INTRODUCTION TO DATA ENGINEERING

DANIEL BEACH

Introduction to Data Engineering

Learn the skills needed to break into Data Engineering.

Daniel Beach

This book is for sale at http://leanpub.com/dataengineeringwithpython

This version was published on 2022-01-22

Contents

Introduction

This book is all about the movement of data, specifically developing data pipelines and how to become an awesome Data Engineer.

With the rise of Business Intelligence, Data Science, Machine Learning, and the general propensity for companies to gather as much data as possible, the ability to design data pipelines has become a valuable skill.

Data engineering is an interesting combination of technical and non-technical skills, and varies from many classic software engineering disciplines. In this book I want to cover the basic topics and discuss at a high level what are the most important skills to a Data Engineer.

What is a Data Engineer?

What is a Data Engineer? That has changed a lot and will continue to change as technology is ever-changing, but there are many things that remain constant. *Data Engineers facilitate the movement of data, and enable businesses to consume that data.*

- Facilitate data movement
- Enable data consumption

The Data Engineer has become a sought-after position and unfortunately, it has not become easier to find those people with the requisite skills to do the job. Learning those skills as an individual is not exactly an easy task either. It seems the training and classes are still lagging behind the demand for real-world Data Engineering knowledge.

This is the gap I'm attempting to fill with the topics in this book. I rewound myself to my first days as a new data developer and thought about how hard it is to even know what topics to learn.

What To Expect

In this book, I want to give you the skills and knowledge to write beautiful, fast, scalable data pipelines. It's impossible to teach everything and cover every topic, but I at least want you to know, what you should focus on. Hopefully, you discover many topics that you can dive into at your leisure.

Data pipelines are so different and varied in their structure, based on technology stacks being used, but most of the concepts are the same. Some people wrongly assume that they should learn how to be a great coder, especially in the beginning, sure, that is helpful. But, as you grow in your career you will quickly realize that it's other skills that enable you to be a good Data Engineer.

- Knowledge and concepts first.
- Writing code second.

What I don't want to teach you is how to write code. You will see me using Python in my examples, and that is just for the ease of code readability. I expect you are a smart and savvy person, you reading this book after all.

The theory and ideas behind many data engineering topics are more important than how well you write code, which comes with time and experience.

Topics We Will Cover

Here are the chapters and topics you can expect to encounter.

- The Theory Of Data Engineering and Pipelines
- Data Pipeline Basics
- Pipeline Architecture
- Storage - Files
- Compute and Resources
- SQL and Databases
- Data Warehousing and Data Lakes
- Data Modeling
- Data Quality
- DevOps - CI/CD
- Servers - The Cloud

The Focus of This Book

This book is going to focus on theory, rather than diving into every detail of writing code for Data Engineering. Programming skills are built over time, if you're looking to sharpen your skills in some language, then, by all means, take some courses or classes.

Many times the harder skills to develop are those that are less obvious, the skills that come with experience. It's hard to know what you don't know when you are starting out. I will insert code snippets and examples where I feel it would make a point or concept clear.

I'm going to give you the headstart you need to help you surpass all your contemporaries and learn the skills that are central to becoming a successful data engineer. The best part is, you can do all this with Python, in which most of our examples will be written, but the choice of language doesn't matter as much as the skill sets and thought processes.

I've personally built a successful career as a Senior Data Engineer, never have taken a Computer Science class in my life, and used Python for 90%+ of my professional life.

Knowledge and Experience

Building data pipelines require a unique set of knowledge that crosses many disciplines and isn't easy to come by without specific experience. What makes it even harder is that many of the skills, like data modeling, for example, are somewhat esoteric, half art and half science. The good news is that I can help speed you down the path to success by giving you the 20,000-foot view of the topics and problems you will encounter in the real world.

I want to share those experiences, tips, and tricks in this book to jump-start you into building reliable, scalable data pipelines.

What are the topics we will cover?

### Theory and Basics
First, we will discuss the theory of data pipelines, I encourage you not to skip this section. It's important to understand where you going before you go on a trip. Next, we will dive into the basic components of every data pipeline regardless of the complexity, I call these the fundamental skills and thought processes.

Architecture and Storage/Files

These two topics will be quickly followed by architecture, it's important to know the high-level choices we make in the beginning will affect every step we take from that point forward. Of course, we will cover the basics of storage options (files). File types play a big part in data engineering, that should be no surprise.

Compute and Resources

In the age of the Cloud, we must covert compute (ram, cpu), and how to think and work with these resources. Big Data processing requires utilizing all available compute resources available.

SQL and Databases

No data engineering book would be complete without a quick overview of SQL and relational databases. Although their importance is waning today you will still find them used for meta-data and highly transactional storage systems. A data engineer who doesn't know their way around SQL queries and tuning is going to run into serious problems. Popular tools like Spark have made SQL even more popular with SparkSQL, so fundamental knowledge about querying datasets will serve you for a long time to come.

Data Warehouses / Lakes

Closely related to SQL and relational databases is the topic of Data Warehousing and Data Lakes. Although the storage layer can range from SQL Server to Parquet files, much of the methodology remains the same. Being able to provide usable analytics is at the crux of solving and providing most businesses with value.

Data Modeling

Another topic near and dear to my heart is Data Modeling. It's half art and half science, easily one of the most important topics in the book. What good is a data pipeline if the model fails to provide the needed value?

Data Quality

Probably a less popular topic, but one of great importance to the longevity and usability of data output by engineers is Data Quality. It's still a fairly new topic even in the data engineering world, with not many good tools to pick from, so I will do my best to give a good overview.

DevOps - CI/CD

Things just wouldn't be complete without taking a look at DevOps-CI/CD and the role it plays in data pipelines. It's an often overlooked and ignored part of data engineering that has a cult-like following in the great software engineering world.

Summary

My goal is to prove to you that anyone who learns the topics in this book can easily build robust data pipelines like a seasoned data engineer. I will show you the tips and tricks you can use for every data pipeline project that will have everyone coming for your help as the expert. Let's dive in!

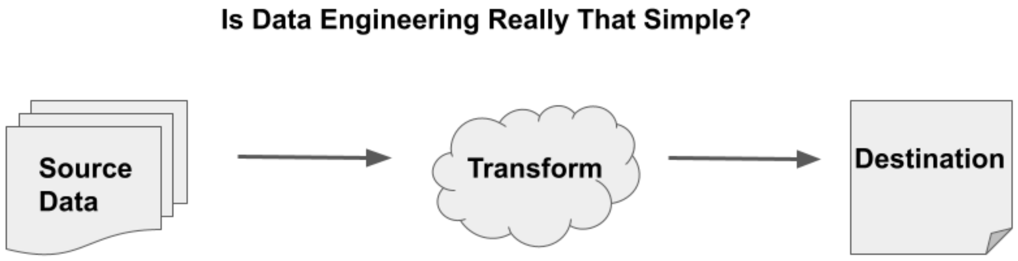
Chapter 1 - The Theory.

Why start a book about data engineering and data pipelines with a chapter on theory? I encourage you not to skip ahead to the next chapters without reading this. Building a solid understanding of why we need data pipelines and what they should accomplish will end up driving every decision made along the way. Every adventure requires a plan and some foresight and your journey into the world of data engineering will be no different.

Theory can be dry and I will do my best to be to the point. Having a high-level understanding of what we are trying to accomplish with our data pipelines should make writing them a little bit more manageable.

What Is a Data Pipeline?

I believe it's important to stop and ask simple questions sometimes, it's easy to miss the forest through the trees. What is a data pipeline? Simply put, moving data from point A to point C, with a transformation B in the middle is a data pipeline at its core.



The movement of data from point A to C with some transformation B.

You might be tempted to think that it's simply the movement of data, but it's more complicated than that. I would rather say that a good data pipeline is about …

“facilitating the movement, storage, and access to data in a repeatable, resilient, and scalable manner.”

These are the main pieces and parts you should think about when it comes to data engineering and data pipelines.

- Movement
- Storage
- Access
- Repeatable
- Resilient
- Scalable

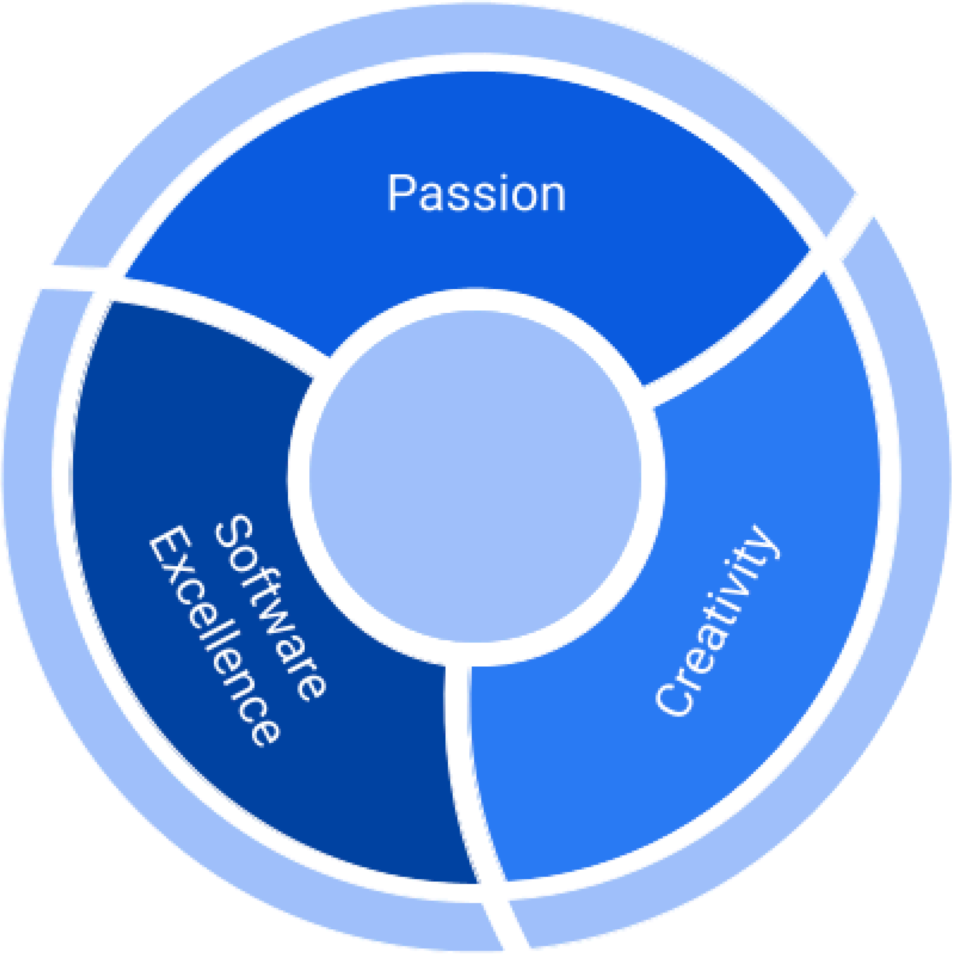
A little more than you thought? Maybe you think I’m nitpicking and throwing out jargon?

I would argue the difference in understanding a data pipeline as “just the movement data” vs “facilitating the movement, storage, and access to data in a repeatable, resilient, and scalable manner” is the difference between an amateur, broken, and unworthy pipeline and one built by a professional that will be in place for years.

Data Pipelines built with Passion and Creativity

The difference between being passionate about what you working on vs doing something for a paycheck is large. When you were in middle school and your mother told you to clean your room or not go out with your friends, I’m sure you “cleaned” it. But did you really clean it? Your heart wasn’t in it.

- Data engineering and pipelines are complex.
- The business relies heavily on functioning pipelines.
- Heart and passion for data goes a long ways.
- Creativity produces better results than rigidity.



Creativity is the spice in life, both in coding and leisure. Data engineering is a hard profession to master, not thinking outside the box and being rigid will curtail your career and results. Complex problems require creative solutions, some of the best data engineers come from non-software backgrounds.

Truly good data pipelines are built with heart and love. You have to see it as more than just another task that is going to make your boss happy. If we see the movement of data in a system as a simple, meaningless, and overhead task, the obvious result is going to be lackluster and the codebase will fail at a critical moment.

Worse, it will compromise trust from business units and lead to incorrect results. Let’s examine our theory of what a data pipeline is and how it will affect the code we will write later.

Data Engineering Standards

- Data Engineering is not “beneath” Software Engineering.
- Pipelines conform to the same level or higher expectations as normal SE code.

There can be an attitude in tech that Data Engineering is somehow beneath Software Engineering and doesn’t ascribe to the same standards. This is a crucial mistake. Data engineers should hold themselves and the complex pipelines they design to the same high level of standards.

Let’s dive into more of what this means and looks like day-to-day.

Movement

The movement of data is basic to the data pipeline. *We are by definition picking up data from somewhere and dropping it off at “home”, with some number of stops in between.* Therefore we can’t brush past the topic of data movement. In Data Engineering some movements will be complex and some will be simple.

For example, reading a CSV and doing something every row is about as basic as it gets. A very small but simple data pipeline.

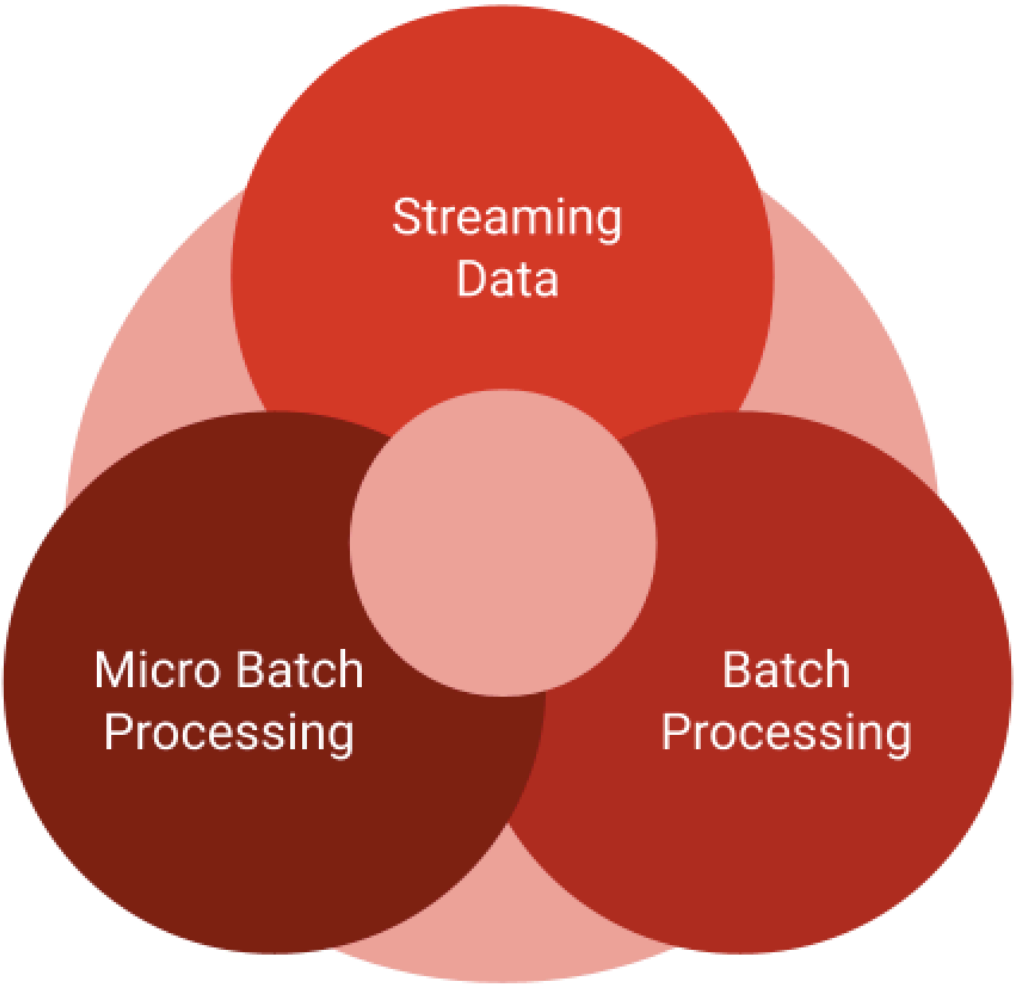
```python
import csv

def open_csv_file(file_location: str) -> object:
    with open(file_location) as f:
        csv_reader = csv.reader(f)
        for row in csv_reader:
            print(row)
```

Usually, the life of a Data Engineer isn’t so easy though.

There are a few obvious topics that come to mind when thinking about moving data from point A to B. Some of the most common, you are probably are familiar with or have heard of …

- Streaming data (real-time).
- Batch processing.
- Micro batch processing (near-real-time).



Don’t be fooled, the basic choice at the foundation of building data pipelines will have a huge impact on how you design the system. Your code will take shape around the technology choice you make here. It’s something you have to think about, “How will my data move point A to B?”

- Technology choices affect code and architecture.

If you choose a messaging service like Pub/Sub, Kafka, Pulsar, etc, that code is going to look completely different and have a different setup requirement than if you decide to push CSV files to Parquet files with Spark. Those options couldn’t be more different.

Complexity in Data Pipelines

When considering what the movement of the data should look like, *remember complexity is a killer.*

Do you already have numerous custom complex transformations that need to take place? Will adding a complex data movement or storage option make the codebase nearly unapproachable? What are the speed requirements of the project? Is some nominal data loss acceptable?

- Data Engineers should reduce complexity.
- Don't over-engineer solutions.

If your dealing with financial data, sure a messaging system where it becomes hard to "lose" data might be a good choice. If you're dealing with terabytes of satellite imagery, missing one image won't have a reasonable effect on the outcome.

Sounds like a lot of jargon just for deciding the data movement stack you will choose for your pipelines?

The point I'm trying to make is that your code is probably a pyramid of complexity. The foundational decisions you make upfront will direct the type and scope of the code that is needed to implement the rest of the project. Don't take it lightly.

Again, the movement of data by code is fundamentally what Data Engineering is all about. We know we have to move and act upon the data, it's the details of how to implement that data movement is where the Data Engineer provides value.

We will explore this topic more later.

Storage and File Types

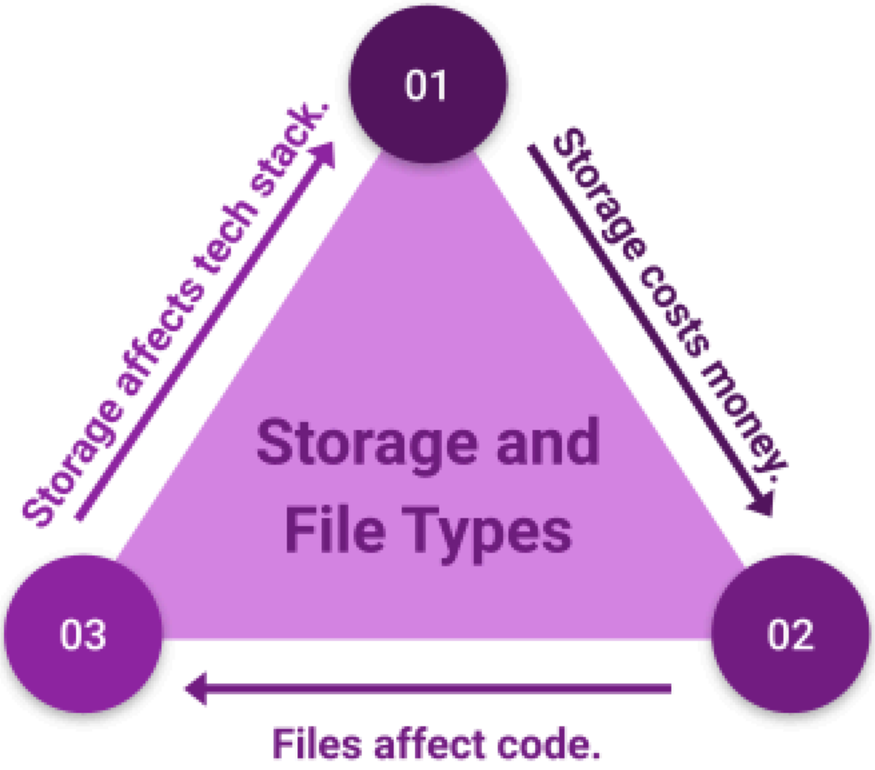
Don't forget our definition of data pipelines. "Facilitating the movement, storage, and access to data in a repeatable, resilient, and scalable manner."

Very closely related to data movement that we just covered is storage, or *data at rest*. Obviously, with the rise of cloud storage options like s3, gs, azure blog, etc, it's been much easier to ignore choices of file and data storage. But, just because it's easy doesn't mean that data pipelines won't depend in many ways on the type of storage chosen.

- Storage affects both time and money.
- Data rest (file) storage affects technology and code access.

The difference between a CSV file and parquet is enormous (we will explore details later), especially as data grows.

Again, many of the file storage choices can be hard to back out of once baked into the software solution. Understanding the data that is flowing and how it is used will always give clues to the storage option needed.



Storage and Files affect Technology and Code

I've seen tens of millions of individual JSON files stored in cloud buckets, and then a year later analytics need to be done. Of course, at this point, even a massive AWS EMR cluster was having trouble reading tens of millions of individual JSON files to retrieve needed business analytics. Obviously, for that project, someone chose JSON because it was easy, and the consequences were large and required a sizable project to regain insights into the data that could have been avoided by proper storage upfront.

Storage can be seen as just an in-consequential decision when building data pipelines, but this is rarely the case once the project is popular and starts to scale.

