# Designing Machine Learning Systems - Chip Huyen

## Chapter 3 – Fundamentals of Data Engineering

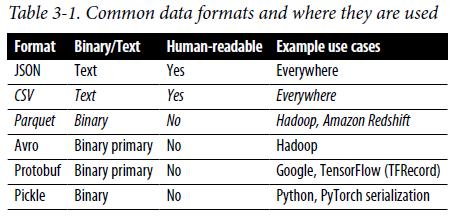
* The **rise of ML in recent years is tightly coupled with the rise of big data**
* **Large data systems, even without ML, are complex** (easy to get lost in acronyms)
* There are **many challenges + possible solutions that these systems generate**
* **Industry standards** (*if there are any*) **evolve 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*
* The basics of data engineering should, hopefully, give a steady piece of land to stand on as you explore the landscape for your own needs
* **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*, **data storage engines/databases specify how the data should be stored on *machines***
* 2 major types of **processing**: **transactional** and **analytical**.
* **When working with data in production, you usually work with data across multiple processes and services**
* Ex: Might have a feature engineering service that computes features from raw data, + a prediction service to generate predictions based on computed features
* This means that you’ll have to **pass** computed features from the feature engineering service to the prediction service
* 2 distinct types of data: **historical data in data storage engines**, and **streaming data in real-time transports**
* These two different types of data **require different processing paradigms**
* **Knowing how to collect, process, store, retrieve, + process an increasingly growing amount of data is essential to people who want to build ML systems in production**

### Data Sources

* An **ML system can work with data from many different sources** that **have different characteristics, can be used for different purposes**, and **require different processing methods**
* **Understanding the sources your data comes from can help you use your data more efficiently**
* **1) User input data: data explicitly input by users** (text, images, videos, uploaded files, etc.)
* **If it’s even remotely possible for users to input wrong data, they are going to do it**
* As a result, **user input data can be easily mal-formatted** (Text is too long or too short, where numerical values are expected, users enter text, upload files in the wrong formats)
* **User input data requires more heavy-duty checking and processing.**
* On top of that, **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**
* **2) System-generated data: data generated by different components of your systems** (including various types of logs and system outputs, such as model predictions)
* **Logs** can **record the state and significant events of the system** (memory usage, number of instances, services called, packages used, etc.)
* Logs can **record the results of different jobs** (like large batch jobs for data processing + model training)
* These types of logs **provide visibility into how the system is doing,** with **main purpose of this visibility is for debugging and potentially improving the application**
* Most of the time, 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’d want to process user input data
* **Many use cases** **=** **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 + be notified whenever something interesting happens** (“Interesting” in production usually means catastrophic, such as a crash or when a cloud bill hits an astronomical amount)
* **Because debugging ML systems is hard, it’s common practice to log everything you can**
* This means that **volume of logs can grow very, very quickly**, which **leads to 2 problems**.
* 1) It can be **hard to know where to look because signals are lost in the noise**
* Many services process + analyze logs, + many of them use ML models to help process + make sense of massive numbers of logs
* 2) **How to store a rapidly growing number 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**
* November 2021: AWS S3 Standard, the storage option that allows you to access your data with the latency of ms, costs about 5X more per GB than S3 Glacier, the storage option that allows you to retrieve your data with a latency from between 1 minute to 12 hours
* The **system also generates data to record users’ behaviors** (clicking, choosing a suggestion, scrolling, zooming, ignoring a pop-up, spending an unusual amount of time on certain pages)
* Even though this is system-generated data, it’s *still considered part of user data and might be subject to privacy regulations*
* It can be argued that a person’s browsing and purchasing activities are extremely personal
* **3) Internal databases**: **generated by various services and enterprise applications in a company**
* These databases **manage assets such as inventory, customer relationship, users, + more**
* This kind of data **can be used by ML models *directly* or by various components of an ML system**
* Ex: Users **enter a search query** on Amazon, **1+ ML models process that query to detect its intention** (someone types in “frozen”, are they looking for frozen foods or Disney’s *Frozen* franchise?), + then Amazon needs to **check its internal databases for the availability of these products before ranking them and showing them to users**
* **4)** **Third-party data**
* **First-party data** = **the data your company already collects about users or customers**
* **Second-party data** = **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** = **data collected on the public who *aren’t direct customers* of the company collecting the data**
* Third-party data is **usually sold after being cleaned and processed by vendors**
* **The rise of the internet + smartphones has made it much easier for all types of data to be collected**
* Used to be especially easy with smartphones since each phone used to have a unique advertiser ID, which acted as a unique ID to aggregate all activities on a phone
* Data from apps, websites, check-in services, etc. are collected + (hopefully) anonymized to generate activity history for each person
* **Data of all kinds can be bought** (social media activities, purchase history, web browsing habits, car rentals, + 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 recommender systems to generate results relevant to users’ interests**

### Data Formats

* **Once you have data**, you might **want to store it** (or “**persist**” it, in technical terms)
* Since **data comes from multiple sources with different access patterns** (the **pattern in which a system or program reads or writes data**), **storing data isn’t always straightforward** and, for some cases, **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**
* Some of the questions you might want to consider:
* **How do I store multimodal data** (e.g., a sample that might contain both images and texts?)
* **Where do I store my data so that it’s cheap and still fast to access?**
* **How do I store complex models so they can be loaded + run correctly on different hardware?**
* **Data serialization** = the **process of converting a data structure or object state into a format that can be stored or transmitted + reconstructed later**
* There are ***many, many* data serialization formats**
* **When considering a format** to work with, you want to **consider different characteristics such as human readability, access patterns, + whether it’s based on text or binary** (influences the size of its files)
* Table 3-1 consists of just a few of the common formats that you might encounter



