Lecture 2. ML and Data Systems Fundamentals

Note: *This note is a work-in-progress, created for the course CS 329S: Machine Learning Systems Design (Stanford, 2022). For the fully developed text, see the book Designing Machine Learning Systems (Chip Huyen, O'Reilly 2022).*

Errata, questions, and feedback -- please send to chip@huyenchip.com. Thank you!

Table of contents

Part 1. ML Systems Fundamentals

Framing ML Problems

To decide which ML model to use to solve your problem, you first need to frame your problem into a problem that ML can solve.

The most general types of ML problems are classification and regression. Classification models classify inputs into different categories. For example, you want to classify each email to be either spam or not spam. Regression models output a continuous value. An example is a house prediction model that outputs the price of a given house.

A regression model can easily be framed as a classification model and vice versa. For example, house prediction can become a classification task if we quantize the house prices into buckets such as under $100,000, $100,000 - 200,000, $200,000 - 500,000, … and predict the bucket the house should be in. The email classification model can become a regression model if we make it output values between 0 and 1, and use a threshold to determine which values should be SPAM (for example, if the value is above 0.5, the email is spam).

Within classification problems, the fewer classes there are to classify, the simpler the problem is. The simplest is **binary classification** where there are only two possible classes. Examples of binary classification include classifying whether a comment is toxic or not toxic, whether a lung scan shows signs of cancer or not, whether a transaction is fraudulent or not. It's unclear whether this type of problem is common because they are common in nature or simply because ML practitioners are most comfortable handling them. Dealing with binary classification problems, including upsampling and downsampling as well as visualizing ROC curves and confusion matrices, is much easier than dealing with multiclass classifiers.

When there are more than two classes, the problem becomes **multiclass classification**. When the number of classes is high, such as disease diagnosis where the number of diseases can go up to thousands or product classifications where the number of products can go up to tens of thousands, the problem can be very challenging. The first challenge is in data collection. In my experience, ML models typically need at least 100 examples for each class to learn to classify that class. So if you have 1000 classes, you already need at least 100,000 examples. The data collection can be especially challenging when some of the classes are rare, and if you have thousands of classes, there's a high likelihood that some of them are rare.

When the number of classes is large, hierarchical classification might be useful. In hierarchical classification, you have a classifier to first classify each example into one of the large groups. Then you have another classifier to classify this example into one of the subgroups. For example, for product classification, you can first classify each product into one of the four main categories: electronics, home & kitchen, fashion, and pet supplies. After a product has been classified into a category, say fashion, you can use another classifier to put this product into one of the subgroups: shoes, shirt, jeans, accessories.

In both binary and multiclass classification, each example belongs to exactly one class. When an example can belong to multiple classes, we have a **multilabel classification** problem. For example, when building a model to classify articles into three topics: [tech, entertainment, finance], an article can be in both tech and finance.

There are two major approaches to multilabel classification problems. The first is to treat it as you would a multiclass classification. In multiclass classification, if there are three possible classes [tech, entertainment, finance] and the label for an example is entertainment, you represent this label with the vector [0, 1, 0]. In multilabel classification, if an example has both labels entertainment and finance, its label will be represented as [0, 1, 1].

The second approach is to turn it into multiple binary classification problems. For the article classification problem above, you can have three models corresponding to three topics, where each model outputs whether an article is in that topic or not.

Changing the way you frame your problem might make your problem significantly harder or easier. Consider the task of predicting what app a phone user wants to use next. A naive setup would be to frame this as a multiclass classification task — use the user's and environment's features (user demographic information, time, location, previous apps used) as input and output a probability distribution for every single app on the user's phone. This means that the last layer of your model will have the shape |number of hidden layer| x |number of apps|.

This is a bad approach because whenever a new app is added, you have to retrain your model, or at least the last layer of your model. A better approach is to frame this as a regression task — the user's, the environment's, and the app's features as input, and output a value between 0 and 1, the higher the value, the more likely the user will open the app given the context.

Decoupling Objectives

When solving a problem, you might have multiple objectives in mind. Imagine you're building a system to rank items on users' newsfeed. Your original goal is to maximize users' engagement. You want to achieve this goal through the following three objectives.

1. Filter out spam
2. Filter out NSFW content
3. Rank posts by engagement: how likely users will click on it

However, you quickly learned that optimizing for users' engagement alone can lead to questionable ethical concerns. Because extreme posts tend to get more engagements, your algorithm learned to prioritize extreme content[1][2]. You want to create a more wholesome newsfeed. So you have a new goal: **maximize users' engagement** while **minimizing the spread of extreme views and misinformation**. To obtain this goal, you add two new objectives to your original plan.

1. Filter out spam
2. Filter out NSFW content
3. **Filter out misinformation**
4. **Rank posts by quality**
5. Rank posts by engagement: how likely users will click on it

Now, objectives 4 and 5 are in conflict with each other. If a post is very engaging but it's of questionable quality, should that post rank high or low?

Let's take a step back to see what exactly each objective does. To rank posts by quality, you first need to predict posts' quality and you want posts' predicted quality to be as close to their actual quality as possible. Essentially, you want to minimize **quality_loss**: the difference between each post's predicted quality and its true quality[3].

Similarly, to rank posts by engagement, you first need to predict the number of clicks each post will get. You want to minimize **engagement_loss**: the difference between each post's predicted clicks and its actual number of clicks.

One approach is to combine these two losses into one loss and train one model to minimize that loss.

$$\textbf{loss} = \alpha \textbf{ quality\_loss} + \beta \textbf{ engagement\_loss}$$

You can randomly test out different values of α and β to find the values that work best. If you want to be more systematic about tuning these values, you can check out Pareto optimization, "*an area of multiple criteria decision making that is concerned with mathematical optimization problems involving more than one objective function to be optimized simultaneously*[4]".

[1] Facebook Employee Raises Powered by 'Really Dangerous' Algorithm That Favors Angry Posts (SFist, 2019)
[2] The Making of a YouTube Radical (NYT, 2019)
[3] For simplicity, let's pretend for now that we know to measure a post's quality.
[4] While you're at it, you might also want to read Jin and Sendhoff's great paper on applying Pareto optimization for ML where the authors claimed that "machine learning is inherently a multiobjective task."

A problem with this approach is that each time you tune α and β — for example, if your users' newsfeed's quality goes up but users' engagement goes down, you might want to decrease α and increase β — you'll have to retrain your model.

Another approach is to train two different models, each optimizing one loss. So you have two models:

- **quality_model** minimizes **quality_loss** and outputs the predicted quality of each post.
- **engagement_model** minimizes **engagement_loss** and outputs the predicted number of clicks of each post.

You can combine the outputs of these two models and rank posts by their combined scores:

$$\alpha\ \textbf{quality\_score} + \beta\ \textbf{engagement\_score}$$

Now you can tweak α and β without retraining your models!

In general, when there are multiple objectives, it's a good idea to decouple them first because it makes model development and maintenance easier. First, it's easier to tweak your system without retraining models, as explained above. Second, it's easier for maintenance since different objectives might need different maintenance schedules. Spamming techniques evolve much faster than the way post quality is perceived, so spam filtering systems need updates at a much higher frequency than quality ranking systems.

Part 2. Data Systems Fundamentals

The rise of machine learning in recent years is tightly coupled with the rise of big data. Big data systems, even without machine learning, are complex. If you haven't spent years and years working with them, it's easy to get lost in acronyms. There are many challenges and possible solutions that these systems generate. Industry standards, if there are any, evolve quickly as new tools come out and the needs of the industry expand, creating a dynamic and ever-changing environment. If you look into the data stack for different tech companies, it might seem like each is doing its own thing.

