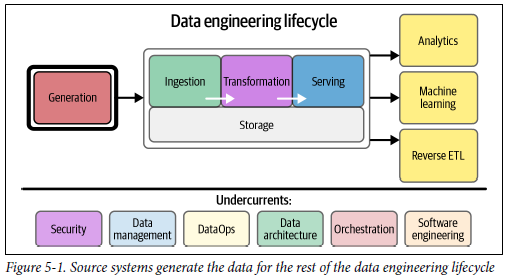
# Fundamentals of Data Engineering - Reis & Housley

## Part II. The Data Engineering Lifecyle in Depth

### Chapter 5 – Data Generation in Source Systems

* Welcome to the 1st stage of the DE lifecycle: **data generation in source systems**
* As described earlier, **the job of a DE is to take data from source systems, do something w/ it, + make it helpful in serving downstream use cases**
* But ***before* you get raw data, you must understand where the data exists, how it is generated, + its characteristics + quirks**
* This chapter covers some popular operational source system patterns + the significant types of source systems
* Many source systems exist for data generation, + we’re not exhaustively covering them all
* We’ll consider the data these systems generate + things you should consider when working w/ source systems, + also discuss how the undercurrents of DE apply to this first phase of the



* **As data proliferates, especially w/ the rise of data sharing (discussed next), we expect that a DE’s role will shift heavily toward understanding the interplay between data sources + destinations**
* The **basic plumbing tasks** of DE (moving data from A to B) **will simplify dramatically**
* On the other hand, **it** **will remain critical to understand the nature of data as it’s created in source systems**

#### Sources of Data: How is Data Created?

* As you learn about the various underlying operational patterns of the systems that generate data, it’s essential to understand how data is created
* **Data is an unorganized, context-less collection of facts + figures** that **can be created in many ways, both analog + digital**
* **Analog data**creation **occurs in the real world**, such as vocal speech, sign language, writing on paper, or playing an instrument
* This analog data is **often transient** (how often have you had a verbal conversation whose contents are lost to the ether after the conversation ends?)
* **Digital data**is either **created by converting analog data to digital form or is the native product of a digital system**
* Analog to digital ex: A mobile texting app that converts analog speech into digital text
* An example of digital data creation is a credit card transaction on an ecommerce platform
* A customer places an order, the transaction is charged to their credit card, + the information for the transaction is saved to various databases
* There’s a few common examples in this chapter, such as data created when interacting with a website or mobile application
* But in truth, data is everywhere in the world around us 🡪 captured from IoT devices, credit card terminals, telescope sensors, stock trades, + more
* **Get familiar with your source system and how it generates data**
* **Put in the effort to read the source system documentation + understand its patterns + quirks**
* **If your source system is an RDBMS, learn how it operates (writes, commits, queries, etc.)**
* **Learn the ins + outs of the source system that might affect your ability to ingest from it**

#### Source Systems: Main Ideas

* Source systems produce data in various ways. This section discusses the main ideas frequently encountered as you work with source systems

##### 1) Files and Unstructured Data

* A **file**is a **sequence of bytes, typically stored on a disk**
* **Applications often write data to files**
* **Files may store local parameters, events, logs, images, + audio**
* In addition, **files are a universal medium of data exchange**
* As much as DE’s wish they could get data programmatically, much of the world still sends + receives files
* Ex: If you’re getting data from a government agency, there’s an excellent chance you’ll download the data as an Excel or CSV file or receive the file in an email
* **Main types of source file formats** you’ll run into as a DE (files that originate either manually or as an output from a source system process) = Excel, CSV, TXT, JSON, + XML
* These files have their quirks and can be structured (Excel, CSV), semi-structured (JSON, XML, CSV), or unstructured (TXT, CSV).
* Although you’ll use certain formats heavily as a DE (such as Parquet, ORC, + Avro), we’ll cover these later + put the spotlight here on source system files.
* Chapter 6 covers the technical details of files

##### 2) APIs

* **Application programming interfaces (APIs)** are **a standard way of exchanging data between systems**
* In theory, APIs simplify the data ingestion task for DE’s
* ***In practice*, many APIs still expose a good deal of data complexity for DE’s to manage**
* Even w/ the rise of various services + frameworks, + services for automating API data ingestion, **DE’s must often invest a good deal of energy into maintaining custom API connections**
* We discuss APIs in greater detail later in this chapter

##### 3) Application Databases (OLTP Systems)

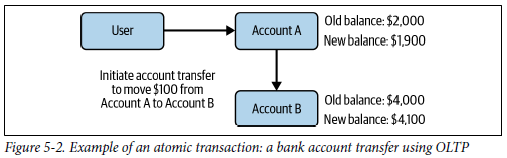
* An **application database****stores the state of an application**
* Standard example = a database that stores account balances for bank accounts
* As customer transactions + payments happen, the application updates bank account balances
* **Typically, an application database is an** **online transaction processing (OLTP)** **system** 🡪 **a database that reads and writes individual data records *at a high rate***
* **OLTP systems are often referred to as transactional databases**, but **this does NOT necessarily imply that the system in question supports atomic transactions**
* More generally, **OLTP databases support low latency + high concurrency**
* An RDBMS database can select or update a row in less than a millisecond (not accounting for network latency) + handle thousands of reads + writes per second
* A document database cluster can manage even *higher* document commit rates at the expense of potential inconsistency
* Some graph databases can also handle transactional use cases
* **Fundamentally, OLTP databases work well as application backends when thousands, even millions of users might be interacting w/ the application simultaneously, updating + writing data concurrently**
* **OLTP systems are *less* suited to use cases driven by analytics at scale, where a single query must scan a vast amount of data**

###### a) ACID

* Support for **atomic transactions** is one of a critical set of database characteristics known together as **ACID** **(atomicity, consistency, isolation, durability)**
* **Consistency means any database read will return the last written version of the retrieved item**
* **Isolation****entails that if 2 updates are in flight concurrently for the same thing, the end database state will be consistent w/ the sequential execution of these updates in the order they were submitted**
* **Durability****indicates that committed data will never be lost, even in the event of power loss**
* Note that **ACID characteristics are NOT required to support application backends, + relaxing these constraints can be a considerable boon to performance + scale**
* *However*, **ACID characteristics guarantee that the database will maintain a consistent picture of the world, dramatically simplifying the app developer’s task**
* **All engineers (data or otherwise) must understand operating w/ + w/out ACID**
* Ex: **To improve performance, some distributed databases use relaxed consistency constraints, such as eventual consistency, to improve performance**
* Understanding the consistency model you’re working with helps you prevent disasters

###### b) Atomic transactions

* An **atomic transaction**is **a set of several changes that are committed *as a unit***
* In the example below, a traditional banking application running on an RDBMS executes a SQL statement that checks 2 account balances, one in Account A (the source) + another in Account B (the destination)



* Money is then moved from Account A to Account B *if sufficient funds are in Account A*
* The entire transaction should run w/ updates to *both* account balances or fail w/out updating either account balance
* **That is, the whole operation should happen as a transaction**

###### c) OLTP and analytics

* **Often, small companies run analytics *directly* *on* an OLTP**
* This pattern **works in the short term but is ultimately not scalable**
* **At some point, running analytical queries on OLTP runs into performance issues due to structural limitations of OLTP or resource contention w/ competing transactional workloads**
* **DE’s must understand the inner workings of OLTP + application backends to set up appropriate integrations w/ analytics systems w/out degrading production application performance**
* As companies offer **more + more analytics capabilities in SaaS applications**, the need for **hybrid capabilities** **(quick updates w/ combined analytics capabilities)** has created new challenges for DE’s
* Use the term **data application**to refer to **applications that hybridize transactional + analytics workloads**

