more complex, and it's more difficult to infer the root cause of the performance concerns.

Report users
: These are normally a subset of SQL users. Their primary goal in life is to create dashboards and visuals to give management insight into how the business is functioning. If done right, these jobs should be simple and not induce performance problems. However, because of the visibility of their output, the failure of these jobs can produce unwanted attention from upper management.

Inner-loop applications
: These are applications that need data to make synchronous decisions (Figure 2-3). These decisions can be made through coded logic or trained machine learning models. However, they both require data to make the decision, so the data needs to be accessible in low latencies and with high guarantees. To reach this end, normally a good deal of data processing is required ahead of time.

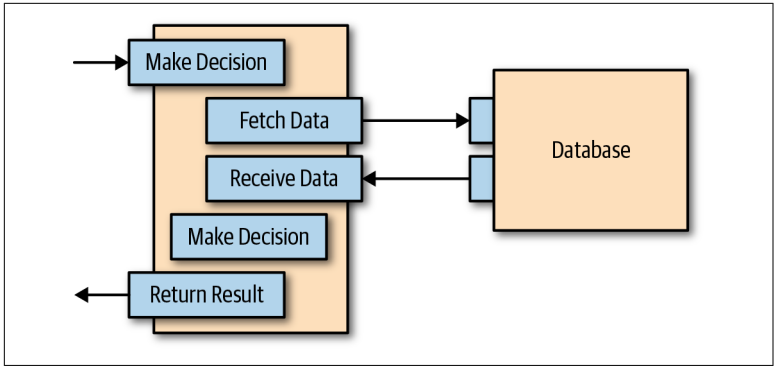


Figure 2-3. Inner-loop execution

Outer-loop applications
: These applications make decisions just like their inner-loop counterparts, except they execute them asynchronously, which offers more latency of data delivery (Figure 2-4).

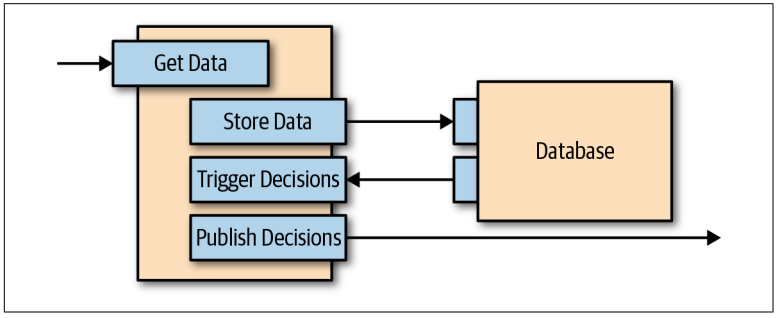


Figure 2-4. Outer-loop execution

Summary

You should now have a sense of the history that continues to shape every technical decision in today's ecosystem. We are still trying to solve the same problems we aimed to address with spreadsheets, except now we have a web of specialized systems and intricate webs of data pipelines that connect them all together.

The rest of the book builds on what this chapter talked about, in topics like the following:

- How to know whether we are processing well
- How to know whether we are using the right tools

- How to monitor our pipelines

Remember, the goal is not to understand a specific technology, but to understand the patterns involved. It is these patterns in processing and pipelines that will outlive the technology of today, and, unless physics changes, the patterns you learn today will last for the rest of your professional life.

CHAPTER 3

The Data Ecosystem Landscape

This chapter focuses on defining the different components of today's data ecosystem environments. The goal is to provide context for how our problem of data processing fits within the data ecosystem as a whole.

The Chef, the Refrigerator, and the Oven

In general, all modern data ecosystems can be divided into three metaphorical groups of functionality and offerings:

Chef
: Responsible for design and metamanagement. This is the mind behind the kitchen. This person decides what food is bought and by what means it should be delivered. In modern kitchens the chef might not actually do any cooking. In the data ecosystem world, the chef is most like design-time decisions and a management layer for all that is happening in the kitchen.

Refrigerator
: Handles publishing and persistence. This is where food is stored. It has preoptimized storage structures for fruit, meat, vegetables, and liquids. Although the chef is the brains of the kitchen, the options for storage are given to the chef. The chef doesn't redesign a different refrigerator every day. The job of the fridge is like the data storage layer in our data ecosystem: keep the data safe and optimized for access when needed.

Oven

: Deals with access and processing. The oven is the tool in which food from the fridge is processed to make quality meals while producing value. In this relation, the oven is an example of the processing layer in the data ecosystem, like SQL; Extract, Transform, and Load (ETL) tools; and schedulers.

Although you can divide a data ecosystem differently, using these three groupings allows for clean interfaces between the layers, affording you the most optimal enterprise approach to dividing up the work and responsibility (see Figure 3-1).

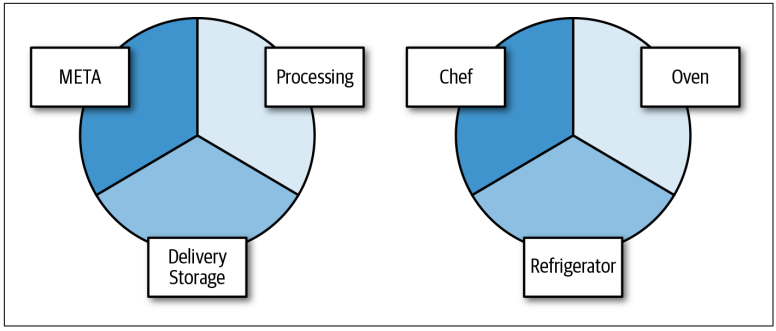


Figure 3-1. Data ecosystem organizational separation

Let's quickly drill down into these interfaces because some of them will be helpful as we focus on access and processing for the remainder of this book:

- Meta <- Processing: Auditing
- Meta -> Processing: Discovery
- Processing <- Persistence: Access (normally through SQL interfaces)
- Processing -> Persistence: Generated output
- Meta -> Persistence: What to persist
- Meta <- Persistence: Discover what else is persisted

The rest of this chapter drills down one level deeper into these three functional areas of the data ecosystem. Then, it is on to Chapter 4, which focuses on data processing.

The Chef: Design Time and Metadata Management

Design time and metadata management is all the rage now in the data ecosystem world for two main reasons:

Reducing time to value
: Helping people find and connect datasets on a meaningful level to reduce the time it takes to discover value from related datasets.

Adhering to regulations
: Auditing and understanding your data can alert you if the data is being misused or in danger of being wrongly accessed.

Within the chef's domain is a wide array of responsibilities and functionality. Let's dig into a few of these to help you understand the chef's world:

Creating and managing datasets/tables
: The definition of fields, partitioning rules, indexes, and such. Normally offers a declarative way to define, tag, label, and describe datasets.

Discovering datasets/tables
: For datasets that enter your data ecosystem without being declaratively defined, someone needs to determine what they are and how they fit in with the rest of the ecosystem. This is normally called *scraping* or *curling* the data ecosystem to find signs of new datasets.

Auditing
: Finding out how data entered the ecosystem, how it was accessed, and which datasets were sources for newer datasets. In short, auditing is the story of how data came to be and how it is used.

Security
: Normally, defining security sits at the chef's level of control. However, the implementation of security is normally implemented in either the refrigerator or the oven. The chef is the one who must not only give and control the rules of security, but must also have full access to know the existing securities given.

The Refrigerator: Publishing and Persistence

The refrigerator has been a longtime favorite of mine because it is tightly linked to cost and performance. Although this book is primarily about access and processing, that layer will be highly affected by how the data is stored. This is because in the refrigerator's world, we need to consider trade-offs of functionality like the following:

Storage formats
: This could be storage in a database or as files. Both will affect data size, read patterns, read speeds, and accessibility.

Compression
: There are a number of compression options, some slower to write, some slower to read. JSON and comma-separated values (CSV)—the formats normally most common for data—can be compressed beyond 80% or 90%. Compression is a big deal for cost, transmission, and to reduce disk input/output (I/O).

Indexing
: Indexing in general involves direction to the data you want to find. Without indexing, you must scan through large subsections of your data to find what you are looking for. Indexing is like a map. Imagine trying to find a certain store in the mall without a map. Your only option would be to walk the entire mall until you luckily found the store you were looking for.

Reverse indexing
: This is commonly used in tools like Elasticsearch and in the technology behind tech giants like Google. This is metadata about the index, allowing not only fast access to pointed items, but real-time stats about all the items and methods to weigh different ideas.

Sorting
: Putting data in order from less than to greater than is a hidden part of almost every query you run. When you join, group by, order by, or reduce by, under the hood there is at least one *sort* in there. We sort because it is a great way to line up related information. Think of a zipper. You just pull it up or down. Now imagine each zipper key is a number and the numbers are scattered on top of a table. Imagine how difficult it would be to put

the zipper back together—not a joyful experience without pre-ordering.

Streaming versus batch
: Is your data one static unit that updates only once a day or is it a stream of ever-changing and appending data? These two options are very different and require a lot of different publishing and persistence decisions to be made.

Only once
: This is normally related to the idea that data can be sent or received more than once. For the cases in which this happens, what should the refrigerator layer do? Should it store both copies of the data or just hold on to one and absorb the other?

That's just a taste of the considerations needed for the refrigerator layer. Thankfully, a lot of these decisions and options have already been made for you in common persistence options. Let's quickly look at some of the more popular tools in the data ecosystem and how they relate to the decision factors we've discussed:

Cassandra
: This is a NoSQL database that gives you out-of-the-box, easy access to indexing, sorting, real-time mutations, compression, and deduplicating. It is ideal for pointed GETs and PUTs, but not ideal for scans or aggregations. In addition, Cassandra can moonlight as a time-series database for some interested entity-focused use cases.

Kafka
: Kafka is a streaming pipeline with compression and durability that is pretty good at ordering if used correctly. Although some wish it were a database (inside joke at the confluent company), it is a data pipe and is great for sending data to different destinations.

Elasticsearch
: Initially just a search engine and storage system, but because of how data is indexed, Elasticsearch provides side benefits of deduplicating, aggregations, pointed GETs and PUTs (even though mutation is not recommended), real-time, and reverse indexing.

Database/warehouse
: This is a big bucket that includes the likes of Redshift, Snowflake, Teradata Database, Exadata, Google's BigQuery, and many more. In general, these systems aim to solve for many use cases by optimizing for a good number of use cases with the popular SQL access language. Although a database can solve for every use case (in theory), in reality, each database is good at a couple of things and not so good at others. Which things a database is good at depends on compromises the database architecture made when the system was built.

In memory
: Some systems like Druid.io, MemSQL, and others aim to be databases but better. The big difference is that these systems can store data in memory in hopes of avoiding one of the biggest costs of databases: serialization of the data. However, memory isn't cheap, so sometimes we need to have a limited set of data isolated for these systems. Druid.io does a great job of optimizing for the latest data in memory and then flushing older data to disk in a more compressed format.

Time-series
: Time-series databases got their start in the NoSQL world. They give you indexing to an entity and then order time-event data close to that entity. This allows for fast access to all the metric data for an entity. However, people usually become unhappy with time-series databases in the long run because of the lack of scalability on the aggregation front. For example, aggregating a million entities would require one million lookups and an aggregation stage. By contrast, databases and search systems have much less expensive ways to ask such queries and do so in a much more distributed way.

Amazon Simple Storage Service (Amazon S3)/object store
: Object stores are just that: they store objects (files). You can take an object store pretty far. Some put Apache Hive on top of their object stores to make append-only database-like systems, which can be ideal for low-cost scan use cases. Mutations and indexing don't come easy in an object store, but with enough know-how, an object store can be made into a real database. In fact, Snowflake is built on an object store. So, object stores, while being a primary data ecosystem storage offering in themselves, are also

a fundamental building block for more complex data ecosystem storage solutions.

The Oven: Access and Processing

The oven is where food becomes something else. There is *processing* involved.

This section breaks down the different parts of the oven into how we get data, how we process it, and where we process it.

Getting Our Data

From the refrigerator section, you should have seen that there are a number of ways to store data. This also means that there are a number of ways to *access* data. To move data into our oven, we need to understand these access patterns.

Access considerations

Before we dig into the different types of access approaches, let's first take a look at the access considerations we should have in our minds as we evaluate our decisions:

Tells us what the store is good at
: Different access patterns will be ideal for different quests. As we review the different access patterns, it's helpful to think about which use cases they would be good at helping and which they wouldn't be good at helping.