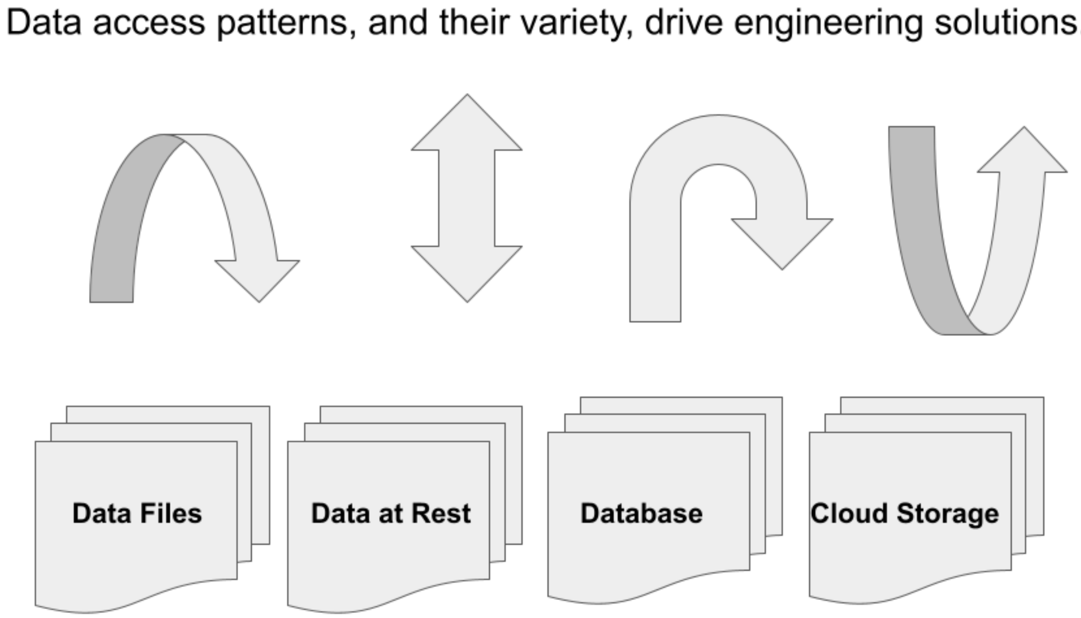
Access

Continuing on with our definition of data pipeline, we come to data *access*.

What does access have to do with data pipelines? Everything. The data output from a pipeline or intermediate steps should be easily and quickly accessible. Why is this so important? Well, what happens when the inevitable bug or business requirements change and you need to troubleshoot?

What happens when the pipeline has scaled to hundreds or even terabytes of data? You will care about data access then.

Simply put, how do people or machines consume the data in question? Data is no good if it cannot be used.


Data access patterns, and their variety, drive engineering solutions.
Data Files
Data at Rest
Database
Cloud Storage

I’ve seen complex projects completed by some very smart people, with code that looks like it should be admired. And those codebases solved very hard problems. But they failed in the end with no adoption even within the same engineering group.

Why? The data access layer was impossible to understand and interact with. It made the whole project useless. Don’t let your project or pipeline fall into that hole.

- Data is of no use without reasonable accessibility and usability.

When you design a data pipeline never forgot about how the data will be accessed and explored. This comes down to how and where the data is stored at rest, and then presented for use by consuming users or applications.

Moral of the story? Choosing storage and file types is one of the most important topics and with far-reaching consequences when designing data pipelines.

Repeatable

Repeatability is arguably one of the most important pieces of any data pipeline. Every pipeline codebase must meet these criteria, it has to be repeatable. What good is the code if the author is the only one who understands, can troubleshoot, or even worse, run the pipeline? This is more common than you think.

- Anyone must be able to run a data pipeline.
- Anyone must be able to troubleshoot a pipeline.
- Pipelines break, they will have to be re-run.
- Pipelines should produce the same results (idempotent).

Some pipeline breaks, everyone is running around like a chicken with their head cut off because that "one person" is on vacation today, and no one else knows what to do.

People and Assumptions Affect Pipelines

Engineers will many times design into their code assumptions about the world that data lives in. It makes the code unreadable, and anyone should be able to clone the code repository, look at the README, and with a few lines at the console, kick off the pipeline.

Anything more inherently means the pipeline is not repeatable. It shouldn't take more than one button click or command line argument issued to re-start or run any data pipeline.

There should not be a myriad of configurations, flags, and files that must be staged in certain ambiguous places for the data pipeline to run.

A good run of thumb is this… if you can't schedule a CRON job to run a single Python file with a main() function and a few arguments, which are noted in the README, then the code is probably not repeatable.

If no one but the person who wrote the code can run it… that is a problem. This is another important subject and we will explore it more later.

Resilent

This might seem obvious to some, but to others, it isn't something they've had experience with. What do I mean when I say a data pipeline must be resilient? I mean the codebase must be written in such a way that it isn't fragile.

There should not be many instances of "hardcoded" values, like dates or numbers in the code. This inevitably means that an engineer will have to make a code change just to run a pipeline for some historic data. Is there no try: except: blocks? This probably means the author didn't think about the pipeline in a resilient way.

Becoming ready for the unknown is important in life and code!

A good data pipeline should be specific enough to get the job done efficiently, but not so specific that minor changes in requirements don't lead to codebase re-writes and refactors at every turn.

Having a resilient pipeline means thinking ahead, without over-engineering a solution. This can end up being half art and half science but it's an easy trap to fall into.

Thinking outside the box and not making assumptions like "I will assume this file always has records in it and is never empty," always make a pipeline less resilient.

Scalable

Last but not least, and probably the most important, scalability is the name of the game. A data pipeline that isn't scalable isn't a pipeline at all,
it's just a one-time script that is doomed from the beginning.

One of the most common mistakes and time wastes I've seen is someone taking a few weeks to write a codebase to solve a problem, testing and testing, releasing the code. Only coming to realize in production when scaled up it either crashes or is so slow as to be useless.

- Data pipelines should be built to handle more data … and more.
- Scalability issues leading to poor performance are common problems.

You can rarely write a pipeline to deal with one piece of code and have it scale up without significant changes. Scalability has to be one of the main tenants driving the creation of the pipeline from the beginning.

When your thinking about a piece of data working its way through the pipeline, ask yourself, what if this record has 10 million friends? Scalability affects file storage options, the processing framework, and everything in between.

In Summary

"Facilitating the movement, storage, and access to data in a repeatable, resilient, and scalable manner."

It's easy to get caught up in the mundane and day-to-day data engineering work of writing the next ETL script. But, I suggest you take a different approach, looking at problems and resolving to make good decisions, not to rush design, and to think through problems.

Reminding yourself daily that your pipelines need to move data efficiently, and what file types are chosen do matter is will save you headaches down the road. Ensuring every pipeline is easily repeatable and resilient from the start will come in handy when something breaks at night or on the weekend.

Of course, when the business starts to scale and your data needs to keep up, you won't be left holding the bag! It's very tempting to get caught up in this and that new technology, chasing the shiny new toy. It's easy to say "been there, done that," when it comes to the next pipeline.

I encourage you to remember these basic ideas we discussed and use them to drive all the decisions you make when tackling the next problem.

Chapter 2 - Data Pipeline Basics

Finally, it's time to dive into the inside workings of data pipelines, what we've all been waiting for! Let's start by talking about the basics of building data pipelines. There are a few tips and tricks that will help you generalize all the pipeline projects you work on.

I consider these things basic 101 level requirements. Chances are if you don't follow these approaches or some variation of them, you're going to struggle to produce reliable and repeatable pipelines that you can be proud of.

The Best Pipelines Start with a Solid Foundation

I have no intention that this book teaches you how to write code, I will leave that for others. What I do want to teach you is how to best write data pipelines possible using best practices, I will use Python in my examples, but these practices can be applied to any language.

It's more about the exercise and thought process than it is about what your language of choice is. Don't let people tell you that X or Y language is better because it's "faster" or some other reason. The more you write code and work on software projects you will realize it's more about the implementation of the code, than the language chosen.

I want to review these data engineering coding best practices that you will run into out in the real world. Practices that will make your life, and the data engineers' lives that come after you more bearable.

- Project Structure.
- Testing.
- Documentation.
- Containerization.
- Architecture First.

These are just some basics that will assist us going forward, as they are the core of what will drive the design and layout of all your data pipeline projects. I know you might be getting tired of the theory, but this is where the theory starts to meet the reality of writing code to move data.

This isn't a book about how to write code, so write code as you like, but adopting a few well-accepted styles can have a positive impact on your success at delivering highly scalable and efficient pipelines.

Project Structure

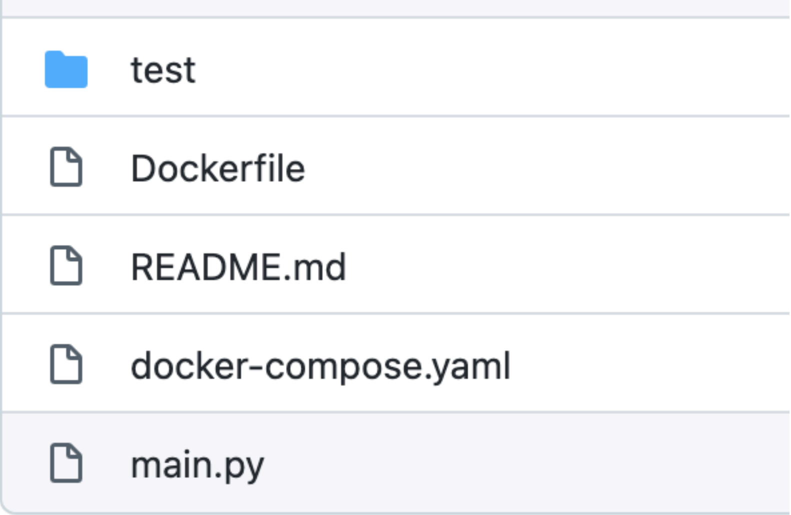
When starting our data pipelines in Python, or any language, we should follow a clear and concise design pattern that shows any reader the flow of the data through the code. Just because something is complex doesn't mean it's acceptable to have our program and data flow is hidden into obscurity.

Be short and to the point when writing code, the data should flow through the code in an obvious and human-readable way. That's why we are using Python!

We will dive into DevOps later but anytime you start a new codebase/pipeline, or work on an old one, you should ensure a few basics. Always start by creating the following resources.

- README.md <- a few well-written sentences can save someone hours.
- Dockerfile <- they are easy to use and level the playing field.
- Tests folder <- as a gentle reminder that tests are important.
- Requirements.txt <- project dependencies should be captured somewhere.
- Folder structure to divide the codebase logically.

Let's see what a very basic project structure might look like, sometimes a picture is worth a thousand words.



```
test
Dockerfile
README.md
docker-compose.yaml
main.py
```

It's a contrived and simple example in the above picture, but it's obvious the `main.py` is where we should look for the code, `test` probably has our tests, there is a `Dockerfile` etc. More than anything we are looking for a project to be *clear*.

Here is an example of what you don't want to see when looking at a new pipeline codebase.

```
run_database.py
general_functions.py
data_file.csv
read_methods.py
configurations.json
```

You don't know where to start, what to look at, what is important or what isn't. It's just messy and not obvious, be clean and organized in your code as well the project structure itself.

What would a better project structure look like applied above?

```
README.md
Dockerfile
requirements.txt
utilties/general_functions.py
configs/configurations.json
sample_data/data_file.csv
database/run_database.py
read_methods.py
```

Not perfect but I'm sure you get the idea.

Build On a Solid Foundation

You might not think this has that much to do with data engineering pipelines, but it does. A solid foundation is a key to the future success of any project. Taking the guessing game out of your pipelines is one of the best things you can do.

There is nothing worse than going to a repository in GitHub and finding random files and folders littered everywhere with no apparent consistency. Remember what your mom always said … "clean your room!"

- You can't have good pipelines without a solid base.
- You have to do the small parts well, to make the whole better.

Before writing the so-called guts of our codebase, large or small, it's important to get a few things right. I do understand that there are many different styles of writing code, but there is no excuse for sloppy and dirty code and organization.

Pipeline code that is thrown together haphazardly is probably code that was also written poorly, not tested, and will break often. Taking ownership and pride in a codebase is important.

So, let's talk about a few basics of laying out data pipeline code. I'm going to use Python, but again, apply these concepts to any language you like.

Let's dive a little deeper into structuring the actual codebase of a pipeline.

Code Structure

If you've never written a data pipeline before, or are just starting, where do you even start? There are so many steps to writing a complete data pipeline end-to-end, how do you accomplish the task? One small step at a time, starting with the basics.

It all starts with main()

Every pipeline should start with a `main.py` file and a `main()` function.
This might sound picky but it's essential to be clear from the beginning, and that starts with a `main.py` file with a `main()` function that anyone can go to for a start. There is nothing worse than opening a file with 20 functions and having no idea where to look.

Or even worse than that, just a bunch of files all in the same directory, each with a different name, and you have no idea where the entry point of the codebase is!

You should always start with something along these lines.

```
>> main.py
if __name__ == '__main__':
    main()
```

I mean call it `entry_point()` if you have to. Just give the obvious indication of where in the world to start.

Good Coding Practices make Good Data Pipelines

I can't stress this enough and I want to make a point here. This is a book about coding practices, but you can't be a successful Data Engineer without embracing a mindset of clean and concise code structures that are easy to read and use.

Your data pipeline is only going to be as good as the code you write. If it's hard to read and follow, then it will leave a bad taste and impression, regardless of how good a job you think you did.

The Code Show Flow with the Data

Simple, but important. We should continue this pattern of obvious code and data flow. Your pipeline code should follow the data flow in a sense. The methods and functions should be named to indicate what they are doing and should be called inside your `main()` method in an obvious pattern.

As the data flows, the code flows, and vice versa.

Simply reading a `main()` function should tell you what's happening to the data and where it is coming from and going too.

Example

Let's say we have a very simple pipeline to read CSV files, get some metrics from the files, and store those results into a SQL database.

- Read CSV files.
- Get metrics from files.
- Store results in SQL.

Don't Focus on the Details Right Away

It can be tempting to start thinking about the details or writing code right away. We should shy away from this practice. Let's sketch our code the way we would sketch a drawing. It helps to understand the work that lies ahead of us and will help keep us on the straight and narrow as we develop the pipeline.

It doesn't matter very much if you write Object Oriented Programming (OOP) or Functional programming with lots of little methods. If you are writing ETL or data pipelines, your code should show the flow of the data when read.

In the above example, we talked about a simple program to open CSV files, get metrics, and put those metrics into the database. The code should be framed in such a way this is obvious with a cursory glance.

```
def main():
    calculate_workload()
    download_csv_files_in_parallel()
    stream_files_to_memory()
    calculate_metrics()
    save_metrics_to_database()
```

Why write code like this? Because it's obvious how the data flows through the program. It also gives a high-level understanding of what the pipeline is going to look like before every bit of code is written.

- Data flows through a data pipeline.
- Human readable data pipelines increase productivity and reduce bugs.

This is a great advantage because it will help us think about how scalable, efficient, and extensible this pipeline will be. I can't emphasize this enough. If you just started by working on `calculate_workload()` and not thinking about what comes after or before, you're on the road to a life of hardship.

You get caught up in "how am I going to read this data", and writing the code straight away, your missing details about what is coming after that could change how you write the code you working on!

Code Readability and Organization

Again, every `main()` function should show the flow of the data through the pipeline in an obvious fashion. Also, if the different steps you will be designing throughout the pipeline require 5 or more functions, you should consider breaking those out into a separate Python Class or file if you writing only functions.

This makes your code more reusable, maintainable, and easier to debug. It gives context and doesn't overwhelm someone else, or you when debugging or feature addition happens.

So perhaps in the above example, streaming files into memory could be a few methods/functions.

```
>> stream_csv_files.py
import csv

def open_csv_file(file_location: str) -> object:
    with open(file_location) as f:
        csv_reader = csv.reader(f)
             for row in csv_reader:
                 yield row

def stream_csv_rows(rows: iter, transform_etl: object) -> iter:
   for row in rows:
       transformed_row = transform_etl(row)
           yield transformed_row
```

This is a simple but powerful idea. Don't have the bad habit of writing data pipelines all in a single file. It isn't helpful and will most certainly leave yourself and others confused as the code grows, group your code in logical units.

- Break up functions into the smallest units possible.
- Write data transformations that are re-usable.

You should not write functions that are 50 lines long. This means your not breaking up your code into small enough logical units.

Functions and methods that are working on data and contain too many lines of business or transformation logic are going to be prone to error and impossible to maintain or troubleshoot.

Tests.

We will talk more about testing in the DevOps chapter, but this topic is so important it's worth a cursory overview now this basics chapter.

You need to have tests for every data pipeline. Why? Because if want to be cut above the rest you will follow best practices every step of the way, even when you don't have to.

- Tests are basic to data pipelines.
- Tests protect pipelines from bugs and errors.
- Tests increase development productivity.

Think about it, if you change code how are you supposed to know it works and that you have not broken something you don't know about, especially if you are new to a codebase?

Unit Testing

Unit tests are the first line of defense. No one should have to run a pipeline to be able to know if it's working or not. Changing the business or transformation logic acting upon data should be tested in such a way that anyone can attempt the change and run tests with a reasonable assumption that unit tests will catch any problems.

Unit tests will not catch every bug, they never will, but they will catch some, and they will protect you from those "silly" mistakes we are all prone to making.

In our above example, we would have at least …

```
    tests()
        test_download_csv_files_in_parallel()
        test_stream_files_to_memory()
        test_calculate_metrics()
        test_save_metrics_to_database()
```

Every language has its own popular testing frameworks, for Python that would be `pytest`. Any Google search or YouTube video can introduce you to the basics of testing for any language and have you writing your own in an hour or less.

Many times it's critical when testing data pipeline code to have example and sample files to process, and the sooner you generate them and start writing tests the better.

It's not a good idea to wait until you are done before thinking about tests.

Many times writing unit tests for the method or function you are writing right away will cause you to refactor your code, to make it simpler and more testable … this is a good thing.

```python
import csv

def open_csv_file(file_location: str) -> object:
    with open(file_location) as f:
         csv_reader = csv.reader(f)
             for row in csv_reader:
                 return row

def test_open_csv_file():
    test_row = open_csv_file(file_location='sample_file.csv')
    assert test_row == [1, 2, 3]
```

If you're new to testing just take some time to read some articles and watch some videos for your language of choice. Once you get into the habit of writing unit tests you won't look back.

Documentation

Having a README.md file in every ETL project isn't asking that much. No matter how well we think we write code, no one, including yourself six months later will remember every nuance and reasoning that caused code to be written in a certain way with certain assumptions.

The `README` is your chance to get these important ideas across.

If your company uses something like Confluence, even better! But everyone has an opportunity to add a `README.md` file to a new or existing pipeline project you working on.

A good place to start with `README.md` contents is here …

- Give a high-level description of what the code's doing.
- Describe the input and output datasets, where they come from, where they go.
- List any reasons and business requirements that drove decisions when writing the code.
- Explain how to test and deploy the code.
- Talk about any configurations or other arguments used in the code.

Even a small amount of context given at the right time to some new engineer who's about to debug code they've never seen is going to be a lifesaver. It will make you a hero.

I know documentation can seem like a boring topic, but it's better to have something rather than nothing. It's better to read a few sentences before digging into a project, to get the general idea of what's happening, rather than having to guess.

Containerzation

You're just going to have to learn to use Docker, containers are the future and are the reality today. This is just as true for a data engineer as is it is for a software engineer. There is no easier way to make a data pipeline repeatable and reliable than to have a Dockerfile to run the code on. It removes all system ambiguity.

Containers are a great equalizer when it comes to systems, requirements, and dependencies. You can essentially freeze every detail about where and how your code runs, assessable to yourself or anyone coming on later to work on a project. I hope this importance doesn't escape you.

I know that some things can feel like a hill too big to climb, especially if you have never used Docker before, containers can be a little bit of black magic. But, I promise it's not that bad.

What does a Dockerfile do for a data engineering project or pipeline?

- Anyone can run the code.
- Anyone can test the code.
- Reduces the chances of something besides the code itself breaking the pipeline. (System updates, new packages, etc.)
- Makes the pipeline more scalable because today's distributed systems are embracing containers.
- Makes the pipeline easier to run in the Cloud, for the same reason as above.

Let's look at a quick example of how Dockerfiles and containerization can remove ambiguity and level the playing field in data engineering. When coming to a new code base for the first time it can be helpful just to review the Dockerfile to see what tools and systems are installed.

Just doing that simple task will usually give you a good idea of what's to come!

```
    >> example Dockerfile
    FROM ubuntu:18.04

    RUN apt-get update && \
    apt-get install -y default-jdk scala wget vim software-properties-     common py\
thon3.8 python3-pip curl unzip libpq-dev build-essential     libssl-dev libffi-dev p\
ython3-dev && \
    apt-get clean

     RUN wget https://archive.apache.org/dist/spark/spark-3.0.1/spark-3.0.1-bin-hado\
op3.2.tgz && \
    tar xvf spark-3.0.1-bin-hadoop3.2.tgz && \
    mv spark-3.0.1-bin-hadoop3.2/ /usr/local/spark && \
```

We can see from the above simple example of a Dockerfile we have an Ubuntu image that gets Java, Scala, and Python installed. We can also see Spark being installed, this gives us a great idea of what this project will be all about, Spark!

Also, it levels the playing field in the sense that the next Developer who comes to work on this project doesn't have to spend a day fighting his machine getting Spark installed before he can start work.

I'm not going to teach you how to become an expert with Dockerfiles, that isn't the point of this book. But I suggest you spend some time learning how to write tests and wrap your projects in a Dockerfile. There are plenty of good resources to help you get started, just Google it!

We will dive more into some high-level Docker examples for data engineering in the DevOps section later.

Architecture First

Surprise, surprise we are talking about architecture, are your eyes glazing over yet? I know you might be rolling your eyes and saying "get on with it already." But, you must remember you can be your own worse enemy. Diving into writing code right away without working through the architecture first is a big mistake.

We all know that code is written with certain assumptions and around certain tech stacks. Writing code is going to run on a Lambda vs code that will be running via a Cron on a EC2 box is probably going to be different. So, we should probably think through the differences first before writing code.

- Architecture before code always.
- Think before you act or write code.
- Ask questions early and often.

Pipeline Architecture Starts with Questions

This is more than how the code is laid out in the files as we talked about earlier. This is a thought process that begins before a single line of code is written and that usually where projects live or die.

I look at the topic of software and data pipeline architecture like that old saying your parents preached at you … "think before you act."

We are comfortable with certain design patterns and ways of solving problems. We usually rely on our history and this is about how we've done something similar in the past. This can be an ok approach, but probably not the best.

It's best when starting a data pipeline or ETL project to take the 10,000-foot view. Ask yourself a few questions.

- What am I trying to accomplish?

- Do you have a tool or tech stack already that can solve this?
- What would be the simplest approach?
- What would be the most complex approach?
- What are the trade-offs of the above?
- How does this plan fit into the bigger picture of my org?