#### JSON

* **JSON (JavaScript Object Notation) is everywhere**
* **Language-independent** (most modern programming languages can generate and parse JSON) + **human-readable**
* Its **key-value pair paradigm** is **simple but powerful**, capable of **handling data of different levels of structuredness**
* Ex: 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:

{

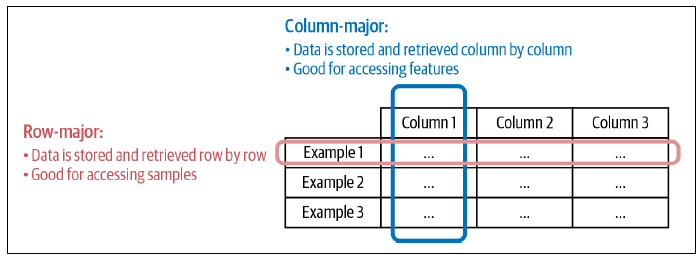
**"text"**: "Boatie McBoatFace, aged 12, is vibing, at 12 Ocean Drive, Port Royal, 10021-3100"

}

* **Because JSON is ubiquitous, the pain it causes can also be felt everywhere**
* Once you’ve committed the data in your JSON files to a schema, it’s pretty **painful to retrospectively go back to change the schema**
* JSON files are **text files**, which means they **take up a lot of space**, as we’ll see later

#### Row-Major vs. Column-Major Format

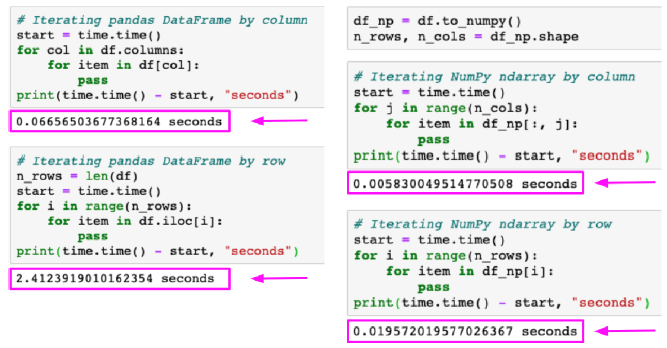
* **2 formats that are common and represent 2 distinct paradigms are CSV and Parquet**
* CSV is **row-major** = **consecutive elements in a *row* are stored next to each other in memory**
* **1 disadvantage of CSV’s = serializes non-text characters poorly**
* Ex: when you write float values to a CSV file, some precision might be los and 0.12345678901232323 could be arbitrarily rounded up as 0.12345678901
* Parquet is **column-major** = **consecutive elements in a *column* are stored next to each other**
* Because **modern CPUs process sequential data more efficiently than nonsequential 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**
* Ex: Dataset of 1K examples, + each example has 10 features
* **If we consider each example as a row + each feature as a column** (as is often the case in ML), then the **row-major formats like CSV are better for accessing *examples*** (e.g., accessing all examples collected today)
* **Column-major formats like Parquet are better for accessing *features*** (e.g., accessing the timestamps of all examples)



* **Column-major formats allow flexible column-based reads**, especially if data is large with thousands, if not millions, of features
* Consider if you have data about ride-sharing transactions that has 1,000 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***
* Plus, *even if you DO 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 where 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**

##### An Aside: NumPy vs. Pandas

* 1 subtle point that a lot of people don’t pay attention to, which leads to misuses of pandas, is that the **pandas is built around the columnar format** via the **DataFrame** (**a 2D table with rows and columns**), a concept inspired by R’s Data Frame, **which is column-major**
* **In NumPy, the major order can be specified**
* When an nd-array 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 nd-array (trying to access data by rows), + find DataFrame slow.
* On the left panel below, **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 nd-array, accessing a row becomes much faster**, as you can see in the right panel below



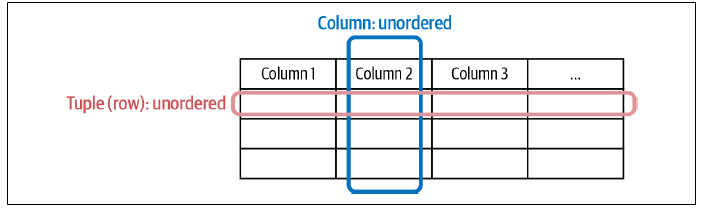
#### Text vs. Binary Format

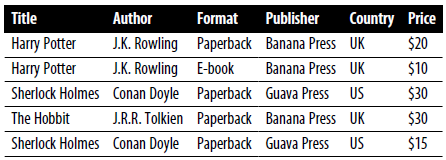
* **CSV and JSON = text files**, whereas **Parquet files = binary files**
* **Text files = in plain text, usually human-readable**
* **Binary files = the catch-all that refers to all non-text files**
* Typically **contain only 0s and 1s,** + are **meant to be read or used by programs that know how to interpret the raw bytes**
* **A program has to know exactly how the data inside the binary file is laid out to make use of it**
* If you open text files in a text editor, you’ll be able to read the texts in them
* If you open a binary file in a text editor, you’ll see blocks of numbers, likely in hexadecimal values, for corresponding bytes of the file
* **Binary files are more compact**
* Ex: You want to store the number 1000000
* To store in a text file, it requires 7 characters, + if each character = 1 byte, it requires 7 bytes
* If you store it in a binary file as int32, it’ll take only 32 bits or 4 bytes
* Ex: interviews.csv, a CSV file of 17,654 rows and 10 columns
* When converted to a binary format (Parquet), the file size went from 14 MB to 6 MB
* AWS recommends using Parquet because “the Parquet format is up to 2x faster to unload + consumes up to 6x less storage in Amazon S3, compared to text formats”