Concurrence
: Different access patterns allow different volumes of different requests at the same time. This can mean that one access pattern is good for a smaller pool of users and another is good at supporting a larger pool of users.

Isolation
: The cost of access in some systems is expensive and/or it can affect other users on that system. This is sometimes called the *noisy neighbor* problem or can be referred to having the level of *isolation of each request*. Normally, higher levels of concurrence are aligned with better degrees of isolation.

Accessibility
: Some access patterns are easier for humans to interact with and some are better suited for machine interaction.

Parallelism
: When accessing data, how many threads or systems can be accessed at once? Do the results need to be focused into one receiver or can the request be divided up?

Access types

Let's look at the different groupings of access patterns we have in our data ecosystem:

SQL
: One of the most popular tools for analysts and machine learning engineers for accessing data, SQL is simple and easy to learn. However, it comes with three big problems:

  - **Offers too much functionality:** The result of having so many options is that users can write very complex logic in SQL, which commonly turns out to use the underlying system incorrectly and adds additional cost or causes performance problems.
  - **SQL isn't the same:** Although many systems will allow for SQL, not all versions, types, and extensions of SQL are transferable from one system to another. Additionally, you shouldn't assume that SQL queries will perform the same on different storage systems.
  - **Parallelism concerns:** Parallelism and bottlenecks are two of the biggest issues with SQL. The primary reason for this is the SQL language was really built to allow for detailed parallelism configuration or visibility. There are some versions of SQL today that allow for hints or configurations to alter parallelism in different ways. However, these efforts are far from perfect and far from universal cross-SQL implementations.

Application Programming Interface (API) or custom
: As we move away from normal database and data warehouse systems, we begin to see a divergence in access patterns. Even in Cassandra with its CQL (a super small subset of SQL), there is

usually a learning curve for traditional SQL users. However, these APIs are more tuned to the underlying system's optimized usage patterns. Therefore, you have less chance of getting yourself in trouble.

Structured files
: Files come in many shapes and sizes (CSV, JSON, AVRO, ORC, Parquet, Copybook, and so on). Reimplementing code to parse every type of file for every processing job can be very time consuming and error prone. Data in files should be moved to one of the aforementioned storage systems. We want to access the data with more formal APIs, SQL, and/or dataframes in systems that offer better access patterns.

Streams
: Streams is an example of reading from systems like Kafka, Pulsar, Amazon's Kinesis, RabbitMQ, and others. In general, the most optimal way to read a stream is from now onward. You read data and then acknowledge that you are done reading it. This acknowledgement either moves an offset or fires off a commit. Just like SQL, stream APIs offer a lot of additional functionality that can get you in trouble, like moving offsets, rereading of data over time, and more. These options can work well in controlled environments, but use them with care.

Stay Stupid, My Friend

As we have reviewed our access types, I hope a common pattern has grabbed your eye. All of the access patterns offer functionality that can be harmful to you. Additionally, some problems can be hidden from you in low-concurrency environments. In that, if you run them when no one else is on the system, you find that everything runs fine. However, when you run the same job on a system with a high level of "noisy neighbors," you find that issues begin to arise.

The problem with these issues is that they wait to point up until you have committed tons of resources and money to the project—then it will blow up in front of all the executives, fireworks style.

The laws of marketing require vendors to to add extra features to these systems. In general, however, as a user of any system, we should search for its core reason for existence and use the system within that context. If we do that, we will have a better success rate.

How Do You Process Data?

Now that we have gathered our data, how should we process it? In processing, we are talking about doing something with the data to create value, such as the following:

Mutate
: The simplest process is the act of mutating the results at a line level: formatting a date, rounding a number, filtering out records, and so on.

Multirecord aggregation
: Here we take in more than one record to help make a new output. Think about averages, mins, maxs, group by's, windowing, and many more options.

Feature generation
: Feature generation boils down to mutations and multirecord aggregations taken to the level of optimizing input for models to be trained or executed on.

Model training
: In a time when machine learning is changing everything, model training is bound to be of importance to you. To train models, you need to be able to put data into any number of machine learning engines to build models to cluster, recommend, decide, and so on.

Logic enhanced decisions
: We look at data in order to make decisions. Some of those decisions will be made with simple human-made logic, like thresholds or triggers, and some will be logic generated from machine learning neural networks or decision trees. It could be said that this advanced decision making is nothing more than a single-line mutation to the extreme.

Where Do You Process Data?

We look at this question from two angles: where the processing lives in relation to the system that holds the data, and where in terms of the cloud and/or on-premises.

With respect to the data

In general, there are three main models for data processing's location in relation to data storage:

- Local to the storage system
- Local to the data
- Remote from the storage system

Let's take a minute to talk about where these come from and where the value comes from.

Local to the storage system. In this model, the system that houses the data will be doing the majority of the processing and will most likely return the output to you. Normally the output in this model is much smaller than the data being queried.

Some examples of systems that use this model of processing include Elasticsearch and warehouse, time-series, and in-memory databases. They chose this model for a few reasons:

Optimizations
: Controlling the underlying storage and the execution engine will allow for additional optimizations on the query plan. However, when deciding on a storage system, remember this means that the system had to make trade-offs for some use cases over others. And this means that the embedded execution engines will most likely be coupled to those same optimizations.

Reduced data movement
: Data movement is not free. If it can be avoided, you can reduce that cost. Keeping the execution in the storage system can allow for optimizing data movement.

Format conversion
: The cost of serialization is a large factor in performance. If you are reading data from one system to be processed by another, most likely you will need to change the underlying format.

Lock in
: Let's be real here: there is always a motive for the storage vendor to solve your problems. The stickiness is always code. If they can get you to code in their APIs or their SQL, they've locked you in as a customer for years, if not decades.

Speed to value
: For a vendor and for a customer, there is value in getting solutions up quickly if the execution layer is already integrated with the data. A great example of this is Elasticsearch and the integrated Kibana. Kibana and its superfast path to value might have been the best technical decision the storage system has made so far.

Local and remote to the data. There are times when the execution engine of the storage system or the lack of an execution engine might request the need for an external execution system. Here are some good examples of remote execution engines:

Apache MapReduce
: One of the first (though now outdated) distributed processing systems. This was a system made up of mappers that read the data and then optionally shuffled and sorted the data to be processed by reducers.

Apache Spark
: In a lot of aspects, this is the successor to Apache MapReduce. Spark improved performance, is easier to use, integrated SQL, machine learning, and much more.

TensorFlow
: Originally developed by Google, this is an external framework for training neural networks to build models.

Python
: A common language used for many use cases, but with respect to data it is used to build models and do advanced machine learning.

R
: R is used for advanced machine learning and statistical use cases.

Flink
: This is used for processing streams in near real time (NRT).

Back in the Hadoop years, there was an argument for bringing the execution to your data. The idea was that Hadoop could store your data, and you then could run your jobs on the same nodes.

There are too many issues with this argument to cover here. However, here are the basics:

Remote happens anyway
: If you do anything beyond a map-only job (such as join, sort, group by, reduce by, or window), the data needs to go over the network, so you're remote anyway.

Nodes for storage might not be optimal for processing
: We are learning this even more in the world of TensorFlow and neural networks, which require special Graphics Processing Units (GPUs) to do their work at a reasonable time and cost. No one should couple their storage strategy to the ever-changing domains of processing power.

Hadoop is no longer king
: Your processing engine might want to execute on other storage systems besides Hadoop, so there is no need to couple the two.

World of the cloud
: Everything in the cloud costs money. Storage and processing both cost money, but they can be charged separately, so why pay for processing when you're not processing?

With respect to the environment

It should be clear that the future for the vast majority of companies is in the cloud. However, a good majority of these companies will still have a sizable footprint in their on-prem environment.

Data be in the cloud versus on-premises

This book isn't a discussion about on-premises versus the cloud. In general, that argument has been decided in favor of the cloud. Here are many things that will force the cloud solution to win in the long term:

New tech
: All new innovations are either happening in the cloud or by the cloud vendors themselves.

On demand
: Within seconds, cloud resources can be assigned to you or released from you. In an on-premises environment you are

forced to make long-term bets that take months to set up and really are near impossible to give back.

Freedom to fail
: As a result of the quick load times and the option to pay only for what you use, you can try new ideas and quickly decide whether that direction will pan out for you. Iterations and allowance of failure are key to development success in today's fast-paced world.

Talent
: Anyone who is a technical person at this point in time is focused on developing their skills in cloud-related technology. So, if you want strong people, you need to be in the cloud to attract them.

Rewarded by good design
: Because you pay for what you use, the model reinforces good design decisions and punishes poor ones.

Thinking about data processing for on-premises. If you are still on-premises for the foreseeable future and are doing data processing, here are your main considerations:

Reuse
: How can you reuse the same hardware for many different use cases? This dilemma comes from the inability to buy and switch out hardware at a moment's notice.

Use what you have
: One of the worst things you can do in an on-premises world is buy a boatload of hardware and then find out that you needed only 10% of it. It is only natural that you fill up what you buy. No one ever got in trouble for asking for more, as opposed to buying more than what was needed.

Running out of capacity
: The "use what you have" rule normally results in resources being constrained, which leaves you managing a delicate balance between using resources and optimizing them so that you can continually put more on the system.

Predicting hardware needs

When it is time to buy more hardware, the big question is how much do you need? This is near impossible to determine because you might not have enough space and resources to experiment. So, in most cases you are reliant on the vendor to tell you, and the vendor has an incentive to sell you as much as possible.

Chargeback

To give an explanation for the sunk cost of the hardware and license fees, there normally is a drive to figure out what value is being created from all that cost. Determining the appropriate chargeback model is difficult enough in the first place; even when done well, it can have the unintended effect of discouraging people from using your solution.

Stopping bad actors

In any data processing system, there will be developers or analysts who write bad queries and either make too much data or use up too many resources. It is difficult to balance addressing this on a shared platform on-premises for a number of reasons:

- **Finger-pointing:** In a shared platform, a number of organizations are normally in the mix, and normally with conflicting goals or highly dependent relationships. In both cases, this can lead to different groups becoming defensive toward each other, resulting in an overly critical environment.
- **Red tape:** Sometimes the auditing of workloads can be seen as unneeded red tape. Not only can red tape lead to additional finger-pointing, it can also encourage "going rogue," with the system user deciding that using the shared platform is too much trouble.
- **Discouragement:** Maybe a bad actor has a great use case, but the solution wasn't implemented in the best of ways. It's difficult to give criticism in a way that doesn't discourage. In addition, the only way for a shared platform to be successful is to celebrate successful implementations.
- **DevOps:** Normally in an on-premises world there is a group within your company that maintains services and a group that develops on them. As long as this exists you will

have stone throwing when things go wrong, and *they will* go wrong.

Thinking about data processing for the cloud. Whereas on-premises is all about using what you bought well and accounting for it, the cloud is about being mindful of what you ask for. This leads to a very different set of considerations:

Auditing
: The cost of cloud usage will catch up with you overnight (literally overnight, because you will sometimes forget to turn off instances). Auditing will lead to a positive question for developers: is the work you are doing worth the money?

Optimization
: Again, with cost in mind, we can look at how we do things and whether there are less expensive ways to do those things. Examples: compress more, run more jobs on one set of hardware, use different instances, and turn off resources that you are not using.

DevOps turns to SRE
: The term Site Reliability Engineer (SRE) is a new focus on building fault-tolerant systems. This push has come from the fact that in the cloud, failure is front and center. Because failure is normal in the cloud, development now takes on more ownership of building fault-tolerant systems, whereas with on-premises, you had DevOps people to blame.

Tech sprawl and duplication
: A side effect of the cloud is that too many people will go in too many directions, and your company will fall into ever-increasing technical debt. In on-premises, there was more focus because of the large up-front costs. Those guardrails are now gone, and there are fewer reasons for alignment (for good or bad reasons). The result is a need for companies to keep an eye on everything that is being developed in hopes of noticing redundant solutions.

Ecosystem and Data Pipelines

I close the chapter with the idea of the chef, the refrigerator, and the oven and the importance of how these groupings affect our data pipelines.

The data pipeline must directly work with all three groups, which makes pipeline management even more complex. Let's dig into what a pipeline designer needs to consider when looking at the three ecosystem groupings.

