A Beginner's Guide to Data Engineering — Part I

Data Engineering: The Close Cousin of Data Science

Robert Chang Jan 8, 2018 · 14 min read ★

Image credit: A beautiful former slaughterhouse / warehouse at Matadero Madrid, architected by Iñaqui Carnicero

Motivation

The more experienced I become as a data scientist, the more convinced I am that data engineering is one of the most critical and foundational skills in any data scientist's toolkit. I find this to be true for both evaluating project or job opportunities and scaling one's work on the job.

In an earlier post, I pointed out that a data scientist's capability to convert data into value is largely correlated with the stage of her company's data infrastructure as well as how mature its data warehouse is. This means that a data scientist should know enough about data engineering to carefully evaluate how her skills are aligned with the stage and need of the company. Furthermore, many of the great data scientists I know are not only strong in data science but are also strategic in leveraging data engineering as an adjacent discipline to take on larger and more ambitious projects that are otherwise not reachable.

Despite its importance, education in data engineering has been limited. Given its nascency, in many ways the only feasible path to get training in data engineering is to learn on the job, and it can sometimes be too late. I am very fortunate to have worked with data engineers who patiently taught me this subject, but not everyone has the same opportunity. As a result, I have written up this beginner's guide to summarize what I learned to help bridge the gap.

Organization of This Beginner's Guide

The scope of my discussion will not be exhaustive in any way, and is designed heavily around **Airflow**, **batch data processing**, and **SQL**-like languages. That said, this focus should not prevent the reader from getting a basic understanding of data engineering and hopefully it will pique your interest to learn more about this fast-growing, emerging field.

- **Part I (this post)** is designed to be a high-level introductory post. Using a combination of personal anecdotes and expert insights, I will contextualize what data engineering is, why it is challenging, and how it can help you or your organization to scale. The primary audience of this post is for *aspiring data scientists* who need to learn the basics to evaluate job opportunities or *early-stage founders* who are about to build the company's first data team.
- **Part II** is more technical in nature. This post will focus on Airflow, Airbnb's open-sourced tool for programmatically author, schedule, and monitor workflows. Specifically, I will demonstrate how to write a Hive batch job in Airflow, how to design table schemas using techniques such as star schema, and finally highlight some best practices around building ETLs. This post is suitable for *starting data scientists* and *starting data engineers* who are trying to hone their data engineering skills.
- **Part III** will be the final post of this series, where I will describe advanced data engineering patterns, higher level abstractions, and extended frameworks that would make building ETLs a lot easier and more efficient. A lot of these patterns are taught to me by Airbnb's experienced data engineers who learned the hard way. I find these insights particularly useful for *seasoned data scientists* and *seasoned data engineers* who are trying to further optimize their workflows.

Given that I am now a huge proponent for learning data engineering as an adjacent discipline, you might find it surprising that I had the completely opposite opinion a few years ago — I struggled a lot with data engineering during my first job, both motivationally and emotionally.

My First Industry Job out of Graduate School

Right after graduate school, I was hired as the first data scientist at a small startup affiliated with the Washington Post. With endless aspirations, I was convinced that I will be given analysis-ready data to tackle the most pressing business problems using the most sophisticated techniques.

Shortly after I started my job, I learned that my primary responsibility was not quite as glamorous as I imagined. Instead, my job was much more foundational — to maintain critical pipelines to track how many users visited our site, how much time each reader spent reading contents, and how often people liked or retweeted articles. It was certainly important work, as we delivered readership insights to our affiliated publishers in exchange for high-quality contents for free.

Secretly though, I always hope by completing my work at hand, I will be able to move on to building fancy data products next, like the ones described here. After all, that is what a data scientist is supposed to do, as I told myself. Months later, the opportunity never came, and I left the company in despair. Unfortunately, my personal anecdote might not sound all that unfamiliar to early stage startups *(demand)* or new data scientists *(supply)* who are both inexperienced in this new labor market.