These are just a few questions to get you started. The best thing you can do is try to poke holes in your ideas and design.

Being able to "step back" from the problem and think high level about pipeline and data architecture seems like a lost art. Too many people rush into decisions and architecture because it's familiar and safe. Don't do that.

The decisions we make upfront have a big impact downstream.

Review

Let's do a quick review of our data pipeline basics.

- Project Structure
- Testing
- Documentation
- Containerization
- Architecture First

These are some of the core tenants of developing top-tier data pipelines that some probably see as "not the point" or as "mere peripheries" of the development work.

I can assure you this is a bad stance to take. Getting certain core principles and practices in place before the real work begins will change the trajectory of the entire project.

Most people say "I will get to that later," and "we can come back to get that." This rarely happens. Typically when a project or codebase starts with a mess of files and no project structure it sets a precedence that the code is no good either.

When you first discover a data pipeline that doesn't have a single test written for it you will be scared to touch it, who knows if something will break?

Also, when you find no documentation or even README it's easy to get overwhelmed, how do you know where to even start? What is the entry point for the code, what is the background, what technologies are used? The amount of questions that can be answered with some simple documentation is amazing.

Never forget containerization with Dockerfiles and how it puts everyone on the same page when it comes to development and requirements. Without containerization, you just rely on people getting things installed on their machine, Apple, Linux, Windows .. who knows all the headaches that will happen?

Of course, it all begins and ends with architecture first. The classic mistake most developers fall into is just jumping in feet first writing code without looking back. Do yourself and others a favor and save time and headaches by working through the ins and outs of the technical details before writing the actual code. It will make your life easier in the long run.

Chapter 3 - Pipeline Architecture

Data pipeline and platform architecture is an important topic that many data engineers shy away from. We all fall into the pit of getting too excited too fast and going straight to writing code, without thinking about the big picture.

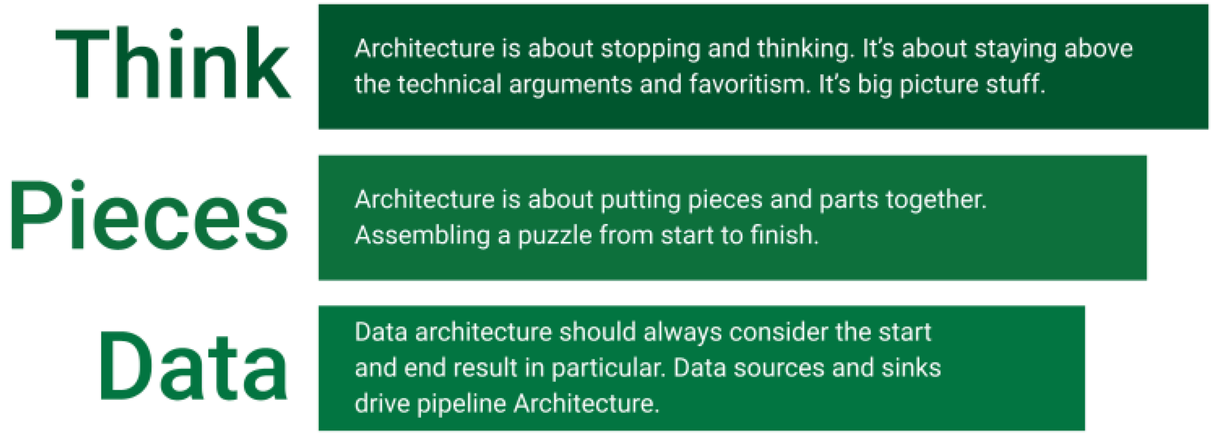
This is what architecture is all about, the big picture. It doesn't matter what your title is, it doesn't need to have an architect in front of it, it's something you should do and a skill you should work on every day. You might feel like your projects are too small to worry about such lofty ideas, but you are wrong.

Data Engineers, especially those at the Senior levels, depending on the company, are given great leeway in deciding what tools are right for the job. Many times everyone will look to the Data Engineers for recommendations about what available pipeline tools and technologies are available, and are best suited to a specific need.

To be able to answer these important questions, and allow the data pipelines to flow free and fast, requires some basic architecture skills and thought processes.

Architecture isn't some high and lofty ideas that have no grounding in reality. The best architects use their knowledge, experience, and research to understand the technical details of what has to happen. They pick the best long-term and simplest solution to get the desired result.

Many times it's about trying to discover problems and roadblocks before they happen.



Think — Architecture is about stopping and thinking. It's about staying above the technical arguments and favoritism. It's big picture stuff.

Pieces — Architecture is about putting pieces and parts together. Assembling a puzzle from start to finish.

Data — Data architecture should always consider the start and end result in particular. Data sources and sinks drive pipeline Architecture.

Data pipeline and platform architecture might sound scary, but it's thinking about your past experiences, and the experiences of others and applying them to the future. It's about trying to objectively poke holes in software before it's written. Think about what could go wrong, where bottlenecks might exist, about worse case scenarios.

Architecture Applied to Data

What does data pipeline architecture come down to?

- What are the business requirements (SLAs), what result is needed.
- How much data how often, how quickly will it grow.
- What types of data format(s).
- Storage needs, compute needs.
- What pieces of technology fit the requirements best.
- What do I have today that will do the job, what are the tradeoffs.
- Code architecture.
- Batch vs streaming.
- Putting puzzle pieces together.

Pipeline architecture is about taking a step back and trying to think about all the possible scenarios and solutions. It's trying not to make a decision and start designing a pipeline in your mind based around some piece of technology that you "like."

This has to be a careful consideration because we are also trying to avoid swinging too far in the wrong direction of over-engineering a solution. The majority of the time the simple solution will out-perform and cause fewer headaches than the fancy new one.

- Architecture provides space to think and plan.
- Architecture provides space to fail early.

In data pipelines, many times data engineers will often have to think about the coupling or connection between systems, or pieces of the pipe. We might have one tool that orchestrates and manages pipeline dependencies like Airflow. We might have another tool like Databricks Spark that runs the transformations of the data. These tools must work together, and such concerns are to be considered during the architecture planning phase of most Data Engineering projects.

Before we jump into the different steps and details of architecture, let's take a look at an example problem, to help us make sense of some common architecture thought processes.

Example Project

Let's work through how we can think about pipeline architecture by using an example. Here is the background.

You work for a medium-sized manufacturing company that makes all sorts of widgets. It's a small IT department of about 15 people, with only 3 data engineers. The company recently purchased some new manufacturing equipment. These new widget machines are fancy and can connect to the company network, sending out a signal/data each time a widget is produced in a TXT or flat file configuration.

Instead of manually counting inventory, the company would like the data engineering team to capture this widget data being output by the new machines and feed it back into the inventory system as well as provide dashboards of machine throughput to the business.

The IT department has still in the middle of a transition to the cloud, they have some infrastructure in AWS, and some local servers still running that host the system used to run the manufacturing system.

As the Senior Data Engineer, you've been tasked with deciding how to implement this system. Let's walk through the architecture thought process of how we might approach the different ways to implement a solution.

The first part of any architecture decisions on data pipelines should be examining the requirements of the business or as much of them as you can get. Of course in the real world, it might take some digging and you only find out half of what you need, but the answers are important nonetheless.

Understanding what the final expectations are is key to making technology and pipeline design decisions. Let's say in our case we find that the business expects a dashboard to be updated every few hours with production numbers from the machines, but you also find out that they want the inventory system to be updated every hour at least.

This already starts to tell us a story, real-time information is not expected for this project. That means when choosing the simplest technology layout to solve our problem, we can probably toss out streaming or message queues. It's always helpful to understand where we can get rid of complexity!

Let's review what we discover about the above requirements. When working on architecture it always helps to make a list that boils down to what we need.

- Not real-time, but hourly updates.
- Need an analytical dashboard.
- Ingest/capture widget production data points.
- Will not be running the cloud.
- TXT files for source raw data.

We can see from the above list that our possible options for the pipeline are becoming more obvious, we can rule in or out certain technologies. We don't need streaming, we need batch. We need code that ingests TXT files, which is fairly straightforward. We also know we need to build a warehouse to house the accumulated data for dashboard analytics.

Now, these are things that a Data Engineer loves to work on!

Let's continue to play along with our contrived example, and dig into each step we listed in the introduction as some of the core tenents and questions we ask and work on during data pipeline architecture.

How much data, how quickly will it grow/ types of data format(s).

We also talked to the manufacturing folks in charge of setting up and installing the machine. They have documentation from the vendor that says the new machines will write out plain text tab-delimited files to a drive on any IP address via ssh.

A new file will be written every 10 minutes in the following format.

machine_id part_number datetime quantity
AB34252AA 1545A03 2021-04-01:13:34:43 50

We know we have 5 new machines, which will output files every 10 minutes, they are text files, so the data will be relatively small and easy to deal with. It appears even if more machines are added, the data will not grow overwhelmingly fast.

As a data engineer, this is not a ton of information, but it allows us to sit down at this point and start making some high-level decisions.

* Where should the data be stored, cloud or local?
* What type of tech platform is good for this type of data
* Calculate rough data sizes and speed of processing needed.

Based on what we know we calculate that each file put out from the 5 machines will be 4KB in size, every 10 minutes. We know the plant will be running 10 hours per day …

60 files x 5 machines x 4KB = 1200KB

This is not much data … even if we are half wrong and our pipeline needs to handle double the data. Also knowing that the data must be put into the locally hosted inventory system every hour is an important piece of information.

This is what we do in architecture planning, we break down what we know into bite-sized chunks, we ask questions, and start making estimates based on what we do know.

What pieces of technology fit the needs best?

Let's make some architectural decisions. When we put a few facts together some things become clear. The data is needed on a local server to populate the inventory system, the data is not big nor is it real-time.

While it might have been tempting when first hearing about this project to select new and shiny technology that would have done the job, it probably would have proved to be complicated and expensive, another over-engineered project.

Option 1 - Cloud-based streaming/messaging system to ingest files, transform and write files to cloud-based storage, system to write results back into locally hosted servers to populate inventory system. Maybe Kafka/Pulsar with Apache Spark/Beam.

Option 2 - Keep everything local, the files are written to a local server, process with locally hosted Python script on the same server, backup files to cloud.

We also have to think about tradeoffs, as data engineers who love the cloud and managed services we want to gravitate towards the fun cool stuff, regardless of if the complexity matches the situation or not. But we have to be realistic what architecting data pipelines.

If we worked at a manufacturing plant that was double the size, and say had 100 machines running all producing these files that *might* change the discussion.

In this case, we could present to our boss some actual numbers and our recommendation. The cost of storing our data in the cloud is known with our rough size calculations, as well as the cost of running messaging services along with Spark or Beam would be fairly easy to obtain a pricing.

This would be compared to the cost of managing and hosting a local server with our custom code to process the file(s).

Review of Example Architecture Problem

This is of course a simple example, and we glossed over many details but I think you get the picture. I also think it's quite common for this type of exercise to never even take place. Usually, it's just up to an engineer who may just start choosing technology regardless of thinking through *all* the options available.

Having a good step back and thinking through the data flow and requirements is a critical step to not ending up with an expensive over-engineered solution, or on the opposite side, a simplistic solution that breaks later because of failure to scale.

I want to point out a few key areas we covered in this example that apply to many data pipeline projects when it comes to architecture decisions.

- Calculating data size and velocity.
- Calculating compute/storage requirements based on data size.
- Understanding the end result.
- Complexity vs simplicity tradeoffs.
- Understanding cost.

Data Size and Velocity

Probably one of the easiest architecture steps when thinking about data pipelines is trying to calculate rough data size and velocity requirements.

Not only is this exercise well because it forces a basic understanding of what the data looks like and potential issues, but understanding the velocity or incoming volume will shed light on what are and are not acceptable solutions.

Data Size

Why should we understand data size? Because usually, it requires us to go through a few exercises that help a person better understand what a pipeline needs to look like.

- What format is the data coming in, can we get a sample?
- Can we simply mock-up a sample file ourselves?
- What types of systems handle data files of this size?

Let's consider another simple example. We find out we have new data sources, someone sends us a sample CSV file that is comma-delimited and contains some 20 million records, about 9GB's in size. They tell us we will be getting two of these files per day.

If we are in a Python shop a few things might come to mind. We have a few options for processing these files…

- CSV reader in Python
- Pandas
- Dask
- PySpark

Most data engineers will understand to processing files of that size without memory issues is probably going to require Dask or PySpark, libraries that good at lazy evaluation.

What else can we understand from data size? Well, the size of the file is also going to tell us generally about what type of technology is best suited for this particular problem. Most of us probably understand that CSV files that are 9GB in size are not suited well to any sort of streaming or messaging system.

But let's say we found our data files were JSON files just a few KB's in size, delivered at the rate of 300,000 per day, well then we might start thinking about a streaming service that seems to fit that data size pattern a little better.

Data Velocity

Just like data size, the amount of incoming data (velocity) is another critical aspect of data pipeline architecture. It makes a very sizeable difference if we are receiving something in only a handful of "batches", or if we will be receiving hundreds of thousands or even millions of files. There are technology stacks that are built to deal with file volumes on each end of this spectrum.

The nice thing about understanding data velocity during the architecture phase is that it informs what some of the rest of the pipeline might have to look like and handle.

If we are looking at a high volume of incoming files, the growth of data may require some sort of compaction or aggregations steps in the pipeline. If we expect to receive only a few files a day, we will most likely not have to think about these steps of compaction or aggregation very much.

So, calculating data size and velocity upfront, even if you end up being half wrong is incredibly helpful. It's almost a given that going through this exercise will bring out the obvious winners and losers of which platforms were made to handle your data needs.

Calculating Compute Requirements

Another area that is ignored during the architecture and layout of data pipelines is the amount of computing that will be required. This can be one of the most important estimations because it's usually the number of one driver of cost. The more computing that is needed the more expensive the pipeline will be to run. Not every surprise is a good one!

There are a few ways to get a rough estimate of the compute requirements. Let's talk about the most straightforward approach to determining resource requirements.

Here are some easy steps to take.

- With rough data sizes, break the work up into logical batches.
- Determine based on transformations and possible tooling is the work CPU or RAM bound.
- Do the simple math.

Resource Requirement = (processing unit size x (number of CPU + RAM)) x number of batches

This will roughly give you an idea of what will be required. Let's walk through a simple example.

Example

We are architecting a new data pipeline, we will be receiving two raw CSV files per day, each with about 15 million records that will need a few different transformations. After the transformations, we will be storing the data in compressed parquet files. Each CSV file uncompressed is about 10GBs.

We have decided that using Apache Spark would be a good option for processing the CSV files with minor transformations in a parquet data lake is S3.

Right away we can understand that this process will be both CPU bound and RAM bound. But knowing that we will only be processing 10GBs at a time, twice a day, getting the rough estimate of resource requirements is very easy.

We know that Spark is a lazy evaluation framework and that the transformations happening to our 15 million records doesn't require anything other than the record itself. Meaning we don't need to hold all 15 million records or 10GB of data in memory at one time.

Knowing these facts the compute resource calculations become straightforward.

Resource Requirement = (processing unit size (1 file at a time) x (number of CPU(4) + RAM(8GB)) x number of batches (2) = 8 CPU and 16GB of ram will be the total resource used during 1 day of use.

Knowing this information along with a guesstimate of how long it might take, say 15 minutes to process such a file, and armed with the cost per hour information from AWS/GCP/Azure we can quickly calculate roughly what it will cost to run this pipeline for a day.

Calculating Storage Requirements

Calculating storage requirements and cost is the same process as compute. Knowing roughly how big the data is, and how often it is received is usually not that hard.

Once you know this information most likely with some compression numbers that are not that hard to find, cloud storage costs can be found out.

In our above example, we know that two files of 10GB each will be ingested and turned into compressed parquet files in S3. A quick Google search and some testing will show you that we can probably conservatively get 70% storage saves moving from CSV to Parquet.

storage cost = 2 files per day x 10GB per file x 365 days in a year x .30 (size of compressed parquets) = 2.2TBs of data.

Many times understanding basic compute and storage requirements during the architecture phase of pipeline work might change some of the decisions we make.

In the example discussed above after thinking about simplicity vs cost and complexity we might decide that processing those files on a Spark cluster is overkill. Spark is more expensive to run in the cloud and our 10GB per file could easily be processed with our Apache Airflow framework we are already running for example.

While all these calculations might be simple, going through the exercise of understanding roughly the storage and compute requirements might change our architecture decisions about what tool(s) to use and our approach to the problem.

Understanding the End Result

Another overlooked part of architecting data pipelines is the simple step of trying to think about the result. Too often as developers and engineers, we can get caught up in the excitement of writing code, new technology, or just how to solve the problem without stepping back first.

Getting a clear picture of the output and usage of the data needed from the pipeline will inform or change some of the choices we make upfront. Knowing if the data is needed for a dashboard or a web application at the very least is going to change how we store the data, and how we store the data might have an impact on what tool we choose to process and output that data.

It’s all connected and should not be overlooked before jumping into designing a data pipeline.

Complexity vs Simplicity Tradeoffs

Many times the culmination of our architecture work will come down to this final question, and it’s probably the most important one. Complexity is that silent killer that sneaks up on you. There are few things in the world of data engineering that will cause problems like over-complexity. It takes many forms and has terrible side effects.

- Extended debugging and error research times.
- Extended development time to add new functionality.
- Extended period to onboard new developers.
- Developers are afraid or unwilling to touch code.
- High cost of running and maintenance.

I am not saying that every complex system should not be complex, some problems and data are inherently hard to work on and require complex solutions.

On the reverse side, overly simplistic solutions have their own set of problems.

- Developer boredom and short tenure.
- Inability to scale.
- Cannot extend the functionality.
- Extended time to develop and add new functionality.
- Stagnation of technology stack.

Both extremes should be avoided and have many pitfalls. But good engineers know how to strike that delicate balance. Choosing new tools and technology simply because they are new is a bad habit and happens too often. It’s important to remember when architecting pipelines that we should let the data and requirements drive the complexity, not the other way around.

Forcing simplicity or complexity onto a problem is the wrong approach.

Data engineers should try to be balanced when designing pipelines. They should take into consideration all that we have talked about, and then setback and evaluate the options they have chosen. Do the data and situation warrant the complexity, or on the other hand the simplicity?

Will future developers and data/business changes break the solution? There are always reasons to incorporate simplicity and complexity depending on the situation. The job of the data engineer is to balance all these requirements and find the middle ground that can be reliable, scalable, and approachable.

Knowing that we will most likely have to come back to a data pipeline to make changes, to add data, the functionality should give us pause when architecting systems.

Understanding Cost

A topic that is important to many businesses is trying to get a handle on costs. I would argue that this might even be one of the easiest steps to complete if all of the other steps above have already been completed.

If you understand the size, volume, and types of data that will be coming, along with what technology choices are likely to be the best fit, figuring out the cost is usually not that hard.

Knowing the size and growth of storage requirements, along with looking at pricing documentation for say s3 or azure storage will be an easy calculation.

The same applies to compute cost, once we understand our data size and volume we can usually figure out what size of machine(s) are needed to process that data. Again with today's cloud platforms, it's very straightforward to find per hour pricing on computing and do some simple and rough calculations.

Again, when picking cloud offerings for certain technologies that might be required, say like AWS EMR, pricing is always available for these decisions and can be added to the cost.

Code Architecture

Another area I want to touch on briefly that is closely related to what we've just discussed is the architecture of our code. This is another good topic to think about after we've completed the steps above and we are ready to dive into writing pipeline code.

Once you've decided to move down a path towards a solution, you've set up Dockerfiles, tests, some documentation, the only thing left is to write code.

Everyone has different stypes of code writing and usually strong opinions about it. Some people practice TDD (Test Driver Development) to the nth degree, others do not.

Some people write in an OOP fashion, others with a more functional approach. It doesn't matter what style you choose and there are gobs of books writing on software architecture. I'm not here to write a book on that, but I do think making a few points will be helpful.

- Approach your data pipeline code cautiously and slowly.
- Plan at a high level before writing code at the low level.
- Keep functions and methods as small as possible.
- Break code into logical units.

I'm generally just advocating that you think carefully and take pride in the data pipeline code you write. Try to stick to normal software engineering best practices while writing your pipeline.

Whether you're using Scala, Python, or Java, just take the time to understand what best practices are and follow them.

Batch vs Streaming Architecture

Another major area of concern you will find in data engineering architecture is the batch vs streaming approach. We will mostly gloss over this, not because it isn't important, but because it's usually a very obvious decision. But, maybe not to everyone, so let's review.

There are only two approaches you can take with data pipelines, batch or streaming, or maybe both? I would say that typically it isn't the data engineer that makes that decision, but that the data sources of the systems where we draw our data dictate that decision.

Streaming

Streaming use cases are fairly obvious, and many times the only choice. If we are tasked with ingesting high volumes of high velocity and relatively small data points, streaming is of course probably your only and best option.

Tools like Kafka, Pulsar, Spark streaming, and the like, as well as the popularity of microservices that produce and send messages, have made streaming data pipelines very common. Typically the data isn't very flat or relational, not like a CSV, by more ragged and fits JSON data structures better.

It's best to think about streaming data pipeline architecture like a garden hose of water. Lots of small messages flying about. Typically in a use-case, this will become quite clear.

Batch

Usually batch is the default and most common type of data pipeline. When real-time isn't that necessary, and data points are produced in batches, or together, in a very flattened and as a file, batch processing will suit you just fine.

This is typically the big difference between streaming and batch systems. Batch data pipelines will be working on ingesting file(s), while streaming pipelines will be working on individual messages or data points.

Most of what you will learn about data engineering applies to both batch and streaming systems. While of course, the technology behind each is different, technologies usually aren't the hard part to learn, it's more about the nuances of the particular data sets that provide most of the challenges.

Puzzle Pieces

Before I close the chapter on Architecture, I want to talk a little more about the big picture. Data engineering can be a difficult subject to master, mostly because of the wide range of technology used.

Many times one must be versed in every Cloud provider product, streaming, batch, data storage, command line, servers, networks, database, programming languages, and the list goes on. One thing will become more clear as you gain more experience, architecture is about the big picture, and putting puzzle pieces together.