The Chef and the Pipeline

The *pipeline* is the path data takes from its sources to its final value-producing destinations. Whenever a pipeline needs to read data, the sources are known by the chef's components, and whenever a pipeline outputs data, the chef's components need to be aware.

If done right, the chef is both the map and the log of the pipeline—the map for pipeline designers to help define their path from sources to destinations, and the log of the pipeline's implementation so that others can evaluate the pipeline and build off of it.

The Refrigerator and the Pipeline

The refrigerator for the pipeline defines all the options for sources, data marts, and destinations. Let's dig in to all three of these.

Sources
: The sources are where the pipeline will source the data. However, depending on the types of source, as mentioned earlier, the source might reduce processing work. For example, if the source is indexed or nested, it can reduce the need for processing. This leads to the idea of data marts.

Data marts
: If your company has more than one pipeline, there are times where the different pipelines could benefit from common optimized stores. This is where the data mart comes in. Although it could be a destination for your output, the data mart could also serve as a middle point for pipelines, allowing multiple pipelines to reuse the joins, indexes, or ordering of the data mart to reduce processing.

Destinations
: The real goal of pipelines is to get the data in a given format to a given destination. As mentioned earlier, different destinations are optimized for different use cases. It's not the pipeline's job to define the destinations, but to ensure that it gets the data there reliably and with quality.

The Oven and the Pipeline

If the chef is the map and the log, and the refrigerator is all the stops along the way, the oven is how you get from one stop to another.

The oven includes extracting, loading, and transforming data. For pipeline engineers, the oven will most likely be the most interesting part of the job. This is the Spark, SQL, and microservices of your pipeline. The next couple of chapters are dedicated to this topic.

Summary

To optimize data processing and pipelines, you need to understand the whole data ecosystem. If you lose focus on any of the three main ecosystem components (chef, refrigerator, or oven), your pipelines will become unbalanced, and the ecosystem as a whole will suffer. In later chapters, you will see an emphasis on visibility and auditing pipelines. In part, this is to help you maintain balance and to evaluate.

CHAPTER 4

Data Processing at Its Core

This book began with a very high-level overview of data processing history and then descended to a more detailed review of the different components of a data ecosystem. We are now diving to a low enough altitude that we will look into the foundations of all data processing at a job level.

The goal of this chapter is to lay a foundation for you to build on later in the book when we begin categorizing different systems and discussing issues that frequently come up in data processing systems. This book is about optimizing the entire pipeline which you can't do without understanding what is happening in its smallest units.

Then, we ascend a bit and look at *Directed Acyclic Graphs* (DAGs) at a pipeline level, which are DAGs that make up many processing DAGs and storage systems.

What Is a DAG?

The first time I heard about a DAG engine, I was super hopeful it was going to be groundbreaking and would change my understanding of data processing. If you have a name as cool as a *DAG engine*, by law you need to be awesome. However, as with many things in life, as you learn the details, reality becomes less exciting than we had hoped.

A DAG is nothing more than a graph of nodes and edges in which you don't loop back, as shown in Figure 4-1.

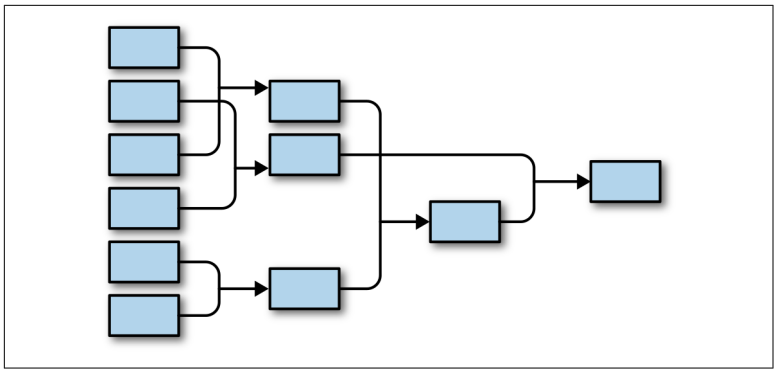


Figure 4-1. Simple DAG

The DAG provides a great way to explain all data processing and even data pipelines. In general, a DAG can represent the execution of a job in the oven or the path data takes throughout the whole data ecosystem.

If the DAG is for a single job, each node represents a unique operation, and the lines between are normally the optimized transmission of data through memory and network transmissions.

If the DAG is for an entire pipeline, each node signifies one job or DAG within that larger DAG. The difference with a pipeline DAG is that the line between nodes is normally an ingest or extraction from a storage system.

The main body of this chapter focuses on the single-job DAG and optimizations when you're designing one. Some of the high-level ideas discussed in the single-job DAG section will translate into the viewpoint of the pipeline DAG. However, how they apply and why they apply can differ. The main idea is to give you a low- then high-level view of DAG optimization, first at a job level and then at multiple job levels in a pipeline.

Single-Job DAGs

The term *single-job DAGs* refers to one SQL statement or one Spark job. The scope in functionality of a single job is not limited by any other factor. It can read in zero to many data sources and output zero to many resulting data sources.

As we discuss in the next major section, the limitations to a single-job's DAG will be separated by a storage system or serialization of the data. For this section, think of a single DAG as a single SQL or Spark job, for which the DAG is the instructions or recipe on how to take the inputs and produce the outputs.

DAGs as Recipes

When making cookies (mmm, cookies), you need directions to tell you which ingredients to add when, and what operations need to take place at every stage of baking.

Consider the DAG in Figure 4-2 for executing word count.

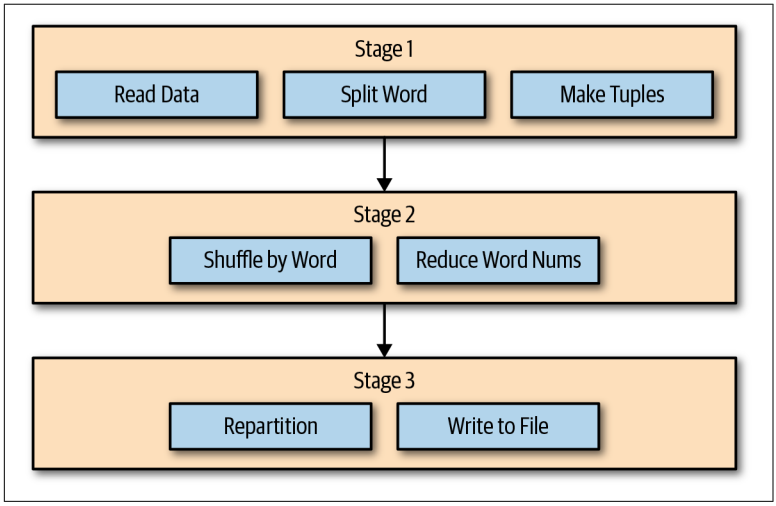


Figure 4-2. DAG stage separation

Let's examine each stage of this process:

1. Stage 1
   a. Spin up map tasks.
   b. Read data from source.
   c. Split each line into words.
   d. Convert each word to a tuple (a value with two subvalues) of the word and the number 1.
2. Stage 2
   a. Shuffle the values by the word.

b. Perform a reduce job on the word and add the numbers of the words together, as shown in Figure 4-3.

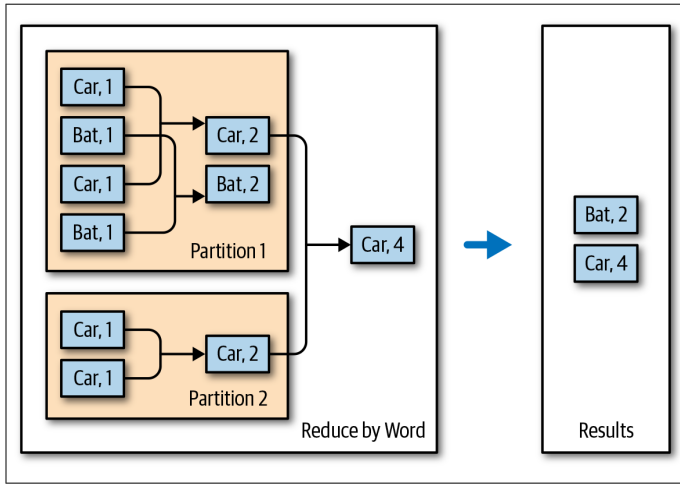


Figure 4-3. Example of reduce by

3. Stage 3
   a. Repartition to one partition.
   b. Write out the results to a single file.

In the end, with many partitions, the data flow for this DAG will look something like Figure 4-4.

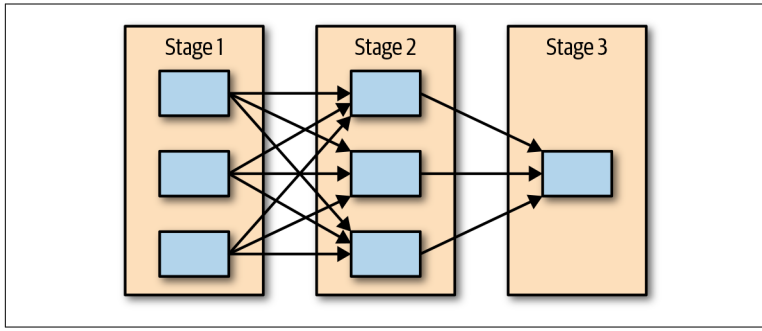


Figure 4-4. : Example of a shuffle (Stage 1 -> 2) and a repartition (Stage 2 -> 3)

I reuse this diagram as we work through the chapter and give a visual for different components of a DAG, which will be important as we troubleshoot later on in the book.

DAG Operations/Transformations/Actions

In this subsection, we look into a number of parts that make up our common DAG, such as the following:

Inputs
: How data comes into our process

Map-only processes
: Processing in place

Shuffle
: Moving data around

Repartitioning
: Properly sizing the parallelism

Advanced techniques
: Extra credit for abnormal DAG activity

The goal of these subsections is to help you get a base understanding of how distributed processing works within the context of a DAG topology.

Inputs

As you see in our word count example, we have three starting points. This is indicating that we are reading from the source with three different threads that might or might not be executing from the same node, as shown in Figure 4-5.

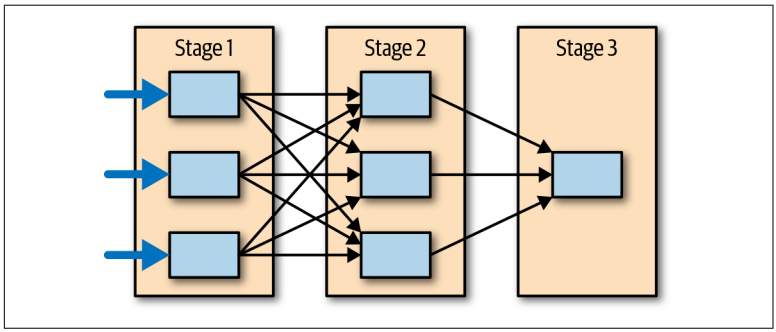


Figure 4-5. Inputs into a DAG

Normally, the number of input streams is defined by the source you are reading from. For example, if you are reading from Amazon Simple Storage Service (Amazon S3), the number of inputs will

depend on the number and sizes of your files and whether your files are splittable.

Finding the appropriate level of separation is not easy, but in general you don't want more than 1 GB of uncompressed data going to one process.

Input Bottlenecks

Earlier, I talked about different storage systems and how they allow for access. If your source system doesn't allow for more than one parallel read, you might be experiencing a bottleneck on the read.

Read bottlenecks are irritating because everything goes slowly, but nothing looks busy—not the CPU, network, disk, or anything else.

Map-only processes

Map-only processes are stages in the DAG during which data is read and processed without exchanging any data between the processing system's nodes. To help understand what a map-only process is, let's look at some common map-only functions in Apache Spark:

Map
: A process in which a record comes into the function and one value comes out, to continue on to the next stage in the DAG.

FlatMap
: A process in which a record comes into the function and zero or more value(s) come out, to continue on to the next stage in the DAG.

Foreach
: A process in which a record comes in and then a process is executed on the record, but no records are processed. This is sometimes used to send data outside the DAG.

A map-only process normally is the first process in the DAG as you read the initial data sources or processes that follow a shuffle, repartition, or reduceBy. Figure 4-6 shows a partition diagram.