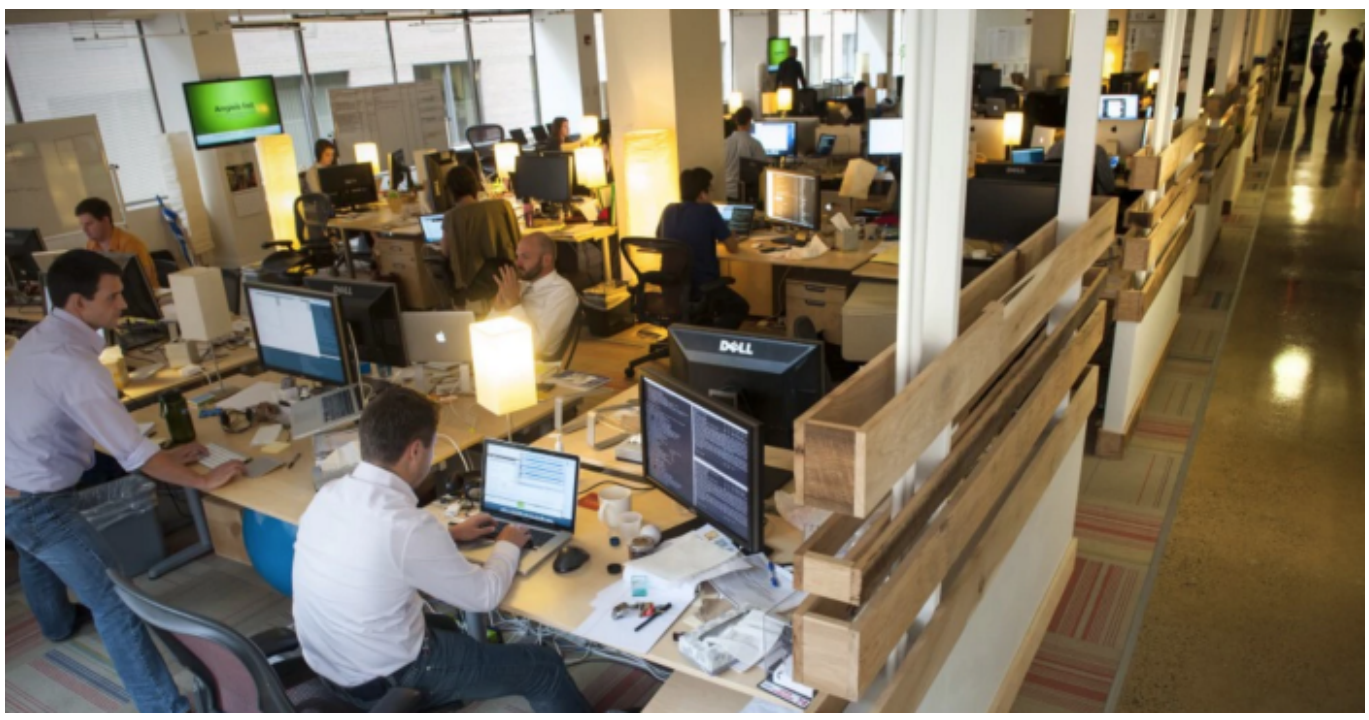


Image credit: Me building ETL pipelines diligently (guy in blue in the middle)

Reflecting on this experience, I realized that my frustration was rooted in my very little understanding of how real life data projects actually work. I was thrown into the wild west of raw data, far away from the comfortable land of pre-processed, tidy .csv files, and I felt unprepared and uncomfortable working in an environment where this is the norm.

Many data scientists experienced a similar journey early on in their careers, and the best ones understood quickly this reality and the challenges associated with it. I myself also adapted to this new reality, albeit slowly

and gradually. Over time, I discovered the concept of instrumentation, hustled with machine-generated logs, parsed many URLs and timestamps, and most importantly, learned SQL (*Yes, in case you were wondering, my only exposure to SQL prior to my first job was Jennifer Widom's awesome MOOC here*).

