Fundamentals of Data Engineering Chapter 5

Favorite	☑
Archived	☐
Fleeting	☐
Area/Resource	
Project	
Type	
Review Date	
Image	
URL	
Created	@April 3, 2023 3:26 PM
Updated	@April 4, 2023 3:14 PM
Root Area	
Project Area	
Updated (short)	04/04/2023
Pulls	
Resource Pulls	
Project Archived	
URL Base	
Recipe Divider	🥗🥗🥗 RECIPE BOOK PROPERTIES 🥗🥗🥗
Recipe Tags	
Book Divider	📚📚📚 BOOK TRACKER PROPERTIES 📚📚📚
Author	

📚 Date Started	
📚 Date Finished	
📚 Book Status	
📚 Rating	

💡 This chapter is about data generation in source systems. As a Data Engineer your job is to take data from source systems, do something with it and make it helpful in serving downstream use cases. Before interacting with raw data you should understand where the data exists, how it is generated and it's characteristics and quirks.

Shifting nature of DE

💡 As data proliferates, the expectation is that a data engineer's role will shift heavily towards understanding the interplay between data sources and destinations. The basic plumbing task of moving data from A to B will dramatically simplify. However, it will still b crucial to understand the nature of data as it's created in source systems.

Sources of Data: How is data created?

✏️ Data is an unorganized, context-less collection of facts and figures. It can created in many way, both analog and digital.

Analog Data

✏️ Analog data creation occurs in real world interaction. Often through speech or physical movement. It is represented by a continuous signal. Analog data is transient meaning it cannot be easily replicated or reproduced as it is temporary and changes over time. Transient analog data is characterized by it’s variability, unpredictability and sensitivity to environmental factors.

Digital Data

✏️ Digital data is either created by converting analog data to digital form or is the native product of a digital system. Digital data is represented in a discrete form through 1s and 0s. It is more easily reproducible and more precisely replicated without loss of data.

🧠 A data engineer should be familiar with the source system and how it’s data is generated. You should put in the effort to read the source system’s documentation and understand its patterns and quirks. If your source system is an RDBMS, learn how it operates (writes, commits, queries, etc..) learn he ins and outs of the source system that might affect your ability to ingest from it.

Source systems: Main Ideas

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 and audio.

- Excel

- CSV
- TXT
- JSON
- XML

APIs

> 💡 Application programming interfaces are a standard way of exchanging data between systems. While an API should abstract away the complexity of dealing with source systems, they can still be complex.

Application Databases (OLTP systems)

> ✏️ An application database stores the state of an application. 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 are often referred to as transaction databases, but this does not mean that the system in question supports atomic transactions.
- OLTP systems are less suited to use cases driven by analytics at scale, due to it’s row based storage

ACID

> 💡 Support for atomic transactions is one of a critical set of database characteristics known together as ACID (atomicity, consistency, isolation, durability).

Consistency

✏️ Any database read will return the last written version of the retrieved item.

Isolation

✏️ If two updates are in flight concurrently for the same thing, the end database state will be consistent with the sequential execution of these updates in the order they were submitted. Each transaction appears to execute in isolation from other concurrent transactions.

Durability

✏️ Indicates that committed data will never be lost, even in the event of power loss.

 Designing Data-Intensive Applications

💡 ACID characteristics ensure that the database will maintain a consistent picture of the world, dramatically simplifying the app developer's task. However relaying these ACID constraints can allow for more performance and scalability.

🧠 Engineers must understand operating with and without ACID. Some distributed databases will use relaxed consistency constrains, such as eventual consistency to improve performance. Understanding the consistency model you're working with helps you prevent disasters.

Atomic transactions

✏️ An atomic transaction is a set of changes that are committed as a unit. It is a way of ensuring consistency and integrity of data within a database. It means that if there are several database operations to be made as part of a single transaction then the transaction will only be successful if each of those operations were successful. Or the entire tx fails.

OLTP and analytics

💡 Running analytical queries on a OLTP system is not scalable. The row based storage of OLTP systems makes queries extremely computationally expensive, when trying to aggregate data across thousands of rows. DE must understand the inner workings of OLTP systems and application backends to set up appropriate integrations with analytics systems without degrading production application performance. Companies are moving towards hybrid capabilities of quick updates with combined analytics capabilities.

✏️ Data application are apps that hybridize transactional and analytics workloads.