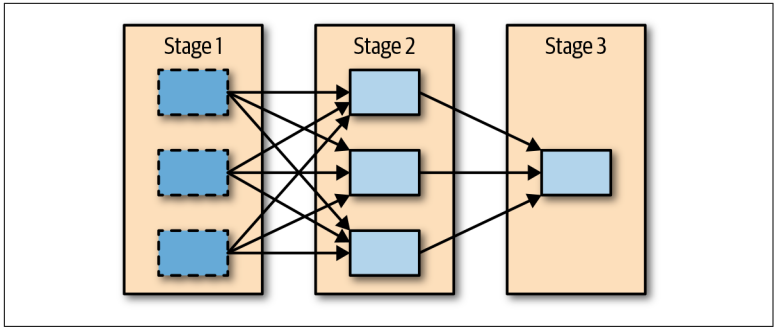


Figure 4-6. Highlighting the first map-only part of a DAG

If you look closely at Figure 4-6, you will note that the boxes in Stage 2 and 3 can also be map-only processes. However, some will not consider the entire job to be map-only if it includes a shuffle or repartition.

Maps are great because they don't require a *shuffle*. A shuffle is that crazy line-crossing thing you see between Stage 1 and Stage 2 in Figure 4-6. The maps read data as it was given to them and write data out as it was given to them. For the most part, map jobs are single-record actions.

Shuffle

The shuffle, as just mentioned, is the crazy line-crossing part of the diagram, as shown in Figure 4-7.

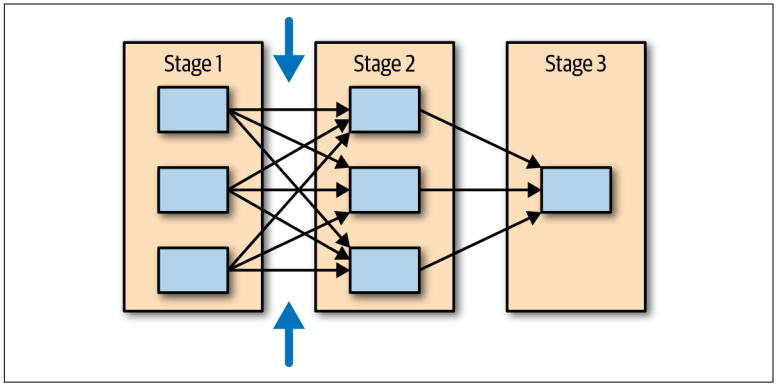


Figure 4-7. Shuffle operation in a DAG

There are a number of things going on here in our word count example:

A map-side reduce
: Because reducing is not super expensive and there are opportunities to reduce within a given partition, the result of a map-side reduce could mean less data going over the wire.

Partitioning the data
: Normally, data is partitioned by a hash of the key value (in our case, it is the word) and then that value will mathematical modulo (%) the number of partitions. This will randomly put all the same key values in the same partitions. This allows for a pseudo-random distribution that is also deterministic in its results. Meaning, if you run the job twice, all the keys and their records would land in the same partitions as they did the first time around.

Sorting or grouped
: Additionally, the data will be sorted or grouped within a partition so that at the end of the shuffle we can reduce by key or in other use cases do group by's or joins.

Send over the network
: The act of sending data over the network is where the real pain comes from. Not only does your data go over the wire, but you need to serialize and deserialize it on the other side. All this serialization is very CPU intensive.

In general, the shuffle is where you will find a good deal of your processing pain. There are so many things that can go wrong with designing shuffles into your workload:

They are expensive
: Use with care because sometimes you send a lot of data over the wire.

How many shuffles
: The goal should always be as few as possible.

Skew
: Skew is when you have too many of one key, so partitions become uneven or lopsided (more on skew coming up soon).

Resulting partitions
: The number of resulting partitions is up to you, unlike the input partitions. So, depending on what you need to do on the other end, you might need more or fewer partitions.

Repartitioning

Not all shuffles are created equal. Sometimes you just need to repartition. The example in Figure 4-8 shows repartitions to write out a single file.

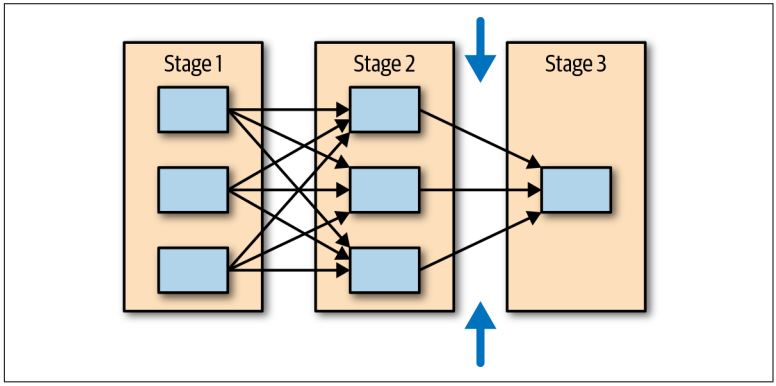


Figure 4-8. Repartitioning in a DAG

Although repartitioning is still expensive because of all the network and CPU involved, we don't need to do the sorting or grouping.

Additionally, you should use repartitioning because you either need a different level of processing power or you want a different type of output.

Advanced techniques

So far we have looked at the word count example to help describe DAG operations, but for the next couple of operations we need to go beyond word count. Here's a short use case with each explanation to give you some background.

Broadcasts. The idea of a *broadcast* is to send out a copy of the data to every executing node in the processing engine. This information can be as simple as connection information; for example, to give each node the instructions to write to a remote store. In the scope of processing data, you can use a broadcast to send records from a smaller dataset. In this case, that's called a *broadcast join*, and it looks something like Figure 4-9.

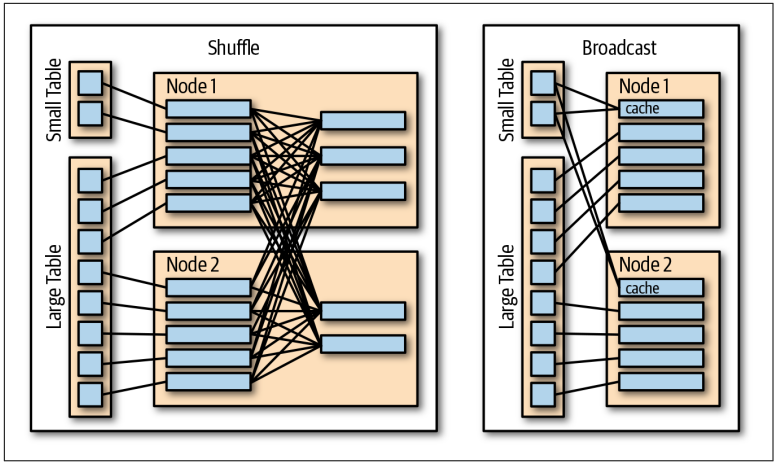


Figure 4-9. Shuffle join versus a broadcast join

The idea is that if one of the tables is small (fewer than a million records), it makes sense to first send the smaller table out to every node and then do a map-only job to join that data to the larger dataset. The result is that you don't need to shuffle the larger dataset.

Remote interactions. Another cheat that people sometimes try is remote fetching of data, which is similar to a broadcast join. In this model, you put the smaller table into a remote low-latency cache and fetch the joins when you need them. It looks something like Figure 4-10.

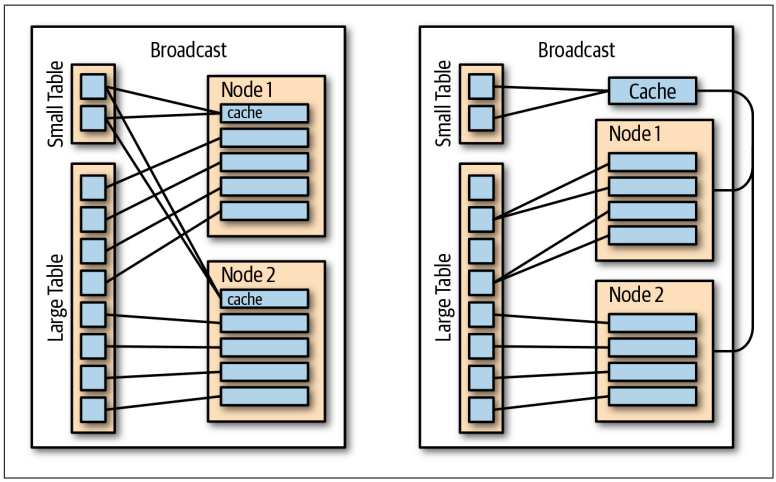


Figure 4-10. Local-to-remote cache in broadcast operation

This approach can sometimes work but is sometimes really bad. The determining factor is whether the remote cache can keep up with the load. Here are some tricks to help in your design:

Scale
: Make sure the cache can scale.

Skew
: Make sure your cache won't fall down if there is key skew.

Round trip
: Make sure the latency round trip between the job and the cache is minimal.

Batching
: The greater the round trip, the more you want to batch requests into a single send.

Overclocking
: Depending on your design, your processing system might be doing a lot of waiting for the remote call, so it might be safe to *overclock* your cores—that is, have more threads than virtual cores.

Using memory. Using memory is always helpful because it removes the need to reach out to a slower storage option; however, memory alone doesn't solve everything. Memory is constrained to a single process, so as data leaves an initial process or the local node, the benefit from memory is mostly lost. With a memory-based system, you want to focus on reduce-by-key operations in order to keep as much in memory as possible.

Checkpointing. *Checkpointing* means taking time and resources to record your state, which allows you to continue from that state if something were to go wrong with your job. This trick comes in handy if you have jobs that take more than 30 minutes.

You don't want to have a 20-hour job that fails in the last hour and requires you to repeat 19 hours' worth of work. Instead, break up the problem into smaller chunks and consider checkpointing. That way, if you do fail, you have a place to recover from that is more optimal than starting from the beginning.

Pipeline DAGs

The pipeline DAG is different from the single-job DAG in that it involves more than one job. With that difference, our pipeline DAG will look a little different, as illustrated in Figure 4-11. Each processing node reads from and writes to persistent nodes.

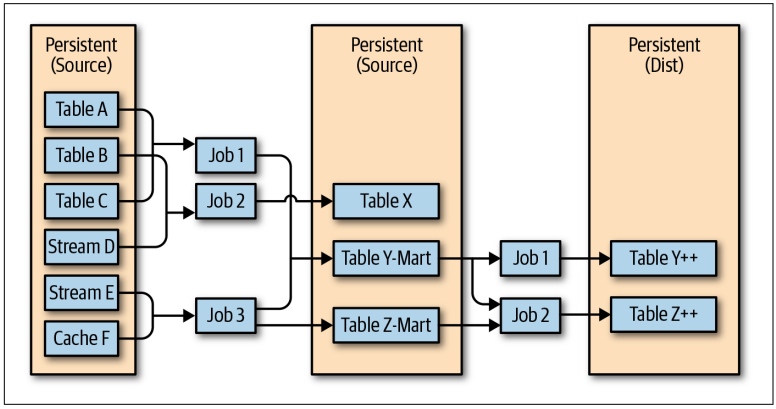


Figure 4-11. Pipeline example (difference in processing versus storage)

There are a few things to note in Figure 4-11:

No direct lines
: There are no lines from a processing node to another processing node or from a storage node to another storage node. You always go from a storage node to a processing node and back.

Start and end with storage
: You always start and end with a storage node, which implies that a processing node exists only to populate a storage node.

Storage reuse
: Sometimes storage nodes (like the data marts in the persistent gray boxes in Figure 4-11) are there only to bridge two different types of processing.

Those three points will serve as reasons to drill down into pipeline DAG optimization. The following subsections look at each one to see how the decisions around each can affect you.

No Direct Lines

Normally, a storage node does not mutate itself. Examples of storage nodes include the following:

- RDBMS
- Kafka topics
- RabbitMQ queue
- Stream
- NoSQL
- Caches
- Streaming in memory stores (Spark Streaming, Flink)
- Elasticsearch

All of those storage systems store data in one form or another. They don't mutate the data or send the data out to anyone else—their main job is to store the data. In some cases, they might notify of data mutation; however, mutation of data and moving data around is the job of the processing systems, which we talk about soon.

What About Triggers?

Some of these storage systems will enable triggers that might mutate or transmit data, but remember our perspective: from a pipeline these triggers are process nodes.