### Data Models

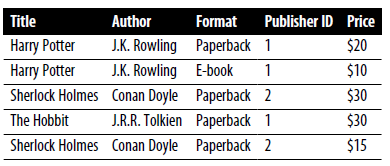
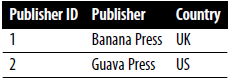
* **Data models *describe* how data is represented**
* Ex: Cars in the real world
* In a database, a car can be described using make, model, year, color, + price
* *These attributes make up a data model for cars*
* Alternatively, you can also describe a car using owner, license plate, + history of registered addresses as 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**
* Ex: The way you represent cars in the 1st data model above makes it easier for people looking to buy cars, whereas the 2nd data model makes it easier for police officers to track down criminals
* 2 types of models that seem opposite to each other but are actually converging = **relational models** and **NoSQL models**

#### Relational Model

* **Relational models** = among the most persistent ideas in CS, invented by Edgar F. Codd in 1970
* Idea is simple but powerful 🡪 **data is organized into relations**, **each relation is a set of tuples**, + **a table is an accepted visual representation of a relation where each row makes up a tuple**
* **Relations are unordered** 🡪 You can **shuffle the order of the rows or 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 or Parquet**.
* It’s **often desirable for relations to be normalized** 🡪 1st normal form (1NF), 2nd normal form (2NF), etc. which **can reduce data redundancy and improve data integrity**
* Consider the relation Book shown below



* There are a lot of duplicates in this data (ex: rows 1 and 2 are nearly identical, except for format + price)
* If publisher information changes (ex: name changes from “Banana Press” to “Pineapple Press”) or its country changes, we’ll have to update rows 1, 2, *and* 4
* If we *separate* publisher information into its own table, as shown below, when a publisher’s information changes, we only have to update the Publisher relation
* Can further normalize the Book relation, such as separating format into a separate relation

* This practice allows us to standardize spelling of the same value across different columns + also makes it easier to make changes to these values, either because these values change or when you want to translate them into different languages
* **One major downside of normalization is that your data is now spread across multiple relations**.
* You *can* **JOIN** data from different relations back together, but **joining can be expensive for large tables**
* **Databases built around the relational data model are relational databases**
* The language you can use to specify the data that you want to retrieve from a database is called a **query language**
* Most popular query language for RDB’s today = **SQL**
* Even though *inspired* by **the relational model**, the data model behind SQL has *deviated* from the original relational model
* Ex: SQL tables can contain row duplicates, whereas *true* relations can’t
* However, this subtle difference has been safely ignored by most people
* The most important thing to note about **SQL** = it’s a **declarative language**, as opposed to **Python**, which is an **imperative language**
* **Imperative paradigm** = **specify the steps needed for an action + the CPU executes these steps to return the outputs**
* **Declarative paradigm = specify the outputs you want + the CPU figures out the steps needed to get you the queried outputs**
* **With a SQL database, you specify the pattern of data you want** (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, + the order in which different parts of the query should be executed**
* With certain added features, SQL can be **Turing-complete** (i.e., in theory, SQL can be used to solve *any* computation problem (without making any guarantee about the time or memory required), but in practice, **it’s not always easy to write a query to solve a specific task, + it’s not always feasible or tractable to execute a query**
* **Figuring out *how* to execute an arbitrary query = hard**, so it’s the job of **query optimizers** = examine all possible ways to execute a query + find the fastest way to do so
* **It’s possible to use ML to improve query optimizers based on learning from incoming queries**
* **Query optimization = 1 of the most challenging problems in database systems**, + **normalization** means 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 = you **generally only need 1 query optimizer and *all* your applications can leverage it**

##### Aside: 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**
* With a **declarative ML system**, users only need to **declare the features’ schema + the task**, + 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 + the number of hidden units) on top of the features’ schema and output
* In H2O AutoML, you don’t specify the model structure or hyperparameters, as it experiments with multiple model architectures + picks out the best model given the features + the task
* Ex: H2O AutoML 🡪 give the system your data (inputs + outputs) + specify the number of models you want to experiment, + it’ll experiment with that number of models and show you the best-performing model:

*# 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

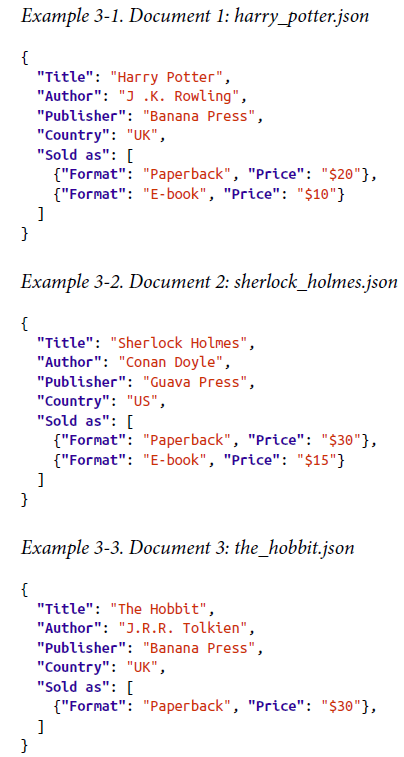
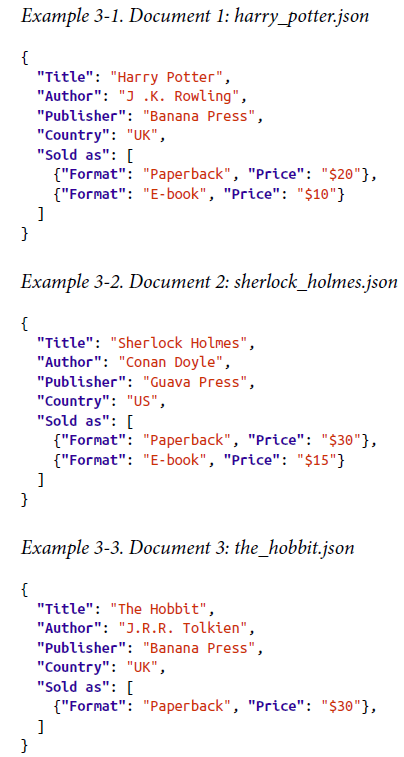
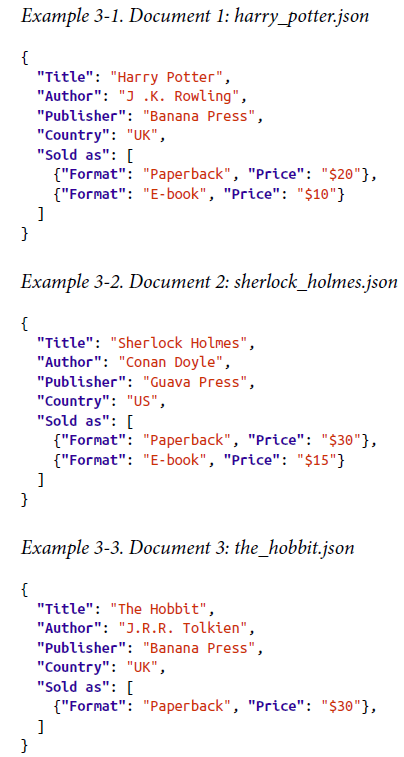
* While **declarative ML** can be useful in many cases, it **leaves unanswered the biggest challenges with ML in production**
* Declarative ML systems today **abstract away the model development part**, and, with **models being increasingly commoditized**, **model development is often the easier part**
* The **hard part = feature engineering, data processing, model evaluation, data shift detection, continual learning, + so on**

#### 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, the relational model can be restrictive**
* Ex: It demands that your data follows a strict **schema**, and **schema management is painful**
* It can also be difficult to write and execute SQL queries for specialized applications.
* **NoSQL** (originally started for NON-relational databases) = **“Not Only SQL”, as many NoSQL data systems also support relational models**
* **2 major types of nonrelational models are the document model + the graph model**
* **Document model targets use cases where data comes in self-contained documents + *relationships* between one document + another are *rare***
* **Graph model targets use cases where *relationships* between data items are *common* + important**

##### Document model

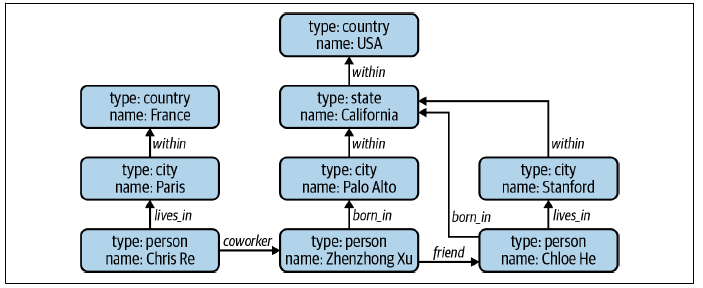
* The document model is **built around the concept of** “**document**” = **often a single continuous string, encoded as JSON, XML, or a binary format like BSON (Binary JSON)**
* All documents in a document database are **assumed to be encoded in the same format**
* Each document has a **unique key** that represents that document, which **can be used to retrieve it**
* A **collection of documents could be considered analogous to a table in an RDN**, + a **document analogous to a row**
* Can convert a relation into a collection of documents that way
* Ex: Convert the book data in the tables from before into 3 JSON documents

* 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**
* Because the document model doesn’t enforce a schema, it’s often referred to as **schema-less.**
* 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 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 examples where information about a book is spread across both the Book and Publisher tables (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 + less efficient to execute JOINs across documents compared to across tables**
* Ex: To find all books with price < $25, you’ll have to read ALL documents, extract the prices, compare them to $25, + return all the documents containing the books with prices < $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 + more DBMS’s, such as PostgreSQL and MySQL, support them both.

##### Graph model

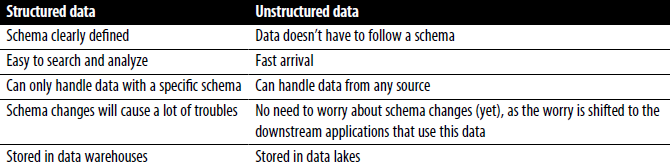
* The graph model is built around the concept of a “**graph**” = **consists of nodes and edges, where the edges represent the relationships between the nodes**
* A **database that uses graph structures to store its data is called a** **graph database**
* If in document databases, the *content* of each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



* 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.
* Imagine you want to find everyone who was born in the USA
* Given this graph, you **start from the node** USA + **traverse the graph following 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 born in the USA, especially given there’re an unknown number of hops between countryand person (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 query with a document database.
* **Many queries that are easy to do in one data model are harder to do in another data model**
* **Picking the right data model for your application can make your life so much easier**

#### Structured vs. Unstructured Data

* **Structured data follows a *predefined* data model, also known as a data schema**
* Ex: A data model might specify that each data item consists of 2 values: the 1st value, “name”, is a string of at most 50 characters, 2nd value, “age”, is an 8-bit integer in the range between 0-200
* The **predefined structure makes your data easier to analyze**
* 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
* Ex: Never kept user email addresses before but now you do, so you have to retrospectively update email information to all *previous* users
* **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 beyond your control, and it’s impossible to make them follow the same schema**
* **Unstructured data doesn’t adhere to a predefined data schema**
* *Usually text but can also be numbers, dates, images, audio, etc.*
* Ex: a text file of logs generated by an 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**
* Ex: The following text is unstructured, but notice the pattern = each line contains 2 values separated by a comma, 1st value is textual, and 2nd value is numerical: {Jack, 23; Huyen, 59}
* ***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
* **Unstructured data also allows for more flexible storage options**
* Ex: 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** 🡪 **can convert all your data, regardless of types + formats, into bytestrings + store them together**
* A **repository for storing structured data** is called a **data warehouse**
* Used to store **data that has been processed into formats ready to be used**
* A **repository for storing unstructured data** is called a **data lake**
* Usually used to store **raw data before processing**
* This table shows a summary of the key differences between structured and unstructured data.