##### 4) OLAP System

* In contrast to an OLTP system, an **online analytical processing (OLAP) system** is **built to run large analytics queries + is typically inefficient at handling lookups of individual records**
* **Ex: Modern column databases are optimized to scan large volumes of data, dispensing w/ indexes to improve scalability + scan performance**
* Any query typically involves scanning a minimal data block, often 100 MB or more in size
* **Trying to look up thousands of individual items per second in such a system will bring it to its knees unless it is combined w/ a caching layer designed for this use case**
* Note that we’re using the term **OLAP**to **refer to any database system that supports**
* **high-scale interactive analytics queries + are *NOT* limiting ourselves to systems that support OLAP cubes (multidimensional arrays of data)**
* The **“online*”* part of OLAP implies that the system constantly listens for incoming queries, making OLAP systems suitable for interactive analytics**
* Although we talk about source systems, **OLAPs are typically storage + query systems for analytics**
* **But in practical use cases, engineers often need to read data from an OLAP system**
* Ex: A DW might **serve data used to train an ML model**
* Or an OLAP system might **serve a reverse ETL workflow**, where derived data in an analytics system is sent back to a source system, such as a CRM, SaaS platform, or transactional application

##### 5) Change Data Capture

* **Change data capture (CDC)** is **a method for extracting each change event (insert, update, delete) that occurs in a database**
* CDC is **frequently leveraged to replicate between databases in near-real-time or create an event stream for downstream processing**
* CDC is **handled differently depending on the database technology**
* **RDB’s** often generate an **event log** **stored directly on the database server** that **can be processed to create a stream**
* Many **cloud NoSQL databases can send a log or event stream to a target storage location**

##### 6) Logs

* A **log captures information about events that occur in systems**
* Ex: A log may capture traffic + usage patterns on a web server
* Ex: A desktop CPU’s OS (Windows, macOS, Linux) logs events as the system boots + when applications start or crash
* **Logs are a rich data source, potentially valuable for downstream data analysis, ML, + automation**
* A few familiar sources of logs: **OS’s, Applications, Servers, Containers, Networks, IoT devices**
* **All logs track events and event metadata**
* **At a minimum, a log should capture who, what, and when:**
* **Who**: The human, system, or service account associated w/ the event (e.g., a web browser user agent or a user ID)
* **What happened**: The event and related metadata
* **When**: The timestamp of the event

###### a) Log encoding

* Logs are **encoded** in a few ways:
* **Binary-encoded logs**: encode data in a **custom compact format for space efficiency and fast I/O**
* Database logs are a standard example.
* **Semi-structured logs**: encoded **as text in an object serialization format** (JSON, more likely)
* Semi-structured logs are **machine-readable + portable**
* However, they are **much less efficient than binary logs**
* And though they are **nominally machine-readable, extracting value from them often requires significant custom code**
* **Plain-text (unstructured) logs**: essentially store the **console output from software**
* As such, no general purpose standards exist
* These logs can provide helpful information for data scientists + MLE’s, though extracting useful information from the raw text data might be complicated

###### b) Log resolution

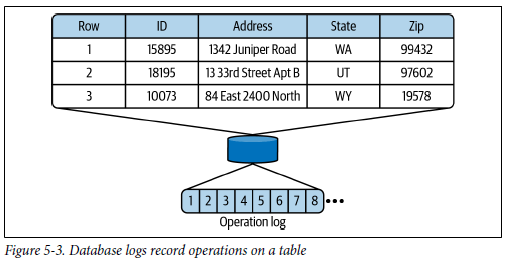
* **Logs are created at various resolutions and log levels**
* The **log resolution**refers to **the *amount* of event data captured in a log**
* Ex: Database logs capture enough information from database events to allow reconstructing the database state at any point in time
* **On the other hand, capturing *all* data changes in logs for a big data system often isn’t practical**
* **Instead, these logs may note only that a particular *type* of commit event has occurred**
* The **log level**refers to **the conditions required to record a log entry, specifically concerning errors and debugging**
* Software is often configurable to **log every event or to log only errors**, for example.

###### c) Log latency: Batch or real time

* **Batch logs are often written continuously to a file**
* **Individual log entries** can be **written to a messaging system** such as Kafka or Pulsar **for real-time applications**

##### 7) Database Logs

* **Database logs**are **essential enough that they deserve more detailed coverage**
* **Write-ahead logs** (typically, **binary files stored in a specific database-native format**) **play a crucial role in database guarantees and recoverability**
* The database server receives write + update requests to a database table (see below), storing each operation in the log before acknowledging the request



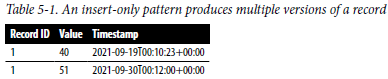
* **The acknowledgment comes a log-associated guarantee: even if the server fails, it can recover its state on reboot by completing the unfinished work from the logs**
* **Database logs are extremely useful in DE, especially for CDC to generate event streams from database changes**

##### 8) CRUD

* **CRUD (create, read, update, delete)** is **a transactional pattern commonly used in programming and represents the 4 basic operations of persistent storage**
* CRUD is **the most common pattern for storing application state in a database**
* A **basic tenet of CRUD is that data must be created before being used**
* **After the data has been created, the data can be read and updated**
* Finally, **the data *may* need to be destroyed**
* **CRUD guarantees these 4 operations will occur on data, regardless of its storage**
* CRUD is a **widely used pattern in software applications**, and you’ll **commonly find CRUD used in APIs and databases**
* Ex: A web app will make heavy use of CRUD for RESTful HTTP requests and storing + retrieving data from a database
* **As w/ any database, we can use snapshot-based extraction to get data from a database where our application applies CRUD operations**
* **On the other hand, event extraction with CDC gives us a complete history of operations + potentially allows for near real-time analytics**

##### 9) Insert-Only

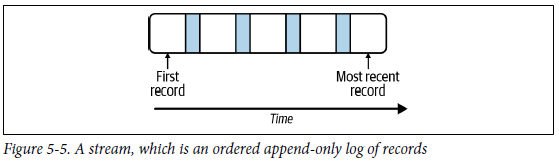
* The **insert-only pattern****retains history directly in a table containing data**
* **Rather than updating records, new records get inserted w/ a timestamp indicating when they were created**



* Ex: Suppose you have a table of customer addresses
* Following a CRUD pattern, you’d simply *update* the record if the customer changed their address
* W/ the insert-only pattern, a new address record is inserted w/ the same customer ID
* To read the current customer address by customer ID, you’d look up the latest record under that ID
* In a sense, **the insert-only pattern maintains a database log directly in the table itself, making it especially useful if the application needs access to history**
* Ex: The insert-only pattern would work well for a banking application designed to present customer address history
* **A *separate* analytics insert-only pattern is often used w/ regular CRUD application tables**
* In the **insert-only *ETL* pattern, data pipelines insert a new record in the target analytics table anytime an update occurs in the CRUD table**
* **Insert-only has a couple of disadvantages**
* 1) Tables can grow quite large, especially if data frequently changes, since each change is inserted into the table
* Sometimes records are purged based on a record sunset date or a maximum number of record versions to keep table size reasonable
* 2) Record lookups incur extra overhead because looking up the current state involves running MAX(created\_timestamp)
* If hundreds or thousands of records are under a single ID, this lookup operation is expensive to run