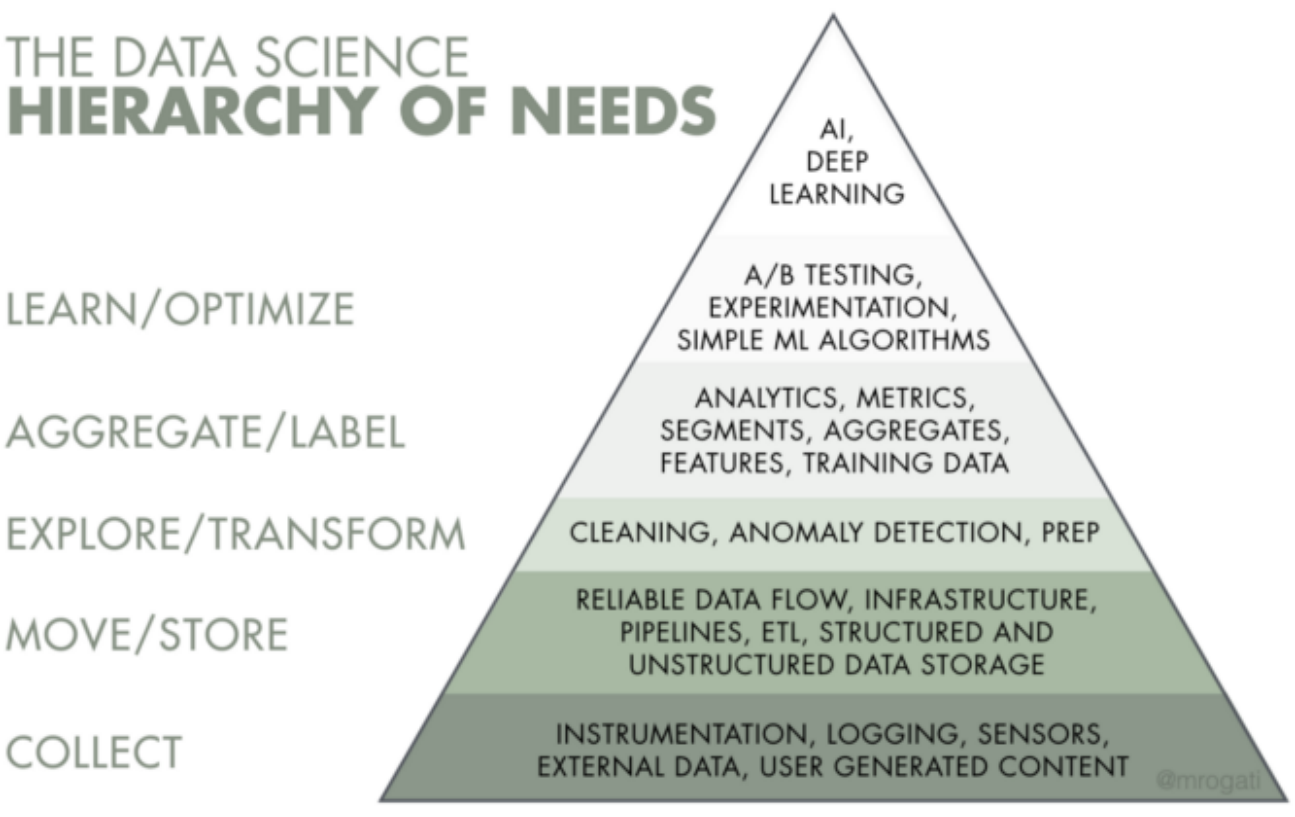
Nowadays, I understand counting carefully and intelligently is what analytics is largely about, and this type of foundational work is especially important when we live in a world filled with constant buzzwords and hypes.

The Hierarchy of Analytics

Among the many advocates who pointed out the discrepancy between the grinding aspect of data science and the rosier depictions that media sometimes portrayed, I especially enjoyed Monica Rogati's call out, in which she warned against companies who are eager to adopt AI:

> *Think of Artificial Intelligence as the top of a pyramid of needs. Yes, self-actualization (AI) is great, but you first need food, water, and shelter (data literacy, collection, and infrastructure).*

This framework puts things into perspective. Before a company can optimize the business more efficiently or build data products more intelligently, layers of foundational work need to be built first. This process is analogous to the journey that a man must take care of survival necessities like food or water before he can eventually self-actualize. This rule implies that companies should hire data talents according to the order of needs. One of the recipes for disaster is for startups to hire its first data contributor as someone who only specialized in modeling but have little or no experience in building the foundational layers that is the pre-requisite of everything else (*I called this "The Hiring Out-of-Order Problem"*).