Online Analytical Processing System

✏️ An OLAP system is built to run large analytical queries and is typically inefficient at handling lockups of individual records. OLAP refers to any database system that supports high-scale interactive analytics queries. The online part implies that the system is constantly listening for incoming queries, making them suitable for interactive analytics.

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 replicated between databases in near real time or create an event stream for downstream processing.

- Relational databases often generate an event log stored on the database server than can be processed to create a stream
- NoSQL databases can send a log or event stream to a target storage location

Logs

💡 A log captures information about events that occur in systems. All logs track events and event metadata. They should capture, who, what and where.

Log encoding

- Binary-encoded logs
- Semi-structured logs
- Plain text

Log resolution

✏️ The log resolution refers to the amount of event data captured in a log.

The log level refers to the conditions required to record a log entry, specifically concerning errors and debugging.

Log latency refers to batch or real time.

Database logs

💡 Database logs ensure integrity and consistency of databases. Write ahead logs which are binary files enables recoverability. The database server receives writes and update requests to a database table storing each operation in the log before acknowledging the request. The acknowledgement comes with a log-associated guarantee: even if the server fails it can recover it’s state on reboot by completing he unfinished work from the logs.

CRUD

💡 CRUD standing for Create, Read, Update and Delete is transactional pattern used in programming and represents the four basic operations of persistent storage. Each of the four transactions are executed as a separate transaction. CRUD represents the basic functions that are necessary to manage data in any type of storage medium.

Insert-Only

💡 The insert only pattern retains history directly in a table containing data. Rather than updating records, new records get inserted with a timestamp indicating when they were created. A current state is made by looking at the most recent records based on ID.

 Designing Data-Intensive Applications

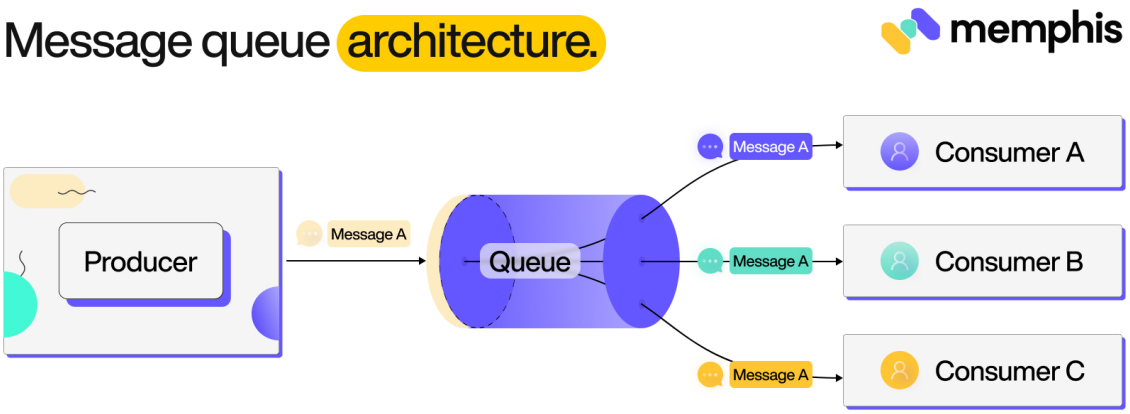
Messages and Streams

- [ ] More research int event-driven architecture

💡 In relation to event-driven architecture you will likely face two terms: message queue and streaming platform, which are often used interchangeably however there lies an essential difference.

Message

✏️ A message is raw data communicated across two or more systems. 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. Messages are not continuous streams of information but rather discrete pieces of information.



Stream

✏️ A stream is an append-only log of event records. As events occur, they are accumulated in an ordered sequence; a timestamp or an ID might order events. (Events may note always be delivered in exact order due to the subtleties of distributed systems).

💡 Streams are to be used when you care about what happened over many events. Due to the append only nature of streams, records in a stream are persisted over a long retention window. Allowing for complex operations on records such as aggregations on multiple records or the ability to rewind to a point in time within the stream.

Types of Time

💡 When we view data as continuous and expect to consume it shortly after produced time becomes an essential consideration for all data ingestion. The key types of time are: the time the event is generated, when it's ingested and processed, and how long processing took.

- Event time: When an event is generated in a source system.
- Ingestion time: When an event is ingested from source into a message queue, cache, memory, object storage, a database or any storage location.
- Process time: When the data is processed (a transformation).