##### 10) Messages and Streams

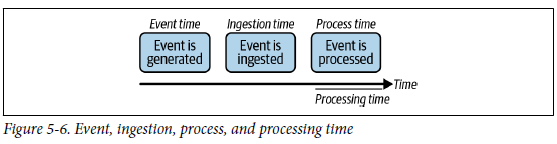
* Related to event-driven architecture, 2 terms that often seen used interchangeably are **message queue** and **streaming platform**, but a subtle but essential difference exists between the two
* Defining and contrasting these terms is worthwhile since they encompass many big ideas related to source systems and practices and technologies spanning the entire DE lifecycle
* A **message**is **raw data communicated across 2 or more systems**
* Ex: System 1 sends a message to System 2
* These systems could be different microservices, a server sending a message to a serverless function, etc.
* A message is typically sent through a **message queue****from a publisher to a consumer**, and **once the message is delivered, it is removed from the queue**
* **Messages are discrete and singular signals in an event-driven system**
* Ex: An IoT device might send a message w/ the latest temperature reading to a message queue
* This message is then ingested by a service that determines whether the furnace should be turned on or off
* This service sends a message to a furnace controller that takes the appropriate action
* Once the message is received, + the action is taken, the message is removed from the message queue
* By contrast, a **stream**is **an append-only log of event records**
* **Streams are ingested + stored in** **event-streaming platforms**
* **As events occur, they are accumulated in an *ordered sequence***



* A **timestamp or an ID might order events**. (*Note that events aren’t always delivered in exact order because of the subtleties of distributed systems*)
* **Use streams when you care about what happened over many events**
* **B/c of the append-only nature of streams, records in a stream are persisted over a long retention window (often weeks or months), allowing for complex operations on records such as aggregations on multiple records or the ability to rewind to a point in time w/in the stream**
* **It’s worth noting that systems that process streams can process messages, + streaming platforms are frequently used for message passing**
* We often accumulate messages in streams when we want to perform message analytics
* In the IoT example, the temperature readings that trigger the furnace to turn on or off might also be later analyzed to determine temperature trends + statistics

##### 11) Types of Time

* While **time** is an essential consideration for *all* data ingestion, it becomes that **much more critical + subtle in the context of streaming, where we view data as continuous + expect to consume it shortly after it is produced**
* **Key types of time** you’ll run into **when ingesting data**: **the time the event is generated, when it’s ingested and processed, + how long processing took**



* **Event time**indicates **when an event is generated in a source system, including the timestamp of the original event itself**
* An undetermined time lag will occur upon event creation, before the event is ingested + processed downstream
* **Always include timestamps for each phase through which an event travels**
* **Log events *as they occur* and *at each stage of time* (when they’re created, ingested, + processed)**
* **Use these timestamp logs to accurately track the movement of your data through your data pipelines**
* After data is created, it is ingested somewhere + **ingestion time****indicates when an event is ingested from source systems into a message queue, cache, memory, object storage, a database, or any place else that data is stored** (see Chapter 6)
* **After ingestion, data may be processed immediately; or within minutes, hours, or days; or simply persist in storage indefinitely**
* **Process time****occurs after ingestion time, when the data is processed (typically, a transformation)**
* **Processing time**is **how long the data took to process, measured in seconds, minutes, hours, etc.**
* **You’ll want to record these various times, preferably in an automated way**
* **Set up monitoring along your data workflows to capture when events occur, when they’re ingested and processed, + how long it took to process events**

#### Source System Practical Details

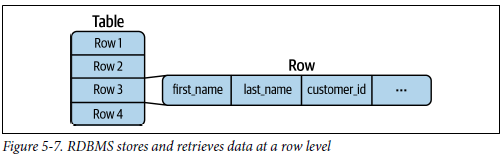
* This section discusses the **practical details of interacting with modern source systems**
* We dig into the details of commonly encountered databases, APIs, + other aspects
* **This information will have a shorter shelf life than the main ideas discussed previously; popular API frameworks, databases, + other details will continue to change rapidly**
* Nevertheless, these details are critical knowledge for working DE’s
* **Study this information as baseline knowledge but read extensively to stay abreast of ongoing developments**

##### 1) Databases

* Now we look at common source system database technologies you’ll encounter as a DE + high-level considerations for working w/ these systems
* **There are as many types of databases as there are use cases for data**
* **Major considerations for understanding database technologies:** Here, we introduce major ideas that occur across a variety of database technologies, including those that back software applications + those that support analytics use cases:
* **Database management system (DBMS)**: **A database system used to store + serve data**
* **Consists of a storage engine, query optimizer, disaster recovery, + other key components** for managing the database system.
* **Lookups: How does the database find and retrieve data?**
* **Indexes can help speed up lookups, but *not all databases have indexes*** **(Know whether your database uses indexes + if so, what are the best patterns for designing + maintaining them?)**
* Understand how to **leverage for efficient extraction**
* Also helps to **have a basic knowledge of the major types of indexes**, including **B-tree** and **log-structured merge-trees (LSM)**
* **Query optimizer:** Does the database utilize an optimizer? What are its characteristics?
* **Scaling and distribution:** Does the database **scale with demand?**
* What **scaling strategy does it deploy?**
* Does it **scale horizontally (more database nodes) or scale vertically (more resources on a single machine)?**
* **Modeling patterns:** What modeling patterns work best w/ the database (e.g., **data normalization or wide tables**)
* See Chapter 8 for a discussion of **data modeling**
* **CRUD:** How is data **queried, created, updated, + deleted** in the database?
* **Every type of database handles CRUD operations differently**
* **Consistency:** Is the database **fully consistent** or does the database support a **relaxed consistency model (e.g., eventual consistency)?**
* Does the database support **optional consistency modes** for reads + writes (e.g., **strongly consistent reads**)?
* We **divide databases into relational and nonrelational categories**
* In truth, nonrelational is far more diverse, but RDB’s still occupy significant space in application backends

###### a) Relational databases

* **A relational database management system (RDBMS)** is **one of the most common application backends**
* Developed at IBM in the 1970s + popularized by Oracle in the 1980s
* The growth of the internet saw the rise of the **LAMP stack (Linux, Apache web server, MySQL, PHP)** + an explosion of vendor and OSS RDBMS options
* Even w/ the rise of NoSQL databases, RDBs have remained extremely popular
* **Data is stored in a table of relations(rows), + each relation contains multiple fields(columns**
* Note that we use the terms *column* and *field* interchangeably



* **Each relation in the table has the same schema(a sequence of columns w/ assigned static types such as string, integer, or float)**
* **Rows are typically stored as a contiguous sequence of bytes on disk**
* **Tables are *typically* indexed by a primary key (PK), a unique field for each row in the table.**
* The **indexing strategy for the PK is closely connected w/ the layout of the table on disk**
* **Tables can also have various foreign keys (FKs)= fields w/ values connected w/ the values of PKs in other tables, facilitating joins, + allowing for complex schemas that spread data across multiple tables**
* In particular, it is possible to design a **normalized schema**
* **Normalization** = a **strategy for ensuring that data in records is not duplicated in multiple places, thus avoiding the need to update states in multiple locations at once + preventing inconsistencies** (see Chapter 8)
* **RDBMS systems are typically ACID compliant**
* **Combining a normalized schema, ACID compliance, + support for high transaction rates makes RDBMSs ideal for storing rapidly changing application states**
* **The challenge for DE’s is to determine how to capture state information over time**
* A full discussion of the theory, history, and technology of RDBMS is beyond the scope of this book
* ***Study RDBMS systems, relational algebra, + strategies for normalization b/c they’re widespread, + you’ll encounter them frequently***