### Data Storage Engines and Processes

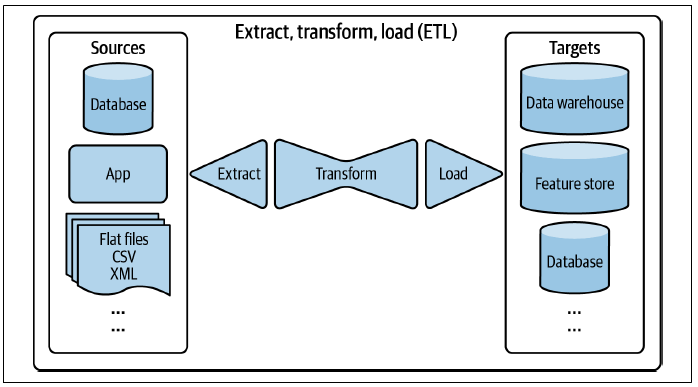
* **Data formats and data models specify the interface for how users can store + retrieve data**
* **Storage engines**, also known as **databases**, **are the implementation of *HOW* data is stored + retrieved on machines**
* It’s **useful to understand different types of databases**, as your team or your adjacent team might **need to select a database appropriate for your application**
* Typically, there are **2 types of workloads that databases are optimized for, transactional processing and analytical processing**, and **there’s a BIG difference between them**

#### Transactional and Analytical Processing

* In the digital world, a **transaction** refers to **ANY kind of action** (tweeting, ordering a ride through a ride-sharing 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**
* **Transactions** are **inserted** **as** they are **generated**, + **occasionally** **updated** when **something changes**, or **deleted** **when no longer needed**
* 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** **(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 + satisfy the low latency, high availability requirements**
* When people hear transactional databases, they usually think of **ACID (atomicity, consistency, isolation, durability)**
* **Atomicity** 🡪 To **guarantee that ALL steps in a transaction are completed successfully *as a group***
* If ***ANY* step in the transaction fails, all other steps *must* fail also**
* Ex: If a user’s payment fails, you don’t want to still assign a driver to that user
* **Consistency** 🡪 To **guarantee that all transactions coming through must follow predefined rules**
* Ex: a transaction must be made by a valid user.
* **Isolation** 🡪 To **guarantee that 2 transactions happen at the same time as if they were isolated**
* 2 users accessing the same data won’t change it at the same time
* Ex: Don’t want 2 users to book the same driver at the same time
* **Durability** 🡪 To **guarantee that once a transaction has been committed, it will remain committed, even in the case of a system failure**
* Ex: After you’ve ordered a ride + 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**
* Martin Kleppmann: “Systems that do not meet the ACID criteria are sometimes called **BASE**, which stands for ***B*asically *A*vailable, *S*oft state, and *E*ventual consistency**. This is even more vague than the definition of ACID”
* **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 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** 🡪 **efficient with queries that allow you to look**
* **at data from different viewpoints 🡪 online analytical processing (OLAP)**
* However, **both the terms OLTP and OLAP have become outdated**, as shown in for **3 reasons**
* **1) The separation of transactional and analytical databases was due to limitations of technology**
* It was **hard to have databases that could handle *both* transactional + analytical queries *efficiently***
* However, **this separation is being closed** 🡪 we **now have transactional databases that can handle analytical** **queries** (CockroachDB) + **analytical databases that can handle transactional queries** (Apache Iceberg and DuckDB)
* **2) 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 using different processing engines to solve different types of queries**
* Interesting paradigm in last decade = to **decouple storage from processing** (or “compute”)
* Adopted by many data vendors like 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**
* **3) “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 + made available: online, nearline, or offline
* “**Online processing”** **=** **data is immediately available for input/output**
* **“Nearline (near-online)”** **= 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**

#### ETL: Extract, Transform, Load

* In the **early days of the relational data model,** **data was mostly structured**
* **ETL** = When data is **extracted**from different sources, it’s first **transformed**into the desired format before being **loaded**into the target destination such as a database or a data warehouse
* Even before ML, ETL was all the rage in the data world, + it’s still relevant today for ML applications
* **ETL refers to the general-purpose processing + aggregating of data into the shape + the format that you want**
* **Extract** **= extracting the data you want from all your data sources**
* **Some** of them will be **corrupted or mal-formatted**
* **In the extracting phase, you need to validate your data + reject the data that doesn’t meet your requirements**
* *For rejected data, you* ***might have to notify the sources***
* **Since this is the 1st step of the process, doing it correctly can save a lot of time downstream**
* **Transform** = the meaty part of the process, where **most of the data processing is done**
* Might want to **join data from multiple sources and clean it**, or to **standardize the value ranges** (e.g., 1 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** **= 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**



* 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 + the nature of data also changed**, while the **number of data sources expanded, and data schemas evolved**
* **Finding it difficult to keep data structured**, some companies thought: “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**” 🡪 process of loading data into storage first *then* processing it later = **ELT(extract, load, transform)**
* **ELT 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** as it’s **inefficient to search through a massive amount of raw data for the data that you want**
* At the same time, **as companies switch to running applications on the cloud + infrastructures become standardized, data structures also become standardized**, so **committing data to a predefined schema becomes more feasible**
* As companies weigh the pros and cons of storing structured data vs. unstructured data, **vendors evolve to offer hybrid solutions that combine the flexibility of data lakes + the data management aspect of data warehouses**
* Ex: Databricks and Snowflake both provide **data lakehouse** solutions.