Because of the numerous sets of tooling, and the fact that one tool can never provide end-to-end modern data pipelines, the ability to fit puzzle pieces together is key. Understanding how storage solutions work together with data transformation and analytics tools, and how those tools work with orchestration tools, these types of decisions are what architecture is really about.

Many times tools are picked for data pipelines and the other puzzle pieces are ignored until the implementation phase happens, at which time roadblocks and other unknowns arise, causing project delays and problems. How do you avoid and overcome such obstacles?

Remember when building data pipelines, no details are too small to plan for. Think about all aspects of the data pipeline, all the pieces, and parts, choose the solution for each puzzle piece, and then test or explore each one. Many times simply glossing over technical documentation for each puzzle piece will reveal pitfalls or pros and cons of such a solution.

This is really what data pipeline architecture boils down to. Being able to design a simple, elegant, cost-effective way to stitch technology together to produce the desired result.

Summary

What makes an exceptional data engineer? Sure coding skills have a lot to do with it, having experience and know-how with different technology stakes is key. But, some of the best engineers I have met are the ones who take their time.

Don't underestimate the value of getting into the habit of doing simple architecture planning on even a small data pipeline project. Planning in all areas of our life always pays back big dividends, the world of data engineering is no different.

Thinking through the pros and cons of approaching problems in different ways is so helpful for our critical thinking skills, and we will always learn something on the journey of trying to understand and plan for that problem at a deeper level.

Chapter 4 - Storage

Probably one of the most overlooked pieces of the data engineering puzzle is storage. Not only the file types and compressions but where to store data, as well as how to store data. In the world of moving data around and transforming it, it's hard to overstate the important role that file types and storage will play in our data pipelines.

If you stop and think about data engineering in general, what is it that you do as a data engineer, you will conclude that it starts and stops with file storage and file types, especially in the age of big data.

The data engineering pipeline typically starts as some sort of file, and the result usually ends up as some sort of file. While this could be database files, in the new era of big data it's becoming more and more common to store data outside of traditional database systems.

Understanding what types of files and compression best fit certain types of data and use cases is a very undervalued skill. Choosing Parquet vs Avro vs JSON vs CSV vs HDF5 vs RDBMS. These decisions should be driven by the data needs and access patterns and should be explored before just "picking" something because it's easy.

With the shift towards a new type of data warehousing using data lakes and file storage instead of the traditional relational database systems, a data engineer who knows how to best layout file storage is priceless.

Many people gloss over the storage aspect of data engineering pipelines, not realizing how much there is to learn and how much productivity and speed gains can be found in properly picking file storage and partitioning strategies.

We will talk through the six basic storage fundamentals that every data engineer needs to take seriously when working on pipelines. While these don't cover every topic, this is a good introduction to what is important for a data engineer to understand.

- Access patterns.
- SQL vs files.
- File types.
- Compression.
- Storage location.
- Partitions.

But, before we dive into the specifics around the different file types, we should talk high level about data access patterns and how they affect our decision-making when thinking about storage. What do I mean when I say access patterns?

Access Patterns

I'm talking about how files need to be read and searched by our data pipelines, and those systems that ingest the results of our data pipelines. This includes a few topics you should always keep in mind.

- How many system(s) will need access to the data storage layer?
- How often will the above system(s) be accessing the data storage?
- How much data will these system(s) be reading?
- How much logic will be applied by those systems to the data?

In general, and usually upfront, data engineers must decide if the storage layer is going to include a relational database system of some kind.

Usually, this is a straightforward decision based on the requirements of the pipeline. If the data warehouse already exists in say MySQL, Postgres, or SQLServer then you may have no choice in the matter. But this is becoming less and less common today with the growth of data, much of it unstructured or semi-structured.

Access patterns can also include how much of the data will be accessed at one time. Will it just be small pieces or large chunks of data that are needed by outside systems? Having a storage system that supports fast single lookups vs serving requests for GB's or TB's of data will change how we think and choose technology.

When to choose a relational database system.

If you are working with data that is on the smaller side, say a few hundred GBS or less, and the data is relational and transactional, a traditional database is probably the obvious and best choice. Traditional databases have been used for decades and have many great features like reliability and ease of use via SQL.

How can I know if my data should be in a SQL database or in some other filesystem or store?

While there is no way to know without exact specifications, there are plenty of rules of thumb you can follow to help make that decision.

- Is the data tabular and relational?
- Data volumes, both in and out?
- Is the data of key-value or ragged structure?

Is the data best suited to fit into multiple tables that have very straightforward relationships together, and is the data very tabular in nature? What I mean by that is that the data can easily be represented in a spreadsheet. If the answer is yes, you might consider a SQL database.

Also, think about how often the data arrives and how often the data is needed. If you are planning on ingesting multi-millions of records a day into the data store and serving up just as many requests for that data … it’s possible a relational database will work, but it won’t be your average setup!

Storing documents that are very relational and transactional are also great candidates for NoSQL systems like MongoDB and DynamoDB. The use cases for these types of data are usually pretty obvious. Many times it comes down to if the data needs to be more on the analytical side of the transactional support side.

Many times data that leans more into the key, value pair with possible built-in hierarchal ragged structures are prime for NoSQL data stores.

I’ve seen 10’s of GB of data stored in DynamoDB because someone thought it was cool. Don’t be this data engineer, take the time to understand that data and what storage best fits the need.

Example

Say we are given a project we are receiving orders from a website in JSON format, and those orders need to be available in a customer portal within 1 minute of the order.

We are also told that the incoming volume will be about 500,000 orders per day.

```
{
“customer_information”:
{“customer_id”: 543267,
“customer_name”: “Billbo Baggins”,
“customer_address”: “100 BagEnd, The Shire, Middle Earth”
},
“order_date”: “2021-04-01 23:45:03:01”
“product_info”: {“product_id”: 567894,
“quantity”: 5}
}
```

Something like MongoDb or DynamoDb would be a great obvious choice for such a use case. Being able to store the incoming order information in its document structure without transforming it, right into a database that can be connected to the customer portal is straightforward.

Of course, we could make this work easily in a relational database like Postgres, but again always try to pick the simple solution. Evaluate which technology best fits the need in a manner that is simple and supports a reasonable expansion of requirements and data.

Any sort of requirement for supporting transactional systems from web apps to manufacturing systems where CRUD and ACID reign supreme, these are classic database use cases. But, as time goes on traditional database storage has given way to feature-rich file storage solutions

I’m now going to side-step relational database systems, we can talk about that more in the SQL chapter. Today’s big data is dominated by different file types.

File Types

Now that the data engineering world turns on storage systems like s3, along with the amazing growth of data, it's become less and less common to store data in traditional databases. Files have become the defacto new data lake. There has been a lot of feature development around file storage, especially when it comes to big data.

It doesn't matter much if files are being stored in the cloud or not, the type of file chosen as the storage layer has a massive impact on data pipeline performance and usability. At a high level, what should you worry about when choosing file storage?

- Data compression.
- Smart files (predicate push down support).
- File type usability by ingestion systems and code.
- Schema and data type support.
- Row vs column storage.

A data engineer needs to understand high-level file storage options and what use case warrants the use of each file storage option. Most of the decisions are straightforward when it comes to choosing a file storage option. If you understand a few basics about most of the different file types, typically a use case will make obvious what the best choice is.

We are going to try and cover a few of the more popular file options in the big data world for storage, we will look at the most common use cases for each, and try them each with Python. By the end you should have a basic understanding of each file storage option and when it's best to use them.

Here are the files you will most often run across and use in your data engineering work day-to-day.

- parquet
- avro
- orc
- csv/flatfile
- json

Before we dive into the specifics and examples for each file type, let's talk about a few more high-level topics related to file storage and types.

Row vs Columnar Storage.

Something you will start to run into as you dive deeper into data engineering and storage solutions are called row vs columnar storage systems. It can all seem a little ephemeral, but the difference is very simple to understand.

Row-based storage stores data in rows next to each other, like you would visualize data normally in your mind. It makes for quick writes (inserts) and reads on those rows of data.

Columnar storage stores the data field or column-wise, that is data from different rows but the same column/field is stored adjacent to each other. This makes queries and computations much faster, for things like analytics and data warehousing applications.

This means at a high level this choice between columnar vs row storage is all about access patterns of the data, how it is going to be used?

- Row Based - quick insert and reads.
- Columnar - analytics, and computations.

Traditional data warehouse or data lake patterns designed for analytical computations would fall into the columnar storage system. High-volume transactional data being streamed from a clickstream would make more sense as row-based storage.

Let's get back to some specific examples and walk through some of the more popular file storage formats you will find in the data engineering world.

Parquet.

Probably the most popular columnar storage file type is Parquet. It was made popular by the rise of Hadoop/Apache Spark and the new type of data warehouse and data lakes and has only continued to grow in popularity.

It's nearly impossible today to be a data engineer without using parquet files. Many times you might be using parquet files and not even know it. For example, the open-source Data Lake tool called Delta Lake that has been made popular by Databricks uses parquet files underneath the hood to provide many of its capabilities.

A few things that have helped make parquet files popular as a storage system.
- schema/data type with file
- great compression
- predicate pushdown (filter pushdown, read what you need)
- selective reads (only read what you need)
- support across platforms (Spark, Pandas, etc.)
- ease of partitioning

Parquet schema and data types.

You can store any kind of data you want in Parquet files. What makes them different from what a lot of people are used to with txt or CSV files is that the schema information about the data is stored with the file.

What do I mean by the schema? I mean the column names and data types, for example, `account_number` as a column and the associated data type which might be a `STRING` or `INT`. In the world of CSV and flat files, everything is a `STRING`, until your code decides it's something else.

Parquet compression.

Parquets also come with built-in compression that is no joke. This compression can turn into big savings when used with big data and you start thinking about it over TB's of data. It is not uncommon to see 70-80% storage savings from a CSV compared to a parquet with the same data.

Parquet files come with three different compression options. Many times snappy will be the default.

- gzip
- snappy
- LZO

Parquet predicate pushdown and projection reads.

Yet even another important feature of parquet files is the ability to only read the data you need. Meaning when calling a `read` function on parquet files with most tools you can say "I want only these 2 columns" and that is all that will be read, not the other 100 that may be there. Again, that is a feature that makes a big difference with big data.

Let's just look at a few parquet file examples with Python. There are two main libraries in Python you can use to interact with parquet files, pandas or pyarrow.

Otherwise many people experience parquet files when working with Spark, which supports parquet files natively.

Most people are used to using CSV files, so let's see how to convert a CSV file to parquet with pandas.

```
    import pandas as pd
    dataframe = pd.read_csv('sample.csv')
    dataframe.to_parquet('sample.snappy.parquet' ,engine='auto',     compression='sn\
appy')
```

Another popular Python package to use with parquets is called pyarrow.
We can also see how to only read columns we are interested in.

```
    import pyarrow.parquet as pq
    data = pq.read_pandas('sample.snappy.parquet', columns=['ID', 'Date', 'Descripti\
on']).to_pandas()
```

Many times parquet files are partitioned into subfolders based on some column, usually a date.

```
    pq.write_to_dataset(data, root_path='dataset', partition_cols=['Date'])
```

Partitioning is an important topic we will cover a little later. What partitioning does for parquets and other file storage systems is ensure your dataset is broken up into physical partitions.

Predicate pushdown is another feature of parquet files that makes them fast and a great tool for data engineers. Because of the metadata stored with parquet files, reading a dataset with filters that are "pushed" down to the file read smartly can create huge performance gains over file formats like CSV that don't have this support.

A pyarrow example might look something like this.

```
    parquet_dataset.to_table(filter=dataset.field('order_amount') >= 7)
```

Also, projection reads that works in much the same way for parquet files. This means that you can select only the columns you require in your data pipeline, and others will be ignored.

You can imagine in large datasets how much performance can be gained from such a feature. Unlike a CSV file, you don't have to read everything, with parquet you can specify what columns you are interested in working with.

An example with PySpark while reading a parquet dataset to sum a customer's order amounts might look like this.

```
    data_set.select('customer_id', 'order_amount')
```

These topics just scratched the surface of what parquet files can do, I encourage you to remember the features and benefits of parquet files and think hard about them before writing your next dataset as a CSV.

Avro.

The Avro file format has been around as long as Hadoop has been here. It is often coined as a "data serialization" framework. It's always touted as being able to handle flexible and wide-ranging data structures, even hierarchical within records. It's also built to be row-oriented (different from Parquet).

- handle schema changes
- row-oriented
- ragged schema structure support (similar to JSON)

It's interesting to note that the Avro format is used widely in the RPC space, to communicate messages and data across networks. This is where it widely differs from parquet.

Where parquet will be found a lot in say large datasets used for data lakes and data warehousing, Avro is going to be found in more highly transactional and messaging systems and architectures.

One of the first things you will notice when starting to read or work with Avro is that the Schema is an integral concept. You can't write data to an Avro file without having or defining a schema first. Most commonly the schema is defined with JSON, which makes it very approachable, but very different from most file systems you are probably used to.

A popular Python package for working with avro is fastavro

```
pip3 install fastavro
```

The first thing you will also have to do when starting to create an Avro dataset is going to be reading in or creating the schema. In most cases, it will be easy to just write a Python dictionary and keep it in memory or a JSON file.

A couple of key points, below you will notice the `parse_schema()` method used to turn the json/dict into the Avro schema. Also of note, you would want some sort of iterator that contains your records/data. Notice how the schema of my records matches the schema set forth at the beginning.

Otherwise, `fastavro` in Python exposes a `reader()` and `writer()` that you will most likely be familiar with. Here is my example of writing records to a file, then reading them back in.

```
import fastavro
>> create json schema from dictionary
avro_schema = {"namespace": "middle_earth.avro",
                 "type": "record",
                 "name": "MiddleEarth",
                 "fields": [
                     {"name": "character", "type": "string"},
                     {"name": "position", "type": "string"},
                     {"name": "dragontreasure",  "type": "int"}
                 ]
                }

avro_schema = fastavro.parse_schema(avro_schema) // turn dict/json into avro schema

>> some sort of records stream/iterable
heros = [
    {'character': 'Gandalf', 'position': 'Wizard', 'dragontreasure': 50},
    {'character': 'Samwise', 'position': 'Hobbit', 'dragontreasure': 1},
    {'character': 'Gollumn', 'position': 'Sneaker', 'dragontreasure': 5},
    {'character': 'Mr.Oakenshield', 'position': 'Dwarf', 'dragontreasure': 1000},
]

avro_file = 'middle_earth.avro'

>> write records to file.
with open(avro_file, 'wb') as write_file:
    fastavro.writer(write_file, avro_schema, heros)

>> read back file
with open(avro_file, 'rb') as in_file:
    for record in fastavro.reader(in_file):
        print(record)
```

Another example would be converting a csv file into avro format.

```
import fastavro
import csv
from datetime import datetime

start = datetime.now()

bike_share_file = 'Divvy_Trips_2019_Q3.csv'
avro_file = 'bike_share.avro'

>> create json schema from dictionary
avro_schema = {"namespace": "bike_share.avro",
                 "type": "record",
                 "name": "BikeShare",
                 "fields": [
                     {"name": "trip_id", "type": "string"},
                     {"name": "bikeid", "type": "string"},
                     {"name": "from_station_id",  "type": "string"},
                     {"name": "gender", "type": ["string", "null"]},
                     {"name": "birthyear", "type": ["string", "null"]}
                 ]
                }

>> turn dict/json into avro schema
avro_schema = fastavro.parse_schema(avro_schema)


def stream_csv_records(file_location: str) -> iter:
    with open(file_location) as f:
        creader = csv.reader(f)
        next(creader) // skip header
        for row in creader:
            yield row


def transformed_stream(record_stream: iter) -> iter:
    for record in records_stream:
        avro_record = {u"trip_id": str(record[0]),
                        u"bikeid": str(record[3]),
                        u"from_station_id": str(record[5]),
                        u"gender": str(record[10]),
                        u"birthyear": str(record[11])
                        }
        yield avro_record

records_stream = stream_csv_records(bike_share_file)
transformed_records = transformed_stream(records_stream)

>> write records to file.
with open(avro_file, 'wb') as write_file:
    fastavro.writer(write_file, avro_schema, transformed_records)

end = datetime.now()
x = end - start
print(f'Write process took {x} seconds.')


>> read back file
with open(avro_file, 'rb') as fo:
    avro_reader = fastavro.reader(fo, reader_schema=avro_schema)
    for record in avro_reader:
        print(record)
```

Another option for Python and avro is the pandavro package.

```
import pandavro as pdx
import pandas as pd
from datetime import datetime

start = datetime.now()

bike_share_file = 'Divvy_Trips_2019_Q3.csv'
avro_file = 'bike_share.avro'


df = pd.read_csv(bike_share_file, usecols=['trip_id', 'bikeid', 'from_station_id', '\
gender', 'birthyear'])
df = df.astype(str)
pdx.to_avro(avro_file, df)

end = datetime.now()
x = end - start
print(f'Write process took {x} seconds.')
```

You’re probably less likely to run into Avro in the wild, especially in the Data Lake world, but it could happen. It’s more built for serial messaging systems and complex schema’s that maybe don’t work well with Parquet for example.

Orc.

Another popular storage option in the big data and data engineering world is ORC, probably not as popular as Parquet, but popular enough to have gained a decent adoption in the community.

- ORC files are made up of Stripes …. groups of row data.
- It supports data types like “datetime, decimal, and the complex types (struct, list, map, and union)“
- The file has/can have “indexes.” Helps in seeking rows and skipping row groups when a reader comes in with a predicate … aka push down filter.
- file footer contains meta-information about the ORC file as a whole.

compression can be ” Snappy, Zlib, or none.“

- supports ACID when used with Hive.

So what do you need to know about ORC?

Stripes are integral to ORC, they are 64MB in size by default, are “independent” from each other … allowing distributed work to happen on a file. Columns are separate from each other in the stripe, allowing only needed data to be read.

The indexes in an ORC file allow push-down read filters into a file, signaling which Stripes need to be read. You will notice this is similar to parquet files.

It’s also worth a note that working with Orc is not as easy as working with parquet or other files. You can’t just work with ORC files with a simple Python setup, it was designed for HIVE and you will most likely not run into it unless you’re working on legacy systems. You will have more luck working with Orc using Java or Scala, JVM-based languages. Otherwise, you can use Spark to work with getting data converted into Orc format.

```
df = spark.read.csv('bikes/*.csv', header='true')
df.write.format("orc").save("bikes/orcs/orcy.orc")
```

If you read about Orc files you will probably run across information indicating that it is faster than the Parquet. While this may be true, be warned, gaining a few seconds here and there is probably less important than how well the file type is supported by different tech stacks and the availability of community to support it.

What I’m trying to say is that unless you have a good reason, choose parquet over Orc.

CSV / Flatfile.

These two file types are probably the most classic file format used in data engineering, even some big data workloads unwisely use CSV and flat files. These are file types that have been around a long time. What are they?

CSV’s and flat files get nothing special, no compression, no built-in headers. They are easy and simple to use, which is why they are so often found to be used in data engineering pipelines. If your data is small, it probably makes sense to use CSV and flat files.

CSV and flat files are just text files with some sort of separator between records, typically a comma is used “,”, although maybe times you will see pipes, like |, or tabs used to delimit records.

It also isn’t uncommon to double-quote records as well. Here are some topics to remember when working with or learning CSV and flat files.

- no compression
- data values separated by comma (,), but also pipe (|) and other values
- no schema or data type support
- many times datapoints will be qualified, maybe surrounded by quotes (“x”)
- very easy to read and write in most languages

An example of what a CSV file would look like is below.

```
ride_id,rideable_type,started_at,ended_at,start_station_name,start_station_id,end_st\
ation_name,end_station_id,start_lat,start_lng,end_lat,end_lng,member_casual
A847FADBBC638E45,docked_bike,2020-04-26 17:45:14,2020-04-26 18:12:03,Eckhart Park,86\
,Lincoln Ave & Diversey Pkwy,152,41.8964,-87.661,41.9322,-87.6586,member
5405B80E996FF60D,docked_bike,2020-04-17 17:08:54,2020-04-17 17:17:03,Drake Ave & Ful\
lerton Ave,503,Kosciuszko Park,499,41.9244,-87.7154,41.9306,-87.7238,member
```

Opening and working with CSV files couldn't be easier in Python. Learning how to work with CSV and flat-file records is key for any data engineer.

```python
import csv

def open_csv_file(file_location: str) -> object:
    with open(file_location) as f:
        csv_reader = csv.reader(f)
        for row in csv_reader:
            print(row)

if __name__ == '__main__':
    open_csv_file(file_location='PortfoliobyBorrowerLocation-Table 1.csv')
```

Doesn't get much easier than that, does it?

Most data engineers stay away from Pandas when working with data, because it's not a big data tool that is scalable. Pandas is well known for having memory issues when working with data. But, that being said CSV and flat files are typically not big and can easily be manipulated with Pandas, and it should be used when it makes sense to do so.

```python
import pandas

def open_csv_file(file_location: str) -> object:
    dataframe = pandas.read_csv(file_location)
    for index, row in dataframe.iterrows():
        print(row['Location'], row['Balance (in billions)'], row['Borrowers (in thou\
sands)'])

if __name__ == '__main__':
    open_csv_file(file_location='PortfoliobyBorrowerLocation-Table 1.csv')
```

What should someone building pipelines remember about using CSV and flat files? They are probably the easiest storage option, and they work great for data that can be represented as a table, with rows and columns.

It's important to remember that they provide no out-of-the-box compression options. If you have large datasets you will have to zip or gzip the files yourself.

Also, you will struggle to support complex data types and structures in CSV and flat files. They are meant to be a basic data file tool and should be kept that way.

json

Another popular format found all over data engineering pipelines is JSON. Many times it's used for configuration, and not as the main storage type, but you will find s3 buckets full of JSON files in your engineering travels.

JSON data is best for ragged hierarchy schemes. It acts just like a Python dictionary. It can contain lists and arrays, as well as depth into its schema. Its popularity rose to meteoric highs with its use in REST API's as the data transmission format of choice.

```
example_json = {
        "some_key": "some_value",
                "customer_id": 1234,
                "order_amount" : 15.05,
                "products": [456, 987, 043]
}
```

How is JSON most likely to be used in Data Engineering? This is an interesting question, and we have to go back to the basics to understand when we should, or should not, use JSON.