###### b) Nonrelational databases: NoSQL

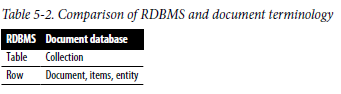
* While **RDBs** are terrific for many use cases, they’re **NOT a one-size-fits-all solution**
* We often see that people start with a RDB under the impression it’s a universal appliance + shoehorn in a ton of use cases + workloads
* **As data + query requirements morph, the RDB can collapse under its weight**
* At that point, you’ll want to use a database that’s appropriate for the specific workload under pressure
* Enter **nonrelational or NoSQL databases**
* **NoSQL (not only SQL)** refers to a **whole class of databases that abandon the relational paradigm**
* On the one hand, **dropping relational constraints can improve performance, scalability, + schema flexibility**
* But as always in architecture, **trade-offs exist**
* **NoSQL databases also typically abandon various RDBMS characteristics, such as strong consistency, joins, or a fixed schema**
* A big theme of this book is that **data innovation is constant**, so let’s take a quick look at the history of NoSQL, as it’s helpful to gain a perspective on *why + how* data innovations impact work as a DE
* In the early 2000s, tech companies such as Google + Amazon began to outgrow their RDBs + pioneered new distributed, nonrelational databases to scale their web platforms
* While the term “NoSQL” first appeared in 1998, the modern version was coined by Eric Evans in the 2000s:
* *I’ve spent the last couple of days at nosqleast and one of the hot topics here is the name “nosql”. Understandably, there are a lot of people who worry that the name is Bad, that it sends an inappropriate or inaccurate message. While I make no claims to the idea, I do have to accept some blame for what it is now being called. How’s that? Johan Oskarsson was organizing the first meetup and asked the question “What’s a good name?” on IRC; it was one of 3 or 4suggestions that I spouted off in the span of like 45 seconds, without thinking. My regret, however, isn’t about what the name says; it’s about what it doesn’t. When Johan originally had the idea for the first meetup, he seemed to be thinking Big Data + linearly scalable distributed systems, but the name is so vague that it opened the door to talk submissions for literally anything that stored data, + wasn’t an RDBMS*
* **NoSQL remains vague in 2022, but it’s been widely adopted to describe a universe of “new school” databases, alternatives to RDBs**
* There are **numerous flavors of NoSQL database designed for almost any imaginable use case**
* B/c there are far too many NoSQL databases to cover exhaustively, **consider the following database types:** **key-value, document, wide-column, graph, search, + time series**
* These databases are all wildly popular + enjoy widespread adoption
* **A DE should understand these types of databases, including usage considerations, the structure of the data they store, + how to leverage each in the DE lifecycle**

i) Key-value stores

* A **key-value database**is a **nonrelational database that retrieves records using a key that uniquely identifies each record (similar to hash map or dictionary data structures presented in many programming languages but potentially more scalable)**
* **KV stores encompass several NoSQL database types** (ex: document stores, wide column databases)
* **Different types of KV value databases offer a variety of performance characteristics to serve various application needs**
* Ex: **In-memory KV databases** are popular for **caching session data for web + mobile apps, where ultra-fast lookup + high concurrency are required**
* **Storage in such systems = typically temporary**, so if a database shuts down, data disappears
* **Such caches can reduce pressure on the main application database + serve speedy responses**
* Of course, KV stores **can also serve applications requiring high-durability persistence**
* An ecommerce application may need to save + update massive amounts of event state changes for a user + their orders
* A user logs into the ecommerce application, clicks around various screens, adds items to a shopping cart, + then checks out
* **Each event must be durably stored for retrieval**
* **KV stores often persist data to disk + across multiple nodes to support such use cases**

ii) Document stores

* As mentioned previously, a **document store**is **a specialized key-value store**
* In this context, a **document**is a **nested object** (can usually think of each document as a JSON object for practical purposes)
* **Documents are stored in collections (roughly equivalent to a table in an RDB) + retrieved by key**



* **One key difference between RDBs + document stores = the latter does NOT support joins**
* This **means that data *cannot* be easily normalized, i.e., split across multiple tables**
* Applications can still join *manually*
* Code can look up a document, extract a property, + then retrieve another document
* ***Ideally*, all related data can be stored in the *same* document**
* **In many cases, the *same* data must be stored in *multiple* documents spread across *numerous* collections**
* **SWE’s must be careful to update a property everywhere it is stored (Many document stores support a notion of transactions to facilitate this)**
* **Document databases generally embrace all the flexibility of JSON + don’t enforce schema or types, which is *a blessing + a curse***
* On the one hand, this **allows the schema to be highly flexible + expressive, + the schema can also evolve as an application grows**
* Flip side = we’ve seen **document databases become absolute nightmares to manage + query**
* **If developers are not careful in managing schema evolution, data may become inconsistent + bloated over time**
* **Schema evolution can also break downstream ingestion + cause headaches for DE’s if it’s not communicated in a timely fashion (before deployment)**
* The following is an example of data that is stored in a collection called *users*
* The collection key is the *id*
* We also have a *name* (along with *first* and *last* as child elements) + an array of the user’s favorite bands within each document:
* {

**"users"**:[

{

**"id"**:1234,

**"name"**:{

**"first"**:"Joe",

**"last"**:"Reis"

},

**"favorite\_bands"**:[

"AC/DC",

"Slayer",

"WuTang Clan",

"Action Bronson"

]

},

{

**"id"**:1235,

**"name"**:{

**"first"**:"Matt",

**"last"**:"Housley"

},

**"favorite\_bands"**:[

"Dave Matthews Band",

"Creed",

"Nickelback"

]

}

]

}

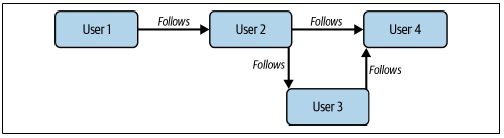
* **To query the data in this example, you can retrieve records by key**
* **Note** that **most document databases also support the creation of indexes + lookup tables to allow retrieval of documents by specific properties**
* This is **often invaluable in application development when you need to search for documents in various ways**
* Ex: Could set an index on *name*.
* **Another critical technical detail for DE’s is that document stores are generally *NOT* ACID compliant, unlike RDBs**
* **Technical expertise in a *particular* document store is essential to understanding performance, tuning, configuration, related effects on writes, consistency, durability, etc.**
* **Ex: Many document stores are “eventually consistent”**
* **Allowing data distribution across a cluster is a boon for scaling + performance but can lead to catastrophes when engineers + developers don’t understand the implications**
* **To run analytics on document stores, engineers generally must run a full scan to extract *all* data from a collection or employ a CDC strategy to send events to a target stream**
* The **full scan approach can have both performance and cost implications**
* The scan often slows the database as it runs, + many serverless cloud offerings charge a significant fee for each full scan
* **In document databases, it’s often helpful to create an index to help speed up queries** (see Chapter 8)

iii) Wide-column

* A **wide-column database**is **optimized for storing massive amounts of data with high transaction rates and extremely low latency**
* These databases **can scale to extremely high write rates + vast amounts of data**
* Specifically, **wide-column databases can support PB of data, millions of requests per second, + sub-10ms latency**
* These characteristics have made wide-column databases **popular in ecommerce, fintech, ad tech, IoT, and real-time personalization applications**
* **DE’s must be aware of the operational characteristics of the wide-column databases they work w/ to set up a suitable configuration, design the schema, + choose an appropriate row key to optimize performance + avoid common operational issues**
* These databases **support rapid scans of massive amounts of data, but they do NOT support complex queries**
* They **have only a *single* index (the row key) for lookups**
* **DE’s must generally extract data + send it to a *secondary* analytics system to run complex queries to deal w/ these limitations**
* **Can be accomplished by running large scans for the extraction or employing CDC to capture an event stream**

iv) Graph databases

* **Graph databases****explicitly store data w/ a mathematical graph structure (as a set of nodes and edges)**
* Neo4j has proven extremely popular, while Amazon, Oracle, + other vendors offer their graph database products
* Roughly speaking, **graph databases are a good fit when you want to analyze the connectivity between elements**
* Ex: Could use a document database to store one document for each user describing their properties
* Could add an array element for “connections” that contains directly connected users’ IDs in a social media context
* It’s pretty easy to determine the number of direct connections a user has, but *suppose you want to know how many users can be reached by traversing 2 direct connections*
* *Could* answer this question by writing **complex code, but each query would run slowly + consume significant resources**
* **The document store is simply NOT optimized for this use case**
* **Graph databases *ARE* designed for precisely this type of query**
* Their **data structures allow for queries based on the connectivity between elements**
* Graph databases are indicated **when we care about understanding complex traversals between elements.**
* In the parlance of graphs, we **store nodes**(users in the preceding example) **+ edges**(connections between users)
* **Graph databases support rich data models for both nodes + edges**
* **Depending on the underlying graph database engine, graph databases utilize specialized query languages** such as SPARQL, Resource Description Framework (RDF), Graph Query Language (GQL), and Cypher
* As an example of a graph, consider a network of 4 users
* User 1 follows User 2, who follows User 3 and User 4; User 3 also follows User 4