In this chapter, we'll cover the basics of data engineering that will, hopefully, give you a steady piece of land to stand on as you explore the landscape for your own needs. We'll start with different sources of data that you might work with in a typical ML project. We'll continue to discuss the formats in which data can be stored. Storing data is only interesting if you intend on retrieving that data later. To retrieve stored data, it's important to know not only how it's formatted but also how it's structured. Data models define how the data stored in a particular data format is structured.

If data models describe the data in the real-world, databases specify how the data should be stored on machines. We'll continue to discuss databases for the two major types of processing: transactional and analytical.

Knowing how to collect, process, store, retrieve, and process an increasingly growing amount of data is essential to people who want to build ML systems in production. If you're already familiar with data systems, you might want to move directly to Chapter 3 to learn more about how to sample and generate labels to create training data. If you want to learn more about data engineering from a systems perspective, I recommend Martin Kleppman's excellent book *Designing Data Intensive Applications* (O'Reilly, 2017).

Data Sources

An ML system can work with data from many different sources. They have different characteristics with different access patterns, can be used for different purposes, and require different processing methods. Understanding the sources your data comes from can help you access and manipulate your data more efficiently. This section aims to give a quick overview of different data sources to those unfamiliar with data in production. If you've already worked with ML in production for a while, feel free to skip this section.

One source is **user input data**, data explicitly input by users, which is often the input on which ML models can make predictions. User input can be texts, images, videos, uploaded files, etc. If there is a wrong way for humans to input data, humans are going to do it, and as a result, user input data can be easily mal-formatted. If user input is supposed to be texts, they might be too long or too short. If it's supposed to be numerical values, users might accidentally enter texts. If you expect users to upload files, they might upload files in the wrong formats. User input data requires more heavy-duty checking and processing. Users also have little patience. In most cases, when we input data, we expect to get results back immediately. Therefore, user input data tends to require fast processing.

Another source is **system-generated data**. This is the data generated by different components of your systems, which include various types of logs and system outputs such as model predictions.

Logs can record the state of the system and significant events in the system, such as memory usage, number of instances, services called, packages used, etc. They can record the results of different jobs, including large batch jobs for data processing and model training. These types of logs provide visibility into how the system is doing, and the main purpose of this visibility is for debugging and possibly improving the application. Most of the time, you don't have to look at these types of logs, but they are essential when something is on fire.

Because logs are system generated, they are much less likely to be mal-formatted the way user input data is. Overall, logs don't need to be processed as soon as they arrive, the way you would want to process user input data. For many use cases, it's acceptable to process logs periodically, such as hourly or even daily. However, you might still want to process your logs fast to be able to detect and be notified whenever something interesting happens[5].

Because debugging ML systems is hard, it's a common practice to log everything you can. This means that your volume of logs can grow very, very quickly. This leads to two problems. The first is that it can be hard to know where to look because signals are lost in the noise. There have been many services that process and analyze logs, such as Logstash, DataDog, Logz, etc. Many of them use ML models to help you process and make sense of your massive amount of logs.

The second problem is how to store a rapidly growing amount of logs. Luckily, in most cases, you only have to store logs for as long as they are useful, and can discard them when they are no longer relevant for you to debug your current system. If you don't have to access your logs frequently, they can also be stored in low-access storage that costs much less than higher-frequency-access storage[6].

Systems can also generatedata to record users' behaviors, such as clicking, choosing a suggestion, scrolling, zooming, ignoring a popup, or spending an unusual amount of time on certain pages. Even though this is system-generated data, it's still considered part of **user data**[7] and might be subject to privacy regulations. This kind of data can also be used for ML systems to make predictions and to train their future versions.

There are also **internal databases**, generated by various services and enterprise applications in a company. These databases manage their assets such as inventory, customer relationship, users, and more. This kind of data can be used by ML models directly or by various components of an ML system. For example, when users enter a search query on Amazon, one or more ML models will process that query to detect the intention of that query — what products users are actually looking for? — then Amazon will need to check their internal databases for the availability of these products before ranking them and showing them to users.

[5] "Interesting" in production usually means catastrophic, such as a crash or when your cloud bill hits an astronomical amount.

[6] As of November 2021, AWS S3 Standard, the storage option that allows you to access your data with the latency of milliseconds, costs about 5 times more per GB than S3 Glacier, the storage option that allows you to retrieve your data from between 1 minute to 12 hours.

[7] An ML engineer once mentioned to me that his team only used users' historical product browsing and purchases to make recommendations on what they might like to see next. I responded: "So you don't use personal data at all?" He looked at me, confused. "If you meant demographic data like users' age, location then no, we don't. But I'd say that a person's browsing and purchasing activities are extremely personal."

Then there's the wonderfully weird world of **third-party data** that, to many, is riddled with privacy concerns. First-party data is the data that your company already collects about your users or customers. Second-party data is the data collected by another company on their own customers that they make available to you, though you'll probably have to pay for it. Third-party data companies collect data on the public who aren't their customers.

The rise of the Internet and smartphones has made it much easier for all types of data to be collected. It used to be especially easy with smartphones since each phone used to have a unique advertiser ID — iPhones with their Apple's Identifier for Advertisers (IDFA) and Android phones with their Android Advertising ID (AAID) — which acts as a unique ID to aggregate all activities on a phone. Data from apps, websites, check-in services, etc. are collected and (hopefully) anonymized to generate activity history for each person.

You can buy all types of data such as social media activities, purchase history, web browsing habits, car rentals, and political leaning for different demographic groups getting as granular as men, age 25-34, working in tech, living in the Bay Area. From this data, you can infer information such as people who like brand A also like brand B. This data can be especially helpful for systems such as recommendation systems to generate results relevant to users' interests. Third-party data is usually sold as structured data after being cleaned and processed by vendors.

However, as users demand more privacy to their data, companies have been taking steps to curb the usage of advertiser IDs. In early 2021, Apple made their IDFA opt-in. This change has reduced significantly the amount of third-party data available on iPhones, forcing many companies to focus more on first-party data[8]. To fight back this change, advertisers have been investing in workarounds. For example, China Advertising Association, a state-supported trade association for China's advertising industry, invested in a device fingerprinting system called CAID that allowed apps like TikTok and Tencent to keep tracking iPhone users[9].

Data Formats

Once you have data, you might want to store it (or "persist" it, in technical terms). Since your data comes from multiple sources with different access patterns — access patterns meaning the patterns in which data is frequently accessed — storing your data isn't always straightforward and can be costly. It's important to think about how the data will be used in the future so that the format you use will make sense. Here are some of the questions you might want to consider. How do I store multimodal data? When each sample might contain both images and texts?

[8] Apple Just Crippled IDFA, Sending An $80 Billion Industry Into Upheaval (Forbes, 2020)
[9] TikTok wants to keep tracking iPhone users with state-backed workaround (Ars Technica, 2021)

Where to store your data so that it's cheap and still fast to access? How to store complex models so that they can be loaded and run correctly on different hardware?

The process of converting a data structure or object state into a format that can be stored or transmitted and reconstructed later is **data serialization**. There are many, many data serialization formats. When considering a format to work with, you might want to consider different characteristics such as human readability, access patterns, and whether it's based on text or binary, which influences the size of its files. Table 2-1 consists of just a few of the common formats that you might encounter in your work. For a more comprehensive list, check out the wonderful Wikipedia page *Comparison of data-serialization formats*.

Format	**Binary/Text**	**Human-readable**	**Example use cases**
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Hadoop, Amazon Redshift
Avro	Binary primary	No	Hadoop
Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Pickle	Binary	No	Python, PyTorch serialization

Table 2-1: Common data formats and where they are used.

We'll go over a few of these formats, starting with JSON.