### Modes of Dataflow

* We’ve been discussing data formats, data models, data storage, + processing for data used within the context of a *single* process
* **Most of the time, in production, you don’t have a single process but multiple**
* A question arises: *how do we pass data between different processes that don’t share memory?*
* **When data is passed from one process to another, we say that the data flows from one process to another,** which **gives us a dataflow**
* 3 main modes of dataflow:
* Data passing **through databases**
* Data passing **through services using requests** such as the requests provided by REST and RPC APIs (e.g., POST/GET requests)
* Data passing **through a real-time transport** like Apache Kafka and Amazon Kinesis

#### Data Passing Through Databases

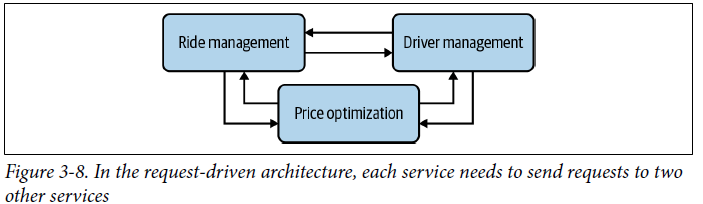
* **Easiest way to pass data between 2 processes** **=** **through databases**
* Ex: To pass data from process A to process B, process A can *write* that data into a database, + process B simply *reads* from that database.
* **This mode, however, doesn’t always work because of 2 reasons**
* 1) **Requires that *both* processes must be able to access the same database**
* This **might be infeasible**, especially if the 2 processes are run by two different companies
* 2) **Requires both processes to access data from databases**
* **Read/write from databases can be slow, making it unsuitable for applications with strict latency requirements** (e.g., almost all consumer-facing applications)

#### Data Passing Through Services

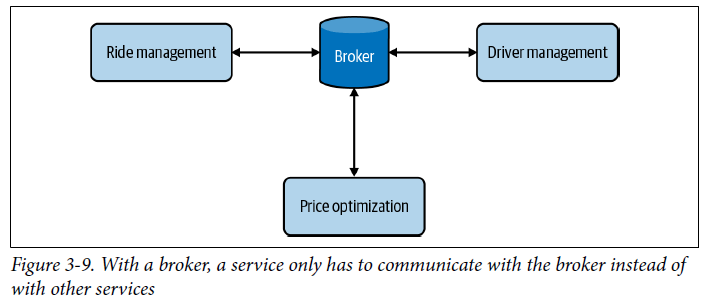
* Another way to **pass data between 2 processes is to send data *directly* through a network that connects these 2 processes.**
* **To pass data from process B to process A, process A first sends a request to process B that specifies the data A needs, and B returns the requested data through the same network**
* Because **processes communicate through requests**, we say that this is **request-driven**.
* This mode of data passing is **tightly coupled with the service-oriented architecture**
* A **service = a process that can be accessed *remotely*** **(e.g., through a network)**
* In this example, **B is exposed to A as a service that A can send requests to**
* **For B to be able to request data from A, A will also need to be exposed to B as a service**
* **2 services in communication with each other can be run by different companies in different applications**
* Ex: A service might be run by a stock exchange that keeps track of current stock prices
* Another service might be run by an investment firm that requests the current stock prices + uses them to predict future stock prices
* **2 services in communication with each other can also be parts of the *same* application**
* **Structuring an application as separate services gives you a** **microservice architecture**
* **Structuring different components of your application as separate services allows each component to be developed, tested, + maintained independently of one another**
* To put the microservice architecture in the context of ML systems, imagine you’re an MLE working on the price optimization problem for a company that owns a ride-sharing application like Lyft
* In reality, Lyft has 100’s of services in its microservice architecture, but for the sake of simplicity, consider only 3 services: **Driver management service** (Predicts how many drivers will be available in the next minute in a given area), **Ride management service** (predicts how many rides will be requested in the next minute in a given area), and **Price optimization service** (Predicts the optimal price for each ride)
* *The price for a ride should be low enough for riders to be willing to pay, yet high enough for drivers to be willing to drive and for the company to make a profit*
* Because price depends on supply (available drivers) *and* demand (requested rides), the **price optimization service needs data from *both* the driver management and ride management services**
* *Each time a user requests a ride, the price optimization service requests the predicted number of rides and predicted number of drivers to predict the optimal price for this ride*
* In practice, price optimization might not have to request the predicted number of rides/drivers *every* time it has to make a price prediction
* It’s a common practice to use the cached predicted number of rides/drivers + request new predictions every minute or so
* **Most popular styles of requests used for passing data through networks = REST (representational state transfer) and RPC (remote procedure call).**
* Detailed analysis is beyond the scope of this book, but **one major difference** is that **REST was designed for requests over networks**, whereas **RPC “tries to make a request to a remote network service look the same as calling a function or method in your programming language*”***
* Because of this, **REST seems to be the predominant style for public APIs**, while the **main focus of RPC frameworks is on requests between services owned by the same organization, typically within the same data center**
* Implementations of a REST architecture are said to be **RESTful**
* Even though many people think of REST as HTTP, REST doesn’t *exactly* mean HTTP because **HTTP is just an implementation of REST**