We should not choose JSON if we are storing large amounts of data into a storage bucket. Why? because JSON files don't provide the partitioning, predicate pushdown, compression of other file formats. You do not want to end up with millions of data points spread across millions of JSON files unless they are later going to be ingested into another system for analytics.

JSON is a great tool for storing configurations for applications, as well as messages and individual data objects being passed around inside a data application. If your data represents tabular formats, don't use JSON, if your data is more document-oriented, it could be a great choice.

Let's look at a few quick examples of using JSON in Python, and how it might fit into a data pipeline application.

```
>> file -> config.json
pipeline_config = {
     "number_of_nodes" = 10,
     "resources" : {
                "cpu": 4,
                "ram": 16GB
    }
}

import json

def read_configuration(config_uri: str = 'config.json') -> json:
     """ read configurations file, data input and output locations """
     with open(config_uri) as cf:
     config = json.load(cf)
     return config
```

JSON makes the perfect data structure for holding configurations and passing information around inside code and your programs. It's used a lot in APIs and the web world. It's a great versatile tool, but as a data engineer, you have to remember its limited usability and big data platforms.

Compression.

The compression of files is an important topic in data engineering. As the size of data grows, and thus the cost, it's important to remember size matters when storing files away.

There are many types of file compression, but in data engineering, you will most likely see the following types...
- gzip
- tar
- snappy
- zip

There are others but these are the ones you will run across daily. Honestly, when data engineers are storing files there should rarely be a case where those files are not compressed in some manner. It's easy to compress files, saves size and money, and there isn't much of a penalty to uncompress those files on read. Many tools like Spark will automatically uncompress files on read for you!

When you can save 80% storage space per file just by compression, it is a no-brainer.

Most compression can be done easily on the command line, if you have a Linux flavor like Ubuntu, these packages are just an `apt-get install` away.

```
bash
>> tar a single file
gzip file1.txt
// gzip multiple files
gzip file1.txt file2.txt

>> tar a directory of files
tar -zcvf example.tar.gz /my/source/data
```

Also never forget you can easily work with these compression types in your pipeline code. For example, say we are downloading a file from AWS s3 with Python and need to gzip and compress that file.

```
python
obj = s3.get_object(Bucket='my-s3-bucket, Key='file-1.csv')
    return gzip.GzipFile(fileobj=StringIO(obj['Body'].read())).read().splitlines()
```

Or again, we have an object of data in Python and we want to write it to a gzip object.

```
python
my_data = "my, row, of, csv, data"
data_io = StringIO()
    with gzip.GzipFile(fileobj=data_io, mode="w") as f:
    f.write("\n".join(my_data) + "\n")
```

The main takeaway for any data engineer on compression should be, to use it. By nature we will be storing and warehousing large amounts of data, it makes sense for cost reasons alone, not to mention the ease of use of compression tools, to find a suitable compression for our files and use it.

Storage location.

Choosing the storage location is a lot easier as a data engineer than it used to be. Actual HDFS and Hadoop file systems are very uncommon and legacy now. Cloud storage provided by AWS, GCP, and Azure rule the day. These are storage locations in the cloud are typically referred to as "buckets."

This is great for a few reasons, high availability and access from anywhere is pretty much standard operating procedure now. No more network mounts and drives to mess with. But, some complexity is added with cloud storage.

You have to learn and become adept at using the command line interfaces provided by the cloud providers to interact with their storage systems. These command-line tools like aws CLI and gsutil are how bulk files are moved around. It could be by copying, moving, deleting, or syncing, but you will have to learn how these operations work. We will cover these CLI tools in more detail in the Cloud Providers and Servers chapter later.

Also, when coding to the storage system in data pipelines, you must learn the packages and libraries provided by the cloud companies, things like the Python package boto3 used to interact with s3. If you are using GCP, gsutil will most likely have a place in your pipelines.

These tools can add some complexity, but what they provide in return is usually worth the hassle. It's also worth a note to think about cost when it comes to storage. Many of the popular cloud storage options, like s3, only charge fractions of a cent per GB stored, but what they don't mention is the cost to read and write that data.

This is something data engineers should think about. Reading and writing massive amounts of storage is going to increase your storage charges and cost, many times the devil is in the details.

Partitions.

Partitioning strategies are probably one of the greatest unsung heroes of data engineering and file storage that isn't talked about enough, nor is it taught with any consistency. What is partitioning? It's how the data files you are working with are physically stored and organized on a disk or in the cloud. How do you know if the data you are working with is partitioned or not?

Just answer this question. Are the files just randomly located in any place or folder … or is their structure to the madness?

One of the most common partitioning strategies in data engineering is date partitioning. So some folder structure has a directory that might be broken down by year, month, and day. Here is an example.

```
year=2021
    month=08
         day=15
             ...file_1.txt
             ...file_2.txt
```

As you can see this makes it easy for people and for computer programs to find data that is just in a single year or even a single, month, and particular day. This is fundamentally what partitioning data is.

- Partitioning keeps programs from reading entire storage directories.
- Partitioning breaks the data up logically and physically.
- Partitioning increases performance and reduces cost (less read).

In another case, we might be working in an environment where we gather lots of information about a few repeat customers or clients. We might run into a data partitioning strategy as follows.

```
client_number=123
    invoices
        year=2021
            month=10
                 day=01
client_number=456
    invoices
        year=2021
            month=10
                 day=01
```

What I’m trying to convey here is that data partitioning is a powerful strategy and is going to be closely related to how your data is either produced or consumed. And when you have terabytes or petabytes of data, you can’t read every file when a pipeline needs some data, that would take too long and make the system unusable. This is the key to data partitioning, it allows for more fine-grained file system seeking and data locating. The program can hone into exactly where the data is that is needed.

This topic is so important that even popular tools like Apache Spark have it built right into their APIs.

```
dataframe.write.partitionBy('year', 'month').parquet('/mnt/my_data/')
```

The idea of grouping data and saving it in partitions is at the core of big data, and most tools now and in the future that data engineers will be working with will have some form of this partitioning idea.

How should I think about data partitioning.

It doesn’t matter if you are engineering pipelines that work with data on disk, in the cloud, or some NFS; the same thought process should be taken when thinking about data organization.

And this is really what lies at the bottom of a data partitioning strategy, trying to understand how data is either produced and consumed and applying some common sense organization around that data to help both humans and software work with that data.

Just writing data and files to a single folder and letting one folder contains thousands upon thousands of files will give yourself or someone else a headache down the road. Think about how your ETL would need to interact with that data later. What kind of predicates or filtering statements could be applied. If you’re familiar with SQL then this would be thinking about the WHERE clause.

Data partitioning can, of course, get very complicated and they might be certain nuances to consider when you are working with specific tools, but in the end, I want you to remember the big picture.

Never forget the importance of just organizing your data and files into commonsense patterns that most likely will present themselves based on the use case of your particular data.

Chapter 5 - Compute and Resources

This chapter is all about managing resources. Believe it or not, this is a big part of what data engineers do on a day-to-day basis and can affect every decision we make when writing code.

Remember in the beginning when we said our pipelines need to be scalable? Well, how do you think most big data gets processed through ETL pipelines? This work happens on clusters and servers, often referred to as compute in the distributed big data processing world.

These principles still apply if you are writing what would be considered small data running on a single machine. Managing the resources and compute available to you is the key to a pipeline that will outlast its creator(s).

Step Back - Overview

We talked a little in the Architecture chapter about calculating compute needs, but let's get a little more specific about what exactly a data engineer needs to take into consideration when writing data pipelines.

- RAM
- CPU
- Storage
- Cluster/Node count (aka how many servers do I have to work with.)

Why should we care about such things? Honestly, because the above resource factors will affect how fast our code runs, how much data it can process, and ensure we are not wasting and leaving unused compute on the table, and of course, there is a little thing called cost!

How do I manage the compute and resources at a high level?

Without getting lost in the weeds, because every situation is different, how can we generally approach compute and resources in a semi-structured way?

- What resources are available?
- What does the tech stack require for resources at a minimum?
- How can I reduce resource consumption in my code and pipelines?

This first step any data engineer should take would be to find out exactly what resources are available on the platform in which your ETL will be running. This of course will widely vary depending on the technology stack, but in the end, you have to find the answer.

Are you using a Spark cluster? How many nodes are there, and what RAM and CPU are available on each. Maybe it's an Apache Airflow setup, what are the size of the workers, and how many workers are available? These questions have answers and you should find them for your use case.

Once you have answered those questions you should do everything in your power when writing ETL and code to use every last piece of RAM and CPU, leaving nothing on the table.

This will probably take the form of two major topics regardless of the programming language of choice.

- Concurrency (how many things can you do at the same time)
- In-memory (how much data can you fit into memory to work on)

Some tools like Apache Spark and other distributed big data platforms make take care of certain pieces of that puzzle, but there are probably many times when you will just be writing custom code and it will be up to you to make those decisions and write your code accordingly.

How does this work out in real life? Let's take a look at an example that has happened to me many times in my career.

Example

Let's say you're just going along with your day as normal and you get a new JIRA ticket assigned. The jist of the ticket is that you will be receiving several raw text files that are fixed-width delimited (say each column of data is separated by 50 white spaces). Your task is to convert each CSV file into a comma-delimited CSV file for processing by some downstream system.

Let's say you start merrily writing code immediately, by writing a few functions that can do such work.

- Open the text file.
- Read the text file.
- Iterate the rows in the text file.
- Convert each row to a comma-separated row.
- Write all the rows back to a file.

Easy enough right? All goes well with your testing until you put it in production. You realize now that you receive about 40 files a day ranging between a few MB up to 2GB in size .. and the folder you are supposed to process has a few hundred historical files already in it.

You do the math and realize it's going to take a month to process each file one by one. Also the more you think about it, you realize the machine that will be processing these files on AWS is a `c4.4xlarge` meaning it has 16 CPU and 30GB of RAM.

It's easy to see that processing files one at a time and not only going to take forever, but it's not efficient and leavings most of the compute resources wasted on the machine the code runs on.

For a Data Engineer this is something we want to avoid, no matter if you are using Python or Scala, Spark or not, it's clear in the above situation we need to find a way to process more files concurrently, use every last CPU and RAM to chew through the files.

RAM/Memory

This topic is probably one that can come and bite data engineers the most often, yet there isn't much training or material on how to manage memory consumption for Data Engineers.

There are plenty of Software Engineering related resources that can teach out about different OOP methods and how certain data structures might be larger or smaller than others, but typically this type of material is directed toward normal SE principles and not that suited to Data Engineering work.

If you think about what Data Engineers do, day-to-day, it's transforming large amounts of data via pipelines. So when a Data Engineer thinks about memory usage it typically comes down to the following question.

- How much memory is available?
- How much memory needs to be left available for the operating system?
- At its peak, what will my program's max memory usage be?
- Can I calculate memory consumption based on my data?

This can be a tricky tight rope to walk. OOM (Out Of Memory) errors are common in data engineering. One of the easiest steps to take is just to calculate the size of the data file, or the piece of data that you are working on, compare it to the size of the resource and do the simple division to see how many pieces of data you can fit into memory while still leaving a little room for overhead.

On the other hand, we all know that memory is faster than disk, and working on data in memory is much faster and is a great option to use when available. Let's take Python for example and explore features for working with in-memory data that every Data Engineering should be aware of.

Even if Python isn't your first language, the ideas should generally transfer to other languages and approaches. What I'm trying to accomplish here is to introduce you to the idea of thinking about memory(RAM) usage when writing your code. Most beginners don't do this, and it never comes up until something is breaking.

Simple Introduction to StringIO and BytesIO in Python.

StringIO and BytesIO are perfect for making your Python faster. Ever heard of something called a File Object in Python? Ever heard of BytesIO or StringIO? If not you're missing out. It's easy, fast, and wonderful, in short, it's the best and a tool that data engineers should be using.

Probably because of perceived complexities, IO streams are an underused feature that rarely comes up in most code. We all know that memory is faster than disk IO, this is what I use IO streams for. Yes, it requires us to think about RAM and how much resource is available on our machine, but this helps ingrain good habits into us early on.

Why would you want to use `from io import StringIO, BytesIO`? Think about it. If you are the typical data engineer, business intelligence engineer, report developer, or data analyst, what is one of the first things you learn? Probably opening a file, maybe a CSV file, a text file, a JSON file, maybe it's a zip file full of other files.

Maybe you're downloading a bunch of data into files. It just usually involves a file of some sort 80% of the time. You may have not ever thought about it, but reading and writing files to disk will be a bottleneck in programs, especially if you are reading and writing the same file multiple times. Say you're downloading a file over HTTP to disk, then unzipping it, then reading the file in. That's a lot of disk IO.

This is where StringIO and BytesIO come in. One way to think about these streams of data is that they act as a File Object. What does that mean? It means you can treat and interact with these objects like you would any other file, they have the File API on top of them. You can read them, write to them, etc. A file but not a file, get it?

- IO streams are just that, streams of data
- They can have File Object (read, write) type features
- They help us get used to thinking about RAM as related to our code
- They are faster than Disk operations

The best part of IO streams is that they live in memory, and that means fast. Of course, you need the resources to do this on your machine or server, but that usually isn't a problem. One minor detail to remember about a StringIO/BytesIO is that when created, it acts like an already opened file!

Let's look at some examples of how this could work. Simplistic, but to the point.

```python
from io import StringIO, BytesIO
import csv

in_memory_file = StringIO()
csv_writer = csv.writer(in_memory_file)
csv_writer.writerows([[1, 2, 3], [4, 5, 6]])
in_memory_file.seek(0)
for row in in_memory_file:
    print(row)
```

So what’s going on here? First, we create an in-memory open file object, `in_memory_file = StringIO()`. Next, I’m showing you how we could create what is a CSV file, just without calling `with open(‘file.csv’, ‘w’) as csv_file`. So we are swapping out the usual step where we create a file on disk to write data too … we are creating a in-memory file!

A CSV writer takes a file object, well we have one of those, don’t we! Next, we call `.writerows()` on our csv_writer object, we write two separate rows. The next part may seem a little strange to you, seek(0). Since we are dealing with a stream, file-like object, and we’ve written to lines, technically that opened file object is at the “end.” To read back out of that file object and print the lines we wrote, we need to be at the beginning.

Let’s take this just slightly further to show how StringIO/BytesIO could be useful.

Example

Let’s say we have a boring job and our boss asks us to download information about Livestock and Meat International Trade data from the government, and insert relevant information into a database. The typical workflow would be to download the zip file, unpack it, read in the relevant CSV file from disk, find the data, and off to the races.

Well, we know better now don’t we. Sounds like a few spots of file IO, like writing the zip to disk, unzipping the files to disk, then reading the CSV file from disk. But, there is a more excellent way.

```python
python
import requests
from io import BytesIO
from zipfile import ZipFile, is_zipfile

url = 'https://www.ers.usda.gov/webdocs/DataFiles/81475/LivestockMeatTrade.zip'

try:
    response = requests.get(url)
except:
    print('Problem downloading zip file.')

if response.status_code == 200:
    in_memory_zip = BytesIO(response.content)
    with ZipFile(in_memory_zip) as zippy:
        for item in zippy.infolist():
            if 'Exports' in item.filename:
                with zippy.open(item.filename) as export_file:
                    for row in export_file:
                        print(row.decode('utf-8'))
```

Easy! It's fast because we are doing everything in memory and it's simple code, straightforward. Really what I'm showing you here is that many of the packages and methods you use in Python can take a file-like object, in this example `with ZipFile(in_memory_zip) as zippy`, doesn't matter if it's an actual file sitting on your disk or a file-like object sitting in memory.

RAM/Memory Review

I know that might have seemed a little bit of a deep dive for a book on data engineering. What I want to get you thinking about is the code that drives your pipelines, it could be PySpark or it could be just plain Python or Scala. If you are working with data, that data is going to be in memory at some point.

Data in-memory is going to be faster than data on disk, most of us have at least heard this in passing, "in-memory is so much faster." But what data engineers need to think about when writing pipeline code is … how much data am I dealing with? How much data is going to be in-memory in a worse cast scenario when doing this specific transformation or calculation? OOM errors are one of the most common problems that data engineers run into early on in their careers.

This is usually because people get in the habit of assuming something works, and the data grows, more data comes in, or just code gets pushed to production, and everything breaks.

What I want you to take away as a data engineer is how to take advantage of RAM, because it is fast, but also be aware of how easily you can out of it and cause OOM errors.

RAM is fast and useful, it's also limited and easy to run out of. You don't want to leave a bunch of memory on the table if you have it as an available resource. You don't want to overuse it and cause memory leaks and other errors, use it wisely.

CPU/Cores

Now that we have talked a little bit about memory and RAM, and how we should pay attention and use it well, let's talk about CPU. The thought process here is the same as with our memory discussions.

In today's world, it's very unlikely that you will be working with resources, servers, or virtual servers, that only have a single core. In most cases, you will probably have a few cores, so leaving those cores unused is another wasted resource that could significantly reduce code run times.

I will say this upfront, no matter what language you use, even Python, concurrency is a tricky topic even for good Software Engineers. Many tears have been spilled on the internet arguing about this topic. But, let's just cover the basics and show a few examples in Python of how you as a Data Engineer can easily use all CPUs available to supercharge your pipelines.

Again, this isn't specifically about Python, this is meant to teach you that many of the machines your data pipelines will run on have multiple cores and CPUs available, and you should use them all. It will make pipelines run faster, reduce compute cost (because you're doing more with what you have) and generally make your life better.

- Use all the CPU/cores available.
- Increase performance with the increased use of CPUs
- Understand what type of code requires more CPU and what doesn't.

Example

Let's say in this example we have a group of CSV files. We need to read all the files, look through each row and find out if that row contains a "member" record or not.

Of course, we could open each file one by one and do this work, but after all our discussions so far we would realize this isn't scalable or fast, and wastes whatever resources we have on our server or box running the code.

This must be balanced by how large the files are and if we start reading more than one file at a time, we should be away from the available CPUs and Memory on our machine so as not to overload the system.

ProcessPoolExecutor's in Python is a great way to introduce concurrency, allowing us to work on more than one file at a time.

```
python
import csv
from glob import glob
from datetime import datetime
from concurrent.futures import ProcessPoolExecutor

def main():
    files = gather_files()
    with ProcessPoolExecutor(max_workers=3) as Thready:
        Thready.map(work_file, files)
```

```python
    for file in files:
        rows = read_file(file)
        for row in rows:
            filter_row(row)

def work_file(file_loc: str) -> None:
    rows = read_file(file_loc)
    for row in rows:
        filter_row(row)

def gather_files(loc: str = 'trips/*.csv') -> iter:
    files = glob(loc)
    for file in files:
        yield file

def read_file(file_location: str) -> iter:
    with open(file_location, 'r') as f:
        data = csv.reader(f)
        next(data)
        for row in data:
            yield row

def filter_row(row: object) -> None:
    if row[12] == 'member':
        print('member ride found')


if __name__ == '__main__':
    t1 = datetime.now()
    main()
    t2 = datetime.now()
    x = t2-t1
    print(f'It took {x} to process files')
```

The point here isn't to teach you how to use ProcessPoolExecutor's, but to teach you that such tools exist and are available for you to use where appropriate. The problem breaks down to this, regardless of tools or language.

You have a list of work that needs to be done. You probably have a method or function that does that actual work, now you need a tool to spread the list of work across several CPU's so you can do more work at one time with your code.

In our case, that's exactly what the ProcessPoolExecutor's `map` function provided for us. We passed it our function `work_file` and our list of work `files` and the rest is done for us! All of a sudden we have multiple files being worked on at once.

Storage

Storage is probably one of the easiest resources to manage, and we won't devote a lot of time to this. With the advent of cloud storage services like Azure Blog, AWS s3, and Google Cloud Storage the idea of running out of disk space has floated into the past.

It's still an important topic to talk about and I've seen disk issues pop up many times in pipelines. This is mostly the case simply because Data Engineers think they don't have to pay attention to storage anymore. But you do.

Most servers and resources come with either a set amount of storage attached, or possibly some elastic storage service that is quite large, or pipelines will just read and write directly to the cloud. Whatever the case it's easy to get caught with disk out-of-space errors if you are not careful.

It could be writing intermediate results to disk or something of that nature, but not cleaning up after yourself or knowing upfront what disk space is available is a good way to get into trouble.

- Always know how much disk space is available on your resource
- If you write results to disk, always clean up after yourself.
- Reading and writing directly to cloud storage is best but creates network bottlenecks possibly.

Cluster/Nodes

Last but not least I want to talk about a difficult topic … cluster size or node count. This can be tricky because so much depends on the tools and technology stack being used. It could be Spark, Kubernetes, Airflow … all these systems have multiple workers, each with their resources.

A general rule of thumb before you start working on a data pipeline is just to understand the system and resources you are working with … what do you have available. Even if only for the simple fact you could greatly speed up your data processing if you knew you had 10 works available and not just 1 or 2.

Any pipeline that isn't just running on a single machine … which is becoming less and less common, no matter the tech, is going to depend on the number and size of the workers available to process data.

You would be surprised at how just by knowing that you have x number of workers, each with x CPU and x RAM will allow you to calculate throughput and possible pipeline problems as the data grows.

Being able to calculate how much data you have, and what each worker could or should handle is something that a Data Engineer should become familiar with calculating. It will save you from wondering why a pipeline is running for more than 1 day or gets OOM errors.

If you never attempt to calculate data size and what resources are available for processing that data, then you are taking a shot in the dark, it may work out, or it may not!

We will cover more in-depth topics around servers and nodes, what common command-line tasks are used to interact with servers and clusters of computers in a later chapter.

Chapter 6 - Mastering SQL

Introduction To SQL

I don't intend to go very deep into SQL or Relational Databases in this chapter. My goal is just to give insights and high-level thoughts to those who are maybe just dipping their feet into the water. Sure, I will cover the basics of SQL and what you should learn, if you are unfamiliar with some of the topics, use that as a guide to show you what you should dig into later.

Feel free to skip this chapter if you have been a DBA or have been writing SQL for the last 15 years, although sometimes a reminder about the basics is helpful!

I've met my fair share of snooty people who poo-poo SQL and databases as second class hand-me-downs. I still remember talking to an academic computer science graduate who was explaining to me how he refused to teach database classes, he was just too good for that. Whatever. We, data engineers, know better!