A processing node could be anything from SQL, Spark, Code, Triggers, and more. It is these processing actions that can take data, do something with it, and deliver it to one or more storage nodes.

Additionally, from the perspective of our pipeline, a processing node cannot send to another processing node. The gap between the two processing nodes must be some type of storage medium, even if that medium is as basic as a stream of messages.

Start and End with Storage

The previous subsection talked about storage nodes always being consumed by a processing node and a processing node sending its output to one or more storage nodes. With that logic, we can derive

that the storage nodes are always our beginnings and our endings to our pipelines.

That's because processing nodes are all about processing and transmission. As soon as it's done, a processing node has no state and is empty. It is the storage node that retains the value.

What About Things Like Flink? Processing or Storage?

Within our perspective of a pipeline, streaming engines like Flink are both a processing system and a storage system. While the processing continues, the processing does not have state. To illustrate, Figure 4-12 shows a typical streaming processing system.

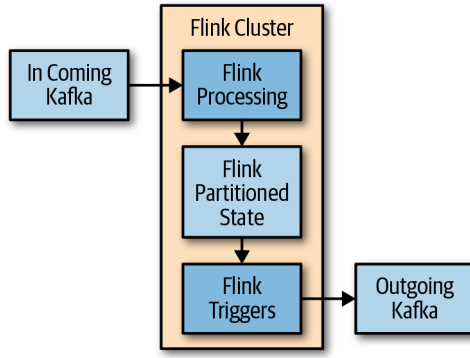


Figure 4-12. Flink state and triggering system

This really is a great example of storage to processing and back to storage. Flink reads a stream from the storage node Kafka and then writes data to its local state. Then, based on logic, Flink will trigger logic off the Flink internal key-value state and then send events through its triggering processing to the outgoing Kafka storage node.

Storage Reuse

In the pipeline DAG diagram shown earlier in Figure 4-11, there is a data mart labeled Y-Mart. This data mart is used by multiple processing nodes, allowing them to reuse the output of processes that are complex, require consistency, or are expensive to run.

Complex
: Complex logic is difficult to implement and would be wasteful to implement multiple times.

Consistent
: Consistent logic is, importantly, calculated only once. Think about a customer's balance. If you had more than one processing system process customer balances, there is a chance that they might differ, and then there could be confusion in the company about which one is right.

Expensive
: The one that is most important to processing speed is performance. A great example would be to create a data mart that is the result of a very expensive join. Many people could use the results without having to go through the cost of independently performing the join.

To recap: the idea here is to look for cases in which the cost of replicating the data is overcome by the cost of reprocessing its results more than once—the cost of complexity, consistency, or processing expense.

Summary

In this chapter, we looked in depth at DAGs, both for a single data-processing job and for executing many different processing jobs in a processing pipeline.

I hope by now that you are beginning to see your systems as DAGs within DAGs, looking for patterns and optimizations at any level of the DAGs that can move the bar at the highest-most DAG.

Chapter 5 uses this foundational information to discuss how things can go wrong with your single jobs, which will hopefully give you more things to think about while in the design stage. Chapter 6 then returns to a higher-level view to look at the running of many jobs in a pipeline.

CHAPTER 5

Identifying Job Issues

Distributed computing is great, until it isn't. The trick is understanding when your jobs have moved from great to not so great. This chapter is all about when things go wrong and all the ways that they can go wrong. I continue to focus on a single job in this chapter; in Chapter 6, we examine failures in the bigger picture of data pipelines.

My goal is that you learn to look out for different patterns of failure and stress in your jobs and in your designs. You will find that a number of the failures are a result of design-time decisions. This insight might help you build better jobs, or at least can tell you when you have one of these issues and how you might go about resolving it in a future rewrite. This chapter should help you spot issues before they manifest into problems that affect your customers—and your career.

Bottlenecks

Consider a stretch of highway that is jammed with traffic, crawling along at 5 miles an hour. That backup will slowly extend back for miles and cause frustration for hundreds if not thousands of people who had nothing to do with whatever caused the bottleneck.

In data processing, the only difference is that the expectations and impact on others can be many magnitudes more painful.

Bottlenecks can easily bring a data processing job or even a data pipeline to its knees with no mercy, bringing good code that has

passed unit tests and run successfully for weeks and months to a grinding halt. To better understand the nature and patterns of bottlenecking, the rest of this section explores some common bottlenecking patterns.

Round Tripping

Chapter 4 addressed remote caches. In the real world, there are many things that we can interact with that require a round trip, such as the following:

- A Kafka cluster
- A database
- A cache
- Security authentication
- Schema repository
- A NoSQL
- An Elasticsearch

If you must interact with one of these, be mindful of the *latency* between your processing system and the external system. Latency is the time it takes for a request to get from your system to the external system and back again, as Figure 5-1 illustrates.

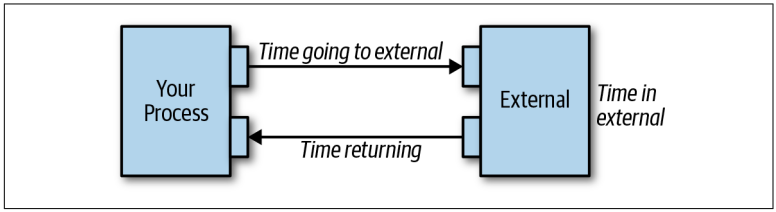


Figure 5-1. External call-time expenses

Let's take Kafka as an example. If we want to commit our offsets every consumer pull, we need to understand that the commit needs to travel all the way to the Kafka cluster and back before we can continue. This might block our code for processing. This is why Kafka has asynchronous commit options—so committing doesn't block processing.

You can determine whether this or other round trips are a problem if you see that your processing system has low CPU utilization. The

CPUs can't do their work because they are always waiting on round trips to finish.

For the most part, there are only five options for resolving round-trip problems:

Move the processing closer to the external system
: Relocating either the external or processing system can reduce travel time, but this is sometimes a difficult task.

Increase the number of threads
: A round trip is limited to a thread. The more threads you have, the more records per second can be processed with the same level of round-trip latency.

Increase batch sizes
: Send more with each round trip. If you need a dozen eggs, go to the store and buy a dozen eggs. Don't go to the store, buy one egg, bring it home, go back to the store for the second egg, and so on.

Move the round trip to a different process
: As in the example of a Kafka asynchronous commit, we can sometimes move the round-trip cost out of the main threads. This allows the main thread to move on to other tasks or even send additional submissions. These additional submissions can result in more threads performing round trips in parallel, with the result of reducing the overall time to submit the collection of submissions and better utilizing the network capacity.

Do fewer trips
: The extreme is just to try to remove the round trips. Take the example of getting a schema to process records. Maybe you can pull all the schemas in the beginning and then read all the records using a local cache of the schemas.

Inputs and Outputs

Sometimes, the problem isn't the processing system at all, but the source or the destination. Some systems were just not designed to extract or ingest data at high rates. If this is your problem, there is little you can do from a Directed Acyclic Graph (DAG) or processing design perspective. The only option is to either work with the vendor to find a faster path in or out, or change your vendor.

Another in and out problem comes from SQL engines. For most SQL engines, if you access them through Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC), the results will come to you in a single stream. It doesn't matter whether the SQL engine is distributed—the bottleneck is the pipe back to you. So be mindful and try not to query more than your little querying box should take.

Over the Wire

A common problem related to inputs and outputs is the *over the wire* problem. This is when your processing system needs to send or receive data from a remote system, but that remote system can be accessed only through a thin network pipe.

Compression

Often, the best way to attack network bottlenecks is with compression. Although compression over the wire will introduce some additional processing, it can greatly reduce the size of your data. In many ways it is a trade-off from CPU to network throughput, but in most systems today the cost of CPUs is well below the cost of an additional network, so it is normally a valid exchange.

How much can compression help? Suppose that you're using LZO and you get two times compression, or GZIP and you get eight times compression. This means that you're going to be two times or eight times faster, respectively, through the network.

I was blown away when I first learned JDBC didn't compress data over the wire—likewise when I heard people send JSON over the wire uncompressed. JSON is a prime target for compression because of all the strings and repetition of the metadata in the messages.

Additionally, if this is an extreme issue for you, you might want to think about compressing in a file format like Parquet to increase compression even more. Whereas Snappy normally gives you 2.5 times, GZIP gives you about 8 times, and BZIP about 9.5 times, with Parquet you might be able to attain 30% more compression because of the columnar approach of the file format.

Send less

Compression is great, but the benefits can get you only so far. After you apply compression, you get the results and then that's it—there is no compressing things a second time. And, unfortunately, it isn't a continuous exchange of CPU for size. If it were, you could always aim for the perfect cost balance of CPU to network.

This means that after you have used the compression trick, the next step is really to send less data.

For example, if you are pulling a full snapshot of a table every day to do research, it might be wise to instead pull just the records that have changed. Depending on the mutation frequency of your dataset, we could be talking about a large ratio of unchanged data compared to data that has mutated. Mixed with compression, you could be looking at a 99% reduction in data over the wire.

There are lots of patterns for sending less data. If the snapshot example just mentioned isn't your problem, you might want to consider the following ideas:

Nesting
: If there are a lot of repeated cells in each record, you might want to nest 10 records together, reducing the send to one-tenth.

Bigger intervals
: If you are sending heartbeats, maybe send them at a less-frequent interval.

Deterministic
: Some use cases can re-create state on both sides. If you have a deterministic use case (like a video game), you could send less data if both sides do the work and then only need to synchronize with checksums.

Move the compute

The idea of moving the compute is to take the code and move it to a location that is closer to the data. If your data is in the cloud, for example, you would run the code in the same region in the cloud as your data instead of one cloud for processing and one cloud for storage.

A good example of compute location is processing data in more than one country. Of course, you might need to for legal reasons,

and if you must compute remotely, be mindful of when the data is transferred. Don't allow the transfer to be an afterthought.

Parallelism

One of the biggest bottleneck problems in distributed computing is the amount of *parallelism* at different stages in your job. Normally, the parallelism of your jobs is set to defaults that you might not even be aware of. If you are dealing with large data, this is one of the easiest bottlenecks to fix—that is, if you can identify that it is the root cause.

Repartitioning

Repartitioning is when you change the number of partitions in your job. Sometimes you might coalesce and get fewer partitions; other times you repartition so that you can get more processing power, as illustrated in Figure 5-2.

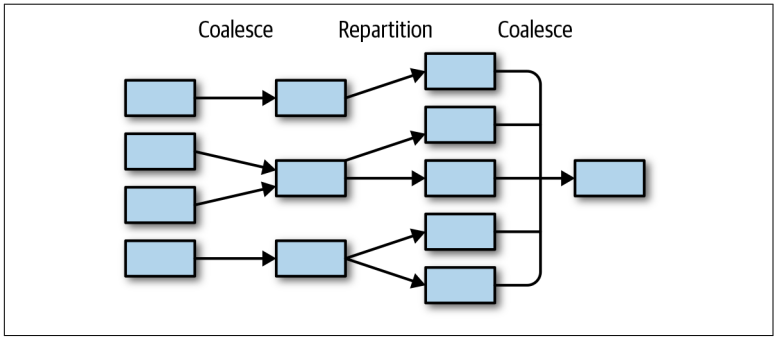


Figure 5-2. Partition examples

What many people don't know is that every time you join, group by, order, or reduce by, you're given another chance to repartition your data. If you don't take the chance to specify to the system how many partitions you want, you can be stuck with a default. Normally, a default is how many partitions you had, or it could be some random number (as in the case of Apache Spark, for which its SQL will pick 200 partitions by default).

If you don't partition properly, you can hurt yourself in a number of ways:

High compute
: If you have an area of the code that needs a lot of compute power, you might want to add more partitions, so allow for more cores to be involved in the process. Doing this wrong, though, will cause bottlenecking.

Output considerations
: Sometimes (as in the use case of writing to an object store), you want more control over how many output threads (or files/objects) that you want to create. You might want to reduce your partitions before writing. The same can go for inputs for some use cases.

Too many partitions
: There is an optimization that is lost with too many partitions. Operations like reduce by key can benefit from good-sized partitions. Getting partitions right is important.

Driver Operations

Another form of bottleneck is countering a distributed system by not using it correctly. This often happens to new Apache Spark users with putting work in the driver instead of the executors, as Figure 5-3 illustrates.