#### Data Passing Through Real-time Transport

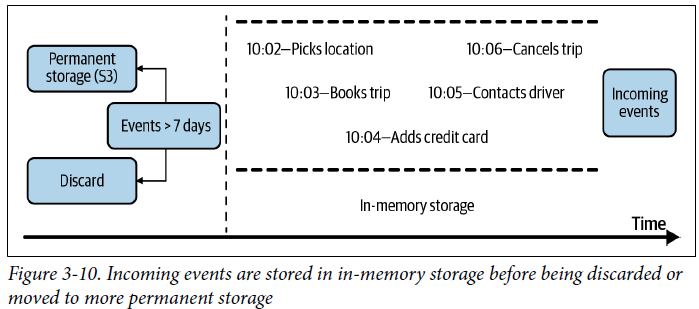
* To understand the motivation for **real-time transports**, go back to the preceding example of the ride-sharing app with 3 simple services: driver + ride management and price optimization
* We discussed how the price optimization service needs data from the ride *and* driver management services to predict the optimal price for each ride
* Now, imagine the *driver* management service *also* needs to know the number of rides from the ride management service to know how many drivers to mobilize
* It *also* wants to know predicted prices from the price optimization service to use them as incentives for potential drivers (e.g., if you get on the road now you can get a 2x surge charge)
* Similarly, the ride management service might *also* want data from the driver management *and* price optimization services
* **If we pass data through services** as discussed in the previous section, **each of these services needs to send requests to the other 2 services**, as shown below



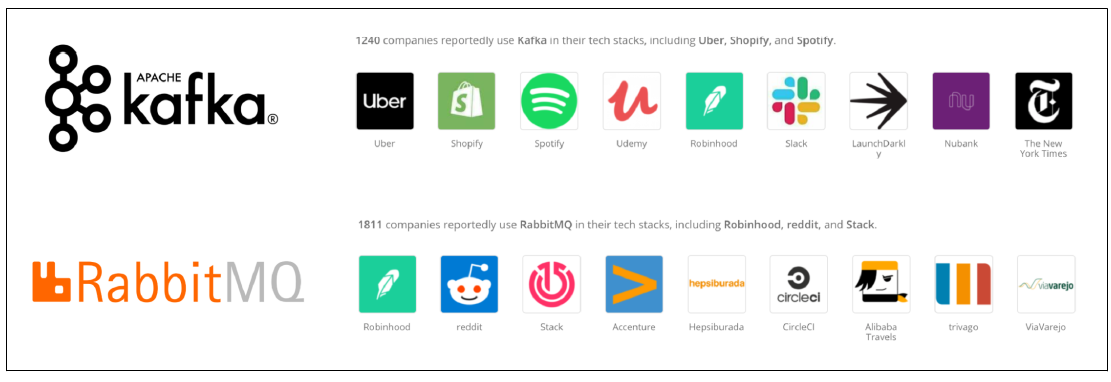
* With only 3 services, data passing is already getting complicated, so imagine having hundreds, if not thousands of services like what major internet companies have.
* **Interservice data passing can blow up and become a bottleneck, slowing down the entire system**
* **Request-driven data passing is synchronous**: the **target service has to listen to the request for the request to go through**
* If the price optimization service requests data from the driver management service and the driver management service is down, the price optimization service will keep resending the request until it times out
* And if price optimization is down before it receives a response, the response will be lost
* **A service that is down can cause all services that require data from it to be down**
* What if there’s a **broker** **that coordinates data passing among services**?
* **Instead of having services request data *directly* from each other and creating a web of complex interservice data passing, each service only has to communicate with the broke**



* Ex: Instead of having *other* services request the driver management services for the predicted number of drivers for the next minute, what if whenever the driver management service makes a prediction, this prediction is broadcast to a broker?
* Whichever service wants data from the driver management service can check that broker for the most recent predicted number of drivers.
* Similarly, whenever the price optimization service makes a prediction about the surge charge for the next minute, this prediction is broadcast to the broker
* **Technically, a database can be a broker** (each service can write data to a database + other services that need the data can read from that database)
* However, as mentioned, **reading + writing from databases are too slow for applications with strict latency requirements**
* **Instead of using databases**, we **use in-memory storage to broker data**
* **Real-time transports can be thought of as in-memory storage for data passing among services**
* A **piece of data broadcast to a real-time transport is called an event**, so this **architecture** is, therefore, also called **event-driven**
* A real-time transport is sometimes called an **event bus**
* **Request-driven architecture works well for systems that rely more on logic than on data**
* **Event-driven architecture works better for systems that are data-heavy**
* The **2 most common types of real-time transports are pubsub (short for publish-subscribe) and message queue**
* **1) In the pubsub model, *any* service can publish to different topics in a real-time transport, and *any* service that subscribes to a topic can read all events in that topic**
* *The services that produce data don’t care about what services consume their data*.
* Pubsub solutions often have a **retention policy** (data will be retained in the real-time transport for a certain period of time (e.g., 7 days) before being deleted or moved to a permanent storage (like Amazon S3)



* **2) In a message queue model,** **an event often has *intended* consumers** (an **event with intended consumers** is called a **message**), and **the message queue is responsible for getting the message to the right consumers**
* Examples of pubsub solutions = Apache Kafka and Amazon Kinesis
* Examples of message queues = Apache RocketMQ and RabbitMQ
* **Both paradigms have gained a lot of traction in the last few years**