* **It’s anticipated that graph database applications will grow dramatically outside of tech companies, + market analyses also predict rapid growth**
* Of course, **graph databases are beneficial from an operational perspective + support the kinds of complex social relationships critical to modern applications**
* Graph structures are also fascinating from the perspective of data science + ML, potentially revealing deep insights into human interactions + behavior
* This introduces **unique challenges for DE’s who may be more accustomed to dealing w/ structured relations, documents, or unstructured data**
* **DE’s must choose whether to do the following**:
* *Map source system graph data into one of their existing preferred paradigms*
* *Analyze graph data within the source system itself*
* *Adopt graph-specific analytics tools*
* **Graph data can be reencoded into rows in an RDB, which may be a suitable solution depending on the analytics use case**
* **Transactional graph databases** are **also designed for analytics, although large queries may overload production systems.**
* **Contemporary cloud-based graph databases support read-heavy graph analytics on massive quantities of data**

v) Search

* A **search database**is a **non-relational database used to search your data’s complex + straightforward semantic + structural characteristics**
* **2 prominent use cases exist for a search database: text search and log analysis**
* **Text search****involves searching a body of text for keywords or phrases, matching on exact, fuzzy, or semantically similar matches**
* **Log analysis**is **typically used for anomaly detection, real-time monitoring, security analytics, + operational analytics**
* **Queries can be optimized + sped up w/ the use of indexes**
* Depending on the type of company you work at, you **may use search databases either regularly or not at all**
* Regardless, it’s **good to be aware they exist in case you come across them in the wild**
* **Search databases are popular for fast search + retrieval**, + can be found in various applications
* Ex: An ecommerce site may power its product search using a search database
* **As a data engineer, you might be expected to bring data from a search database (such as Elasticsearch, Apache Solr or Lucene, or Algolia) into downstream KPI reports or something similar**

vi) Time series

* A **time series** is a **series of values organized by time**
* Ex: Stock prices might move as trades are executed throughout the day, or a weather sensor will take atmospheric temperatures every minute
* ***Any* events that are recorded over time (either regularly or sporadically) are time-series data**
* A **time-series database**is **optimized for retrieving + statistical processing of time-series data**
* **While time-series data** such as orders, shipments, logs, + so forth **have been stored in RDBs for ages, these data sizes + volumes were often tiny**
* **As data grew faster + bigger, new special-purpose databases were needed**
* **Time-series databases address the needs of growing, high-velocity data volumes from IoT, event + application logs, ad tech, + fintech, among many other use cases**
* **Often these workloads are write-heavy** + **as a result, time-series databases often utilize memory buffering to support fast writes + reads**
* We should distinguish between **measurement** and **event-based data**, common in time-series databases
* **Measurement data**is **generated regularly, such as temperature or air-quality sensors**
* **Event-based data**is **irregular + created every time an event occurs** (ex: when a motion sensor detects movement)
* The **schema for a time series typically contains a timestamp and a small set of fields**
* ***Because the data is time-dependent, the data is ordered by the timestamp***
* **Makes time-series databases suitable for operational analytics but *not great for BI use cases***
* **Joins are not common**, though some quasi time-series databases such as Apache Druid support joins
* Many time-series databases are available, both as OSS and paid options

##### 2) APIs

* **APIs are now a standard + pervasive way of exchanging data in the cloud, for SaaS platforms, + between internal company systems**
* Many types of API interfaces exist across the web, but we are **principally interested in those built around HTTP, the most popular type on the web and in the cloud**

###### a) REST

* **REST (representational state transfer)** **= currently (2022-2023) the dominant API paradigm**
* This **set of practices + philosophies for building HTTP web APIs** was laid out by Roy Fielding in 2000 in a PhD dissertation
* REST is **built around HTTP verbs, such as GET and PUT**
* *In practice, modern REST uses only a handful of verb mappings outlined in the dissertation*
* **One of the principal ideas of REST is that interactions are stateless**
* *Unlike in a Linux terminal session***, there is no notion of a session w/ associated state variables such as a working directory**, +**each REST call is independent**
* **REST calls can change the system’s state, but these changes are *global*, applying to the full system rather than a current session**
* **Critics point out that REST is in no way a *full* specification**
* **REST stipulates *basic* properties of interactions, but developers utilizing an API must gain a significant amount of domain knowledge to build applications or pull data effectively**
* There is great variation in levels of API abstraction
* In some cases, APIs are merely a thin wrapper over internals that provides the minimum functionality required to protect the system from user requests
* In other examples, a REST data API is a masterpiece of engineering that prepares data for analytics applications + supports advanced reporting
* A couple of developments have simplified setting up data-ingestion pipelines from REST APIs
* **1) Data providers frequently supply client libraries in various languages**, especially in Python.
* Client libraries **remove much of the boilerplate labor of building API interaction code**
* Client libraries **handle critical details such as authentication + map fundamental methods into accessible classes.**
* **2) Various services + OSS libraries have emerged to interact w/ APIs + manage data synchronization**
* Many SaaS and OSS vendors provide off-the-shelf connectors for common APIs
* **Platforms also simplify the process of building custom connectors as required**
* There are numerous data APIs *without* client libraries or out-of-the-box connector support
* **DE’s would do well to reduce undifferentiated heavy lifting by using off-the-shelf tools**
* However, **low-level plumbingtasks still consume many resources**
* **At virtually any large company, DE’s will need to deal with the problem of writing + maintaining custom code to pull data from APIs, which requires understanding the structure of the data as provided, developing appropriate data-extraction code, + determining a suitable data synchronization strategy**

###### b) GraphQL

* **GraphQL**was created at Facebook as **a query language for application data + an alternative to generic REST APIs**
* **Whereas REST APIs generally restrict your queries to a *specific* data model, GraphQL opens up the possibility of retrieving multiple data models in a single request**
* This **allows for more flexible + expressive queries than w/ REST**
* GraphQL is **built around JSON + returns data in a shape resembling the JSON query**
* There’s something of a holy war between REST and GraphQL, w/ some engineering teams partisans of one or the other + some using both
* **In reality, engineers will encounter both as they interact with source systems**

###### c) Webhooks

* **Webhooks**are **a simple event-based data-transmission pattern**
* The data source can be an application backend, a web page, or a mobile app
* **When specified events happen in the source system, this triggers a call to an HTTP endpoint hosted by the data consumer**
* ***Notice* the connection goes *from the source system to the data sink*, *the opposite of typical APIs***
* For this reason, **webhooks are often called reverse APIs**
* **The endpoint can do various things w/ the POST event data, potentially triggering a downstream process or storing the data for future use**
* **For analytics purposes, we’re interested in collecting these events**
* **Engineers commonly use message queues to ingest data at high velocity + volume** (talked about later in this chapter)