JSON

JSON, JavaScript Object Notation, is everywhere. Even though it was derived from JavaScript, it's language-independent — most modern programming languages can generate and parse JSON. It's human-readable. Its key-value pair paradigm is simple but powerful, capable of handling data of different levels of structuredness. For example, your data can be stored in a structured format like the following.

```
{
  "firstName": "Boatie",
  "lastName": "McBoatFace",
  "isVibing": true,
  "age": 12,
  "address": {
    "streetAddress": "12 Ocean Drive",
    "city": "Port Royal",
    "postalCode": "10021-3100"
  }
}
```

The same data can also be stored in an unstructured blob of text like the following.

```
{
  "text": "Boatie McBoatFace, aged 12, is vibing, at 12 Ocean Drive,  Port Royal, 
10021-3100"
}
```

Row-major vs. Column-major Format

The two formats that are common and represent two distinct paradigms are CSV and Parquet. CSV is row-major, which means consecutive elements in a row are stored next to each other in memory. Parquet is column-major, which means consecutive elements in a column are stored next to each other.

Because modern computers process sequential data more efficiently than non-sequential data, if a table is row-major, accessing its rows will be faster than accessing its columns in expectation. This means that for row-major formats, accessing data by rows is expected to be faster than accessing data by columns.

Imagine we have a dataset of 1000 examples, each example has 10 features. If we consider each example as a row and each feature as a column, then the row-major formats like CSV are better for accessing examples, e.g. accessing all the examples collected today. Column-major formats like Parquet are better for accessing features, e.g. accessing the timestamps of all your examples. See Figure 2-1.

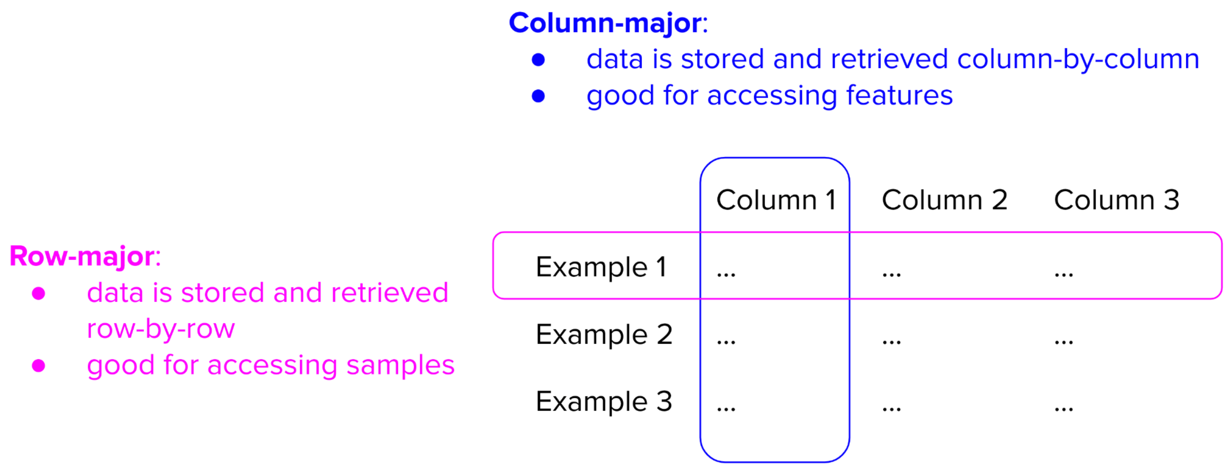

Column-major:
data is stored and retrieved column-by-column
good for accessing features
Row-major:
data is stored and retrieved row-by-row
good for accessing samples
Column 1
Column 2
Column 3
Example 1
Example 2
Example 3
...

Figure 2-1: Row-major vs. column-major formats

[SIDEBAR]
I use CSV as an example of the row-major format because it's popular and generally recognizable by everyone I've talked to in tech. However, some of the early reviewers of this book got upset by the mention of CSV because they believe CSV is a horrible data format. It serializes non-text characters poorly. For example, when you write float values to a CSV file, some precision might be lost — 0.12345678901232323 could be arbitrarily rounded up as "0.12345678901" — as complained about here and here. People on Hacker News have passionately argued against using CSV.
[/SIDEBAR]

Column-major formats allow flexible column-based reads, especially if your data is large with thousands, if not millions, of features. Consider if you have data about ride-sharing transactions that has 1000 features but you only want 4 features: time, location, distance, price. With column-major formats, you can read the 4 columns corresponding to these 4 features directly. However, with row-major formats, if you don't know the sizes of the rows, you will have to read in all columns then filter down to these 4 columns. Even if you know the sizes of the rows, it can still be slow as you'll have to jump around the memory, unable to take advantage of caching.

Row-major formats allow faster data writes. Consider the situation when you have to keep adding new individual examples to your data. For each individual example, it'd be much faster to write it to a file if your data is already in a row-major format.

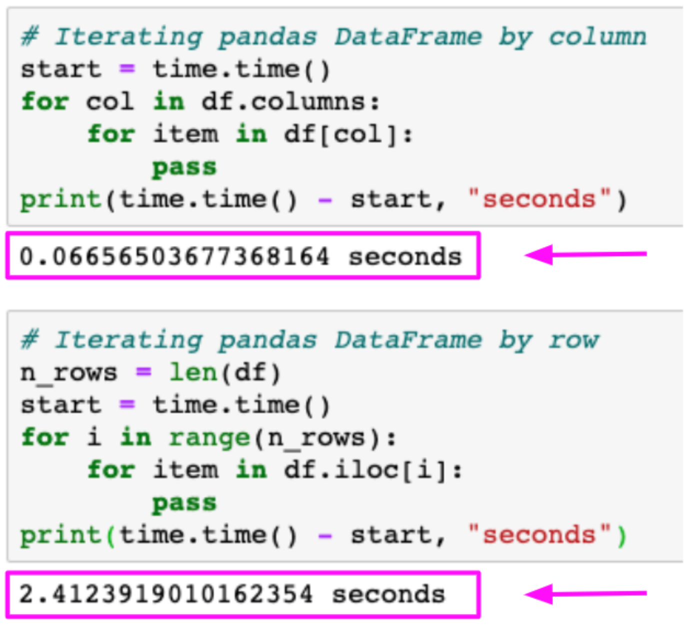
Overall, row-major formats are better when you have to do a lot of writes, whereas column-major ones are better when you have to do a lot of column-based reads.

[SIDE BAR]
NumPy vs. Pandas
One subtle point that a lot of people don't pay attention to, which leads to misuses of Pandas, is that this library is built around **the columnar format**.

Pandas is built around DataFrame, a concept inspired by R's Data Frame, which is column-major. A DataFrame is a two-dimensional table with rows and columns.

In NumPy, the major order can be specified. When an `ndarray` is created, it's row-major by default if you don't specify the order. People coming to pandas from NumPy tend to treat DataFrame the way they would `ndarray`, e.g. trying to access data by rows, and find DataFrame slow.