Refusing to accept how 90% of companies can operate as data-driven businesses just isn't important to some people. There is probably nothing more important in the tool belt of a data engineer than being above average at SQL and databases.

Tuning queries, writing queries, indexing, designing data warehouses. I'm sure some Hadoop data engineers skipped this step of RDBMS world, but that is not the normal path of a data engineer. Let's dive into the fundamentals of SQL and databases.

Does the type of database matter?

Someone might ask this question. Shouldn't we worry about Postgres, MySQL, SqlServer, doesn't that matter? Not really. Being good at the fundamentals of SQL has little to do with a brand name slapped on top. Remember, we aren't talking about being a DBA here, we are talking about being a data engineer. We've aren't setting up automatic failover clusters. We are worried about understanding how to model tables, design indexes, and tune queries.

Think of it as working on your car, there are differences and you might have to check the manual a few times. But once you figure out how to change the oil, the brakes, air-filter, the light bulbs, you can work on almost any car.

All the angry DBA's right now can just send me emails that I will promptly delete. No, the type of RDBMS doesn't matter to learn SQL/database fundamentals.

The fundamentals of SQL/Databases.

We are going to cover a few basic topics, give you a ten thousand foot overview. There are plenty of books written on the inner workings of SQL and have devoted chapters to understanding table joins. I'm not going to do that. I want to teach you the concepts that can be applied very easily to any relational database.

OLTP vs. OLAP.
Table design/layout.
Understanding indexing basics.
How to write fast/tune queries.
Where to look for common problems.

OLTP vs. OLAP

I won't spend much time here, but understanding OLTP vs OLAP is pretty important. There are plenty of long-winded explanations about this, but it isn't that hard to grasp this concept. If you've designed a new set of tables in SQL for a project, it should fall into one of two categories.

OLTP = highly transactional in nature.
OLAP = analytical in nature, think data warehousing, aggregation.

Transactions vs analytics are the difference and should drive differences in how you think about modeling and handling the design of SQL tables. Highly transactional systems are very concerned about allowing for large volumes of inserts. Being able to support analytical queries in an OLTP system just isn't in the cards.

Many times OLTP tables will be wide with lots of columns, designed to handle high volumes of inserts. Think maybe an online order coming into a system, financial or banking transactions hitting a table. OLAP on the other hand is designed for aggregating and summarizing data, think analytics, and data warehousing. Hopefully, that makes sense to you.

For most data engineering work, we find ourselves working in the OLAP world and design. We think about structuring our data in a way that can answer questions with the aggregation of records.

Table design/layout.

I'm going to mostly talk about table design and layout that data engineers will run into. If you start reading a lot about the database and table design you will probably run into people talking about normal forms, "you should design this in the third normal form."

"Normalizing" a table is really about data de-duplication, and the extreme that you take it to in your table designs. Without worrying about exactly which normal form level you should be designing in your SQL tables, I suggest a more pragmatic and common-sense approach.

If you don't normalize your SQL tables, and up with just two large tables that hold the information needed, there is a good chance you didn't normalize your data model enough. If you end up with 15 tables for a relatively simple data set, you normalization too far.

Shoot for the middle ground. The easiest approach is just to group the datasets into their logical units. Think about a warehouse that holds information about customers and their orders. What is a logical way to group and design the SQL tables in a relational database?

- Customer information.
- Order information.
- Product information.

It isn't any harder than that. Complexity is always the enemy, especially if the situation doesn't warrant that complexity.

I do suggest you read up on this concept of database and table normalization if you plan on working in and around classical relational databases and data warehouse environments. I believe table designs are closely related to the OLTP and OLAP thoughts above. It isn't that hard to avoid bad table design, most of it is common sense. Here are some rules I've used in the past.

You have to know the queries first (Try to understand data access patterns, even at a high level before trying to design table layouts).
Break the data up into logical units (customer data, orders, products).

Don't put everything in one table (this use case is rare). When you're designing a table think about how it relates to other tables (ex. how will it join to other data). Simple is usually better than fancy (and means faster queries at the end most likely.) Understand data types (if you haven't thought through the data type of each value, then you aren't ready to design the table yet).

Note

I want to pause and talk about primary keys. This is glossed over by a lot of people but is essential to table design. One of the keys to understanding the correct design for a table is if you can easily list what columns make a row of data unique or different from all the other rows.

I'm going to repeat that because it is so key to table designs in relational databases. What piece of data, or pieces of data combined, make a record unique in your data set?

This is a concept of a primary key.

The primary key is the first thing you should define when designing a table, if you can't, then you don't understand the data well enough yet, or your table has a poor design.

Let's take a simple example. We want a set of tables in SQL that will help us hold information about hobbits, their friends, and the quests they go on.
## Example

```
SQL

CREATE TABLE middle_earth.Hobbits (hobbit_id INT NOT NULL,
                                   first_name VARCHAR(150),
                                   last_name VARCHAR(150),
                                   age INT DEFAULT 1,
                                   favorite_food VARCHAR(250)
                                  );
CREATE TABLE middle_earth.Quests (quest_id INT NOT NULL,
                                  quest_name VARCHAR(150),
                                  location VARCHAR(200),
                                  hobbit_id INT,
                                  treasure_value BIGINT
                                  );
CREATE TABLE middle_earth.Hobbit_Companions (companion_id INT NOT NULL,
                                             first_name VARCHAR(150),
                                             last_name VARCHAR(150),
                                             power_level INT DEFAULT 10
                                            );
CREATE TABLE middle_earth.Adventures (hobbit_id INT NOT NULL,
                                      companion_id INT NOT NULL,
                                      quest_id INT NOT NULL
                                     );
```

Hobbits, so I made a hobbit table. Hobbits always go on quests, so that is an obvious one. And of course, a hobbit will never leave his hobbit-hole without a companion, bingo.

Finally, they will set off on an adventure together, the last table. Just apply that simple logic to database table design and layout. Think about them as pieces of a puzzle and consider how they relate to each other.

So which hobbits have been going on adventures with whom? Can we answer this question for the tables designed above?

```
sql
SELECT h.first_name, h.last_name, h.first_name, h.last_name, q.quest_name
FROM Hobbits h
INNER JOIN Adventures a ON h.hobbit_id = a.hobbit_id
INNER JOIN Hobbit_Companions c ON c.companion_id = a.companion_id
INNER JOIN Quests q ON q.quest_id = a.quest_id AND q.hobbit_id = a.hobbit_id;
```

You get the idea. Table design and layout are half art and half science.

Here are good rules of thumb for Data Engineers when it comes to database table design.

- keep it simple
- break tables up into logical units
- always think about primary keys (what makes something unique)

Understanding indexing basics.

Yes, this is what can make a break your database design, OLAP or OLTP. The best tables in the world can quickly become useless with no or incorrect indexing. And honestly, the basics don't care about what kind of RDBMS you are running, SqlServer, Postgres, MySQL, it doesn't matter.

- Every table needs a primary key (uniqueness).
- Every table probably needs secondary indexes to support queries and joins.
- Don't go overboard, indexing too much can have a negative impact.

Every table should have indexes (I hope you are not surprised at this novel idea). Table join keys should be the primary and most important index (how two tables link together and relate to each other).

- Columns that show up in GROUP BY and WHERE clauses should be in indexes.

If you have to put 10 indexes on your table go back and read the section on table design and layout because you failed (or google Data Mart).
Think about what makes each table unique. This is a critical topic in database design.

Honestly, taking the very simple steps outlined above will put you way ahead of most people. Always define your primary keys, the uniqueness for each table. Understanding the queries that will be used on the tables, especially the WHERE and GROUP BY columns, indexes should be created to support those.

Think about the simple example from above. If you go back and look at the hobbit tables we did in the table design/layout section above it should be very obvious where to place indexes, hobbit_id , quest_id, and companion_id are pretty obvious choices, right? These columns are how some of the tables are related together and joined.

I think learning the very basic and obvious approach to database indexing will get you 80% of the way there. Of course, the topic is complicated and the examples and use cases can take years of learning to fine-tune. Lots to learn but stick with the basics and they will serve you well.

How to write fast/tune queries.

If you write SQL you will run into slow queries. There are can be two causes of this!

- Poorly written queries.
- Underlying database problems (administrative).

Firstly, let's talk about poorly written queries, again, regardless of the database system. A key concept is that databases were designed to have you work on data in sets, groups of data, not row by row.

- Never do SELECT * , the more data you are asking for the more time your query will take. Ask for what you need, nothing else.
- If you're going to do a subquery, don't put it in another SELECT statement, requiring it to be run for every row.
- Always have a WHERE clause. Since when do you need to run a query without a WHERE clause? Add more things to the WHERE clause. This is filtering your dataset to only what is needed, every time!
- Stop using VIEWS, they suck and always will.
- If you have complex data that requires periphery information in the end, don't pull that data in until the end. Aggregate first, pull in the details later.
- Stop using udf's, they suck as well.
- Do you have 7+ table joins in your first query before you've even started to subquery? Think again.
- Understand the data types and the data types of the table joins.

I could go on but you get the point again I'm sure. Just like in good table design, good query design and tuning starts with the simple approach. Think about the very basic things you need to accomplish first, do that, and then add the detail later. Try to use SQL that applies to the entire dataset, like GROUP BY, WHERE instead of using a udf or a SELECT in the SELECT statement.

Always try to pair the data down as much as possible before getting fancy.

Where to look for common problems.

I usually consider this the last resort. If everything else above is done properly you shouldn't have to dig farther down very often. But it will happen. And the problems are always side effects of what was not done above.

- Check indexes first. Are you joining on some column with no index?
- Are the index stats being updated?
- Look for bad SQL. Do the joins and subqueries look reasonable?
- Inspect the complex parts of the query. If something is complex it's probably like that for a reason. Find out why.
- Learn how to read query execution plans and stats.

Some SQL query problems can only be solved by getting way more into the weeds than you want to.

SQL has always been near and dear to my heart, being on data warehousing teams is how I cut my teeth on working in IT. After years of working around data warehousing, business intelligence, watching software engineers struggle with database design and queries, while at the same time being the best programmers, it becomes obvious to me that the simple things are what matters.

There are always cases where even the DBA's have problems, that's just life. But, most of the time sticking to the fundamentals will get you most of the way there.

SQL Fundementals

What I want to do in this section is just list the fundamentals of writing SQL. I'm going planning on teaching you how to learn each of the topics in depth. I just want to bring them up, give you a list of SQL features that you should be comfortable with. If you are unfamiliar with any of these, use it as an indication you need further study in this area.

Here are the SQL features I want to glance over at a high level.

- SELECT statements
- JOINS
- GROUP BY and WINDOW functions
- AGGREGATE functions
- WHERE clause
- SubQuerys
- CTEs

SELECT statements

SELECT statements in SQL are probably one of the first topics you learn. What should a good data engineer know about SELECT statements?

- Only ever SELECT the columns you need.
- Avoid putting complex logic in your SELECT statements unless necessary.

Not paying attention to SELECT statements in the culprit of many poor-performing queries. When you `SELECT *` from three different tables in your query, it should be no surprise things are slow.

Also, putting complex logic in SELECT statements usually means that logic must run on each row individually, in some cases, like CASE WHEN, this is needed, in my cases it is not, because it is expensive.

JOINS

JOIN statements are another topic most seasoned engineerings are very familiar with, yet it is surprising how many times the simple JOIN types can cause problems and bugs.

You should learn like the back of your hand, the different JOIN types, and when you should and shouldn't use them.

- INNER (only matching rows join)
- LEFT or RIGHT OUTER (keep only those records that have a match in one or the other)

- FULL (give me everything on both sides)
- ANTI JOINS (give me records only found on one side or the other)

What it boils down to is realizing when you need to use a INNER JOIN vs any other join. INNER joins will most likely always `drop` records (because it likely not everything matches between two tables). This is a simple concept, yet causes lots of problems in applications from Postgres to Spark SQL.

GROUP BY and WINDOW functions and AGGREGATE functions

I'm talking quickly about GROUP BY, WINDOW, and AGGREGATE functions because they are all used in conjunction with each other most of the time. You only would have GROUP BY or use a WINDOW function unless you are aggregating or "working on" some subsets of data.

Understanding generally what aggregation functions are available to you in most systems is key to SQL success, most data queries written by engineers will require some GROUP BY or WINDOW functions, usually in combination with some aggregate function.

These methods almost always "collapse" the data set, or in other terms, give some sort of smaller summary dataset.

WHERE clause

WHERE clauses are arguably one of the most important features of SQL, in a relational database or big data system like Spark. Why? Because any filtering or subsetting that can be done with a WHERE clause, as early as possible in the process, is going to ensure the best performance and run time of a query.

Figuring out WHERE clauses also typically give great insight into how the data should be indexed or partitioned. Many times it is advisable to spend time understanding common WHERE clauses and filters while a data model is being built before anything goes to production. This is how important the WHERE clause is.

SubQuerys vs CTEs

SubQuerys and CTEs are very much the same things, although each tool that provides them may have its nuances and performance issues.

CTEs

At a basic level, there isn't anything that special about a CTE, other than the possibility of recursion, which only applies to %.01 of the persons reading this.

- Defined using the WITH x AS (…) syntax.
- Are a temporary result set.
- Can be thought of as a temporary table.
- Can be referenced multiple times in a query.
- Encapsulate logic and code.

Subquery

Also known as the infamous nested query, usually gets a bad rap. Why? Overused just like anything else. They are a powerful way to slowly and methodically build up complex queries and logic.

There are two types of sub-queries.

- Correlated.
- Un-correlated.

What does that mean? It means that a subquery might or might not have some specific relation to the query "above" it, or "over" it.

CTE vs SubQuery Summary

The hard part about SQL is that most people don't consider it a "programming language," we can argue about that one later. The unfortunate downside of SQL not being considered as a programming language is that it misses out on some of the best software engineering principles. It's probably why the old Database Developers got such a bad rap back in the day. Piles of queries that look ugly with no testing.

Newer technology like SparkSQL and dbt are changing that. But, using CTEs is probably one of the best ways to apply the idea of encapsulating logic to the world of SQL.

Just like long, jumbled, drawn-out code written in Python with functions that are 100 lines long … SQL queries that might look impressive but take scrolling and scrolling and scrolling with sub-queries here and there and everywhere become ripe for bugs and headaches.

CTEs can help with this mess. You can name them what they are, logically group them, encapsulate specific code

Chapter 7 - Data Warehousing / Data Lakes

Closely related to the last topic of SQL is Data Warehousing and Data Lakes. Many things have changed since the days of data warehouses existing solely on relational databases. Now, many data lakes and data warehouses exist in cloud storage like s3. Even with these large changes, at the core, many of the same concepts still apply when designing classic RDBMS data warehouses or Data Lakes in the cloud on s3. But let's try to cover the basics upfront.

- Data Warehouse vs Data Lake vs Lake House
- Facts
- Dimensions
- Constraints and Schema
- Id's and Keys
- CDC / History Tracking
- Key Takeaways

Data Warehouse vs Data Lake vs Lake House

You've probably noticed that two terms come up a lot, many times they are mixed, and it's hard to truly know the difference, Data Warehouses vs Data Lakes, or even the new Lake House.

I would say most of the time you can chalk up these discussions at a high level to marketing. But, once you get past the marketing, there are usually some underlying assumptions based on if the datastore is a Data Lake vs Data Warehouse.

Many times when someone speaks of a Data Warehouse vs a Data Lake, they are probably talking about the data model of the data store. Data Warehouses are known to be strict Kimball or Immam style schemas, whereas many times Data Lakes start to blur this line and are less strict.

It depends on the context of the discussion for what the actual definition of a Data Warehouse is or a Data Lake. What the more important question is sometimes, what is the technology underlying your Data Warehouse or Data Lake.

For years a Data Warehouse was just another name for a relational database of some sort that had a Kimball or Imam style design applied to it. It's isn't so clear cut anymore. But, there are some things that both Data Lakes and Data Warehouses have in common.

Both places are the dumping ground for an organization's data. It's supposed to be the single source of truth to which all data flows and is captured. So what's the difference between a Data Warehouse and a Data Lake from there?

I don't know if there is a correct answer to this question, as many people define each differently, but it's safe to assume it has a lot to do with the data model applied. So, the data model (which is many times driven by the underlying technology) is the main difference between a Data Warehouse and a Data Lake.

You are probably aware that Data Warehouses and Data Lakes are there for OLAP, aka aggregation purposes. They are mostly used to answer high-level questions across the business that requires many times years of data. So, it comes down to how that data is stored in its "tables" and what the relationship looks like between the different data sets.

Generally speaking, both Data Warehouses and Data Lakes seek to at a minimum de-duplicate data, clean it, and store it logically, many times capturing historical versions of that data as well (SCD), called Slowly Changing Dimensions, aka data that changes over time.

With all that being said, let's dive into some topics you will most certainly run across as a Data Engineer, regardless of if you end up working with Data Warehouses or Data Lakes.

Facts and Dimensions.

Here are two concepts that should be a part of any Data Warehouse or Data Lake, fact and dimension tables. They will end up being the beating heart of any analytical platform, regardless of the semantics of the underlying data model.

Fact Tables.

Fact tables are the central dataset that grows the quickest and is used to aggregate data. Many times they have some sort of date or time series associated with them and are transactional in most cases.

Many times the data in a Fact table could also be known as an "event," a record that something happened. It could be a sales order, an invoice, a widget being produced, a user clicking on a button. These are data points that would naturally be summarized or aggregated.

```
transaction_id | transaction_date | amount | quantity
   6678932        2021-11-01         55.55     3
   9283234        2021-12-12         65.75     1
```

Dimension Tables.

Dimension tables are datasets that help describe and extend information in a Fact table. They are ancillary, like the leaves on a branch. They help us understand and interpret certain values in a Fact table and their primary goal is to be used as a "look-up" table.

```
customer_id | customer_name | customer_state | country
  55353      Billbo, Bagins        Iowa         USA
```

What is a Data Warehouse or Data Lake all about? It's all about aggregating values. Think about fact tables (that will be aggregated) as the hub in a spoke of a wheel. All the spindles coming off dimensions.

Let's look at a straightforward example of a Fact table and Dimension table. Of course, we will use hobbits as our example, we know that hobbits go on adventures and amass treasure for themselves.

```
sql
CREATE TABLE middle_earth.Adventures (
                                      hobbit_id INT NOT NULL,
                                      quest_id INT NOT NULL,
                                      date_quest_completed DATE,
                                      amount_of_treasure BIGINT DEFAULT 0,
                                     );
```

A good fact table in a data warehouse should consist of some nondescript ids, maybe a date, and the values you would want to aggregate on. Say like the amount of treasure a hobbit was able to compile over different time intervals.

What about the hobbit dimension that would describe a hobbit for us?

```
sql
CREATE TABLE middle_earth.Hobbit (
                                      hobbit_id INT NOT NULL,
                                      name STRING NOT NULL,
                                      age INT
                                     );
```

The Hobbit table is a good example of a dimension, a table that has important information that better describes and extends the data, but may not be critical for the calculation and aggregation of values.

How do we combine the two?

```
SELECT h.name, MONTH(a.date_quest_completed) as mnth, YEAR(a.date_quest_completed) a\
s yr, SUM(a,amount_of_treasure) as treasure,
FROM middle_earth.Adventures a
INNER JOIN middle_earth.Hobbit h on a.hobbit_id = b.hobbit_id
GROUP BY h.name, MONTH(a.date_quest_completed), YEAR(a.date_quest_completed);
```

So we can easily formulate how much treasure our hobbits are gathering over time. Remember simple = better = faster, when it comes to database queries. And data warehouses and data lakes tend to get large, so designing your Fact and Dimension tables correctly becomes important.

Constraints and Schema.

Another major role of the Data Warehouse or Data Lake is the ability to enforce rules upon data from various sources. This is what sets a Data Warehouse apart, you can have data from many different places all in a single spot, ready to act upon.

Constraints.

One of the best parts of the Data Warehouse is the ability to enforce constraints upon our data. It gives the ability to expect uniform values in the most critical data points. This concept is so key to the Data Lake and Data Warehouse, that even file-based storage systems like Delta Lake support constraints.

Now depending on the underlying technology, different constraints are supported in different systems, but it's good to have a general understanding of what's available.

Here are some common constraints that can be useful in ensuring data integrity.

- NULL or NOT NULL
- CHECK (is a value in some range or meets certain criteria.)

Here is a simple example to make the idea of constraints more concrete.

```
ADD CONSTRAINT dateEmployee CHECK (hireData > '1900-01-01')
```

These two basic constraints in themselves will go a long way into ensuring the correctness of data entering into the system. As you can see from the above example, you can be very basic or fairly complex in the constraints you add to your data points. Remember, each constraint has to run on INSERT, so don't go overboard either.

Schemas.

Schemas are the other important topic when talking about Data Warehouses or Data Lakes. What I'm mostly referring to when I talk about schemas in this context is the ability to control and enforce column names and data types.

Part of the problem with ingesting data from many different sources is that the data is, well, different. We are trying to take data points that are different and somehow commonize them into a single store from which we can answer any question.

- data types
- column names

In data engineering, you will soon realize it's always a few small but problematic issues that continue to plague most data systems decade after decade.

Data Types.

A Data Warehouse or Data Lake allows you to define all the data types you expect for each data point beforehand. Many times mixing of INTEGERS with STRINGS, say 1,2,3 with a,b,c will cause major issues that need to be fixed downstream. How can you SUM a column if it includes both INTEGERS and letters from the alphabet?

Column Names.

Another major area of concern in data engineering is the change of column names over time. It happens and will always happen, for many reasons. It could be a bug or it could be the business changing. Very few data sources stay static forever.

Example

```
CREATE TABLE ....
    ...
    order_amount DECIMAL,
    order_type STRING,
    ...
```

In the above example, we know that `order_amount` is going to be aggregated regularly, so we don't want STRINGS ending up in that column. As for `order_type` what if a new data set comes with a column titled `order_description`? These are the types of problems that a Data Warehouse or Data Lake aims to solve in the long run.

Nothing is more confusing than the "same" data source with different "versions" of data. This introduces complex transformations and logic that has to "combine" data sets so a full view can be taken of the single data source.