###### d) RPC and gRPC

* A **remote procedure call (RPC) is commonly used in distributed computing** + **allows you to run a procedure on a remote system**
* **gRPC**is a **remote procedure call library developed internally at Google** in 2015 + later released as an open standard
* Its use at Google alone would be enough to merit inclusion in discussions
* Many Google services, such as Google Ads + GCP, offer gRPC APIs
* **gRPC** is **built around the Protocol Buffers open data serialization standard, also developed by Google.**
* **gRPC emphasizes the efficient bidirectional exchange of data over HTTP/2**
* **Efficiency** refers to aspects such as **CPU utilization, power consumption, battery life, + bandwidth**
* **Like GraphQL, gRPC imposes much more specific technical standards than REST**, thus **allowing the use of common client libraries + allowing engineers to develop a skill set that will apply to *any* gRPC interaction code**

##### 3) Data Sharing

* The **core concept of cloud data sharing** is that **a multitenant system supports security policies for sharing data among tenants**
* Concretely, ***any* public cloud object storage system w/ a fine-grained permission system can be a platform for data sharing**
* **Popular cloud DW platforms also support data-sharing capabilities**
* Of course, **data can also be shared through download or exchange over email, but a multitenant system makes the process much easier**
* **Many modern sharing platforms (especially cloud DWs) support row, column, + sensitive data filtering**
* Data sharing also streamlines the notion of the **data marketplace**, available on several popular clouds and data platforms
* **Data marketplaces provide a centralized location for data commerce, where data providers can advertise their offerings + sell them w/out worrying about the details of managing network access to data systems.**
* Data sharing **can also streamline data pipelines within an organization**
* Data sharing **allows units of an organization to manage their data + selectively share it w/ other units while still allowing individual units to manage their compute + query costs separately, facilitating data decentralization**
* This **facilitates decentralized data management patterns such as** **data mesh**
* **Data sharing + data mesh align closely w/ the philosophy of common architecture** **components**
* **Chapter 3 = *Choose common components that allow the simple + efficient interchange of data + expertise, rather than embracing the most exciting + sophisticated tech***

##### 4) 3rd-Party Data Sources

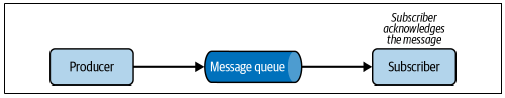
* The consumerization of technology means every company is essentially now a tech company
* The consequence is that these companies (+ increasingly government agencies) want to make their data available to their customers + users, either as part of their service or as a separate subscription
* Ex: US Bureau of Labor Statistics publishes various statistics about the US labor market, NASA publishes various data from its research initiatives, + Facebook shares data w/ businesses that advertise on its platform
* *Why would companies want to make their data available?*
* Data is sticky, + a **flywheel is created by allowing users to integrate + extend their application into a user’s application**
* **Greater user adoption and usage means more data, which means users can integrate more data into their applications + data systems**
* The **side effect is there are now almost infinite sources of 3rd-party data**
* **Direct 3rd-party data access is commonly done via APIs, through data sharing on a cloud platform, or through data download**
* APIs often provide deep integration capabilities, allowing customers to pull + push data
* Ex: Many CRMs offer APIs that their users can integrate into their systems + applications.
* A common workflow is to get data from a CRM, blend the CRM data through the customer scoring model, + then use reverse ETL to send that data back into CRM for salespeople to contact better-qualified leads

##### 5) Message Queues and Event-Streaming Platforms

* **Event-driven architectures** **are pervasive in software applications + are poised to grow their popularity even further**
* 1) **Message queues and event-streaming platforms (critical layers in event-driven architectures) are easier to set up + manage in a cloud environment**
* 2) **The rise of data apps (applications that directly integrate real-time analytics) are growing from strength to strength**
* Event-driven architectures are ideal in this setting because **events can both trigger work in the application and feed near real-time analytics**
* Please note that **streaming data** (in this case, **messages** and **streams**) **cuts across many DE lifecycle stages**
* Unlike an RDBMS, which is often *directly* attached to an application, the **lines of streaming data are sometimes less clear-cut**
* **These systems are *used* as source systems, but will often cut across the DE lifecycle b/c of their transient nature**
* Ex: You can use an event-streaming platform for message passing in an event-driven application, a source system
* The same event-streaming platform can be used in the ingestion + transformation stage to process data for real-time analytics
* ***As source systems*, message queues and event-streaming platforms are used in numerous ways,** from routing messages between microservices ingesting millions of events per second of event data from web, mobile, + IoT applications

###### a) Message queues

* A **message queue**is a **mechanism to *asynchronously* send data (usually as small individual messages, in the KBs) between discrete systems using a publish + subscribe model**
* **Data is published to a message queue + is delivered to 1+ subscribers**
* **The subscriber acknowledges receipt of the message, removing it from the queue**



* **Message queues allow applications + systems to be decoupled from each other, + are widely used in microservices architectures**
* The message queue **buffers messages to handle transient load spikes + makes messages durable through a distributed architecture w/ replication**
* Message queues are a **critical ingredient for decoupled microservices + event-driven architectures**
* **Some things to keep in mind w/ message queues are frequency of delivery, message ordering, and scalability.**

i) Message ordering and delivery

* The **order in which messages are created, sent, + received can significantly impact downstream subscribers**
* ***In general,* order in distributed message queues is a tricky problem**
* **Message queues often apply a *fuzzy* notion of order and first in, first out (FIFO)**
* ***Strict* FIFO** means that **if message A is ingested before message B, message A will always be delivered before message B**
* **In practice, messages might be published + received out of order, especially in highly distributed message systems**
* Ex: Amazon SQS **standard queues make the best effort to preserve message order**
* SQS also offers **FIFO queues**, which **offer much stronger guarantees *at the cost of extra overhead***
* ***In general,* DON’T assume your messages will be delivered in order unless your message queue technology guarantees it**
* **You typically need to design for out-of-order message delivery**

ii) Delivery frequency

* **Messages can be sent *exactly* once or *at least* once**
* If a message is sent **exactly once**, then **after the subscriber acknowledges the message, the message disappears + won’t be delivered again**
* Messages sent **at least once****can be consumed by multiple subscribers or by the same subscriber more than once**
* This is **great when duplications or redundancy don’t matter**
* Ideally, systems should be **idempotent** 🡪 the **outcome of processing a message *once* is identical to the outcome of processing it *multiple times***
* This helps to account for a variety of subtle scenarios
* Ex: Even if our system can guarantee exactly-once delivery, a consumer might fully process a message but fail right before acknowledging processing
* The message will effectively be processed twice, but an idempotent system handles this scenario gracefully

iii) Scalability

* The **most popular message queues utilized in event-driven applications are horizontally scalable, running across multiple servers**
* This **allows these queues to scale up + down dynamically, buffer messages when systems fall behind, + durably store messages for resilience against failure**
* However, this **can create a variety of complications**, as mentioned previously (**multiple deliveries** and **fuzzy ordering**)

###### b) Event-streaming platforms

* In some ways, an **event-streaming platform** is a continuation of a message queue in that **messages are passed from producers to consumers**
* As discussed previously, the **big difference between messages and streams is that a message queue is primarily used to route messages w/ certain delivery guarantees, while, in contrast, an event-streaming platform is used to ingest + process data in an ordered *log* of records**
* In an event-streaming platform, **data is retained for a while, + it is possible to replay messages from a past point in time**
* Let’s describe an **event** related to an event-streaming platform
* An **event** is “**something that happened, typically a change in the stateof something**”
* An event **has the following features**: a **key**, a **value**, + a **timestamp**
* ***Multiple* key-value timestamps might be contained in a single event**
* Ex: An event for an ecommerce order might look like this:
* {