In Figure 2-2a, you can see that accessing a DataFrame by row is so much slower than accessing the same DataFrame by column. If you convert this same DataFrame to a NumPy `ndarray`, accessing a row becomes much faster, as you can see in Figure 2-2b.[10]



```
# Iterating pandas DataFrame by column
start = time.time()
for col in df.columns:
    for item in df[col]:
        pass
print(time.time() - start, "seconds")
```

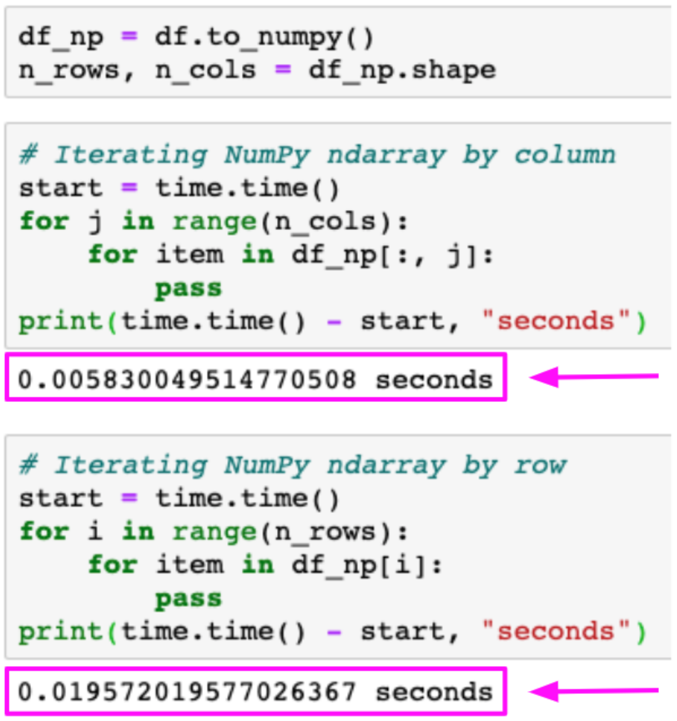
```
0.06656503677368164 seconds
```

```
# Iterating pandas DataFrame by row
n_rows = len(df)
start = time.time()
for i in range(n_rows):
    for item in df.iloc[i]:
        pass
print(time.time() - start, "seconds")
```

```
2.4123919010162354 seconds
```

Figure 2-2a: Iterating a pandas DataFrame by column takes 0.07 seconds but iterating the same DataFrame by row takes 2.41 seconds.
[FIGURE 2-2a and 2-2b should be put side by side]

[10] For more Pandas quirks, check out just-pandas-things (Chip Huyen, GitHub 2020).



```
df_np = df.to_numpy()
n_rows, n_cols = df_np.shape
```

```
# Iterating NumPy ndarray by column
start = time.time()
for j in range(n_cols):
    for item in df_np[:, j]:
        pass
print(time.time() - start, "seconds")
```

```
0.005830049514770508 seconds
```

```
# Iterating NumPy ndarray by row
start = time.time()
for i in range(n_rows):
    for item in df_np[i]:
        pass
print(time.time() - start, "seconds")
```

```
0.019572019577026367 seconds
```

Figure 2-2b: When you convert the same DataFrame into a NumPy ndarray, accessing its rows becomes much faster than accessing rows of the DataFrame.

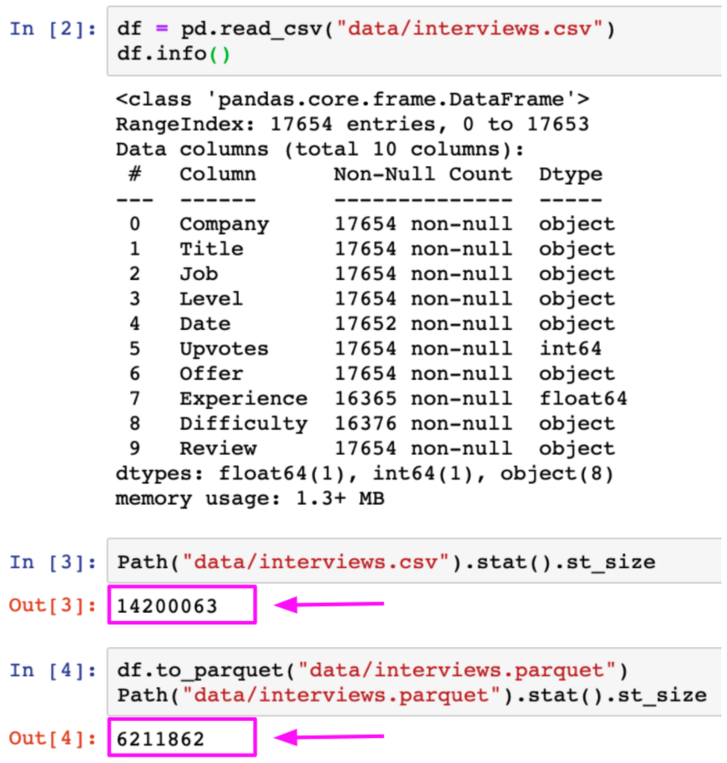
[/SIDE BAR]

Text vs. Binary Format

CSV and JSON are text files whereas Parquet files are binary files. Text files are files that are in plain text, which usually mean they are human-readable. Binary files, as the name suggests, are files that contain 0's and 1's, and meant to be read or used by programs that know how to interpret the raw bytes. A program has to know exactly how the data inside the binary file is laid out to make use of the file. If you open text files in your text editors (e.g. VSCode, Notepad), you'll be able to read the texts in them. If you open a binary file in your text editors, you'll see blocks of numbers, likely in hexadecimal values, for corresponding bytes of the file.

Binary files are more compact. Here's a simple example to show how binary files can save space compared to text files. Consider you want to store the number `1000000`. If you store it in a text file, it'll require 7 characters, and if each character is 1 byte, it'll require 7 bytes. If you store it in a binary file as int32, it'll take only 32 bits or 4 bytes.

As an illustration, I use `interviews.csv`, which is a CSV file (text format) of 17,654 rows and 10 columns. When I converted it to a binary format (Parquet), the file size went from 14MB to 6MB, as shown in Figure 2-3.



```
In [2]: df = pd.read_csv("data/interviews.csv")
        df.info()

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 17654 entries, 0 to 17653
        Data columns (total 10 columns):
         #   Column      Non-Null Count  Dtype
        ---  ------      --------------  -----
         0   Company     17654 non-null  object
         1   Title       17654 non-null  object
         2   Job         17654 non-null  object
         3   Level       17654 non-null  object
         4   Date        17652 non-null  object
         5   Upvotes     17654 non-null  int64
         6   Offer       17654 non-null  object
         7   Experience  16365 non-null  float64
         8   Difficulty  16376 non-null  object
         9   Review      17654 non-null  object
        dtypes: float64(1), int64(1), object(8)
        memory usage: 1.3+ MB

In [3]: Path("data/interviews.csv").stat().st_size

Out[3]: 14200063

In [4]: df.to_parquet("data/interviews.parquet")
        Path("data/interviews.parquet").stat().st_size

Out[4]: 6211862
```

Figure 2-3: When stored in CSV format, my interview file is 14MB.
But when stored in Parquet, the same file is 6MB.

AWS recommends using the Parquet format because "*the Parquet format is up to 2x faster to unload and consumes up to 6x less storage in Amazon S3, compared to text formats*."[11]

Data Models

Data models describe how data is represented. Consider cars in the real world. In a database, a car can be described using its maker, its model year, its color, and its price. These attributes make up a data model for cars. Alternatively, you can also describe a car using its owner, its license plate, and its history of registered addresses. This is another data model for cars.

How you choose to represent data not only affects the way your systems are built, but also the problems your systems can solve. For example, the way you represent cars in the first data model makes it easier for people looking to buy cars, whereas the second data model makes it easier for police officers to track down criminals.

[11] Announcing Amazon Redshift data lake export: share data in Apache Parquet format (Amazon AWS 2019).

In this section, we'll study two types of models that seem opposite to each other but are actually converging: relational models and NoSQL models. We'll go over examples to show the types of problems each model is suited for.

Relational Model

Relational models are among the most persistent ideas in computer science. Invented by Edgar F. Codd in 1970[12], the relational model is still going strong today, even getting more popular. The idea is simple but powerful. In this model, data is organized into relations, each relation is a set of tuples. A table is an accepted visual representation of a relation, and each row of a table makes up a tuple[13], as shown in Figure 2-4. Relations are unordered. You can shuffle the order of the rows or the order of the columns in a relation and it's still the same relation. Data following the relational model is usually stored in file formats like CSV, Parquet.