Data Warehouses and Data Lakes aim to solve this problem using data modeling, among other techniques.

The Role of ID's in a Data Warehouses or Data Lake.

Know that we know the basics of what a Data Warehouse or Data Lake is, how they are made up of Facts and Dimensions, and their primary purpose is the aggregation of data, we should touch on another topic that is critical to the function of a Data Warehouse or Data Lake. That topic is ids.

- Uniqueness

- Relationships

It’s critical to understand that the data repository that is the central hub of truth for an organization needs to be just that, the source truth. What do we mean by truth?

Uniqueness.

The data in the Lake or Warehouse is going to be de-duplicated and somewhat transformed. When we talked about primary keys and uniqueness in our chapter on SQL, this same topic applies and is even more important in the Warehouse or Lake world.

We can’t de-duplicate and have unique data records without having keys, primary or otherwise that identify our data. How can we relate Fact tables to their dimensions? This will be done by ids either in a SQL table or as hashes generated by code in a Data Lake.

It isn’t important how our particular technology requires us to generate an id for each data row, but that one takes that step in the first place. Why? The difference between data dumped into cloud storage and a Warehouse or Lake is that in the latter we’ve taken the time to understand and transform the data into a unique and usable dataset that can be related to other datasets easily.

Example

Let’s look at an example of some rows of data, and the challenge that uniqueness can pose.

```
# dataset 1
customer_id | order_amount | order_date | order_source
   12234         10.00       2021-12-01     56
   12234         10.00       2021-12-01     67
```

As you can see in the above example, if we had these records in an incoming data file, being ingested into our data warehouse, we might not have thought about this problem beforehand. A customer places 2 orders on the same day for the same amount.

This requires id’s to be generated for each row of data, in a Fact or Dimension table. This could be primary and foreign keys in a relational database or hashes of unique column(s) produced by our code and stored along with the data in our Lake in s3.

Being able to uniquely identify each Fact and join to Dimension on ideally a single column will require an id to be generated during the creation and loading of our Data Warehouse or Data Lake.

In Spark SQL code this might be simply creating a hash of the columns that make a record unique.

```
sha2(concat_ws("|", `customer_id`, `order_amount`, `order_date`, , `order_source`), \
256) as hash_primary_key,
```

Relationships.

We already mentioned this in passing above, but relationships are the crux of a good Data Warehouse or Data Lake. You are going to have a few main datasets or tables in your data store, how these data sets and tables relate to each other is the key.

That is why the keys or hashes we use in our table structures to uniquely identify each row are so important. We need to be able to JOIN our datasets together, without producing duplicates in our Data Lakes.

Data is really about relationships. Sure, we might be able to answer a few simple questions using a single table, but most business requirements require datasets to be joined to derive deep analysis that provides the best insights and answers.

When building out a Data Warehouse or Data Lake, much time should be spent on defining the relationships between our data tables and stores. We will dive into this more during the Data Modeling chapter.

CDC / History Tracking.

Probably one of the hardest parts of creating a Data Warehouse or Data Lake that provides the greatest challenge is the requirement for history tracking, also known as CDC, or change data capture.

It isn’t too bad once you’ve implemented it a few times, but in the beginning, it can be a little difficult to grasp how you can provide such features. There are a few different approaches to capturing data changes over time, I will pick one to go over to give you a good idea of what it can look like in the real world.

There are two key parts to history tracking.

- Effective start and end dates/times.
- Active flag.

This might make more sense with an example. Say we have a Dimension table of Customers or Accounts in our Data Lake, periodically someone moves and changes their postal code and address. How do we keep track that this customer’s information changed?

```
customer_id | ... | postal_code | ... | start_date | end_date | active_flag
 123                 50023              2020-12-12   2021-5-4      0
 123                 50035              2021-5-5     9999-9-9      1
```

Of course, this is a contrived example, but it gives a good idea of how important a start and end date/time, as well as an active flag, can be for tracking CDC changes over time in a Data Lake or Warehouse.

Active flag.

The active flag is important because if you just want the "current" snapshot of what is more recent, you can simply add a WHERE or FILTER to say `active_flag = 1`.

Effective date/times.

The effective start and end times of a record also provide a great history tracking of what "happened" to a customer or any data record for that matter. It gives the ability to time travel as well, to see "what this looked like on X date."

CDC implementation notes.

I'm not going to dive into details of all the technical details behind actually implementing CDC and history tracking in code. I will just give you a few highlights and hints that will work most of the time.

Of course, this can be a complex set of transformations to be able to track history records, but here is generally how you can accomplish such a task.

- get records in the source that ARE NOT in the target.
  - these become your INSERT new records (aka they have no previous)
- get records in sources THAT ARE in the target.
  - UPDATE those records in the TARGET to set "end" or "stop" datetime and turn OFF the active flag.
  - INSERT new records similar to step 1

Of course, there are many ways to do this, some people use MERGE statements, and technically you can combine steps 1 and 3. But generally speaking, you simply need to break up our source records into two groups .. those that exist in the target and those that do not. Then it's simply a matter of ending the records (via UPDATE) in the target that already exists, and inserting the new records.

Key Takeaways.

There are a few high-level key concepts that should be kept while designing Data Lakes or Data Warehouses. We glossed over a lot of topics, but I hope you have a better idea of what to expect when working on a Data Lake or Warehouse.

- Facts and Dimensions

It doesn't matter if you will be working on a Data Warehouse or Data Lake, the nuances are small between them, it's the details that matter. You should identify where the transactional data will accumulate with the most growth, these will be your fact tables. The dimensions will be more slowly growing, with ancillary data that will help extend and describe your fact tables.

Don't go overboard with data normalization, more on that later, the fewer the tables the better, they will be large. You should expect to have a few fact tables in the middle, with dimension tables that "surround" your facts, extending and describing them.

- Constraints and Schema

Never forget that placing constraints and the correct data types in your schemas of a Data Warehouse or Lake will save you time and trouble that will pay dividends in the end.

Validating inputs and commonizing the data to be the same is a key part of what a Data Lake provides and the value it brings. Spend the extra time to put value constraints in at high-value targets, columns that find themselves in WHERE clauses or filters.

- Id's and Keys

Having the correct id's or keys in our Warehouse is of utmost importance. Not having duplicates records, and having a tool that can uniquely identify each record solves many problems that can arise during a Data Lake project.

Ids and keys are critical to being able to reliable join different datasets, like Fact and Dimensions together, without producing duplicate and erroneous results.

Chapter 8 - Data Modeling

It might seem like this chapter rehashes a few topics we have already talked about during Data Lake/Warehousing and SQL chapters, and this is true because at the core of data modeling there are a few ideas and techniques that seem to apply to 90% of the Data Modeling that you will do as a Data Engineer.

You can use the same skills to model small sets of data stored in a few Postgres tables and hundreds of TB's of data stored in Parquet format in s3. Get these few basic data modeling skills right and they will help you in almost every data project you work on.

Don't ever forget that data modeling is half art and half science, many topics related to data modeling are heavily dependent on the use case in question. We will call this concept "access patterns," how is the data used by the consumers. A data model should reflect how the data is going to be used.

Is the data in question going to be aggregating many values over specific time series, and products? Then the data model should reflect that need and provide a schema for the data that can support such needs.

Here are the highlights before we dig in.

- Understand data types and schema.
- Group data logically.
- Think about the grain of the data (lowest level of detail).
- Think about the uniqueness of data.
- Think about access patterns (query).
- Think about normal forms.
- File-based modeling vs SQL.

Data Types and Schema.

When you have data that needs to be stored for use by downstream systems, before starting to design any part of the data model, begin by trying to understand the schema and data types of the data you will be working with.

This most often consists of understanding the schema, usually made up of a data type and definitions. *Each piece of data should be assigned a data type, constraints around that data, and a definition, including a name.*

- datatype
- constraints
- definition and name

Data Types.

Generally speaking, whatever data storage solution you are using, relational database or parquet, basic data types don't change very much. You should become familiar with a few of the most common.

- STRING or CHAR
- INT or BIG INT
- DECIMAL or FLOAT
- ARRAY
- BOOLEAN or TRUE/FALSE

Example

Usually, it's obvious based on the column name what the data type is, it's easy to fall into traps though. We should think carefully about assigning data types, should an `amount` by INT or DECIMAL?

```
account_id -> INT -> 1459345
account_name -> STRING -> "ABCDEFG"
purchase_amount -> DECIMAL -> 55.05
is_customer -> BOOLEAN -> True
product_ids -> ARRAY -> [1, 3, 6, 7]
```

Of course, these data types specifically will change slightly if you're using Postgres vs a parquet file, but generally, you will be using some form of a string or char, int or decimal, true and false boolean values, and sometimes arrays or list, or other more complex types.

Data Size.

Rember to consider data size when defining what data type should hold which data points. It's going to be a delicate balance, you don't want to run out of room with a `STRING` or `CHAR` with a certain size, but you also don't want to waste a lot of space by calling every column a `STRING` just to make things easy.

All data is stored on disk eventually, it could be cloud storage in `s3` or just a local drive on a server. Either way, data can add up over time, choosing the correct data type for your data points will have a large impact on storage size.

Data Engineers should spend time with their source data, examining values, looking for the min and max's of each column, looking at the lengths of strings, and using good old common sense.

It can be easy to fall into the trap of just glossing over data types, assigning certain values to `STRINGs` just because you can, or not understanding the `DECIMAL` points involved in a numeric column, even understanding if `INT` is a `BIG INT` or not.

Take your time to understand each data type in-depth, it will help you start with a solid data model at its foundation.

Constraints.

After the data types have been identified, the next task is to understand the constraints that surround each data point. This will be a familiar topic for those savvy SQL developers, but I want to extend the idea a little further.

Most people, especially those working in SQL systems, usually take a laxidazical approach to define constraints, both physical and theoretical. Yet even the easy and obvious constraints around data points are often ignored. But, anyone with experience can tell you that having robust constraints around your data is the first step in data quality.

What type of constraints should be thought about for each data point?

- NULLs or NOT NULL.
- CHECK (controls what values)
- uniqueness (keys and ids)
- foreign key (SQL - key/id integrity)

Constraints are a great way in either Data Warehouses or Data Lakes to control the quality and ensure the integrity of the data being stored. Dirty data is a classic problem, so catching problems early can be accomplished easily with constraints. Think of constraints as your first line of defense.

Constraints for numeric columns.

For numeric columns, constraints can check to make sure a value is in some range of sanity, it's a great way to ensure mistakes in source systems don't cascade farther into other parts where they are harder to extract and correct.

```
CONSTRAINT checkRange CHECK (product_id BEWTEEN 1 AND 50)
```

Constraints for string columns.

In string columns, constraints can also help ensure the filters and where clauses we will work. Miss spelling and typos have a way of finding a home in every system, constraints can catch those errors.

Take the simple example of a `response` column, let's say we define a constraint that says the value of the `response` column must be one of the following items … `['approved', 'declined', 'unknown']`. Now when a new data source comes in with the value of `decline` our constraint ran stop that problem in its track.

```
CONSTRAINT repCheck CHECK (response IN ['approved', 'declined', 'unknown'])
```

What if we didn't have the constraint in place? Now we have a column with both `declined` and `decline`, some unknowing analyst using our data source with a query that says `WHERE response = 'declined'` and is now getting incorrect results.

Data Definitions.

This topic is might seem a little basic, but it's a very important one nonetheless. Having a definition of the data points is a two-fold task.
Readable column names, that carry meaning, as well as descriptions of data points.

- legible column names with meaning
- descriptions

Legible Names.

Ensure you define column names that tell someone what the data point is without too much problem. Simply reading a column name should tell you a lot of what you need to know. If the column is a code, call it `..._code`, if it's a description, call it `..._description`. Simply calling something `code` instead of `response_code` or `description` instead of `product_description` just causes ambiguity.

Descriptions.

Descriptions should also be kept for each data point. This is usually easiest to do when creating the schema of the data at the first. The more time that goes by the less likely someone is just going to remember or know what a particular data set represents. Putting together documentation or comments in a codebase will come in handy later.

This topic seems boring and pointless until later when you are asked by the business what a definition is or where to find some obscure data point.

Modeling Data Logically.

Modeling data logically is probably one of the more obvious ways to enable an above-average data storage solution. What does it mean to define a data logical data model?

Logical data modeling is grouping sets of data that logically should be together. This is probably the easiest way to start a data model, for SQL or otherwise, as a high-level first pass attempt when starting a brand new model.

Here are a few obvious examples of how that might look. Let's say we work at a widget manufacturing company. That company makes and sells on their website many widgets. What kind of logical data model would we start to put together?

- Customer information
- Product information
- Order information
- Bill of material information
- Marketing information
- Accounting and Pricing information

This is a simple example of the logical data model. In your mind, you can already see the tables forming that would fall into these different groups.

Logical data models lead to physical relationships.

This is a great way to start because it gives a sense of the type of information that will be available, and even the relationships (joins) between the different logical groups of data.

Once you have broken data sets into logical units, the `JOINs` between those data sets start to present themselves to the designer. If you have a dataset of `orders` and a dataset of `customers`, you already know there is going to be a relationship between these two tables, even if you don't understand yet every nuance of what that `JOIN` might look like.

Logical data models are a good starting point for talks and reviews with businesses and stakeholders about the data needs and think about how the logical model could answer those questions. The logical data model will probably make the most sense to a nontechnical audience, they are also great tools for documentation when introducing new technical engineers to a data set.

Grain of Data.

The idea of the "grain" of the data might be a new one to you but is an important thought processing while data modeling. This becomes especially important during a Data Warehouse or Data Lake project, even with some smaller relational database designs as well.

The grain of data is the lowest level of information available on a topic. What's the mean in real life?

If you have a table that holds order information from customers, does the table hold a single record describing the overall order, even if multiple items were ordered? Or does the data table hold a record for each item that a customer orders, and to get the total order summary you have to roll up those individual order lines?

```
# orders table
order_id | product_id | quantity|
 456        4AB-C          3
 456        DCA-01         1
```

In the above example, we might assume that the table holds one record for each order, but this is not the case is it? The "grain" of this dataset is that each order can have multiple lines in the orders table, one for each product ordered. This is an important distinction.

That might seem like a small difference, but it becomes very important when data modeling. If you design a data model that relies on an Orders table having only a single line for each order, that is a summary of multiple items, what happens later when you realize you are missing important information that can only be obtained from a "lower" grain of data, means you need a data record for each item ordered.

Knowing the lowest level of information that is required by the business logic and requirements that are driving the data model is important. Identifying the lowest level of grain required should be one of the first tasks when data modeling because it directly affects the final schema of the data model.

Uniqueness of Data.

Another data modeling technique that should always be thought about before writing any code is the uniqueness of your data, and more specifically, each table or data sink you are trying to design and model should have something that uniquely identifies each record or row.

- Identify each record.
- Helps with `JOIN` relationships.
- Reduces the age-old duplicate row problem.
- Many times uniqueness is a combination of multiple columns.

This is important for several reasons. First, if you are unable to identify what makes a piece of data, or row, unique, then you most likely don't understand enough about the data. Second, the uniqueness factor of the data will tie directly into what are the primary keys that will eventually be needed for each record, or join later.

Duplicates in data are one of the oldest and most common problems that have plagued databases and data warehouses since the beginning of time. What was the problem, how did the duplicates get there?

I would argue it was a failure during the data modeling process to identify for each logical group of data, what made each record unique. It's typically a combination of data points, sometimes even combined with a date. Anyone working on a data model should ensure they understand what makes the data unique, and what information might have to be added to make a data record unique, even if that requires the addition of outside data like file name or insert date.

When duplicate records start to appear in data models, every join to other data sources starts to compound the problem. And, once duplicate records are placed into the system, they are typically painful to back out of and remove.

Access Patterns.

Access patterns are one of the least used data modeling techniques, mainly because they can be difficult to ascertain at the beginning of a project when a data model is taking place.

- Understand queries.
- How do downstream systems consume the data.
- Understand business needs and requirements for the data.

What is a data access pattern? It's how the data is queried and used in the end. When you're just starting to work on a data model for a system, this can be sometimes hard to know exactly what type of queries will show up, and how the data will be used. But, usually, you can get at least a decent idea by talking to the downstream people and applications that will be using your data and data model.

Example

It doesn't matter if your data modeling parquet files in Delta Lake or Postgres, if you can see the queries that will be run, or mock them up yourself, the access patterns become quite clear.

```
SELECT year, month, day, product, SUM(sales_amount) as sales
FROM company_orders
WHERE product IN [1, 2, 3] AND year >= YEAR(NOW());
GROUP BY year, month, day, product;
```

In the above example, we can see that our indexes or partitions will most likely break down along date parts like year and month, as well as product.

Talking to the Business.

For example, if you are working on a data warehouse or data lake, and you have a dataset with customer orders in it, you could talk to the product team, marketing team, and sales team, asking them what kind of reports and information they expect and want to answer.

Most likely you will hear them talk about how much product is sold by date, or what the top-selling products are, maybe which customers order a lot during which times of the year. If you keep having those conversations some things will start to become clear. Date and Product are very important to our end-users, and many of their questions revolve around those data points.

Business Requirements impact the Data Model.

How does a data model use this information? We know now that most of our data is going to be aggregated by date and product and customer. So these data values become prime targets for indexes or partitioning strategies depending on our technical stack.

Thinking about the columns and data points used in WHERE and GROUP BY clauses is key for data modeling correctly. If you don't understand how the data is going to be used there is a good chance the data model won't support the queries and access patterns that try to use that data source. This will lead to many problems, not least of all performance, and difficulty retrieving correct data with reasonable logic.

Normal Forms.

Many folks get glazed over their eyes when the topic of normalization forms comes up. I am one of those people, but it's important to take away some of the basics of normal form design when data modeling.

- De-duplication
- Join integrity

This topic was more important in the days of the classic relational data warehouse on SQL Server or Oracle, but has become less so today. You can loosely follow the levels of data normalization or de-duplication, and benefit from much of what they provide in a file-based Data Lake.

What are some of those old database normalization techniques that can be applied to even new Data Lakes in the cloud? There are a few timeless classic database normalizations that should be followed.

De-Duplication of Data.

Duplicate data creates many problems in a data sink, some of them not as obvious as you think. I'm not talking about duplicate data records that we discussed earlier, this is a different problem, I'm referring to data definitions that exist in more than one place.

Let's take an example of a database that holds information about specific products. What happens if we make the easy mistake of having product descriptions in more than one spot? This would be easy to miss. We might have a product table and an order table that holds the product_id and its description. What's the problem with this?

The product description changes, what do we do now? Now we have a problem if we didn't follow proper data normalization we could have product descriptions in several if not many tables, and now we have inconsistent data. The process of updating product descriptions becomes a very painful and expensive operation.

As you can see, one of the main tenants of normalization, the removal of redundant data is integral to data models of all kinds.

Join Integrity.

Probably one of the most complex and yet most important parts of data modeling is understanding how different data sets related to each or the joins.

May times there can be many joins between many tables, giving all sorts of results. Part of normalizing your data is controlling how these joins occur and the type of results they will give back.

Will joining two datasets cause one dataset to duplicate many rows? Will joining another table cause records to drop? In real terms, can someone assume that every single record in your orders table, that has a customer column, will have one and only one record in the joined customer table?

Think closely about your data model and how these joins will affect everyone and every query downstream.

Keys - Primary and Foreign.

Before someone starts complaining that their new Data Lake doesn't support foreign keys, let's remember we are trying to review proven techniques for data modeling that solve issues in the long run. Hint: relational databases support the concept of Primary and Foreign keys, but many new Data Lake technologies, like Delta Lake, do not.

Primary and foreign keys, regardless of if they are technically supported or not, and incredibly important topics that we should go through the exercise of designing, if for no other reason than to understand the data more closely during the modeling process.

The Idea Behind Keys.

What we are talking about here at a high level is the relationship between all the entities in our data model. *It doesn't matter what the underlying technology is beneath the model, the fact is that the data is only useful enough as far as it can be reliably related to other entities.*

Understanding how each table or sink of data can be related to others, and how that "join" would function is critical to any data model. Relationships between entities are core to any data product and should be looked at in-depth during the data modeling phase.

Generate your own keys if you must.

Just because you are using Delta Lake for example, on the cloud, and there isn't any official implementation of a foreign key that is enforced, doesn't mean that you shouldn't design a key yourself that acts as a foreign key, in fact, this is a good idea.

Having a key in each entity that can be used to relate to other entities easily will force better practices and long-term results and data usability. Being able to design keys of unique data the related entities to each other requires a deeper understanding of data sources. The simple process of going through this exercise of designing keys will make you the subject matter expert and result in a better data model.

If you are not familiar with data normalization I recommend you take some time to do some reading on the subject, the more you learn the more you will be able to pick and choose topics to apply to your next data model.

Relational Databases (SQL) vs Data Lake (File Based) Modeling.

Because of the new world we live in, I want to take time to review the differences between data modeling for SQL and relational databases, and file-based models, like Delta Lake.

The differences between a classic relational database data model and the new Data Lake data model are somewhat obscure, but important and obvious once you observe them.

The Differences

- SQL models with many dimensions, data norms, and de-dups to the extreme.
- File-Based models have fewer dimensions, normalization in moderation.
- SQL Database models table size doesn't matter much, small or big.
- File-Based data models table size and file size matter in the extreme.
- SQL Database models are centered around indexing and indexes.
- File-Based data models are centered around partitions and partitioning.

The number of Fact tables and Dimensions and normalization.

This just comes down to the technical implementation of say Postgres vs Apache Spark. The technology is just different, and of course, that is going to impact the data model greatly, this isn't much you can do about it.

Relational SQL databases blazing fast and joining many small tables is no big deal. That's typically why the classic Data Warehouse running on a SQL database had a look-up code and dimension for everything. Data deduplication and normalization were taken to the extreme in these data models … because they fit the technology.

The following queries are quite common in the classic data warehouse world.

```
SELECT fact.*, dim2.blah, dim3.blah, lookup1.blah, lookup2.blah
FROM fact
INNNER JOIN dim1 on fact.key = dim1.key
INNER JOIN dim2 on fact.key2 = dim2.key
INNER JOIN dim3 on fact.key3 = dim2.key
INNER JOIN lookup1 on fact.key4 = lookup1.key
INNER JOIN lookup2 on fact.key5 = lookup2.key
WHERE dim1.blah = x AND dim2.blah = y and lookup2.key = z
```

File size and table size matter in the new File-Based Data Lakes.