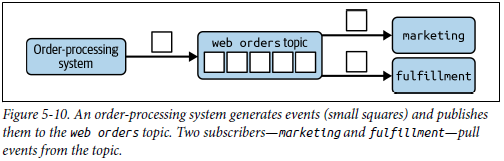
**"Key"**:"Order # 12345",

**"Value"**:"SKU 123, purchase price of $100",

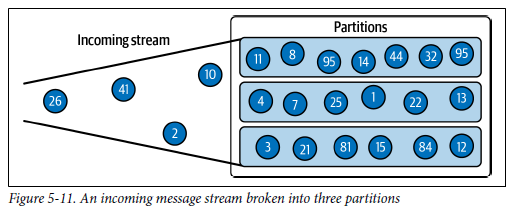
**"Timestamp"**:"2023-01-02 06:01:00"

}

* Let’s look at some of the **critical characteristics of an event-streaming platform that you should be aware of as a DE**
* **Topics: In an event-streaming platform, a producer streams events to a topic** **= a collection of related events**
* A topic might contain fraud alerts, customer orders, or temperature readings from IoT devices, for example
* **A topic can have 0, 1, or multiple producers + customers on most event-streaming platforms**
* Using the preceding event example, a topic might be “web orders”
* Also, let’s send this topic to a couple of consumers, such as ”fulfillment” and “marketing”
* This is an **excellent example of blurred lines between analytics + an event-driven system**
* The ”fulfillment” subscriber will use events to trigger a fulfillment process, while “marketing” runs real-time analytics or trains + runs ML models to tune marketing campaigns



* **Stream partitions**are **subdivisions of a stream into multiple streams**
* A good analogy is a multilane freeway
* Having multiple lanes allows for **parallelism + higher throughput**
* **Messages are distributed across partitions by partition key**
* **Messages with the same partition key will *always* end up in the same partition**
* Below, for example, each message has a numeric ID (shown inside the circle representing the message) that we use as a partition key



* To determine the partition, we divide by 3 + take the remainder
* Going from bottom to top, the partitions have remainder 0, 1, and 2, respectively
* **Set a partition key so that messages that should be processed together have the same partition key**
* Ex: It is common in IoT settings to want to send all messages from a particular device to the same processing server
* Can achieve this by using a device ID as the partition key, + then setting up one server to consume from each partition
* A **key concern with stream partitioning is ensuring that your partition key does not generate hotspotting = a disproportionate number of messages delivered to one partition**
* Ex: If each IoT device were known to be located in a particular US state, we might use the state as the partition key
* Given a device distribution proportional to state population, the partitions containing California, Texas, Florida, + New York might be overwhelmed, w/ other partitions relatively underutilized
* **Ensure that your partition key will distribute messages evenly across partitions**
* **Fault tolerance and resilience**
* **Event-streaming platforms are typically distributed systems, w/ streams stored on various nodes**
* **If a node goes down, another node replaces it, + the stream is still accessible**
* This means **records aren’t lost** (may choose to delete records, but that’s another story)
* **This fault tolerance + resilience make streaming platforms a good choice when you need a system that can reliably produce, store, + ingest event data**

#### Whom You’ll Work With

* **When accessing source systems, it’s essential to understand the people with whom you’ll work**
* **Good diplomacy + relationships w/ the stakeholders of source systems are an underrated and crucial part of successful DE**
* Typically, you’ll deal with **2 categories of stakeholders: systems and data stakeholders**
* A **systems stakeholder****builds + maintains the source systems**
* Might be SWEs, application developers, + 3rd parties
* **Data stakeholders** **own + control access to the data you want**
* Generally handled by IT, a data governance group, or 3rd parties
* Systems + data stakeholders are often different people or teams, but sometimes, they are the same
* You’re **often at the mercy of the stakeholder’s ability to follow correct SWE, database management, + development practices**
* ***Ideally*, the stakeholders are doing DevOps + working in an agile manner**
* **Create a feedback loop between DE’s + stakeholders of the source systems to create awareness of how data is consumed + used**
* This is **among the single most overlooked areas where DE’s can get a lot of value**
* **When something happens to the upstream source data (and something *will* happen,** whether it’s a schema or data change, a failed server or database, or other important events)**, you want to make sure you’re made aware of the impact these issues will have on your DE systems**
* It might help to have a **data contract** in place with your upstream source system owners
* *A data contract is a written agreement between the owner of a source system + the team ingesting data from that system for use in a data pipeline. The contract should state what data is being extracted, via what method (full, incremental), how often, as well as who (person, team) are the contacts for both the source system + the ingestion. Data contracts should be stored in a well-known + easy-to-find location such as a GitHub repo or internal documentation site. If possible, format data contracts in a standardized form so they can be integrated into the development process or queried programmatically.*
* In addition, **consider establishing an SLA with upstream providers** **which provides expectations of what you can expect from the source systems you rely upon**
* An SLA example might be “data from source systems will be reliably available + of high quality”
* A **service-level objective (SLO) measures performance against what you’ve agreed to in an SLA**
* Ex: Given the example SLA, an SLO might be “source systems will have 99% uptime”
* If a data contract or SLA/SLO seems too formal, at least verbally set expectations for source system guarantees for uptime, data quality, + anything else of importance to you
* **Upstream owners of source systems need to understand your requirements so they can provide you with the data you need**

#### Undercurrents and Their Impact on Source Systems

* Unlike other parts of the DE lifecycle, **source systems are generally *outside* the control of the DE**
* **Implicit assumption** (or hope) that the **stakeholders + owners of the source systems** (+ the data
* they produce) **are following best practices concerning data management, DataOps (+ DevOps), Data Observability Driven Development (DODD), data architecture, orchestration, + SWE**
* The **DE should get as much upstream support as possible to ensure that the undercurrents are applied when data is generated in source systems**
* **Doing so will make the rest of the steps in the DE lifecycle proceed a lot more smoothly**
* *How do the undercurrents impact source systems?*

##### Security

* **Security is critical, + the last thing you want is to accidentally create a point of vulnerability in a source system**
* Some areas to consider:
* Is the **source system architected so data is secure + encrypted**, both with **data at rest + while data is transmitted?**
* Do you have to **access the source system over the public internet**, or are you **using a VPN?**
* **Keep passwords, tokens, + credentials to the source system securely locked away**
* Ex: If using Secure Shell (SSH) keys, use a key manager to protect keys
* Same rule for passwords 🡪 use a password manager or a single sign-on (SSO) provider
* **Do you trust the source system?**
* **Always be sure to trust *but verify* that the source system is legitimate**
* You don’t want to be on the receiving end of data from a malicious actor

##### Data Managements

* **Data management of source systems is challenging for DE’s**
* In **most cases, you will have only peripheral control (if any control at all) over source systems + the data they produce**
* **To the extent possible, you should understand the way data is managed in source systems since this will directly influence how you ingest, store, and transform the data**
* Some areas to consider:
* **Data governance:** Are **upstream data + systems governed in a reliable, easy-to-understand fashion?** **And who manages the data?**
* **Data quality:** How do you **ensure data quality + integrity in upstream systems?**
* **Work w/ source system teams to set expectations on data + communication**
* **Schema: Expect that upstream schemas will change**
* Where possible, collaborate w/ source system teams to be notified of looming schema changes
* **Master data management (MDM):** Is the creation of upstream records controlled by an MDM practice or system?
* **Privacy and ethics:** Do you have access to raw data, or will the data be obfuscated?
* What are the implications of the source data? How long is it retained? Does it shift locations based on retention policies?
* **Regulatory:** Based upon regulations, **are you supposed to access the data?**