### Batch Processing vs. Stream Processing

* Once data arrives in data storage engines like databases, data lakes, or data warehouses, it becomes **historical data**
* This is opposed to **streaming data** (data that is still streaming in)
* **Historical data is often processed in batch jobs** **(jobs that are kicked off periodically)**
* Ex: Once a day, kick off a batch job to compute average surge charge for all rides in the last day
* When **data is processed in batch jobs**, we refer to it as **batch processing**
* This has been a research subject for many decades, + companies have come up with distributed systems like **MapReduce and Spark to process batch data efficiently**
* When you have **data in real-time transports like Apache Kafka and Amazon Kinesis,** we say that you have **streaming data**
* **Stream processing** refers to doing **computation on streaming data**, which can ***also* be kicked off periodically,** but **periods are usually much shorter than the periods for batch jobs** (e.g., every 5 minutes instead of every day)
* **Computation on streaming data can also be kicked off whenever the need arises**
* Ex: Whenever a user requests a ride, you process your data stream to see what drivers are currently available
* **Stream processing, *when done right*, can give low latency because you can process data as soon as data is generated, without having to first write it into databases**
* **Many people believe that stream processing is less efficient than batch processing because you can’t leverage tools like MapReduce or Spark**
* ***This is not always the case*, for 2 reasons**
* **1) Streaming technologies** like Apache Flink are **proven to be highly scalable + fully distributed**, which means they **can do computation in parallel**
* **2) The strength of stream processing is in stateful computation**
* Consider the case where you want to process user engagement during a 30-day trial
* If you kick off this batch job every day, *you’ll have to do computation over the last 30 days every day*
* **With stream processing, it’s possible to continue computing only the *new* data each day + joining the new data computation w/ the older data computation, preventing redundancy**
* **Because batch processing happens much less frequently than stream processing, in ML, batch processing is usually used to compute features that change less often**
* Ex: Drivers’ ratings (if a driver has had 100’s of rides, their rating is less likely to change significantly from one day to the next)
* **Batch features** **(features extracted through batch processing**) **are also known as static features**
* **Stream processing is used to compute features that change quickly**
* Ex: How many drivers are available right now, how many rides have been requested in the last minute, how many rides will be finished in the next 2 minutes, median price of the last 10 rides in this area, etc.
* **Features about the current state of the system are important to make the optimal predictions**
* **Streaming features** **(features extracted through stream processing**) **are also known as dynamic features**
* **For many problems, you need not only batch features or streaming features, but *both***
* You **need infrastructure that allows you to process streaming data as well as batch data + join them together to feed into your ML models**
* **To do computation on data streams, you need a *stream* computation engine** (the way **Spark and MapReduce are *batch* computation engines**)
* For *simple* streaming computation, you might be able to get away with the built-in stream computation capacity of real-time transports like Apache Kafka
* *But Kafka stream processing is limited in its ability to deal with various data sources.*
* **For ML systems that leverage streaming features, the streaming computation is *rarely* simple**
* The **number of stream features** used in an application such as fraud detection and credit scoring **can be in the hundreds, if not thousands**
* **Stream feature extraction logic can require complex queries** with join and aggregation along different dimensions
* **To extract these features requires efficient stream processing engines**
* Might want to look into tools like Apache Flink, KSQL, and Spark Streaming
* Of these 3 engines, Apache Flink and KSQL are more recognized in the industry + provide a nice SQL abstraction for data scientists.
* **Stream processing is more difficult because the data amount is unbounded + the data comes in at variable rates and speeds**
* **It’s easier to make a stream processor do batch processing than to make a batch processor do stream processing**
* Apache Flink’s core maintainers have been arguing for years that batch processing is a special case of stream processing

### Summary

* This chapter is built on the foundations established in Chapter 2 around the importance of data in developing ML systems
* **We learned 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** and the **pros and cons of row-major versus column-major formats** as well as **text versus binary formats**
* We continued to cover **3 major data models: relational, document, graph**
* Even though the relational data model is the most well-known given the popularity of SQL, **all 3 models are widely used today**, and each is good for a certain set of tasks
* When talking about the relational model compared to the document model, many people think of the former as structured and the latter as unstructured
* The **division between structured and unstructured data is quite fluid** 🡪 **main question is who has to shoulder the responsibility of assuming the structure of data**
* **Structured data = the code that *writes* the data has to assume the structure**
* **Unstructured data = the code that *reads* the data has to assume the structure**
* We continued the chapter with **data storage engines and processing**
* We studied **databases optimized for 2 distinct types of data processing: transactional processing and analytical processing**
* We studied **data storage engines** and **processing** together because traditionally storage is coupled with processing: transactional databases for transactional processing and analytical databases for analytical processing
* However, **in recent years, many vendors have worked on decoupling storage and processing**
* **Today, we have transactional databases that can handle analytical queries and analytical databases that can handle transactional queries**
* When discussing data formats, data models, data storage engines, + processing, data is assumed to be within a single process
* However, **while working in production, you’ll likely work with multiple processes, and you’ll likely need to transfer data between them**
* We discussed **3 modes of data passing**
* The **simplest mode is passing through databases**
* The **most popular mode of data passing for processes is data passing through services**
* In this mode, a **process is exposed as a service that another process can send requests for data**
* This mode of data passing is **tightly coupled with microservice architectures**, where **each component of an application is set up as a service**
* A **mode of data passing that has become increasingly popular over the last decade is data passing through a real-time transport** like Apache Kafka and RabbitMQ
* This mode of data passing is **somewhere between passing through databases and passing through services 🡪 allows for asynchronous data passing with reasonably low latency**
* **As data in real-time transports have different properties from data in databases, they require different processing techniques** **(Batch vs. Stream)**
* **Data in databases is often processed in batch jobs + produces static features**, whereas **data in real-time transports is often processed using stream computation engines and produces dynamic features**
* **Some argue that batch processing is a special case of stream processing, and stream computation engines can be used to unify both processing pipelines.**
* Once we have our data systems figured out, we can collect data and create training data