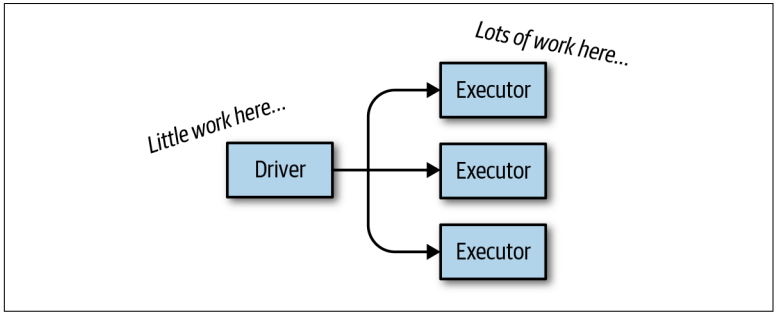


Figure 5-3. Driver versus executor work

It is often difficult for new coders to distinguish where the code is running. To get a hint about whether you are experiencing driver bottlenecks, just look at your driver. The driver's CPU should always be low, and your executor's CPU should be high. If you see the opposite, it's time for a code review.

Also look for the `collect`, `parallelize`, and `take` commands in Spark. These are all commands to take data from the driver to the executors and from the executors to the driver. If you don't use these commands with care, bad things can happen.

Skew

I briefly mentioned skew earlier and promised we would come back to it. In short, *skew* is when your key fields have a large number of values that are the same. A great example of skew is the number of people in each state in the United States. In 2018, the state of California had 37 million people living within its borders. You could add up all of the following states and get around the same number of people:

- Utah
- Iowa
- Nevada
- Arkansas
- Mississippi
- Kansas
- New Mexico
- Nebraska
- West Virginia
- Idaho
- Hawaii
- New Hampshire
- Maine
- Montana
- Rhode Island
- Delaware
- South Dakota
- North Dakota
- Alaska
- District of Columbia

- Vermont
- Wyoming

The problem with skew is that it takes away the power of partitioning. In Figure 5-4, partitioning and parallelism are reflected on the left, and skew results in the picture are on the right.

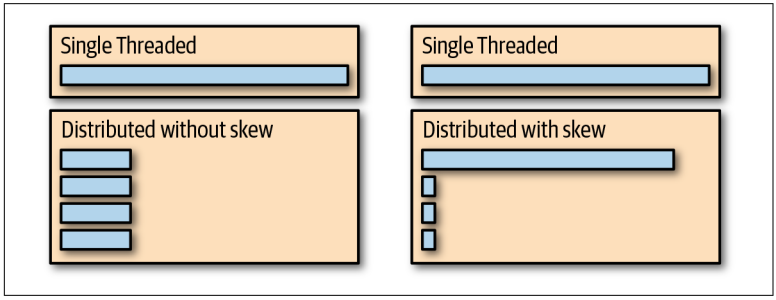


Figure 5-4. Skew example

There are a number of answers to skew, but the real trick to combating skew is knowing that you have skew.

If you take the California example, maybe you could filter out the top 10 states and give each of those states 20 threads, versus 1 thread for the remaining 40 states. Or you could pick a smaller key; for example, instead of using states, use zip codes.

Also with skew cases, the double reduce can be your friend. Do work at the zip code level, and then if you really need states, roll up a second time and take all the zip codes to the state. The advantage here is that on the second roll up, you're working with 42,000 zip codes, versus the original dataset of 330 million people.

Nonlinear Operations

The last big bottleneck is the nonlinear operation. Most of the time this turns out to be a *Cartesian join*.

If you don't know what a Cartesian join is, the short answer is that it is bad. The long answer is that it is a join that multiplies the inputs to make a much larger output. A common example is a *many-to-many join*. In a many-to-many join, you could have a case in which 100 records joins to 100 other records, which would give you 10,000 records.

From that you should be able to see how you can quickly get yourself into trouble. In the long run, Cartesian joins never end well. You might have the budget to pay for the compute and storage *now*, but depending on the skew of the data, soon there might not be enough money or processing power on the planet.

You will know if you come across one of these because all of your cores will be firing and your disks will be filling up, but your progress bar doesn't seem to move an inch.

Normally the only answer to Cartesian joins is to rethink what you're doing to avoid the joins. Use nesting, use grouping—break the problem up.

Failures

Sometimes, the problem with your job isn't the performance, but just keeping it running. There are so many things that can go wrong in a distributed environment: bad input data, node failures, back pressure, and running out of memory, for example. Let's take a moment to look through some of these failure cases to see what causes them and what might be done to help prevent them in the future.

Input Failures

Input failure is when you have data coming into your workflow that you didn't anticipate. A common example of values that can cause failures are nulls in fields that were assumed to all be populated with values.

How it will fail

In our null example, the job would fail with a null pointer exception, and depending on the engine, it might give up right there. In other cases, such as Apache Spark, it will try again three more times. Spark will attempt to retry a configurable amount of time in the hopes that the failure was because of the node or the environment. However, with a data error, we won't be so lucky, and the job will fail on the same line every time.

Depending on how long it takes to arrive at the failure point, this failure process can turn out to be a real time suck. For example, if the job normally runs 10 hours and it fails at the last minute, the job

will retry 3 more times, burning up another 29 hours or so. For this reason, if you can determine that the failure is an input failure, it makes sense to cancel the retries and avoid the waste of resources.

How to protect your jobs

This is where an Extract, Transform, and Load (ETL) job moves into the realm of regular software development, and we return to the basics of unit testing, input data quality checks, and filtering bad data into dead-letter queues.

Here are some common data quality issues you will want to check for:

- Null checking
- Dividing by zero
- Negative values when you expect positive
- Decimal precision
- Date formats
- Strings expected to be other types
- Data of the wrong schema

Environment Errors

In any distributed environment, you must worry about failure. Remember from the beginning of the book that players like Google decided that with scale it had to plan for failure. So, in general, distributed systems will try to handle environment failure, but none handle it very well.

There are the best cases, like with Apache Spark when you lose a Spark executor. In that case, Spark will try to spin up a new executor or move the workload to another executor.

However, in systems like Presto, Impala, or even Spark, when you lose the driver process or the query execute process, your job's state is lost and it results in a failure. It might try to restart from the beginning, but in some cases that is not helpful.

How to protect your jobs

If you know jobs can fail and there is honestly nothing that you can do to prevent this, how should you respond? The answer is run shorter jobs:

- Break your jobs up and consider using checkpoints
- Make your jobs run faster by improving the implementations
- Make your jobs run faster by giving them more resources

Resource Failures

Resource allocation can always be an issue with execution engines, and even more so if more than one job or user is giving the system overlapping work. The most common resource allocation issue has to be memory issues.

Memory is one of those things that is very difficult to get right. Keeping track of how much is being used and what to do when there is none left becomes even more troublesome when more than one job is running on the same system. Now the system might take down both jobs or have to decide if one of the jobs should die. Most engines will default to the easier path and just die.

How to protect your jobs

Resource issues are normally rooted in a misconfiguration that is triggered by a misuse of the system. This is important to understand because a misconfigured system can run just fine for a long time, but that doesn't mean it was configured right; it just means it wasn't misused enough to cause the problem.

Here are a couple of things to note:

Job resource limits
: Put resource limits on jobs if you plan to have more than one job running on the same system.

Job concurrency limits
: You will want to limit the number of concurrent jobs to a reasonable amount to avoid resources being divided up too much.

Leave headroom
: With systems that you need to depend on greatly, it is wise to overprovision. In the memory example, this is especially true given that Java (the language for most of these engines) likes to eat up more memory than you allot it.

Don't let a resource issue pass
: Resource failures will seem to happen randomly, and if they happen, you should focus on making sure the root cause is addressed. If you don't, you will find that random issues will continue frequently. Over time, that will reduce the confidence in the system as a whole.

Summary

By now it should be clear that there are a large number of things that can slow down or bring down your jobs. They all have different sources, and will affect you in different ways. It is important that you are able to determine how to react to the varying issues.

And remember, everything we talked about in this chapter was about the failure of a single job. In Chapter 6, we raise the stakes and talk about managing the success of many jobs.

CHAPTER 6
Identifying Workflow and Pipeline Issues

From its beginning, this book has been about extracting value from your data. That journey to value sometimes requires your data to flow through many different systems and processings. Chapter 5 was all about the failures for a given process. This chapter steps back and looks at the bigger picture, asking how to identify issues at a pipeline level.

Typically, to extract value from data, you need a number of operations to happen first, tasks that include but aren't limited to the following:

Join and enrich
: Joining and enriching the data with external data sources.

Optimal persist
: Getting the raw data in the common desired format. This could be storing the data in different storage systems for different optimal access patterns.

Feature generation
: Enriching data with logic (human coded or artificial intelligence generated)

Toxinization/Mask
: The act of removing personal data and replacing it with universally unique identifiers (UUIDs) to better protect the person the data relates to and to protect the company from legal issues.

Data marting
: The action of joining, ordering, or filtering datasets to get them to a more consumable state.

To make things more difficult, the larger an organization is, the more separated these operations will be. Compound that by the fact that you will most likely have hundreds, if not thousands, of datasets with thousands of value-producing approaches and outcomes. That will leave you with many jobs firing all at the same time, with a good deal of those jobs being on the critical path to getting value-generated data products.

This chapter covers considerations for making all these jobs run well and with minimum budgets.

Considerations of Budgets and Isolations

This section digs into the different options for isolating jobs and the pros and cons of each. This will help us get an idea of how close our different jobs will need to be and how they will need to share resources.

In general, there are three isolation options that are pretty common across execution engines: *node isolation*, *container isolation*, and *process isolation*. You will most likely be using all these to some degree, so let's take a second and dive into the details.

Node Isolation

This is when your job runs on its own nodes. This is as close to complete isolation as you can get in the distributed-computing world. With this design, there is no worry about other people grabbing your resources, giving you more control over your environment.

The downside of this approach is that you might not have enough work to use up the resources in your isolated block of nodes. From a cost perspective, that means this is the least optimal option.

Node isolation in the cloud

Though the cloud does help with a number of cost issues related to node isolation, it doesn't come free of issues. Let's take, for example,

the idea of spinning up an Amazon Elastic MapReduce (EMR) cluster for the purpose of a single execution of an Apache Spark job.

As of this writing, it takes about 10 to 20 minutes to get an EMR cluster fully up and running. Then, suppose that you run your job for 5 minutes and shut down the cluster. Here the cloud does offer a good deal of benefit because in this model you pay for about 15 to 25 minutes of nodes for that 5-minute job.

Whether you consider this model to be good or not is more of a personal decision. The fact is, this is a very common model because it's easy and safe. As an optimal use of resources, time, and money, it is dead last among our options even in the cloud world. It also leads to added issues on debugging and operations in a few areas:

- Keeping track of the logs
- Time to iterate or retry
- Ability to look in the nodes after a job to establish the issues

Node isolation on-premises

If we compare that to on-premises, you wouldn't really do node isolation, because it wouldn't make sense. It would be very difficult for someone to pitch the purchase of hundreds of thousands of dollars of hardware for one job. This is why on-premises you will most likely see only container and process isolations.

Container Isolation

Whereas node isolation aims to grab a physical instance as its boundary, the container isolation approach makes a claim within a physical instance, as you can see in Figure 6-1.

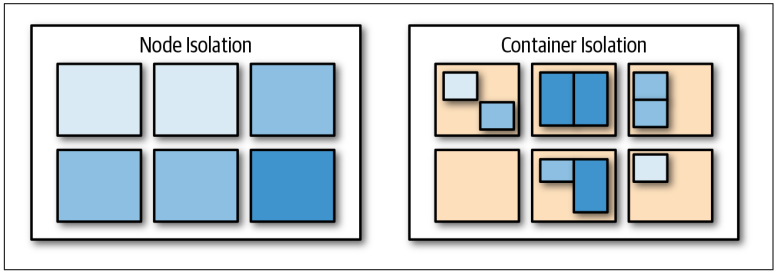


Figure 6-1. Node versus container isolation

A good common example of a container solution would be Apache Spark on Yarn, Kubernetes, or Mesos. When a new Spark job is started, it is assigned container slots on existing nodes to run. These jobs are completely isolated at a memory and CPU level from the other processes. When the Spark job finishes, the resources are returned to the container scheduler to assign to the next job.