Source: Monica Rogati's fantastic Medium post "The AI Hierarchy of Needs"

Unfortunately, many companies do not realize that most of our existing data science training programs, academic or professional, tend to focus on the top of the pyramid knowledge. Even for modern courses that encourage students to scrape, prepare, or access raw data through public APIs, most of them do not teach students how to properly design table schemas or build data pipelines. As a result, some of the critical elements of real-life data science projects were lost in translation.

Luckily, just like how software engineering as a profession distinguishes front-end engineering, back-end engineering, and site reliability engineering, I predict that our field will be the same as it becomes more mature. The composition of talent will become more specialized over time, and those who have the skill and experience to build the foundations for data-intensive applications will be on the rise.

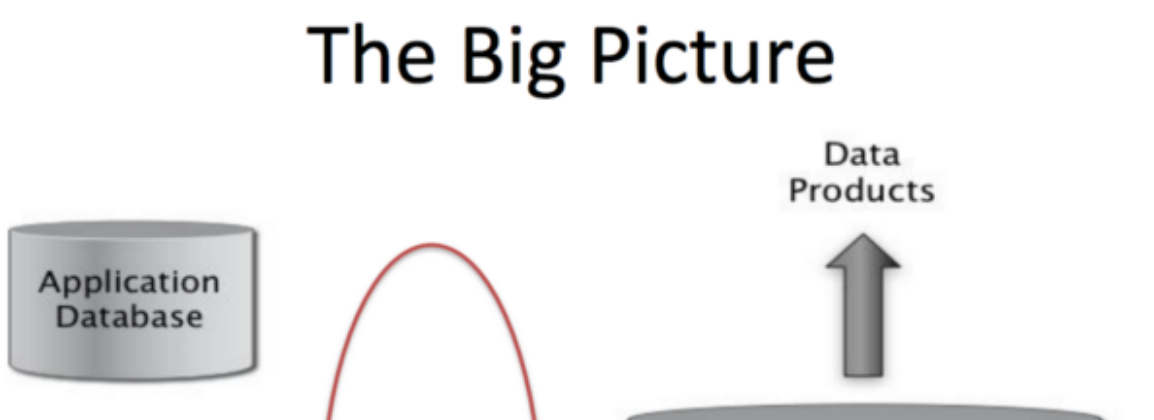
What does this future landscape mean for data scientists? I would not go as far as arguing that every data scientist needs to become an expert in data engineering. However, I do think that every data scientist should know enough of the basics to evaluate project and job opportunities in order to maximize *talent-problem fit*. If you find that many of the problems that you are interested in solving require more data engineering skills, then it is never too late then to invest more in learning data engineering. This is in fact the approach that I have taken at Airbnb.