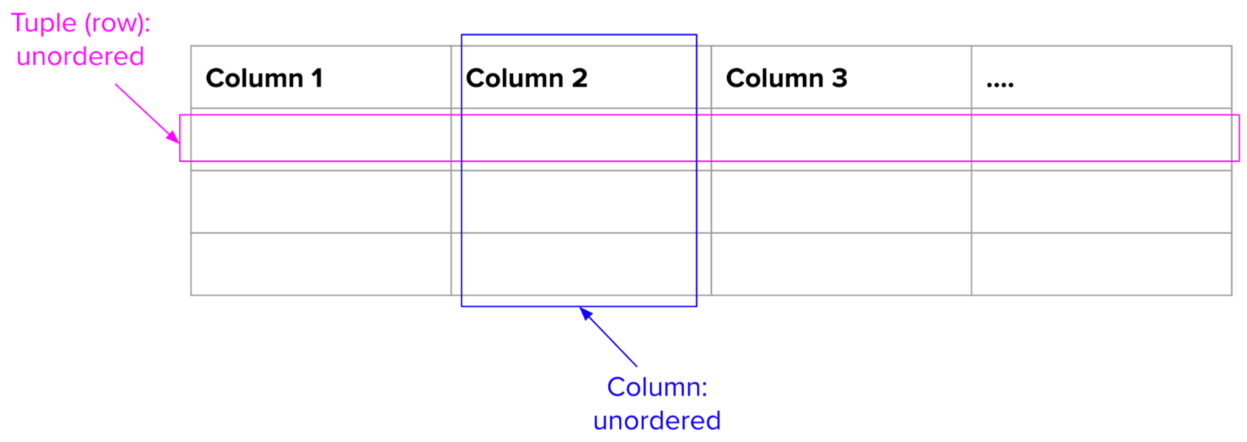


Figure 2-4: In a relation, the order of neither the rows nor the columns matter.

It's often desirable for relations to be normalized. Data normalization can follow normal forms such as the first normal form (1NF), second normal form (2NF), etc., and readers interested can read more about it on Wikipedia. In this book, we'll go through an example to show how normalization works and how it can reduce data redundancy and improve data integrity.

Consider the relation Book shown in Table 2-2. There are a lot of duplicates in this data. For example, row 1 and 2 are nearly identical, except for format and price. If the publisher information changes, for example, its name changes from "Banana Press" to "Pineapple Press" or its country changes, we'll have to update rows numbered 1, 2, and 4 because they contain "Banana Press". If we separate publisher information into its own table, as shown in Table 2-3a and Table 2-3b, when a publisher's information changes, we only have to update the Publisher

[12] *A Relational Model of Data for Large Shared Data Banks* (Edgar F. Codd, Communications of the ACM 1970)
[13] For detail-oriented readers, not all tables are relations.

relation[14]. This practice allows us to standardize spelling of the same value across different columns. It also makes it easier to make changes to these values, either because when these values change or when you want to translate them into different languages.

Title	**Author**	**Format**	**Publisher**	**Country**	**Price**
Harry Potter	J.K. Rowling	Paperback	Banana Press	UK	$20
Harry Potter	J.K. Rowling	E-book	Banana Press	UK	$10
Sherlock Holmes	Conan Doyle	Paperback	Guava Press	US	$30
The Hobbit	J.R.R. Tolkien	Paperback	Banana Press	US	$30
Sherlock Holmes	Conan Doyle	Paperback	Guava Press	US	$15

Table 2-2: Initial Book relation

Title	**Author**	**Format**	**Publisher ID**	**Price**
Harry Potter	J.K. Rowling	Paperback	1	$20
Harry Potter	J.K. Rowling	E-book	1	$10
Sherlock Holmes	Conan Doyle	Paperback	2	$30
The Hobbit	J.R.R. Tolkien	Paperback	1	$30
Sherlock Holmes	Conan Doyle	Paperback	2	$15

Table 2-3a: Updated Book relation

Publisher ID	**Publisher**	**Country**
1	Banana Press	UK
2	Guava Press	US

Table 2-3b: Publisher relation

One major downside of normalization is that your data is now spread across multiple relations. You can join the data from different relations back together, but joining can be expensive for large tables.

[14] You can further normalize the Book relation, such as separating format into a separate relation.

Databases built round the relational data model are relational databases. Once you've put data in your databases, you'll want a way to retrieve it. The language that you can use to specify the data that you want from a database is called a **query language**. The most popular query language for relational databases today is SQL. Even though inspired by the relational model, the data model behind SQL has deviated from the original relational model. For example, SQL tables can contain row duplicates, whereas true relations can't contain duplicates. However, this subtle difference has been safely ignored by most people.

The most important thing to note about SQL is that it's a declarative language, as opposed to Python which is an imperative language. In the imperative paradigm, you specify the steps needed for an action and the computer executes these steps to return the outputs. In the declarative paradigm, you specify the outputs you want, and the computer figures out the steps needed to get you the queried outputs.

With a SQL database, you specify the pattern of data you want — the tables you want the data from, the conditions the results must meet, the basic data transformations such as join, sort, group, aggregate, etc. — but not how to retrieve the data. It is up to the database system to decide how to break the query into different parts, what methods to use to execute each part of the query, and the order in which different parts of the query should be executed.

With certain added features, SQL can be Turing-complete, which means that in theory, SQL can be used to solve any computation problem (without making any guarantee about the time or memory required). However, in practice, it's not always easy to write a query to solve a specific task, and it's not always feasible or tractable to execute a query. Anyone working with SQL databases might have nightmarish memories of painfully long SQL queries that are impossible to understand and nobody dares to touch for fear that things might break[15].

Figuring out how to execute an arbitrary query is the hard part, which is the job of query optimizers. A query optimizer examines all possible ways to execute a query and finds the fastest way to do so[16]. It's possible to use ML to improve query optimizers based on learning from incoming queries[17]. Query optimization is one of the most challenging problems in database systems, and normalization means that data is spread out on multiple relations, which makes joining it together even harder. Even though developing a query optimizer is hard, the good news is that you generally only need one query optimizer and all your applications can leverage it.

[SIDEBAR]

---

[15] Greg Kemnitz, a co-author of the original Postgres paper, shared on Quora that he once wrote a reporting SQL query that was 700 lines long and visited 27 different tables in lookups or joins. The query had about 1000 lines of comments to help him remember what he was doing. It took him 3 days to compose, debug, and tune.

[16] Ioannidis, Yannis E. "Query optimization." ACM Computing Surveys (CSUR) 28.1 (1996): 121-123.

[17] Marcus, Ryan, et al. "Neo: A learned query optimizer." arXiv preprint arXiv:1904.03711 (2019).

From declarative data systems to declarative ML systems

Possibly inspired by the success of declarative data systems, many people have looked forward to declarative ML[18][19]. With a declarative ML system, users only need to declare the features' schema and the task, and the system will figure out the best model to perform that task with the given features. Users won't have to write code to construct, train, and tune models. Popular frameworks for declarative ML are Ludwig, developed at Uber, and H2O AutoML. In Ludwig, users can specify the model structure — such as the number of fully connected layers and the number of hidden units — on top of the features' schema and output. In H2O AutoML, you don't need to specify the model structure or hyperparameters. It experiments with multiple model architectures and picks out the best model given the features and the task.

Here is an example to show how H2O AutoML works. You give the system your data (inputs and outputs and specify the number of models you want to experiment. It'll experiment with that number of models and show you the best performing model.