💡 Recording these various times, in an automated way will enable more effective monitoring along your data workflows allowing you to identify bottlenecks etc…

Source System Practical Details

Databases

💡 This is a list of major ideas that occur across a variety of database technologies. It is important to understand these as database technology will evolve but the main ideas will likely stay.

Database management system

💡 A database system used to store and serve data. Known as DMBS, consists of a storage engine, query optimizer, disaster recovery and other key components for manaing the database system.

Lookups

💡 Lookups enable the database to find and retrieve data. Indexes speed up lookups, but this is not available in all databases. You should know what type of indexes your database uses, and what are the best patterns for designing and managing them. How do you efficiently extract information? You should also know the basic type of indexes such as B-trees and log-structured merge-trees (LSM).

Chapter 3 - Design scalable systems

Query optimizer

✏️ A database optimizer is a database management system that is responsible for analyzing queries and selecting the most efficient query execution plan to optimize performance.

💡 You should know if a database uses an optimizer, and what it’s characterisitcs are

Scaling and distribution

💡 You should know what scaling strategy the database deploys, if it scales with demand, and if it scales vertically or horizontally.

Modeling patterns

💡 It is important to understand which modeling patterns work best with the database.

CRUD

💡 CRUD operations are handled differently by every database, you should assess how they are handled in a given database.

Consistency

💡 It is important to understand if the database is fully consistent, or if it supports a relaxed consistency model (e.g. eventual consistency). Does the database support optional consistency modes for reads and writes (e.g. strongly consistent reads).

✏️ **Consistency** relates to the accuracy and correctness of data in a database. A fully consistent database requires expensive compute in terms of performance and resources. Especially in large distributed systems. Therefore, this requirement is relaxed in some solutions such as the **eventual consistency** model which allows for some time lag between updates, meaning changes may not be immediately reflected in all queries. **A consistency mode** relates to the option the database provides to change the consistency constraints for different operations.

Relational databases

💡 Data in a relational database is stored in a table of relations (rows), and each relation contains multiple fields (columns). Each relation in the table has the same schema. Rows are typically stored as a contiguous sequence of bytes on disk.

✏️ Schema: A sequence of columns with assigned static types such as string, integer, or float).

✏️ Contiguous: In the context above contiguous means that the rows are stored one after the other without any space in between.

💡 RDBMS are typically ACID compliant. Combining a normalized schema, ACID compliance and support for high transaction rates makes relational database systems ideal for storing rapidly changing application states. The data consistency gained from ACID compliance, and normalized schema ensures that the changing states will be accurately captured.

- [ ] study RDBMS systems (database course in your comp sci lessons)
- [ ] Relational algebra
- [ ] Strategies for algebra

Nonrelation databases: NoSQL

💡 As data and query requirements morph, the relational database collpases under its weight. You'll want to use a database that's appropriate for the specific workflow under pressure. NoSQL which stands for not only SQL is a class of databases that abondon the relational paradigm.

Tradeoffs

💡 On one hand dropping relational constraints improve performance, scalability and schema flexibility. On the other NoSQL solutions often abondon strong consistency, joins or a fixed schema.

🧠 DE should understand these types of databases, including usage considerations, structure of the data they store and how to leverage each in the DE lifecycle.

Different types of NoSQL:

- key-value
- document
- wide-column
- graph
- search
- time-series

Key-value stores

✏️ A key-value database is a nonrelational database that retrieves records using a key that uniquely identifies each record. It is similar to hash maps or dictionary data structures, but more scalable.

💡 Different types of key-value stores offer advantages. In memory databases are popular for caching session data for web and mobile apps. Storage is temporary but can reduce pressure on the main application database and serve speedy responses. Key-value stores can also serve applications requiring high-durability persistence.

Document stores

✏️ A document store is a specialized key-value store. A document in this context is a nested object, which can be thought of as a JSON object for practical purposes. Documents are stored in collections and retrived by a key.

💡 A key difference between RDBMS and document stores is that the latter does not support joins, which eliminates the ability to normalize data. These leads to the same data being duplicated across many collections. SE must be careful to update the stores in every location they exist.

💡 Document stores don’t enforce schema or types. While this could be a strength it is important to manage schema evolution, or data will become inconsistent and bloated over time. Schema evolution can also break downstream ingestion.

Indexes in document stores

💡 Document stores allow for indexes that enable look up based on specific properties.