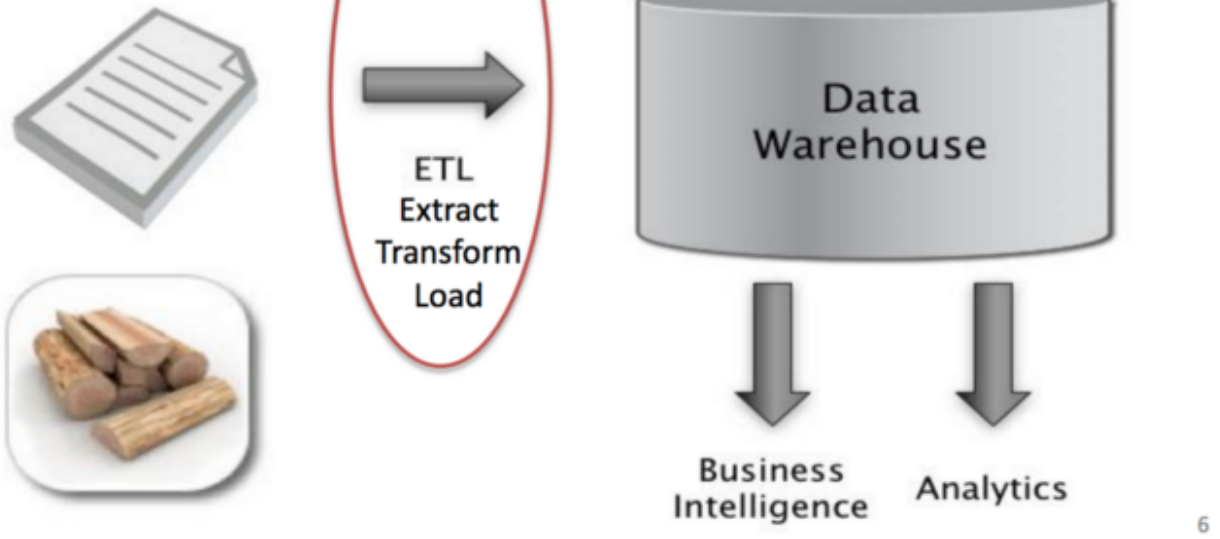
Building Data Foundations & Warehouses

Regardless of your purpose or interest level in learning data engineering, it is important to know exactly what data engineering is about. Maxime Beauchemin, the original author of Airflow, characterized data engineering in his fantastic post **The Rise of Data Engineer**:

> *Data engineering field could be thought of as a superset of* ***business intelligence*** *and* ***data warehousing*** *that brings more elements from* ***software engineering****. This discipline also integrates specialization around the operation of so called "big data" distributed systems, along with concepts around the extended Hadoop ecosystem, stream processing, and in computation at scale.*

Among the many valuable things that data engineers do, one of their highly sought-after skills is the ability to design, build, and maintain **data warehouses**. Just like a retail warehouse is where consumable goods are packaged and sold, a data warehouse is a place where raw data is transformed and stored in query-able forms.





Source: Jeff Hammerbacher's slide from UC Berkeley CS 194 course

In many ways, data warehouses are both the engine and the fuels that enable higher level analytics, be it business intelligence, online experimentation, or machine learning. Below are a few specific examples that highlight the role of data warehousing for different companies in various stages:

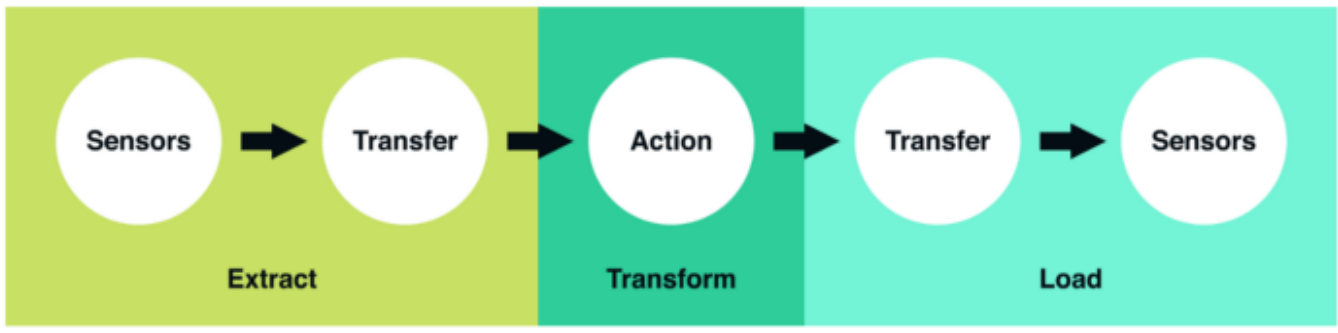
- **Building Analytics at 500px**: In this post, Samson Hu explains the challenges 500px faced when it tried to grow beyond product-market fit. He describes in details the process how he built the data warehouse from the ground and up.
- **Scaling Airbnb's Experimentation Platform**: Jonathon Parks demonstrates how Airbnb's data engineering team built specialized data pipelines to power internal tools like the experimentation reporting framework. This work is critical in shaping and scaling Airbnb's product development culture.
- **Using ML to Predict the Value of Homes on Airbnb**: Written by myself, I explain why building batch training, offline scoring machine learning models requires a lot of upfront data engineering work. Notably, many tasks associated with feature engineering, building, and backfilling training data resemble data engineering works.