This allows for many advantages over the node isolation option:

Avoid node creation time
: The node stays up, but the containers switch in and out so that you need to start up the node only once.

Less idle time
: With more than one workload on our shared nodes, there is more opportunity to fully utilize the resources on the node—allowing us to get more processing output for our dollar, as illustrated in Figure 6-2.

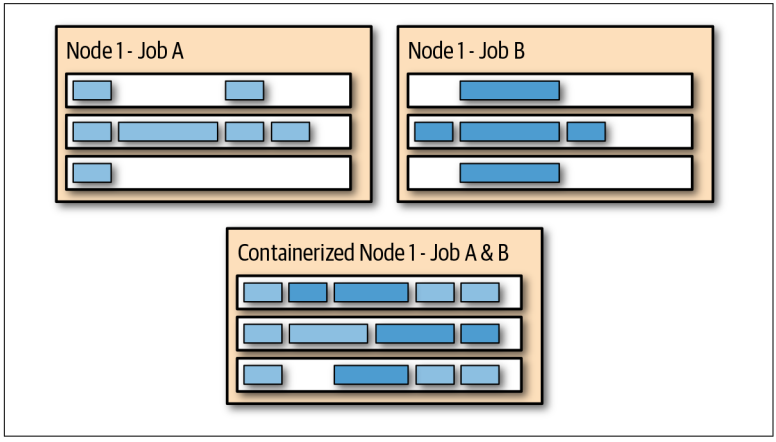


Figure 6-2. Multiple job distributions in a container isolation world

Consistent environments
: The common reply "it works on my machine" is often uttered by developers when their code has issues running on external environments. Containers can help remove that issue as the environment goes with the code, allowing for repeatable code and environment on any node, local library set, or operating systems. This also helps with repeatability when troubleshooting.

More options for auditing and monitoring
: Because we are all scheduling on shared nodes, there are tools that can help watch the execution of the processes. With node isolation, it is less common to get repeatable auditing and monitoring of the different ways different groups deploy and maintain their clusters.

Faster startup times
: To spin up a container takes seconds as compared to tens of minutes for the node option, decreasing the cost of switching from one job to the next.

Scheduling Containers

Now that we are sharing common resources, we need a way to decide which jobs run before others. This becomes a problem when we have more jobs being asked to run than resources available to handle them. If the scheduling is done wrong, in a container-based world we could have jobs be delayed for unacceptable periods of time, misconfigured jobs take all the resources, and less important jobs being put in front of more important ones.

When considering using a container isolation approach, think about the following configurations to help address scheduling issues:

Setting thresholds
: These configurable limits will force different users or jobs within the cluster of nodes to be restricted to a limited group of resources, enforcing protection of resources from being consumed by one user or job.

Setting job priorities
: Priorities can be given with weights to allow more important jobs to get more resources than less important jobs. A good example would be 2 jobs that need 100 cores each in a shared cluster that has only 100 cores to give. In an equally weighted use case, both jobs would be allotted 50 cores each, but if one job has a weight of 2 and the other has a weight of 1, the higher-weighted job will receive 66 cores and the lower-weighted job will receive 34.

Preemptive killing
: This type of configuration allows for resources to be taken away from existing running jobs to relocate them to jobs with higher

priority. If you don't have this set correctly and you have long-running jobs of low importance consuming the cluster, new, more important jobs will have to wait until the old jobs are finished before getting a shot at resources.

Kill triggers
: In some systems it makes sense to have kill triggers that discontinue jobs if they take too long so that the resources are given back. This is also helpful to set a precedent to the query runners that long-running jobs are not permitted, helping encourage better coding and design decisions to better optimize jobs or break them up.

Scaling Considerations for Containers

In the container isolation strategy, there are two kinds of scaling to consider: job-level scaling and cluster-level scaling. Neither is required, but both have the ability to help you better use resources and prepare for upcoming loads:

Autoscaling jobs
: Any given job can have Directed Acyclic Graphs (DAGs) that require more resources for one stage in their life cycle and fewer resources in other parts of their life cycle. Allowing the job to ask for more or fewer resources within its running state allows for better use of resources across the larger cluster.

Autoscaling the container cluster
: In cloud environments, it becomes possible to ask for more nodes to add to the shared resource pool and to release nodes from the shared resource pool. Though it can take minutes to add nodes, this pattern has huge savings potential for handling different resource demands throughout a given day or week.

Limits to Isolation in Containers

The last item to think about with containers is what the container can't isolate. Containers do a great job of limiting the CPU and memory access to the processes that reside within them, but access disks, remote storages, and networks are outside of a container's ability to limit.

If not monitored correctly, it is possible to suffer from a *noisy neighbor problem*, in which one set of containers will negatively affect the

performance of other containers, resulting in unexpected execution times.

Process Isolation

The last level of isolation is at the process level, which means that you can have different jobs running in the same container that belong to different use cases or even users. If we take the same diagram we used to compare nodes to containers and add in processes, we would get something like Figure 6-3.

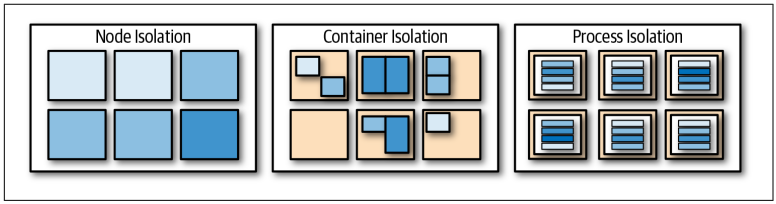


Figure 6-3. Node versus container versus process isolation

Process isolation is like container isolation in many ways but gives up running in the protection of a container for allowing an engine to handle the isolation and scheduling. Some good examples of a process isolation system are Apache Spark and Apache Presto. Both deployments allow for more than one job to run on the same process at the same time.

The big win for process isolation is that the startup time between jobs is now in the milliseconds, whereas nodes are in the tens of minutes, and containers are in the tens of seconds. Process isolation is around 500 times better at transitioning than containers, and about 3,000 times better than nodes.

It is this high transaction speed that allows for higher-concurrence environments, which is ideal for having many users execute analytical queries or process jobs of varying sizes.

Death by a Million Small Jobs

One place where process isolation has served me well is when dealing with very small jobs. It is a misnomer that in the big data world all jobs are acting on super large datasets. In fact, a good majority of the jobs will run on datasets that might need only one or more cores.

If you take the container isolation approach on these smaller jobs, it might take longer to start the job than to actually process the data. This is where the process isolation pattern really excels. By being able to transition between jobs at the millisecond level, it is giving the appropriate level of processing power to the problem at hand instead of wasting that processing power on waiting.

Considerations of Dependent Jobs

The preceding subsection dealt with isolation to make sure you're using your resources correctly and that your jobs are working in parallel. This section talks about how to make sure your jobs that depend on other jobs are firing in the proper order and integrating well. To reach this goal, we focus on three areas: *dependency management*, *integration tests*, and *resource utilization*.

Dependency Management

One of the worst signs of a poorly performing data processing shop is a lack of understanding of which jobs depend on which other jobs. The result is that at some point in time a job will fail, or slow down, or produce bad data. When that happens, jobs that are dependent on the output of that earlier job can be affected.

Dependency triggering

The first step in getting dependency management right is scheduling. We want to make sure that jobs that are dependent on others are triggered after the earlier job is finished. Even though this makes perfect sense, it is normally not done!

We want scheduling systems that allow us to interchain dependent jobs through notifications or larger DAGs that compose many jobs. As of this writing, the tool of choice is Apache Airflow, which was originally made by AirBnB.

Audit trails

After adding the dependency links, you might get the joy of creating a documented dependency graph. This offers so much at an auditing level, giving you and your company the following:

Critical paths
: With audit trails you can have visibility into your process-critical paths or long polls. This allows you to know from a total data processing front what level of SLAs you will be able to provide.

Notifications
: The whole audit map now allows you to send out failure or delay notifications to downstream processing jobs, enabling them to react and/or notify their customers of the issues.

Resource strategies
: Having an audit trail also gives you insight into what resources you're going to need, and when. If you know after job A that jobs B, C, and D will fire, you can scale up 15 minutes before job A is scheduled to finish, and you can predict when job A will finish based on its past execution times.

How to react
: Having a view of the dependency map and the critical path can give you insight into how to adjust resources when reacting to issues. For example, you might know that increasing resources for some of the jobs in the pipeline can help counter a failure earlier in the day.

Notification of build or configuration changes
: The largest chance of failure in the software world is when new code or configurations are added to the mix, and this is only magnified in a multidependent workload environment. That's why it is important to have integration testing between dependent jobs, and to make sure before you publish your changes that downstream jobs have also passed integration tests.

Summary

This chapter has shown that pipeline management and process management interconnect. The pipeline is the lines between the processes, and the success or failure of pipeline management is about isolation, dependency management, and auditing.

These seem like easy things to say—of course, we all want those things. The trick is not in the technology, it's in the discipline to put a bar of quality for these three components and have standards and

processes that enable developers to easily and quickly develop while continually using good practices.

Where everything will fall apart is if everyone decides to build their pipelines their own way, with no central auditing or dependency management. Although it might seem that allowing developers to implement independently is enabling, it will soon produce an unmaintainable tech department that could swallow your organization whole.

CHAPTER 7

Watching and Learning from Your Jobs

The whole rationale behind big data and data processing is the belief that having data gives us insights that allow us to make better decisions. However, sometimes organizations dedicated to analyzing data don't make the extra effort to collect data about *their acting* on the data.

This data about how we process data can allow you to discover game-changing insights to improve your operations, reliability, and resource utilization. The next chapter digs down into different insights we can get from our data, but before we dig into the outputs, this chapter focuses on the inputs.

This chapter also talks about different types of data that we can collect on our data processing and data storage ecosystem that can help us improve our processes and manage our workloads better.

But before we dig into the different metrics that we want to collect, let's talk about how you can go about collecting this data and changing the culture of collecting data of workflows.

Culture Considerations of Collecting Data Processing Metrics

We all know that having metrics is a good thing, and yet so few of us gather them from our data processing workloads and data pipelines.

Below are some key tips for changing the culture to encourage increased metric coverage.

Make It Piece by Piece

As you read through the upcoming list of data, try not to become overwhelmed if you are not currently collecting all this data. If you are not even collecting a majority of what I'm about to describe, don't worry. Address your collection piece by piece—there is no need to make a huge push to grab all the data at once. Select the ones that can lead you to the outcomes you care about the most first.

Make It Easy

The more important thing is to make the collection of this data easy and transparent to your developer and analyst communities. Otherwise, you will forever be hunting down gaps in your data collection. You might even want to make data collection a requirement for deploying jobs to production.

Make It a Requirement

Although this book has focused on the performance optimization of jobs, there is another angle that is even more important: making sure your data processing organization is in alignment with the law. In recent years, a number of new laws have been passed around the world relating to data protection and data privacy. Perhaps the most notable are the European Union's General Data Protection Regulation (GDPR) and the California consumer protection laws. I can't cover the many details of these laws, but there are a number of high-level points of which you should be aware:

- Knowing what personal data you are collecting
- Knowing where your personal data is being stored
- Knowing what insights you are getting from personal data
- Knowing what internal/external groups have access to your personal data
- Being able to mask personal data
- Being able to inform the customer what data of theirs you have
- Being able to remove a customer's personal data if requested

- Being able to inform a customer what you are doing with their personal data

As we review the different metrics you should be collecting, it will be easy to see how some of them will directly help you answer some of these problems in a timely manner.

When Things Go South: Asking for Data

This is a book about processing data for people who build data systems. So, it should be no surprise that when something goes wrong in our organizations, we should default to asking for the data to understand the problem more.

In this effort, it should be a requirement. Then, when looking back on issues, we demand charts, reports, and alert configuration. In addition, we should track how long it takes to gather such information. With time, as issues pop up, more and more coverage will result as a process of executing reviews that require data.

What Metrics to Collect

Not all data has the same value. We know this as we work with other people's data, and the same applies to our data. There are a number of different metric data points we can collect while monitoring our jobs and pipelines. The following subsection digs into the most important categories of these datasets.

It wouldn't be a bad idea to use the groupings below to evaluate completeness of production code and readiness. If you don't have any one of the following datasets, there is a cost associated with that lack of information. That cost should be known for every job.