Wide-colum

✏️ A wide-column database is optimized for storing massive amounts of data with high transaction rates and extemely low latency. These databases can scale to extremely high reads and vast amounts of data. DE should understand the operational characteristics of the wide-column database and set up suitable configurations, design the schema and choose an appropriate row key to optimize performance.

💡 These databases support rapid scans of data but do not support complex queries. They only have a single index.

☐ Learn more about this

Graph databases

✏️ Graph databases store data with a mathematical graph structure (as a set of nodes and edges). Graph databases are a good fit when you want to analyze the connectivity between elements. 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.

🧠 There is an anticipation that graph databases will grow dramatically outside of tech companies. ML and Data science applications are also interesting.

Search

✏️ A search database is a nonrelational database used to search your data's complex and straight forward semantic and structural characterisitcs. Text search and log analysis are prominent use cases.

✏️ Text search involves search 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 and operational analytics.

💡 Search databases are popular for fast search and retrieval. An ecommerce site may power it’s product search using a search database.

Time series

✏️ A time series database is optimized for retrieving and statistical processing of time-series data.

💡 As data grew faster and bigger, new special-purpose databases were needed. Time series databses adress the needs of growing, high-velocity data volumes from IoT, event and application logs, ad tech and fintench. Often these workflows are write-heavy. As a result, they often utilize memory buffering to support fast writes and reads.

Measurement vs event-based data

💡 **Measurement data** is generated regularly, such as temperature or air-quality sensors. **Event-based data** is irregular and created every time an event occurs, such as a mtion sensor detecting movement.

APIs

💡 There are many different types of APIs, but we are mostly interested in those built around HTTP.

REST

✏️ REST stands for Representational State Transfer. 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 with associated state variables such as a working directory; each REST call is independent. A REST call can change the systems stage; but these changes are global, applying to the full system rather than a current session.

💡 Stateless means that each request to a server contains all the information it needs, and that the receiving server does not need to store any information about the client. Meaning each REST call is independent and self maintained, which enables better reliability, scalability and performance of web services.

🧠 DE may need to write custom code to interact with REST APIs when a built connector is not available. Meaning DE will require an understanding of the structure of the data as provided, developing appropriate data extraction code and determining a suitable data synchronization strategy.

GraphQL

💡 GraphQL enables a query to to retrive multiple data models in a single request as opposed to REST APIs which restricts it to one. This allows more flexible and expressive queries than with REST. It is built around JSON and returns data in a shape similar to JSON.

Webhooks

✏️ Webhooks are a event-based data transmission pattern. The data source can be an application backend, web page or app. When specified events happen in the source system, this triggers a call to an HTTP endpoint hosted by the data consumer. Webhooks are called reverse APIs but cause the connection goes from the source system to the data sink.

💡 The event based transmission pattern reduces overhead, as it allows real time communication based on events as opposed to monitoring the state and sending requests based on that.

RPC and gRPC

✏️ A remote procedure call (RPC) is commonly used in distributed computing. It allows you to run a procedure on a remotre system.

💡 The ability to trigger procedures and functions as if they were built directly on a machine abstracts away network complexity and enables more efficitive writting of code. It will also enable microservices to be developed independently, and reduce network traffic.

💡 gRPC is built around the Protcol Buffers open data serialization standard. This is a binary encoding format that offers a compact way to store data. This enables lower resource network usage in a distributed system.

💡 gRPC emphasizes the efficient bidirectional exchange of data over HTTP/2. Efficiency refers to aspects such as CPU utilization, power consumption, battery life and bandwdth. Like GraphQL, gRPC imposes much more specific technical standards than REST, thus allowing the use of common client libraries and allowing engineers to develop a skill set that will apply to any gRPC interaction code. The increase in technical standards allows for the development of code that is operable in different environments.

Data sharing

💡 The core concept of cloud data sharing is that a multitenant system supports policies for sharing data among tenants. Concretely, any public cloud object storage system with a fine-grained permission system can be a platform for data sharing. Many modern sharing platforms support row, column and sensitive data filtering.

Data marketplace

✏️ A data marketplace is a centralized platform that faciliates the buying and selling of data.

Data decentralization

💡 Data sharing streamlines data pipelines within an organization. Data sharing allows units of an organization to manage their data and selectively share it with other units while still allowing individual units to manage their compute and query costs separately, facilitating data decentralization.

Message Queues and Event streaming platforms

🧠 Event-driven architecture are pervasive in software applications and are poised to grow their popularity even further.