Without these foundational warehouses, every activity related to data science becomes either too expensive or not scalable. For example, without a properly designed business intelligence warehouse, data scientists might report different results for the same basic question asked at best; At worst, they could inadvertently query straight from the production database, causing delays or outages. Similarly, without an experimentation reporting pipeline, conducting experiment deep dives can be extremely manual and repetitive. Finally, without data infrastructure to support label collection or feature computation, building training data can be extremely time consuming.

ETL: Extract, Transform, and Load

All of the examples we referenced above follow a common pattern known as ETL, which stands for **E**xtract, **T**ransform, and **L**oad. These three conceptual steps are how most data pipelines are designed and structured. They serve as a blueprint for how raw data is transformed to analysis-ready

data. To understand this flow more concretely, I found the following picture from Robinhood's engineering blog very useful:



Source: Vineet Goel's "Why Robinhood uses Airflow?" Medium Post

- **Extract**: this is the step where sensors wait for upstream data sources to land (e.g. a upstream source could be machine or user-generated logs, relational database copy, external dataset … etc). Upon available, we transport the data from their source locations to further transformations.
- **Transform**: This is the heart of any ETL job, where we apply business logic and perform actions such as filtering, grouping, and aggregation to translate raw data into analysis-ready datasets. This step requires a great deal of business understanding and domain knowledge.
- **Load**: Finally, we load the processed data and transport them to a final destination. Often, this dataset can be either consumed directly by end-users or it can be treated as yet another upstream dependency to another ETL job, forming the so called data lineage.

While all ETL jobs follow this common pattern, the actual jobs themselves can be very different in usage, utility, and complexity. Here is a very simple toy example of an Airflow job:

```
"""
A DAG definition file in Airflow, written in Python.
"""
from datetime import datetime, timedelta
from airflow.models import DAG  # Import the DAG class
from airflow.operators.bash_operator import BashOperator
from airflow.operators.sensors import TimeDeltaSensor

default_args = {
    'owner': 'you',
    'depends_on_past': False,
    'start_date': datetime(2018, 1, 8),
}

dag = DAG(
    dag_id='anatomy_of_a_dag',
    description="This describes my DAG",
    default_args=default_args,
    schedule_interval=timedelta(days=1))   # This is a daily DAG.

# t0 and t1 are examples of tasks created by instantiating operators
t0 = TimeDeltaSensor(
    task_id='wait_a_second',
    delta=timedelta(seconds=1),
    dag=dag)

t1 = BashOperator(
    task_id='print_date_in_bash',
    bash_command='date',
    dag=dag)
```

```
31
32  t1.set_upstream(t0)
```

airflow_toy_example_dag.py hosted with ♡ by GitHub view raw

Source: Arthur Wiedmer's workshop from DataEngConf SF 2017

The example above simply prints the date in bash every day after waiting for a second to pass after the execution date is reached, but real-life ETL jobs can be much more complex. For example, we could have an ETL job that extracts a series of CRUD operations from a production database and derive business events such as a user deactivation. Another ETL can take in some experiment configuration file, compute the relevant metrics for that experiment, and finally output p-values and confidence intervals in a UI to inform us whether the product change is preventing from user churn. Yet another example is a batch ETL job that computes features for a machine learning model on a daily basis to predict whether a user will churn in the next few days. The possibilities are endless!

Choosing ETL Frameworks

When it comes to building ETLs, different companies might adopt different best practices. Over the years, many companies made great strides in identifying common problems in building ETLs and built frameworks to address these problems more elegantly.

In the world of batch data processing, there are a few obvious open-sourced contenders at play. To name a few: Linkedin open sourced Azkaban to make managing Hadoop job dependencies easier. Spotify open sourced Python-based framework Luigi in 2014, Pinterest similarly open sourced Pinball and Airbnb open sourced Airflow (also Python-based) in 2015.