[CODE - Python]
```
# Identify predictors and response
x = train.columns
y = "response"
x.remove(y)

# For binary classification, response should be a factor
train[y] = train[y].asfactor()
test[y] = test[y].asfactor()

# Run AutoML for 20 base models
aml = H2OAutoML(max_models=20, seed=1)
aml.train(x=x, y=y, training_frame=train)

# Show the best performing models on the AutoML Leaderboard
lb = aml.leaderboard

# Get the best performing model
aml.leader
```
[/CODE]

While declarative ML can be useful in many cases, it leaves unanswered the biggest challenges

[18] Declarative Machine Learning - A Classification of Basic Properties and Types (Boehm et al., 2016)
[19] Declarative Machine Learning Systems (Piero Molino and Christopher Ré, 2021)

with ML in production. Declarative ML systems today abstract away the model development part, and as we'll cover in the next six chapters, with models being increasingly commoditized, model development is often the easier part. The hard part lies in feature engineering, data processing, model evaluation, data shift detection, continual learning, etc.
[/SIDEBAR]

NoSQL

The relational data model has been able to generalize to a lot of use cases, from ecommerce to finance to social networks. However, for certain use cases, this model can be restrictive. For example, it demands that your data follows a strict schema and schema management is painful. In a survey by Couchbase in 2014, frustration with schema management was the #1 reason for the adoption of their nonrelational database. It can also be difficult to write and execute SQL queries for specialized applications.

The latest movement against the relational data model is NoSQL. Originally started as a hashtag for a meetup to discuss nonrelational databases, NoSQL has been retroactively reinterpreted as Not Only SQL[20] as many NoSQL data systems also support relational models. Two major types of nonrelational models are the document model and the graph model. The document model targets use cases where data comes in self-contained documents and relationships between one document and another are rare. The graph model goes in the opposite direction, targeting use cases where relationships between data items are common and important. We'll examine each of these two models, starting with the document model.

Document Model

The document model is built around the concept of "document". A document is often a single continuous string, encoded as JSON, XML, or a binary format like BSON. All documents in a document database are assumed to be encoded in the same format. Each document has a unique key that represents that document, which can be used to retrieve that document.

A collection of documents could be considered analogous to a table in a relational database, and a document analogous to a row. In fact, you can convert a relation into a collection of documents that way. For example, you can convert the book data in Table 2-3a and Table 2-3b into three JSON documents as shown in Figure 2-5. However, a collection of documents is much more flexible than a table. All rows in a table must follow the same schema (e.g. have the same sequence of columns), while documents in the same collection can have completely different schemas.

[20] Designing Data-Intensive Applications (Martin Kleppmann, O'Reilly 2017)

```
# Document 1: harry_potter.json
{
    "Title": "Harry Potter",
    "Author": "J.K. Rowling",
    "Publisher": "Banana Press",
    "Country": "UK",
    "Sold as": [
        {"Format": "Paperback", "Price": "$20"},
        {"Format": "E-book", "Price": "$10"}
    ]
}

# Document 2: sherlock_holmes.json
{
    "Title": "Sherlock Holmes",
    "Author": "Conan Doyle",
    "Publisher": "Guava Press",
    "Country": "US",
    "Sold as": [
        {"Format": "Paperback", "Price": "$30"},
        {"Format": "E-book", "Price": "$15"}
    ]
}

# Document 3: the_hobbit.json
{
    "Title": "The Hobbit",
    "Author": "J.R.R. Tolkien",
    "Publisher": "Banana Press",
    "Country": "UK",
    "Sold as": [
        {"Format": "Paperback", "Price": "$30"},
    ]
}
```

Figure 2-5: Representing the book data in Table 2-3a and 2-3b in the document data model.

Because the document model doesn't enforce a schema, it's often referred to as schemaless. This is misleading because, as discussed previously, data stored in documents will be read later. The application that reads the documents usually assumes some kind of structure of the documents. Document databases just shift the responsibility of assuming structures from the application that writes the data to the application that reads the data.

The document model has better locality than the relational model. Consider the book data example in Table 2-3a and Table 2-3b where the information about a book is spread across both the Book table and the Publisher table (and potentially also the Format table). To retrieve information about a book, you'll have to query multiple tables. In the document model, all information about a book can be stored in a document, making it much easier to retrieve.

However, compared to the relational model, it's harder and less efficient to execute joins across documents compared to across tables. For example, if you want to find all books whose prices are below $25, you’ll have to read all documents, extract the prices, compare them to $25, and return all the documents containing the books with prices below $25.

Because of the different strengths of the document and relational data models, it’s common to use both models for different tasks in the same database systems. More and more database systems, such as PostgreSQL and MySQL, support them both.

Graph Model

The graph model is built around the concept of a “graph”. A graph consists of nodes and edges, where the edges represent the relationships between the nodes. A database that uses graph structures to store its data is called a graph database. If in document databases, the content of each document is the priority, then in graph databases, the relationships between data items are the priority.

Because the relationships are modeled explicitly in graph models, it’s faster to retrieve data based on relationships. Consider an example of a graph database in Figure 2-6. The data from this example could potentially come from a simple social network. In this graph, nodes can be of different data types: person, city, country, company, etc.

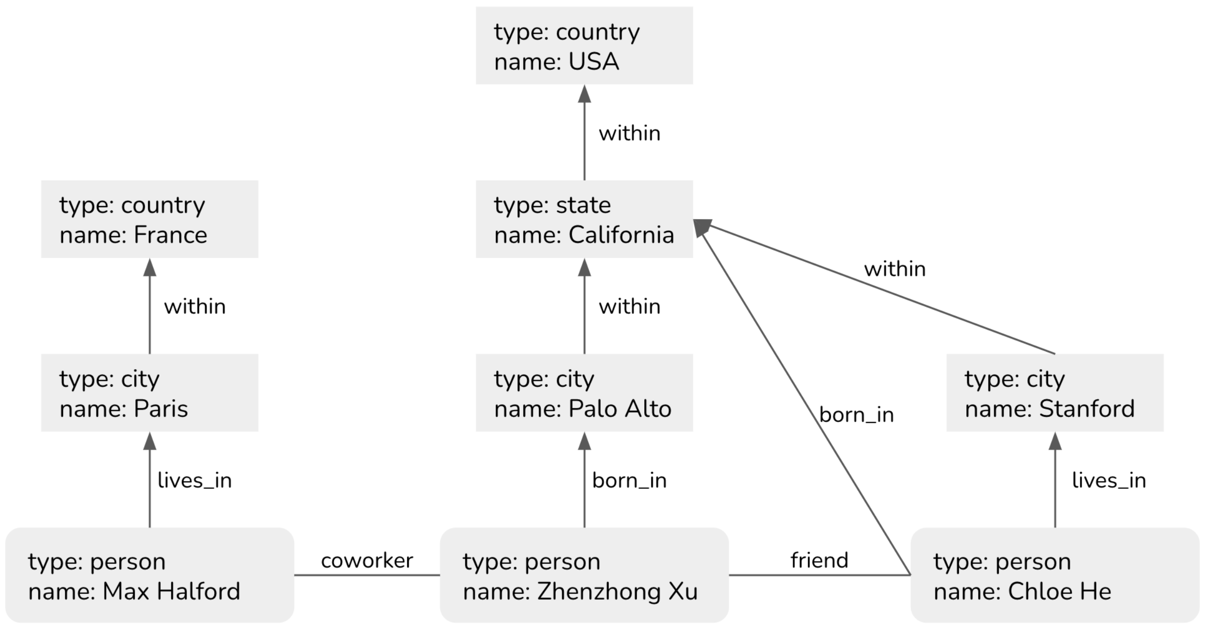


Figure 2-6: An example of a simple graph database.
This data could potentially come from a social network.

Imagine you want to find everyone who was born in the USA. Given this graph, you can start from the node USA and traverse the graph following the edges “within” and “born_in” to find all the nodes of the type “person”. Now, imagine that instead of using the graph model to represent this data, we use the relational model. There'd be no easy way to write a SQL query to find everyone who was born in the USA, especially given that there are unknown number of hops between *country* and *person* — there are 3 hops between Zhenzhong Xu and USA while there are only 2 hops between Chloe He and USA. Similarly, there'd be no easy way for this type of queries with a document database.