Job Execution Events

The easiest and most basic metric you should aim to collect is information on when jobs are running in your ecosystem. You should be able to know the following things about any job that starts on your system:

User
: Who is requesting that the job run? This could be a person, an organization, or a system account.

Privileges
: What permissions that user has.

Job code and version info
: What version of the code or SQL statement is running.

Start time
: Time the job was started.

This information will begin to give you some understanding of what is going on in your data processing ecosystem. As I write this, I wonder how many people reading it have a clear view of all the data processing jobs that are happening on a daily basis on their data. I'll wager that if you belong to a company of any considerable size, even this first stage of monitoring seems out of reach. But I assure you it is achievable, and with the new laws it will even be required soon.

Job Execution Information

After you know what jobs are running, you can move to the next level and begin collecting information on the job's execution. This would include information like the following:

Start and end times
: Or the length of the execution

Inputs
: The data sources for the job

Outputs
: The data destination information

This input and output information will be key in developing *data lineage*. This data lineage will help you to answer some of those legal questions about what data is being processed and how did a feature get to be created.

As for the start and stop information, when it comes to performance and operations, we are going to find a lot of usage for this data.

Comparing Run Times with Thresholds and Artificial Intelligence

Another opportunity to look for issues that might not rise up as exceptions is *runtime difference*. Between runs of the same job over time, you take into account the input data and the output data.

Using thresholds is a good way to begin on this, looking at past execution times and triggering warnings or alerts to operations teams if the job takes longer than expected.

You can take this even one step further by labeling the warnings and alerts. You can even feed that data to a machine learning model to help the system learn from your reactions whether the alert is a real one or a false alarm.

Job Meta Information

Job meta information includes some or all of the following:

- Organization owner of the job
- SLA requirements
- Execution interval expectation
- Data quality requirements
- Run book meta
  - What to do in case of failure
  - What to do in case of slowness
  - What to do in case of data quality checks
- Impact of failure
- Downstream dependencies
- Notification list in case of failure
- Fallback plan in case of failure
- Deployment pattern
- Unit testing quality and coverage
- Execution tool

Data About the Data Going In and Out of Jobs

If you can capture the following information, there is a lot more you can do to identify possible job slowness and data quality issues before they become larger problems:

Record counts
: How many records are going in and out of a job

Column carnality levels
: The amount of uniqueness in a field

Column top N
: Top more common values for a given column

File sizes
: When reading files, the sizes of files can affect job performance in a number of ways, so it is good to check counts coming in and out of your jobs

Job Optimization Information

The following data collection information is more focused on job resource utilization and job optimization:

CPU utilization
: How much are the CPU cores being used on your nodes?

Locked resources
: How many resources does a job block?

Locked but unused resources
: How many resources are being reserved by active jobs but are not being used?

Job queue length
: How many jobs are queued up because there aren't enough resources for them to start?

Job queue wait time
: How long does a job wait before it is allowed to start because of lack of resources?

Degrees of skew
: When a group by, join, order by, or reduce by action takes place, how even is the execution of all the threads/partitions? If there

are partitions that take much longer than others that is a great sign that you have a skewed dataset.

Keys used in shuffles
: Whenever you sort and shuffle on a key, you should mark that down. There might be an opportunity to avoid shuffles by creating data marts instead of performing the same painful shuffle multiple times a day.

Scaling better

A big issue with scaling a cluster of shared resources is getting it right. If you scale leaving too much buffer, you are leaving money on the table because of resources that are not being used. But if you scale too tightly, you risk long wait times for jobs when the load increases.

The answer is to predict your load requirements before they hit you. You can do that by taking the load expectations from past days, weeks, and months and feeding them to a number of different machine learning models. Then, you can train these models to minimize cost while optimizing availability.

You will be able to evaluate whether you are getting better at scaling by comparing queue times and queue lengths against your unused resources.

Job optimization

Job quality is never an easy thing to define. The wrong metric is often used to rate whether a job is worrisome. For example, job execution time is not always the best indicator of a job's optimization. In fact, some engineers will proudly boast that their jobs are so big and important.

Three metrics can go a long way toward getting past that echo base defense. Let's look at those metrics again from the angle of reviewing a job's design and approach:

Locked but unused resources
: This is a good indication that the job developer doesn't understand the load profile of the job or is not parallelizing the workload correctly. If you find a long-running job that is locking resources but not using them, that is a great indicator that something isn't right.

Degrees of skew
: If there is heavy skew in a job, there is also risk of increasing skew. This could be a sign that the job is not taking into account the unbalanced nature of the data or is using the wrong shuffle keys. Heavy skew is never a good sign.

Keys used in shuffles
: If you find the same shuffle key being used many times in a job or workflow, this is most likely a sign of poor design. Ideally, you want to shuffle on a given key only once. There might be some excuse for shuffling twice, but beyond that there should be a review. Shuffles are expensive, and a repeated shuffle on the same key is normally a sign of wastefulness.

Resource Cost

In any organization with a budget, there will be concerns about the cost of workloads and the datasets they produce. Thankfully, there is a lot of data that you can collect that can help the bean counters while also providing additional information on how to focus financial resources on the tasks and workflows that matter the most. The following are some of the more important metrics that you will want to capture.

Processing cost
: This is the direct cost of the processing of the workflow. Depending on your configuration, this could be per query or a percentage of the whole in a serverless or shared environment. For cases in which you have node isolation, you might be able to allocate the direct cost of those nodes to the operator of the workflow.

Cost of data storage
: After the datasets are created, there is normally a cost to storing them. Additionally, depending on the access patterns of the data store, the cost can vary widely and you might need to count the storage count more than once if you hold that data in more than one store or data mart.

Cost of data transmission
: One of the costs that people sometimes miss is the cost of transmitting data. Though sending the data in a datacenter might be cheap, as soon as you cross regions or send the data externally,

the cost of data transmission has the potential to grow to be more than the storage and processing combined. These costs can lead to fewer moves and higher compression, but don't let this quiet cost drain your money without keeping an eye on it.

Cost of dataset
: This is really a sum of the cost to create the dataset and the cost to stage and store it. Not all data has the same value. Commonly, larger data like security logs can take the lion's share of the cost but might not provide as much value compared to customer-related data. By tracking the cost of data, you can review each dataset to see whether it is paying for itself.

Operational Cost

Not on my list of costs is the cost that comes from running your systems. Ideally, if deployment is completely automated and you never get a failure, the costs listed next would be so small they wouldn't be worth counting. In real life, though, you can lose 40% to 80% of your IT time on the following issues. In a lot of ways you can use the metrics in this list to judge the development discipline of your organization:

Resource failure
: This is a count of how often your computing or storage resources have failures, how long they live, and how long it takes to recover.

Job failure
: This is when a job fails for any reason. You want the ability to break that down into resource failures, coding issues, and data-quality issues.

Effort to deploy
: Tracking the number of human steps to deploy a new version of the SQL or data processing code or pipeline. When tracking this number, note that every human step is not only a cost on the company, but most likely a source of errors.

Issue tickets count
: A metric for keeping track of issues being reported on your data pipeline.

Issue ticket coverage
: A metric that defines the percentage of real events to ones that were covered by tickets. This will help identify whether you have issues that are being resolved by well-intentioned actors but that might be an indication of large foundational problems.

Issues run book coverage
: This metric tracks how many issues had a corresponding *run book* (repeatable instructions of how to handle the issue). This is a good indication of how disciplined your organization is related to quality and uptime.

Time to identify issue
: This metric tracks how long it took for your organization to understand that there was an issue. This is normally counted from the time the issue started to when an issue ticket was filed (manually or through an automated process).

Time to resolve issue
: This covers the time after the issue ticket's creation to the time the issue is confirmed resolved. This metric is key for understanding how ready your teams and processes are to handle events.

SLAs missed
: Issues might not always result in missed SLAs. In short, this metric is to cover how many times issues affect customers and operational dependability promises.

Labeling Operational Data

Data collection is going to give you a great deal of information, but there are simple steps you can take to increase the value of that data and empower structured machine learning algorithms to help you. This is the act of *labeling* your data. The idea behind labeling is you take an event, such as an error notification message, and allow a human to add meta information to that event like the following:

Noise
: Means this error notification is not worth my time and should be filtered.

Context value
: Means that this error has value for discovery and might have value in relation to other events.

High value
: This notification is very important and should be raised to the highest levels.

This idea of labeling works very much like marking certain mail as spam or when you tell your music app that you like or dislike a song. From your inputs, you can use different machine learning tools to build correlations, clusters, and logic.

Possible value of labeling

Labeling data is really adding hints to what you should be learning from your data. Let's take operational events for an example. What are all the things that you might want to know about your operations events?

Is the event important?
: Should you show this event to an operator?

Is this event related to other events?
: Normally, events cause additional problems, and finding the root cause of issues is typically an important factor in resolving the problem. If you could give hints in the data to make these connections easier, that would be game changing. In fact, if done right, event relations might even be able to help you know whether future issues are on the horizon.

Linking the event to a resolution
: Even more optimal would be linking events to their solutions, possibly giving the operator insight into potential actions to resolve the issue. Think about the autofill hints you see when replying to an email or text. How often are they pretty close to what you would have written anyway? It is likely there is even more repeatability and standardization in your operational events than in human-generated emails.

Technique for Capturing Labeling

Labeling data requires a certain level of consistency and coverage to be effective. To do it right, you're going to want it ingrained into

your processes. It would be best if you could capture data from the operator's steps in the ticket-making and resolving process.

Here are some things to consider when evaluating your labeling strategy:

Consistent inputs
: If possible, use simple inputs to link or flag events or records to given goals.

Multiple supporting inputters, times, and methods
: To increase the odds the input is correct, you should be able to collect inputs from different people over time. You don't want your machine learning systems getting skewed by too few results or a skew of the viewpoints of only one personal labeling.

Part of the normal operator's tasks
: The more integrated labeling is to the process, the better the odds that it will happen and be done correctly.

Evaluate coverage
: Another angle to increase labeling coverage is to monitor its completeness after every issue.

CHAPTER 8
Closing Thoughts

If you have made it this far, thank you. I tried very hard to fit a lot of content into a small package and deliver it in a short time frame. I'm sure there might be areas where you feel as if questions were not answered or detail was missing. It would be nearly impossible to reach every angle of data processing and pipelines in a 70-page book.

I want to leave you with the following key points:

A better understanding of expectations and their origins
: The early chapters looked at how many current problems stem from Excel and the mountain of expectations we are still trying to reach today.

A better understanding of the evolution of tech
: From the single node, to distributed, to the cloud, so much has changed. And data has only become bigger, more complex, and more difficult to understand.

A different perspective of the data organization's different parts
: The metaphor of the chef, the oven, and the refrigerator shows how meta management, data processing, and storage need to play together in a carefully orchestrated dance in order to make anything work.

A deeper viewpoint on the complexity of data processing jobs
: This book went deep into DAGs, shuffles, and bottlenecks. The aim was to help you understand why and how problems pop up in distributed processing.

A scary look into all the ways things can fail
: Understanding and expecting failure is the first step to a more maintainable life in development. I hope that the last couple of chapters instilled in you the importance of breaking problems into smaller parts and of making sure you have visibility into your processes from different angles.

It could be that the ideals espoused in this book seem a long way off from how your company currently runs things. The important thing is not to feel overwhelmed. Instead, focus on small wins, like the following:

- Getting one solid pipeline
- Adding visibility into your processes
- Increasing your auditing
- Knowing what jobs are firing and failing
- Mapping out a dependency graph of jobs
- Rethinking your processing and storage systems
- Measuring and prioritizing your tech department

I wish you well, and I wish you a world without 2:00 A.M. support ticket calls.

About the Author

Ted Malaska is Director of Enterprise Architecture at Capital One. Previously, he was Director of Engineering of Global Insights at Blizzard Entertainment, helping support titles such as *World of Warcraft*, *Overwatch*, and *Hearthstone*. Ted was also a principal solutions architect at Cloudera, helping clients find success with the Hadoop ecosystem, and a lead architect at the Financial Industry Regulatory Authority (FINRA). He has also contributed code to Apache Flume, Apache Avro, Apache Yarn, Apache HDFS, Apache Spark, Apache Sqoop, and many more. Ted is a coauthor of *Hadoop Application Architectures* (O'Reilly), a frequent speaker at many conferences, and a frequent blogger on data architectures.