Different frameworks have different strengths and weaknesses, and many experts have made comparisons between them extensively (see here and here). Regardless of the framework that you choose to adopt, a few features are important to consider:

	Luigi	Airflow	Pinball
repo	https://github.com/spotify/luigi	https://github.com/airbnb/airflow	https://github.com/pinterest/pinball
docs	http://luigi.readthedocs.org	https://airflow.readthedocs.org	none
my review	http://bytepawn.com/luigi.html	http://bytepawn.com/airflow.html	http://bytepawn.com/pinball.html
github forks	750	345	58
github stars	4029	1798	506
github watchers	319	166	47
commits in last 30 days	lots of commits	lots of commits	3 commits
architecture			
web dashboard	not really, minimal	very nice	yes
code/dsl	code	code	python dict + python code
files/datasets	yes, targets	not really, as special tasks	?
calendar scheduling	no, use cron	yes, LocalScheduler	yes
datadoc'able [1]	maybe, doesn't really fit	probably, by convention	yes, dicts would be easy to parse
backfill jobs	yes	yes	?
persists state	kindof	yes, to db	yes, to db
tracks history	yes	yes, in db	yes, in db
code shipping	no	yes, pickle	workflow is shipped using pickle, jobs are not?

Source: Marton Trencseni's comparison between Luigi, Airflow, and Pinball

- **Configuration**: ETLs are naturally complex, and we need to be able to succinctly describe the data flow of a data pipeline. As a result, it is important to evaluate how ETLs are authored. Is it configured on a UI, a

domain specific language, or code? Nowadays, the concept of *configuration as code* is gaining popularity, because it allows users to expressively build pipelines programmatically that are customizable.

- **UI, Monitoring, Alerts:** Long running batch processes inevitably can run into errors (e.g. cluster failures) even when the job itself does not have bugs. As a result, monitoring and alerting are crucial in tracking the progress of long running processes. How well does a framework provide visual information for job progress? Does it surface alerts or warnings in a timely and accurate manner?
- **Backfilling:** Once a data pipeline built, we often need to go back in time and re-process the historical data. Ideally, we do not want to build two separate jobs, one for backfilling historical data and another for computing current or future metrics. How easy does a framework support backfilling? Can it do so in a way that is standardized, efficient, and scalable? All these are important questions to consider.

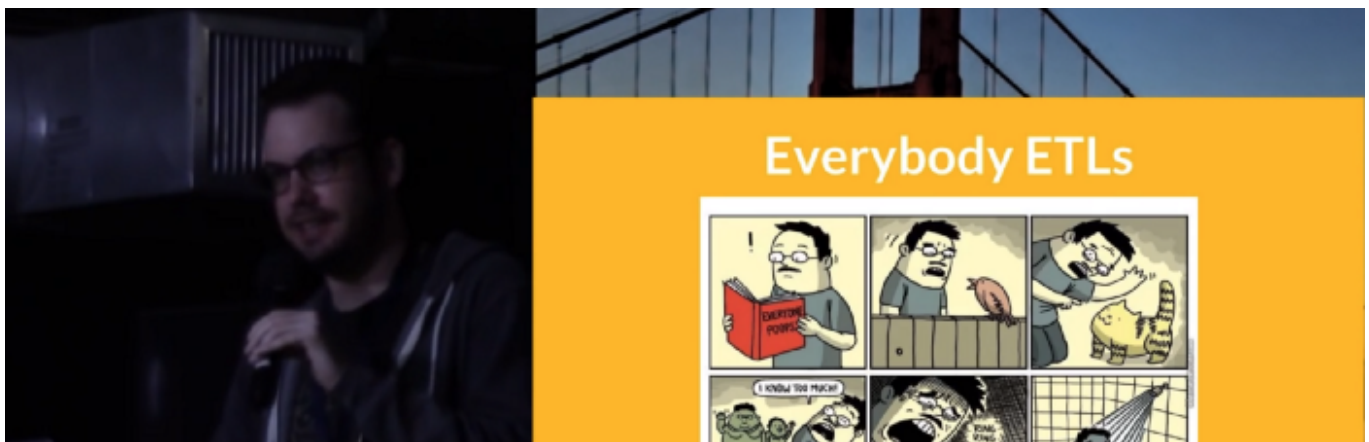
Naturally, as someone who works at Airbnb, I really enjoy using Airflow and I really appreciate how it elegantly addresses a lot of the common problems that I encountered during data engineering work. Given that there are already 120+ companies officially using Airflow as their de-facto ETL orchestration engine, I might even go as far as arguing that Airflow could be the standard for batch processing for the new generation start-ups to come.