Many queries are easy to do in one data model but harder to do in another. Picking the right data model for your application can make your life so much easier.

Structured vs. Unstructured Data

Structured data is data that follows a predefined data model, also known as a data schema. For example, the data model might specify that each data item consists of two values: the first value, "name", is a string at most 50 characters, and the second value, "age", is an 8-bit integer in the range between 0 and 200. The predefined structure makes your data easier to analyze. If you want to know the average age of people in the database, all you have to do is to extract all the age values and average them out.

The disadvantage of structured data is that you have to commit your data to a predefined schema. If your schema changes, you'll have to retrospectively update all your data, often causing mysterious bugs in the process. For example, you've never kept your users' email addresses before but now you do, so you have to retrospectively update email information to all previous users. One of the strangest bugs one of my colleagues encountered was when they could no longer use users' ages with their transactions, and their data schema replaced all the null ages with 0, and their ML model thought the transactions were made by people of 0 years old.

Because business requirements change over time, committing to a predefined data schema can become too restricting. Or you might have data from multiple data sources, many of which are beyond your control, and it's impossible to make them follow the same schema. This is where unstructured data becomes appealing. Unstructured data is data that doesn't adhere to a predefined data schema. It's usually text but can also be numbers, dates, etc. For example, a text file of logs generated by your ML model is unstructured data.

Even though unstructured data doesn't adhere to a schema, it might still contain intrinsic patterns that help you extract structures. For example, the following text is unstructured, but you can notice the pattern that each line contains two values separated by a comma, the first value is textual and the second value is numerical. However, there is no guarantee that all lines must follow this format. You can add a new line to that text even if that line doesn't follow this format.

```
"Lisa, 43
Jack, 23
Nguyen, 59"
```

Unstructured data also allows for more flexible storage options. For example, if your storage follows a schema, you can only store data following that schema. But if your storage doesn't follow a schema, you can store any type of data. You can convert all your data, regardless of types and formats into bytestrings and store them together.

A repository for storing structured data is called a data warehouse. A repository for storing unstructured data is called a data lake. Data lakes are usually used to store raw data before processing. Data warehouses are used to store data that have been processed into formats ready to be used.

Table 2-2 shows a summary of the key differences between structured and unstructured data.

Structured data	**Unstructured data**
Schema clearly defined	Data doesn't have to follow a schema
Easy to search and analyze	Fast arrival
Can only handle data with a specific schema	Can handle data from any source
Schema changes will cause a lot of troubles	No need to worry about schema changes (yet) as the worry is shifted to the downstream applications that use this data
Stored in data warehouses	Stored in data lakes

Table 2-2: The key differences between structured and unstructured data

Data Storage Engines and Processing

Data formats and data models specify the interface for how users can store and retrieve data. Databases (storage engines) are the implementation of how data is stored and retrieved on machines. It's useful to understand different types of databases as your team or your adjacent team might need to select a database appropriate for your application.

Typically, there are two types of workloads that databases are optimized for: transactional processing and analytical processing, and there's a big difference between them. In this section, we will discuss the difference between transaction processing and analytical processing. We will then cover the basics of the ETL (Extract, Transform, Load) process that you will inevitably encounter when building an ML system in production.

Readers tuned into data engineering trends might wonder why batch processing versus stream processing is missing from this chapter. We'll cover this topic in **Chapter 6: Deployment** since I believe it's more related to other deployment concepts.

Transactional and Analytical Processing

Traditionally, a transaction refers to the action of buying or selling something. In the digital world, a transaction refers to any kind of actions that happen online: tweeting, ordering a ride through a ridesharing service, uploading a new model, watching a YouTube video, etc. Even though these different transactions involve different types of data, the way they're processed is similar across applications. The transactions are inserted as they are generated, and occasionally updated when something changes, or deleted when they are no longer needed[21]. This type of processing is known as **OnLine Transaction Processing (OLTP)**.

Because these transactions often involve users, they need to be processed fast (low latency) so that they don't keep users waiting. The processing method needs to have high availability — e.g. the processing system needs to be available any time a user wants to make a transaction. If your system can't process a transaction, that transaction won't go through.

Transactional databases are designed to process online transactions and satisfy the low latency, high availability requirements. When people hear transactional databases, they usually think of ACID (**A**tomicity, **C**onsistency, **I**solation, **D**urability). Here are their quick definitions of ACID for those needing a quick reminder.

- **A**tomicity: to guarantee that all the steps in a transaction are completed successfully as a group. If any step between the transaction fails, all other steps must fail also. For example, if a user's payment fails, you don't want to still assign a driver to that user.
- **C**onsistency: to guarantee that all the transactions coming through must follow predefined rules. For example, a transaction must be made by a valid user.
- **I**solation: to guarantee that two transactions happen at the same time as if they were isolated. Two users accessing the same data won't change it at the same time. For example, you don't want two users to book the same driver at the same time.
- **D**urability: to guarantee that once a transaction has been committed, it will remain committed even in the case of a system failure. For example, after you've ordered a ride and your phone dies, you still want your ride to come.

However, transactional databases don't necessarily need to be ACID, and some developers find ACID to be too restrictive. According to Martin Klepmann, *systems that do not meet the ACID*

[21] This paragraph, as well as many parts of this chapter, is inspired by Martin Kleppman's *Designing Data Intensive Applications* (O'Reilly, 2017)

*criteria are sometimes called BASE, which stands for **Basically Available, Soft state, and Eventual consistency**. This is even more vague than the definition of ACID*[22].

Because each transaction is often processed as a unit separately from other transactions, transactional databases are often row-major. This also means that transactional databases might not be efficient for questions such as "What's the average price for all the rides in September in San Francisco?". This kind of analytical question requires aggregating data in columns across multiple rows of data. Analytical databases are designed for this purpose. They are efficient with queries that allow you to look at data from different viewpoints. We call this type of processing **OnLine Analytical Processing (OLAP)**.

However, both the terms OLTP and OLAP have become outdated, as shown in Figure 2-7, for three reasons. First, the separation of transactional and analytical databases was due to limitations of technology — it was hard to have databases that could handle both transactional and analytical queries efficiently. However, this separation is being closed. Today, we have transactional databases that can handle analytical queries, such as CockroachDB. We also have analytical databases that can handle transactional queries, such as Apache Iceberg.

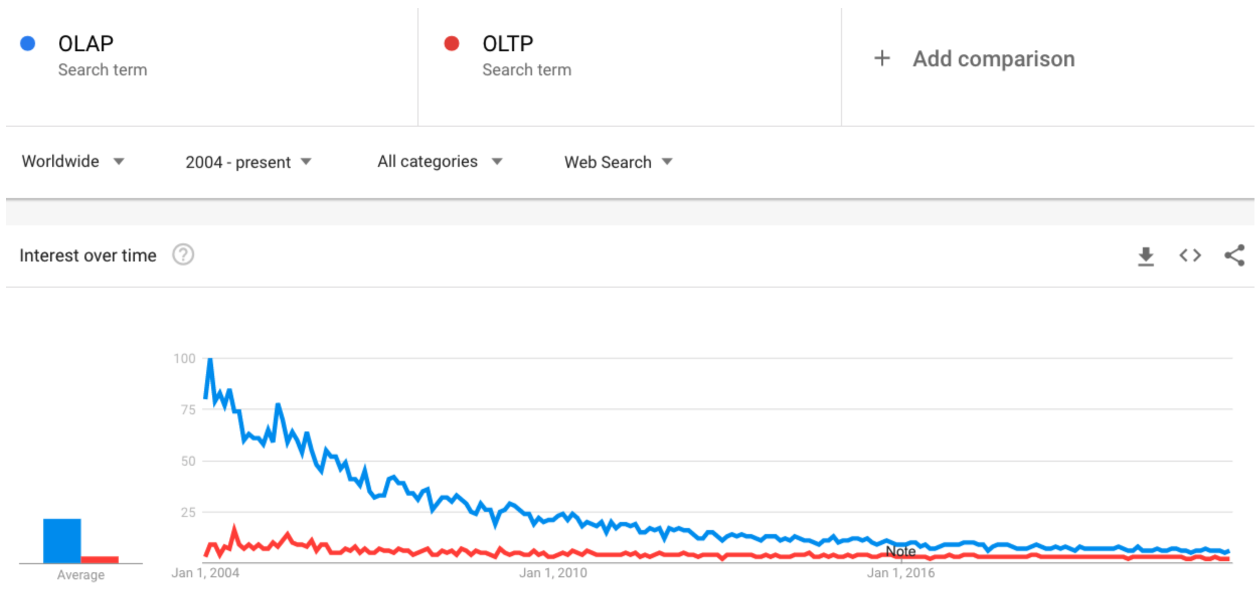


Figure 2-7: OLAP and OLTP are outdated terms, as of 2021, according to Google Trends