##### DataOps

* **Operational excellence (DevOps, DataOps, MLOps, XOps) should extend up + down the entire stack + support the DE lifecycle**
* This is **ideal, but often not fully realized**
* Because you’re working w/ stakeholders who control both the source systems + the data they produce, you **need to ensure that you can observe + monitor the uptime + usage of the source systems + respond when incidents occur**
* Ex: When the application database you depend on for CDC exceeds its I/O capacity + needs to be rescaled, how will that affect your ability to receive data from this system?
* Will you be able to access the data, or will it be unavailable until the database is rescaled?
* How will this affect reports?
* Ex: If the SWE team is continuously deploying, a code change may cause unanticipated failures in the application itself
* How will the failure impact your ability to access the databases powering the application?
* Will the data be up-to-date?
* **Set up a clear communication chain between DE + the teams supporting the source systems**
* **Ideally**, these **stakeholder teams have incorporated DevOps** into their workflow + culture
* This will go a long way to accomplishing the **goals of DataOps (a sibling of DevOps), to address + reduce errors quickly**
* As mentioned earlier, **DE’s need to weave themselves into the DevOps practices of stakeholders, and vice versa**
* **Successful DataOps works when ALL people are on board + focus on making systems holistically work**
* A few DataOps considerations:
* **Automation**
* There’s the **automation impacting the source system**, such as **code updates + new features**
* Then there’s the **DataOps automation you’ve set up for your data workflows**
* **Does an issue in the source system’s automation impact your data workflow automation?**
* **If so, consider decoupling these systems so they can perform automation independently**
* **Observability:** How will you *know* when there’s an issue with a source system, such as an outage or a data-quality issue?
* **Set up monitoring for source system uptime** (or use the monitoring created by the team that owns the source system)
* **Set up checks to ensure that data from the source system conforms with expectations for downstream usage**
* Ex: Is the data of good quality? Is the schema conformant? Are customer records consistent? Is data hashed as stipulated by the internal policy?
* **Incident response:** What’s **the plan if something bad happens?**
* Ex: How will your data pipeline behave if a source system goes offline?
* What’s your plan to backfill the “lost” data once the source system is back online?

##### Data Architecture

* Similar to data management, **unless you’re involved in the design + maintenance of the source system architecture, you’ll have little impact on the upstream source system architecture**
* You **should also understand how the upstream architecture is designed and its strengths + weaknesses**
* **Talk often w/ the teams responsible for the source systems to understand the factors discussed in this section + ensure that their systems can meet your expectations**
* **Knowing where the architecture performs well + where it doesn’t will impact how you design your data pipeline**
* Some things to consider regarding source system architectures:
* **Reliability**: All systems suffer from entropy at some point
* Outputs will drift from what’s expected, bugs are introduced, + random glitches happen
* **Does the system produce predictable outputs?**
* **How often can we expect the system to fail?**
* **What’s the mean time to repair to get the system back to sufficient reliability?**
* **Durability**:
* Everything fails 🡪 A server might die, a cloud’s zone or region could go offline, or other issues may arise
* **Need to account for how an inevitable failure/outage will affect managed data systems**
* **How does the source system handle data loss from hardware failures or network outages?**
* What’s the **plan for handling outages for an extended period + limiting the blast radius of an outage?**
* **Availability**: ***What* guarantees that the source system is up, running, + available when it’s supposed to be?**
* **People**: Who’s in charge of the source system’s design, + **how will you know if breaking changes are made in the architecture?**
* **A DE needs to work w/ the teams who maintain the source systems + ensure that these systems are architected reliably**
* **Create an SLA w/ the source system team to set expectations about potential system failure**

##### Orchestration

* When orchestrating within your DE workflow, you’ll **primarily be concerned w/ making sure your orchestration can access the source system, which requires the correct network access, authentication, + authorization**
* Some things to think about concerning orchestration for source systems:
* **Cadence and frequency**: **Is the data available on a fixed schedule, or can you access new data whenever you want?**
* **Common frameworks**: Do the SWEs + DE’s use the same container manager, such as K8s?
* Would it make sense to integrate application + data workloads into the same K8s cluster?
* If using an orchestration framework like Airflow, does it make sense to integrate it with the upstream application team?
* **There’s no correct answer here, but you need to balance the benefits of integration with the risks of tight coupling**

##### Software Engineering

* **As the data landscape shifts to tools that simplify + automate access to source systems, you’ll likely need to write code**
* A few considerations when writing code to access a source system:
* **Networking**: Make sure your code will be *able* to access the network where the source system resides
* Also, always think about **secure networking**
* Are you accessing an HTTPS URL over the public internet, SSH, or a VPN?
* **Authentication and authorization**: Do you have the proper credentials (tokens, username/passwords) to access the source system?
* Where will you store these credentials so they don’t appear in your code or version control?
* Do you have the correct IAM roles to perform the coded tasks?
* **Access patterns**: How are you accessing the data?
* **Using an API**? Then, how are you handling REST/GraphQL requests, response data volumes, + pagination?
* If **accessing data via a database driver**, is the driver compatible w/ the database you’re accessing?
* *For either access pattern*, how are things like retries + timeouts handled?
* **Orchestration**: Does your code integrate w/ an orchestration framework, + can it be executed as an orchestrated workflow?
* **Parallelization**: How are you **managing + scaling parallel access to source systems**?
* **Deployment**: How are you handling the deployment of source code changes?

#### Conclusion

* **Source systems + their data are vital in the DE lifecycle**
* **DE’s tend to treat source systems as “someone else’s problem” 🡪 do this at your peril**
* DE’s who abuse source systems may need to look for another job when production goes down
* **Better collaboration w/ source system teams can lead to higher-quality data, more successful outcomes, + better data products**
* **Create a bidirectional flow of communications w/ your counterparts on these**
* **Teams set up processes to notify of schema + application changes that affect analytics + ML**
* **Communicate your data needs proactively to assist application teams in the DE process**
* Be aware that the **integration between DE’s + source system teams is growing**
* One example = **reverse ETL**, which has **long lived in the shadows but has recently risen into prominence**
* The **event-streaming platform** **could serve a role in event-driven architectures + analytics**
* **A source system can also be a DE system**
* **Build shared systems where it makes sense to do so**
* **Look for opportunities to build user-facing data products**
* **Talk to application teams about analytics they’d like to present to their users or places where ML could improve the user experience**
* **Make application teams stakeholders in DE, + find ways to share your successes**

#### Additional Resources

* Confluent’s “Schema Evolution and Compatibility” documentation
* <https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html#schema-evolution-and-compatibility>
* *Database Internals* by Alex Petrov (O’Reilly)
* <https://www.oreilly.com/library/view/database-internals/9781492040330/>
* *Database System Concepts* by Abraham (Avi) Silberschatz et al. (McGraw Hill)
* <https://www.mheducation.com/highered/product/database-system-concepts-silberschatz-korth/M9780078022159.html>
* “The Log: What Every Software Engineer Should Know About Real-Time Data’s Unifying Abstraction” by Jay Kreps
* <https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>
* “Modernizing Business Data Indexing” by Benjamin Douglas and Mohammad Mohtasham
* <https://engineeringblog.yelp.com/2021/06/modernizing-business-data-indexing.html>
* “NoSQL: What’s in a Name” by Eric Evans
* <https://blog.sym-link.com/posts/2009/30/nosql_whats_in_a_name/>
* “Test Data Quality at Scale with Deequ” by Dustin Lange et al.
* <https://aws.amazon.com/blogs/big-data/test-data-quality-at-scale-with-deequ/>
* “The What, Why, and When of Single-Table Design with DynamoDB” by Alex DeBrie
* <https://www.alexdebrie.com/posts/dynamodb-single-table/>