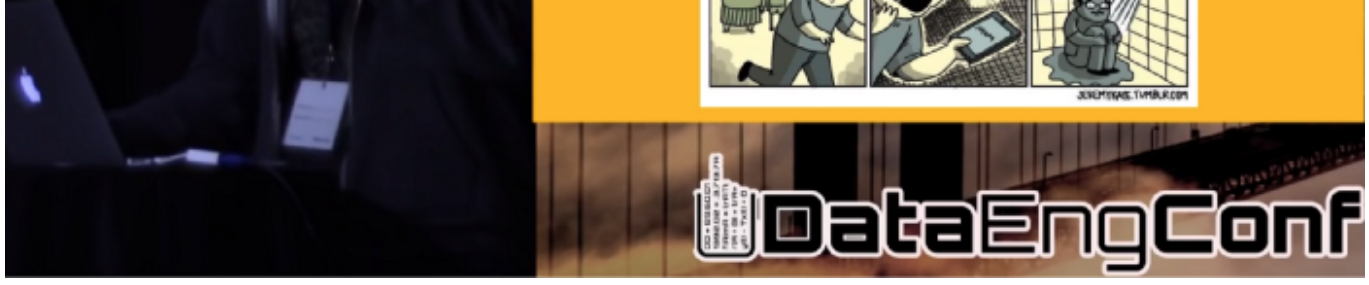
Two Paradigms: SQL- v.s. JVM-Centric ETL

As we can see from the above, different companies might pick drastically different tools and frameworks for building ETLs, and it can be a very confusing to decide which tools to invest in as a new data scientist.

This was certainly the case for me: At Washington Post Labs, ETLs were mostly scheduled primitively in Cron and jobs are organized as Vertica scripts. At Twitter, ETL jobs were built in Pig whereas nowadays they are all written in Scalding, scheduled by Twitter's own orchestration engine. At Airbnb, data pipelines are mostly written in Hive using Airflow.

During my first few years working as a data scientist, I pretty much followed what my organizations picked and take them as given. It was not until much later when I came across Josh Will's talk did I realize there are typically two ETL paradigms, and I actually think data scientists should think very hard about which paradigm they prefer before joining a company.





Video Source: Josh Wills' Keynote @ DataEngConf SF 2016

- **JVM-centric ETL** is typically built in a JVM-based language (like Java or Scala). Engineering data pipelines in these JVM languages often involves thinking data transformation in a more imperative manner, e.g. in terms of key-value pairs. Writing User Defined Functions (UDFs) are less painful because one does not need to write them in a different language, and testing jobs can be easier for the same reason. This paradigm is quite popular among engineers.
- **SQL-centric ETL** is typically built in languages like SQL, Presto, or Hive. ETL jobs are often defined in a declarative way, and almost everything centers around SQL and tables. Writing UDFs sometimes is troublesome because one has to write it in a different language (e.g. Java or Python), and testing can be a lot more challenging due to this. This paradigm is popular among data scientists.

As a data scientist who has built ETL pipelines under both paradigms, I naturally prefer SQL-centric ETLs. In fact, I would even argue that as a new data scientist, you can learn much more quickly about data engineering when operating in the SQL paradigm. Why? Because learning SQL is much easier than learning Java or Scala (unless you are already familiar with them), and you can focus your energy on learning DE best practices than learning new concepts in a new domain on top of a new language.

Wrapping Up Beginner's Guide — Part I

In this post, we learned that analytics are built upon layers, and foundational work such as building data warehousing is an essential prerequisite for scaling a growing organization. We briefly discussed different frameworks and paradigms for building ETLs, but there are so much more to learn and discuss.

In the second post of this series, I will dive into the specifics and demonstrate how to build a Hive batch job in Airflow. Specifically, we will learn the basic anatomy of an Airflow job, see extract, transform, and load in actions via constructs such as partition sensors and operators. We will learn how to use data modeling techniques such as star schema to design tables. Finally, I will highlight some ETL best practices that are extremely useful.

If you found this post useful, stay tuned for **Part II** and **Part III**.