Second, in the traditional OLTP or OLAP paradigms, storage and processing are tightly coupled — how data is stored is also how data is processed. This may result in the same data being stored in multiple databases and use different processing engines to solve different types of queries. An

[22] Designing Data-Intensive Applications (Martin Kleppmann, O'Reilly 2017)

interesting paradigm in the last decade has been to decouple storage from processing (also known as compute), as adopted by many data vendors including Google's BigQuery, Snowflake, IBM, and Teradata. In this paradigm, the data can be stored in the same place, with a processing layer on top that can be optimized for different types of queries.

Third, "online" has become an overloaded term that can mean many different things. Online used to just mean "connected to the Internet". Then, it grew to also mean "in production" — we say a feature is online after that feature has been deployed in production.

In the data world today, "online" might refer to the speed at which your data is processed and made available: online, nearline, or offline. According to Wikipedia, online processing means data is immediately available for input/output. Nearline, which is short for near-online, means data is not immediately available, but can be made online quickly without human intervention. Offline means data is not immediately available, and requires some human intervention to become online.

As the speed at which applications respond to users queries has become a competitive advantage, it's become more and more important to make data available for use as fast as possible. In many use cases, companies want online processing not just for transactional queries but also for analytical queries. Online, in this case, is synonymous to "real-time". Both online processing and nearline processing are covered by stream processing that we'll cover in Chapter 6.

ETL: Extract, Transform, Load

Even before ML, ETL (extract, transform, load) was all the rage in the data world, and it's still relevant today for ML applications. ETL refers to the general purpose processing and aggregating data into the shape and the format that you want.

Extract is extracting the data you want from data sources. Your data will likely come from multiple sources in different formats. Some of them will be corrupted or malformatted. In the extracting phase, you need to validate your data and reject the data that doesn't meet your requirements. For rejected data, you might have to notify the sources. Since this is the first step of the process, doing it correctly can save you a lot of time downstream.

Transform is the meaty part of the process, where most of the data processing is done. You might want to join data from multiple sources and clean it. You might want to standardize the value ranges (e.g. one data source might use "Male" and "Female" for genders, but another uses "M" and "F" or "1" and "2"). You can apply operations such as transposing, deduplicating, sorting, aggregating, deriving new features, more data validating, etc.

Load is deciding how and how often to load your transformed data into the target destination, which can be a file, a database, or a data warehouse.

The idea of ETL sounds simple but powerful, and it's the underlying structure of the data layer at many organizations. An overview of the ETL process is shown in Figure 2-8.

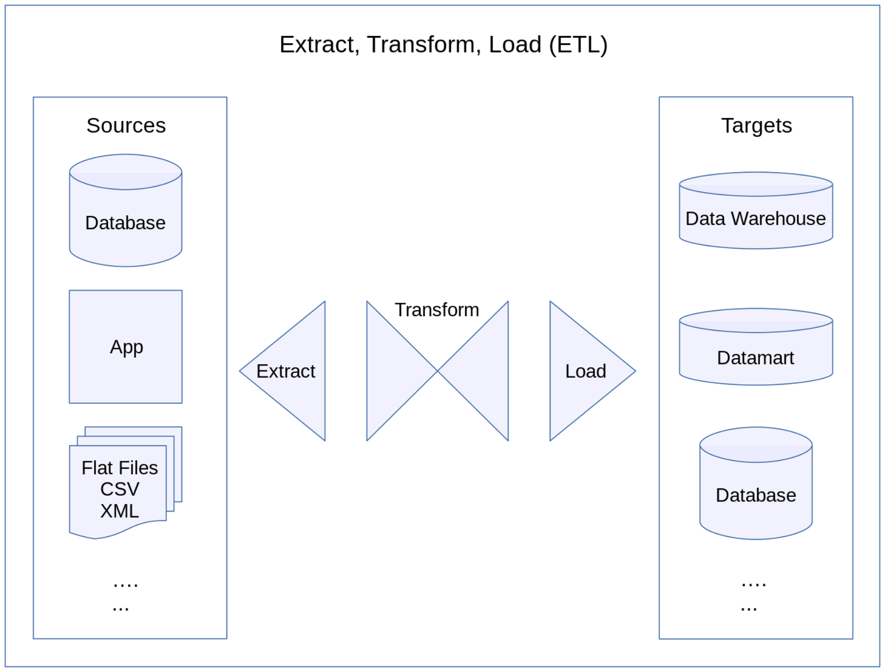


Figure 2-8: An overview of the ETL process

ETL to ELT

When the Internet first became ubiquitous and hardware had just become so much more powerful, collecting data suddenly became so much easier. The amount of data grew rapidly. Not only that, but the nature of data also changed. The number of data sources expanded, and data schemas evolved.

Finding it difficult to keep data structured, some companies had this idea: "Why not just store all data in a data lake so we don't have to deal with schema changes? Whichever application needs data can just pull out raw data from there and process it." This process of loading data into storage first then processing it later is sometimes called ELT (extract, load, transform). This paradigm allows for the fast arrival of data since there's little processing needed before data is stored.

However, as data keeps on growing, this idea becomes less attractive. It's expensive to store everything, and it's inefficient to search through a massive amount of raw data for the piece of data that you want. At the same time, as companies switch to running applications on the cloud and infrastructures become standardized, data structures also become standardized. Committing data to a predefined schema becomes more feasible.

Summary

In chapter 1, we discussed the importance of data in developing ML systems. This chapter covers the ABC of a data system that lays to building bricks to handle the data needed for ML models.

We started the Data Systems Fundamentals section with different sources where our data might come from. To use data, we often need to store it and retrieve it later. It's important to choose the right format to store our data to make it easier to use the data in the future. We discussed different data formats, the pros and cons of row-major vs. column-major formats as well text vs. binary formats.

We continued to cover three major data models: relational, document, and graph. Even though the relational model is the most well-known given the popularity of SQL, all three models are widely used today, and each is good for a certain set of tasks. One model can be emulated in terms of another model, and we went through an example to show how a relational database can be represented using the document model. However, the wrong data model can make it difficult to do our job.

When talking about the relational model compared to the document model, many people think of the former as structured and the latter as unstructured. Even though the discussion of structured vs. unstructured data can get quite heated, the division is quite fluid. Some people even argue that there's no such thing as unstructured or structured data. The question is who has to shoulder the responsibility of assuming the structure of data. Structured data means that the code that writes the data has to assume the structure. Unstructured data means that the code that reads the data has to assume the structure.

We ended the chapter with data storage and processing. We studied databases optimized for two distinct types of data processing: transactional processing and analytical processing. We also studied a process that is ubiquitous in data science and analytics workloads today: ETL. These fundamentals will hopefully help readers become better prepared when facing seemingly overwhelming data in production.