Most data engineering practitioners who've worked around Big Data and those tools like Hive, Spark, BigQuery, and the like will recognize that both small file sizes and small datasets can wreak havoc on those tools.

If you have 5 small-to-medium size datasets that have to be broadcast and joined with every single query in your Spark Delta or otherwise Data Lake … things probably are not going to work as you think. Technically most big data tools get slow when you start to include large numbers of small files or datasets that have to make their way across a distributed cluster network and disk that is behind those big data tools.

This wasn't a big deal back in the SQL relational database days because very few systems required sharding and the data was sitting on a drive or two on a single machine. This is not the case in the big data Data Lake. Hundreds of terabytes of data can be scattered across many hundreds to thousands of nodes depending on workload size.

Partitions vs Indexes.

Another major role in the data model differences between the new Data Lakes and the old Data Warehouses is partitioning vs indexes. The file-based Data Lakes are very much designed around the Partitions applied to the data, typically a minimum of some datetime series, and usually extending to other data attributes, like a customer, for example.

Because of the size of the data in the Data Lake file systems, it's an absolute necessity that the data be "broken-up" aka partitioned according to one or more attributes in that data table. Otherwise, the data of that size cannot be queried and accessed in a reasonable timeframe. Think of entire scans running on every single file of a few hundred terabyte data sources, of course, is not acceptable.

SQL relational databases on the other hand rely on primary and secondary keys to be able to seek, join, and filter the datasets quickly. The data models of classic relational databases are typically normalized, leading to many smaller tables with one or two large fact tables, typically designed around the primary key, or uniqueness value of a data set.

A typical Spark SQL query on a Data Lake might look as follows.

```
SELECT fact.*
FROM fact.{s3://location}
INNER JOIN dim1.{s3://location}
WHERE fact.date BETWEEN x AND Y AND fact.partition_key = x AND dim1.partition_key = y
```

Walking the data model line between old and new.

It's a fine line. There are many attributes of the old school Data Warehouse that tried and true, tested for decades, that should be brought into the new Data Lake. What it comes down to is that it's a fine line to walk. The differences in the technology stacks behind the scenes affect the data model.

We can all agree that modeling data as Facts and Dimensions is a tried and true way of organizing data for a Data Lake or Data Warehouse, the thought process around understanding what is a Fact table and what is a Dimension table enables high-value data models that can support a myriad of analytic use cases.

With file-based data models though, we cannot support a large number of small lookup and dimension tables, our big data tools will not respond well to such workloads. Whereas with a SQL relational data model, it would be foolish not to normalize the data to the extreme.

If file-based data models our data tables will be centered around and designed with Partition keys at the forefront. In file-based big data models, we don't care about the size of large tables, the bigger the better. The tooling built around Data Lakes was MADE to work a the terabyte level and above. In our SQL relational models, we have to normalize the tables and break them up as much as possible in an effort to keep the data size from exploding.

Chapter 9 - Data Quality

Data quality is a topic that is becoming more and more popular. In the past, it's always been sort of an afterthought, with solutions put in place after the fact that try to solve and identify problems with data quality. But, with the explosion of the size and types of data stored in the cloud, data quality has become a real and formidable problem for anyone to tackle. Expect to see more and more discussion and products around this topic of data quality in the future.

- What is Data Quality

- Business Quality
- Data Value Quality

- Measures of Data Quality

This chapter will be shorter than others. We will cover just some high-level basics about what to look for and what you can try to achieve when implementing data quality yourself. This is a new area of data engineering that has come into focus, so there is still a lot to learn and very few reliable tools available.

Most Data Quality tools today are made and implemented inside organizations big and small. So, really what we need is an understanding of what Data Quality is, how we can measure most data quality, and what some different implementations might look like.

What is Data Quality.

What is data quality? This is a difficult topic to put into precise words, mostly because every data source is different and has different business logic and requirements that surround it.

But, we can probably reason about some common Data Quality issues that come up again over and over regardless of data source and type. Data Quality can probably be split up into two main headings, the "business" quality standards, and the "data value" qualities of each data source and sink. Let's dive into each of these a little more.

Data Quality is also extremely important from an end-user and business point of view. It's typically poor data quality that users of our data notice first and immediately reduce their trust in our data products. Once that trust is lost, it's very hard to gain back.

Good Data Quality can ensure that trust and long-term use of the data and pipelines engineers work so hard on every day.

Business Quality

When you think about data quality, you might rightly think about a column that should not be NULL, and this is correct. But, such simple constraints can easily be built into most data stores. And yes, these simple constraints are core to data quality but are usually not the problems that are difficult to solve.

What is at the core of data quality?

- Human reasoning about data.
- Double meanings or standards.

Reasoning about Data.

The business end of Data Quality is more about how we "reason" about the data as humans who consume the data. We and the applications we build expect certain pieces of information to be true about the data we use. For example, we might have a data field called `response_code`, and we expect that field to contain a STRING value, but not just any string, either APPROVED or DECLINED, not a blank string '', or even a misspelled word like APPROVED.

This sort of Data Quality seems silly but think of the downstream aspects. If we have a Data Lake or Warehouse that end-users and scientists use to gather data and information, often using queries that filter by `response_code` of APPROVED, guess what, now all that downstream data is wrong because have records that say APPROVED that don't meet that criteria when they should.

Double meanings.

It's also imperative in our quest for data quality to identify and deal with all kinds of misunderstandings. Miss-understandings can come from many sources, it could be bad field names, data points that are too similar, or just ambiguity.

Many times businesss' struggle with data quality simply because there are no concrete quality definitions and rules surrounding the data. Just like in life, in data, many people may refer to the same data, but be talking about different things, or believe different things about that data.

This is a simple example but shows the importance of Business Data Quality and how we reason about the business values contained in each data point. It is an often overlooked part of Data Quality that leads to poor Data Quality and user experience that throws a shadow on the reliability of our data.

Data Value Quality.

On the other end of the spectrum from the business data quality surrounding how we reason about quality is the physical data values and ranges contained in each data point.

These value data qualities could be simple ideas and constraints that many are familiar with like NOT NULL or NULL, or even value CONTAINS x, y, or z. Typically you will find that a data set is a mix of two general types of data.

- Can contain any value.
- Values are in a range.

If you have a description column, there is a good chance that the values will range all over the board, a mix of numbers and characters of varying lengths. On the other hand, you might have an amount column that holds dollar values. Some obvious constraints are that it should be a positive decimal number holding up the 2 precision points.

When you break data values down, many times there is some sort of expected range of values for each column or data point. Maybe all the values should be one from a list of strings like YES and NO. It could be that it's a numerical value that should always hold an integer in the range of 1 to 1000. Whatever the case is, as your data grows in size and complexity it becomes a very serious and complex business to know what is happening over time to all those data points.

Measures of Data Quality.

Let's talk about a few basic data quality checks that should be performed on all data before it's ingested into any pipeline.

- Correct header or column names.
- Correct file formatting.
- Correct datatypes.
- Values ranges and values integrity.

All of the above measures are core to ensuring long-term data quality, and the best part is they are all fairly easy to approach and check with code in an automated fashion. These factors should be checked on each data set before it's ingested, avoiding costly pipeline downtime and bad data being propagated downstream causing trust and reliability issues.

Correct Header or Column Names.

Probably one of the most common causes of pipeline and data failures any data engineer will experience is unknown and unanticipated changes in column names. It could be new columns, columns going away, or just simple name changes.

The problem is that most pipelines and data transformations that are even moderately complex rely on an expected setlist of column names from which the source data is pulled. The minute a column name changes typically means a pipeline or some process shortly thereafter is going to break.

Luckily this is probably one of the easiest data quality tests to implement in any system. Each time a new file or data set hits our staging environment, a server, a cloud bucket, it's fairly simple to quickly open that file and pull the first line of the file, the header. Any change at this point, which is easy to check, we can automate alerts and quartine or move the "bad" file or data into another area out of harm's way.

Correct Data/File Formatting.

Another easy topic that is often the root cause of many data pipeline breakages in data engineering is simply the format of the file or data being ingested. This is probably the second most common error seen in most data ingestion systems.

Typically this can simply the take form of the wrong file extension showing up. It could be we get .txt files normally and all of sudden a .csv file appears, or the opposite. Whatever the case might be for the expected file formats, it's very easy to write automated checks on file extensions to determine if we have received what our system is expecting.

Closely related to the file extension we are expecting is the actual data format, typically the delimiters if applicable, to that dataset in question. Even if a pipeline receives a .txt or .csv file as in input as expected, what's to say that data contained in the file is actually in the correct format.

Did the file delimiter change from a comma to a pipe, or visa versa? These are easy mistakes to make, and therefore it is a very common problem data engineers will deal with. A simple read of a dataset and grabbing the first line, after the header, if it exists, we can easily and quickly check the format without reading the entire file.

Chapter 10 - DevOps for Data Engineers

DevOps for Data Engineers covers a lot of topics and technologies. Depending on the size of the company a Data Engineer works in they may, or may not, get very involved in DevOps and CI/CD. Many companies have a team of Platform Engineers and others whose job it is to make the development of other engineers simple and easy.

But, many companies on the smaller side don't have the resources to have a whole group of people working in DevOps and CI/CD. What I want to do in this chapter is give a quick run-through of some of the most important DevOps topics that will give you the biggest bang for your educational bucks.

- Dockerfiles and docker-compose
- Unit and Integration tests
- CI/CD options
- Automation

Dockerfiles and Docker-compose.

The lack of a Dockerfile in any data pipeline and repo I explore tell's me everything I need to know about the quality and setup of the codebase. Most people in the data world live their life without it, thinking that containerization is for the software engineers of the world, but this is not the case. If anything the Data Engineering and Data Science worlds have more of a use case for Dockerfiles than most.

Dockerfiles level the playing field by pre-packing all the OS and system requirements and dependencies into a single easy-to-use source.

Why data needs Dockerfiles.

It's pretty common today for most Data Engineering/Data Science/ML workloads to be Python-heavy. What are the best and worst part about PyPI and Python packages? They are incredibly finicky, break easily, cause requirement conflicts, and require a large amount of magic to not break over time.

What else is common for data workloads and pipelines, relational databases, and the connections that go with them. Anything else? An amazing number of command-line tools.

Why could there be more reasons? I'm glad you asked, yes there are more reasons. Typical complex data pipelines and codebases require environment variables, configurations, specific directories, and code layout.

Dockerfiles solve the complexity problem.

This is what a Dockerfile is for. Why not make life for yourself and others easier? With a simple docker run or docker-compose up command, everything that is needed to run and test pipeline code is at your fingertips. All the setup complexity is written once and hidden away rarely to be messed with again.

Reasons to use a Dockerfile for data pipeline(s).

- No surprise updates and breakages due to OS or package updates
- Easier onboarding new engineers into the codebase
- Requirements, configuration, env vars all become easier to manage.
- Everyone is on the same page, no windows vs mac vs Linux gotchas.
- Easier to transition code into distributed environments (think Kubernetes).
- Better DevOps (code deployment) and unit/integration testing.
- Makes you better at the command line (with makes you better in general)

Getting started with Dockerfiles.

Getting started with Dockerfiles. The first thing to do is install Docker desktop easy to use and easy to install.

There are two (probably three) options to write/use Dockerfiles for data pipelines.

First, it's good to understand DockerHub, it's where pretty much every project under the sun, plus some, stores official Dockerfile(s) for your use. Need to run Apache Spark? Why install it on your machine when you can get a Dockerfile with it already installed? Got a Python-based project? Why not just use the many Python Dockerfiles available.

These pre-build Dockerfiles can be obtained by a simple …

```
docker pull python  # or whatever else
```

The other option is to build your Dockerfile, based on whatever OS you want, with whatever packages and tools you need, even layered on top of some Dockerfile from someone else.

Let's take the example of someone who builds pipelines that run in AWS on Linux-based images. You want a good development base that is as close as possible or exactly like production correct? So you build a Dockerfile that has say Python and Spark-based on Linux with the aws cli installed.

```
FROM ubuntu:18.04

RUN apt-get update && \
    apt-get install -y default-jdk scala wget vim software-properties-common python3\
.8 python3-pip curl unzip libpq-dev build-essential libssl-dev libffi-dev

RUN wget https://archive.apache.org/dist/spark/spark-3.0.1/spark-3.0.1-bin-hadoop3.2\
.tgz && \
    tar xvf spark-3.0.1-bin-hadoop3.2.tgz && \
    mv spark-3.0.1-bin-hadoop3.2/ /spark && \
    ln -s /spark spark

RUN curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip\
" && \
    unzip awscliv2.zip && \
    ./aws/install

WORKDIR code
COPY . /code

RUN pip3 install -r requirements.txt

ENV MY_CODE=./code
```

It's just an example but you get the point, defining a complex set of tools that won't easily be broken that all developers and users of the pipeline can use is a very simple and powerful way to make development, testing, and code usage easy for all.

Usually, a Dockerfile written like this stored with the code can be built using a simple command.

```
docker build --tag my-special-image .
```

Docker-compose.

Also, make sure to read up on `docker-compose`. A great way to automate running tests and bits of code. Docker-compose is simply a way to have multiple services and images all work together and talk together, a great tool for more complex projects.

Dockerfile summary.

Dockerfiles are far from rocket science, they are probably one of the easiest things to learn, even as a new developer. Like anything else they can get complicated when running multiple services, but the basic usage of a Dockerfile will give you the 80% of what you need upfront.

I also believe Dockerfiles in general forces a more rigid development structure that is missing from a lot of data engineering code bases. When you find Dockerfiles you are more likely to find unit tests, documentation, requirements files, and generally better design patterns.

Unit Testing.

There are few things in life that are worse than cracking open some serious pipeline code, and then realizing there isn't a single function written to encapsulate logic, wondering if some change you are about to make will bring down the whole pipeline.

Why unit testing?

- increase learning
- leads to better quality code
- reduce bugs
- increase development time
- increase developer confidence
- can be automated into the DevOps and CI/CD

Increase learning.

When you are new to a codebase you don't know what you don't know, you don't have any backstory and you are usually flying by the seat of your pants in the beginning. When you have no unit tests, usually the only other way to test changes on a pipeline is to run it, this is sometimes easier said than done in a development environment. The first line of defense should be unit testing the entire pipeline.

Also simply reading unit tests can be a great way to start learning a codebase, before simply just trying to read the whole codebase.

Leads to better quality code.

When it comes to unit testing pipeline code, there is at least a baseline that must be followed. The critical ETL transforms of a script should be encapsulated inside a method/function.

It doesn't matter if you are more of a OOP or a Functional programmer, or both, anything is better than not breaking your code up into logical units that can be tested. This is how unit testing pushes the quality level of code up a few notches. You can't test code that isn't testable, so you have to write pipelines in such a way that they can be.

This is what I mean.

```
dataframe = dataframe.where(dataframe['that_column'] == 'this_value').groupBy('anoth\
er_column').agg(F.sum('yet_another').alias('new_column')).select('another_column', '\
new_column', when(dataframe['new_column'] > 1, 'yes').otherwise('no'))
```

Now imagine a pipeline with line after line of this type of code, with not a single piece of logic encapsulated in a method/function. This is way more common than people would think, I would venture a guess that 80% of pipelines are written this way.

Instead if we want testable code, we would do the following.

```
def filter_and_aggregate(input_df: DataFrame) -> DataFrame:
   output_df = input_df.where(dataframe['that_column'] == 'this_value').groupBy('ano\
ther_column').agg(F.sum('yet_another').alias('new_column')).select('another_column',\
 'new_column', when(dataframe['new_column'] > 1, 'yes').otherwise('no'))
   return output_df
```

Now, this is code that can be called in a unit test, and is reusable, making the code more clean and modular.

Reduce bugs.

An important part of DevOps is automating processes, and that allows for the reduction of bugs make it into production. Automated unit tests are probably the number one way we can use DevOps to reduce the chances of bugs getting out.

By making sure we unit test our code and making the running of those tests automated, reduces or risk of making errors and protects developers from making human mistakes.

You might be new to testing in general, so I want to take this chance to do a deep dive example in unit testing an imaginary PySpark pipeline.

Deep Dive PySpark Unit Testing.

Say we have a new PySpark pipeline we've had to develop for work, we want to be good DevOps and unit testing advocates, so here is what we do.

Setup directories and pytest.

Obviously, we need a test directory, a file to hold our unit tests test_pipeline.py, and a file specific to pytest called conftest.py.

```
>> mkdir tests
>> touch tests/test_pipeline.py
>> touch tests/conftest.py
```

Let’s talk about conftest.py for a minute.

Our conftest.py is going to hold a pytest fixture, this is a way to initialize each test function with whatever we want. In our case, to test PySpark code we are going to need a Spark Session.

```
# >> vim tests/conftest.py

import pytest
from pyspark.sql import SparkSession

@pytest.fixture(scope="session")
def spark_session():
    spark = SparkSession.builder.master("local[*]").appName("test").getOrCreate()
    return spark
```

Now we have a fixture that will provide a Spark Session for each of our unit tests, so we can test our code on actual Spark dataframes.

Write your first PySpark unit test.

Let’s go back to that original function we wrote, the first step would be to simply put the logic inside a function, no rocket science here.

```
import pyspark.sql.functions as F
from pyspark.sql import DataFrame

def sample_transform(input_df: DataFrame) -> DataFrame:
    inter_df = input_df.where(input_df['that_column'] == \
                              F.lit('hobbit')).groupBy('another_column').agg(F.sum('\
yet_another').alias('new_column'))
    output_df = inter_df.select('another_column', 'new_column', \
                                F.when(F.col('new_column') > 10, 'yes').otherwise('n\
o').alias('indicator')).where(
                F.col('indicator') == F.lit('yes'))
    return output_df
```

Now we actually have a function to unit test. So let’s write the test, we will crack open our test file…

```
# >> vim tests/test_pipeline.py

import pytest
from mycode import sample_transform

@pytest.mark.usefixtures("spark_session")
def test_sample_transform(spark_session):
    test_df = spark_session.createDataFrame(
        [
            ('hobbit', 'Samwise', 5),
            ('hobbit', 'Billbo', 50),
            ('hobbit', 'Billbo', 20),
            ('wizard', 'Gandalf', 1000)
        ],
        ['that_column', 'another_column', 'yet_another']
    )
    new_df = sample_transform(test_df)
    assert new_df.count() == 1
    assert new_df.toPandas().to_dict('list')['new_column'][0] == '70'
```

If we run our function sample_transform against our sample dataframe the following is the output.

```
+--------------+----------+---------+
|another_column|new_column|indicator|
+--------------+----------+---------+
|        Billbo|        70|      yes|
+--------------+----------+---------+
```

This is what we are trying to validate, that our filters, switching logic, and filter logic are done correctly and we get the expected outcome.

Keys to writing good unit tests.

As you can see writing unit tests for pipeline code isn’t very hard at all. It doesn’t take much effort and protects you and other people to ensure changes and initial code being written, with transform after transform, are being done correctly.

Setting up Docker to run our PySpark unit tests.

The next piece of unit testing our code is having somewhere to test it that insnt a production environment, somewhere anyone can do it, Docker of course. You will need a few items.

- Dockerfile
- Docker compose file

The Dockerfile doesn't need to be rocket science, a little Ubuntu, Java, Python, Spark …

using the below file run `docker build --tag spark-test .`

```
FROM ubuntu:18.04

RUN apt-get update && \
    apt-get install -y default-jdk scala wget vim software-properties-common python3\
.8 python3-pip curl unzip libpq-dev build-essential libssl-dev libffi-dev python3-de\
v&& \
    apt-get clean

RUN wget https://archive.apache.org/dist/spark/spark-3.0.1/spark-3.0.1-bin-hadoop3.2\
.tgz && \
    tar xvf spark-3.0.1-bin-hadoop3.2.tgz && \
    mv spark-3.0.1-bin-hadoop3.2/ /usr/local/spark && \
    ln -s /usr/local/spark spark

WORKDIR app
COPY . /app
RUN pip3 install cython==0.29.21 numpy==1.18.5 && pip3 install pytest pyspark pandas\
==1.0.5
ENV PYSPARK_PYTHON=python3
```

And our docker-compose file

```
version: "3.9"
services:
  test:
    environment:
      - PYTHONPATH=./src
    image: "spark-test"
    volumes:
      - .:/app
    command: python3 -m pytest
```

Now if you build your Dockerfile in your project, you should be able to just run `docker-compose up spark-test` and there you have it, your unit tests for PySpark running with one easy command.

```
test_1  | ============================= test session starts ========================\
======
test_1  | platform linux -- Python 3.6.9, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
test_1  | rootdir: /app
test_1  | collected 1 item
test_1  |
test_1  | test/test_pipeline.py .                                                  [\
100%]
test_1  |
test_1  | ============================== 1 passed in 5.08s =========================\
======
```

Unit testing your pipeline code really isn't that hard, and it saves a lot of time later down the road. Being able to make changes to code and have some idea if you breaking anything, without running the entire pipeline, is kinda nice. Data Engineers have gotten a bad rap over the years because unwillingness to incorporate basic software engineering principles like unit tests.

Obviously, from above it doesn't take much effort to add unit tests for your Spark code. Encapsulate your logic, write a few docker files, and you are off to the races.

CI/CD.

You may not like it but CI/CD is a major part of DevOps and has become critical to good Data Engineering pipelines. Let's face it, we all know we do too many things manually, and it gets even harder when we have multiple people working in some repositories.

At some point you have cannot rely on some person, no matter how smart, to do a manual sync of code or build a new Docker image if that is the case. This is where CI/CD comes into play.

Now there is a myriad of tools that will probably continue to grow and expand, Jenkins, GitLab/GitHub CI/CD runners etc. Many tools have been built around taking actions when something happens in a repository of data. This is something that all Data Engineers need to learn.

Automation is the name of the game.

What most CI/CD and DevOps boils down to is automation. It doesn't matter what the problem is, if you have developers spending a lot of time running the same command over and over again, said to build, tag, and deploy a Docker image, this is something DevOps can help with.

You need to learn to be comfortable around the command line and bash, and with different OS environments.

Being able to stitch together multiple commands, API calls is key to being a good Data Engineer.