1. Message queues and event-streaming platforms are easier to set up and manage in a cloud environment.
2. 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.

💡 Streaming data cuts across many data engineering lifecycle stages. These systems are used as source systems, but they will often cut across the lifecycle stages due to their transient nature. An event streaming platform for message passing can act as a source system but can also be used in the ingestion and transformation state.

Message queues

✏️ A message queue is a mechanism to asynchronously send data (usually as small individual messages, in the kb) between discrete systems using a publish and subscribe model. Data is published to a message queue and is delivered to one or more subscribers. The subscriber acknowledges receipt of the message, removing it from the queue.

💡 Message queues allow applications and systems to be decoupled from each other and are widely used in microservice architecture. Some things to keep in mind with them are frequency of delivery, message ordering and scalability.

Message ordering and delivery

💡 The order in which messages are created, sent and received can significantly impact downstream subscribers. This is a tricky problem and we usually 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 alwaysbe delivered before message B. In practice messages are published and received out of order, especially in highly distributed systems. You need to design for out-of-order message delivery.

Delivery frequency

💡 Messages can be sent exactly once or at least once. If a message is sent exactly once, the the subscriber acknowledges it and it disappears. 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.

Idempotent systems

✏️ A idempotent system can take the same request or be executed multiple times without changing the result after the first execution.

💡 Ideally systems should be idempotent. If a system fails after receiving and executing based on a message before acknowledging it, then another message can be sent and handled gracefully.

Scalability

💡 The best message queues are horizontally scalable. Allowing for dynamic scaling, buffer messages when systems fall behind and durably store messages for resilience against failure.

✏️ Buffering messages refers to temporarily storing messages in a queue when the system falls behind in it’s processing of messages.

Event streaming platforms

💡 In an event streaming platform is used to ingest and process data in an ordered log of records. Data is retained for a while, and it is possible to replay messages from a past point in time.

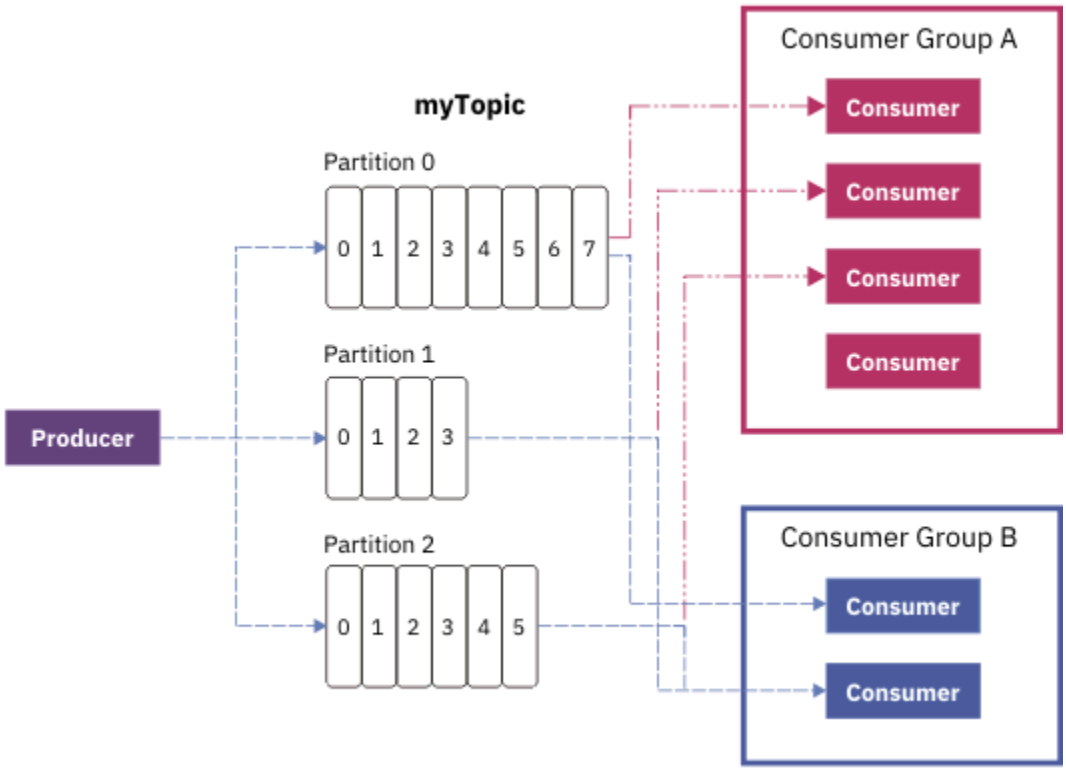
Topics

💡 A producer streams events to a topic, a collection of related events. A topic can have zero, one, or multiple producers and customers on most event-streaming platforms.

Stream partitions

✏️ Stream partitions are subdivisions of a stream into multiple streams. A good analogy is a multilane freeway. Having multiple lanes allows for parallelism and higher throughput. Messages are distributed across partitions by **partition key.** Messages with the same partition key always end up in the same partition.

⚠️ A concern with stream partitioning is ensuring that your partition key does not generate hotspotting. Which is a disproportionate number of messages delivered to one partition.



Fault tolerance and resilience

Event streaming platforms are typically distributed systems, with streams stored on various nodes. If a node goes doewn, another node replaces it, and the stream is still accessible. This makes platforms good solutions for reliability producing, storing and ingesting event data.

Whom you'll work with

When working with source systems, it's essential to understand the people with whom you'll work. Good diplomacy and relationships with stakeholders are crucial to data engineering. You will typically deal with two categories of stakeholders: systems and data stakeholders.

Stakeholders

Systems stakeholders

✏️ A system stakeholder builds and mantains the source systems. These can be SE, app devs and third parties.

Data stakeholders

✏️ Data stakeholders own and control access to the data you want.

💡 You are often at the mercy of stakeholders ability to follow correct software engineering, database management and dev practices. If an upstream system goes down you want to make sure you are aware of the impact these issues will have on your DE systems.

Data contracts

✏️ A data contract is a written agreement between the owner of a source system and 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 contracts for both the source system and the ingestion. They should be stored in easy to find location such as GitHub, or an internal documentation site.

SLA

💡 You should also consider Service level agreements with data providers. It will provide you with the expectations for the the source system you rely on.

Undercurrents and their impact on source systems

🧠 A data engineer should get as much upstream support as possible to ensure that the undercurrents are applied when data is generated in source systems.

Security

💡 You want to avoid accidentally creating a point of vulnerability in a source system.

- Is the source system architected so data is secure and encrypted?
- How do you access the data, web or VPN

Data management

💡 You should understand the way data is managed in source systems since this will influence how you ingest, store and transform the data.

- Data governance: Who manages the data?
- Data quality: How is data quality ensured?
- Schema: Expect a change in schema. Where possible, collaborate with source system teams to be notified of looming schema changes.

DataOps

💡 Operational excellence should extend up and down the entire stack. You need to ensure that you can observe and monitor the uptime and usage of the source systems and respond when incidents occur. Set up clear communication chain between data engineering and the teams supporting the source system. **Successful DataOps works when all people are on board and focus on making system holistically work.**

- Automation
- Observability
- Incident response: How will your pipeline behave if your source system goes offline?

Data Architecture

> 💡 You should understand how the upstream architecture is designed and it’s strengths and weaknesses.

- Reliability
- Durability
- Availability
- People

Data Orchestration

> 💡 You will be concerned with making sure your orchestration can access the source system, which requires the correct network access, authentication and authorization.

- Cadence and frequency
- Common frameworks

Software engineering

> 💡 Considerations when you are writing code to access a source system.

- Networking: Make sure your code can access the network
- Authentication and authorization

- Access patterns
- Orchestration: Does your code integrate with an orchestration framework and can it be executed in an orchestrated workflow?
- Parallelization
- Deployment

Conclusion

Better collaboration with source systems teams can lead to higher-quality data, more successful outcomes and better data products. Create bidirectional flow of communications with your counterparts on these teams; set up processes to notify of schema and application changes that affect analytics and ML. Communicate your data needs proactively to assist application teams in the data engineering process.

Appendix

Type of database	Real world solution	GCP equivalent
RDBMS	MySQL	CloudSQL
Key-value store	Redis	Memory store
Document store	MongoDB	Firestore
Wide column	Cassandra	BigTable
Graph database	Neo4J	Datastore
Search database	Elasticsearch	Cloud Search
Time series database	InfluxDB	Bigtable or Spanner
Message Queue	Kafka	Pub/Sub
Event streaming platform	Kafka Streams	Pub/Sub or Dataflow

- [ ] Know all of these and use cases for exam