Vinay Ram Gazula

Interview Preperation Cheatsheet

August 10, 2025

Contents

Part I
Data Engineering

References:

1. GeeksForGeeks Data Engineer Interview Questions
2. GeeksForGeeks Python Interview Questions

Contents

Chapter 1
DE Basics and Problem Solving

1.1 What is data engineering?

Data engineering focuses on designing, building, and maintaining systems for collecting, storing, and analyzing large data volumes. This includes creating data pipelines, optimizing storage, and ensuring data quality and accessibility for data scientists and analysts.

1.2 What are the main responsibilities of a data engineer?

A data engineer's core responsibilities involve:

- Data Pipeline Management: Designing and implementing robust data pipelines.
- Data Warehousing: Creating and maintaining efficient data warehouses.
- Quality Assurance: Ensuring the quality and consistency of data.
- System Optimization: Optimizing data storage and retrieval systems for performance.
- Cross-functional Collaboration: Collaborating with data scientists and analysts to fulfill their data requirements.
- Security and Governance: Implementing comprehensive data security and governance measures.

1.3 What is a data pipeline?

A data pipeline facilitates the seamless flow of data from its source to its destination, where it can be utilized for analysis or other functions. This involves a sequence of processes that extract, transform, and load data, ensuring its smooth movement and readiness for use.

1.4 What are some common challenges in data engineering?

Data engineering frequently encounters challenges such as:

- Efficiently processing large data volumes: Managing and processing massive datasets effectively is a core hurdle.
- Ensuring data quality and consistency: Maintaining accuracy, reliability, and uniformity across all data is crucial.
- Managing real-time data: Handling and processing data streams as they arrive presents significant complexity.
- Scaling systems: Adapting infrastructure to accommodate increasing data demands is a continuous challenge.
- Integrating diverse data: Combining information from various sources and formats requires robust solutions.
- Maintaining data security and privacy: Protecting sensitive data and adhering to privacy regulations is paramount.

1.5 How do you approach learning new technologies in the rapidly evolving field of data engineering?

Here are several ways to stay current with the latest technological advancements:

- Read tech blogs and articles regularly.
- Enroll in online courses and pursue certifications.
- Attend industry conferences and workshops.
- Experiment with new tools through personal projects.
- Collaborate with colleagues and engage in knowledge sharing.
- Follow industry experts on social media.

1.6 How do you ensure data quality in your projects?

To ensure data quality, consider these strategies:

- Implement Robust Validation: Perform data validation checks at the point of ingestion.
- Understand Data Characteristics: Utilize data profiling tools to gain insights into data characteristics.
- Monitor Quality Metrics: Establish and consistently monitor clear data quality metrics.
- Cleanse Data: Implement processes for data cleansing.
- Conduct Regular Audits: Perform regular data audits.
- Establish Governance: Set up a comprehensive data governance framework.

1.7 How do you handle conflicts in a team environment?

Effective conflict resolution involves:

- Understanding diverse perspectives: Actively listen to grasp all viewpoints.
- Problem-focused approach: Concentrate on the issue at hand, avoiding personal attacks.
- Identifying commonalities: Seek shared goals and areas of agreement.
- Collaborative solutions: Propose and discuss potential remedies.
- Management involvement (when necessary): Escalate to leadership with suggested resolutions if internal efforts fail.

1.8 How do you prioritize tasks in a data engineering project?

Effective prioritization involves:

- Understanding Task Impact: Evaluating the business impact and urgency of each task.
- Managing Dependencies: Considering the relationships and dependencies between different tasks.
- Optimizing Resources: Assessing available resources and any constraints.
- Utilizing Prioritization Frameworks: Employing methods such as the Eisenhower Matrix or MoSCoW method.
- Stakeholder Alignment: Maintaining consistent communication with stakeholders to ensure aligned priorities.

1.9 How do you stay updated with the latest trends and best practices in data engineering?

To stay current in your field, consider these methods:

- Engage with online communities (e.g., Stack Overflow, Reddit)
- Follow industry blogs, podcasts, and YouTube channels
- Attend webinars and virtual conferences
- Subscribe to industry newsletters
- Network with other professionals
- Experiment with new tools and technologies in personal projects

1.10 How would you design a system to handle real-time streaming data?

When designing a system for real-time streaming data, the following aspects are crucial:

- Distributed Streaming Platform: Utilize platforms such as Apache Kafka or Amazon Kinesis.
- Stream Processing: Implement processing with tools like Apache Flink or Spark Streaming.
- Low-Latency: Ensure both data ingestion and processing operate with minimal delay.
- Fault Tolerance & Scalability: Design the system to be resilient to failures and capable of handling increased loads.
- Error Handling & Data Validation: Implement robust mechanisms for managing errors and validating data integrity.
- Data Storage: Plan for the storage of both raw and processed data.

1.11 What strategies do you use for optimizing query performance in large datasets?

To optimize query performance, consider these strategies:

- Indexing: Properly index columns that are frequently queried.
- Partitioning: Divide large tables into smaller, more manageable partitions.
- Materialized Views: Employ materialized views for complex queries that are run often.
- Query Optimization: Focus on optimizing and rewriting queries for efficiency.
- Caching: Implement caching mechanisms to store frequently accessed data.
- Columnar Storage: Utilize columnar storage formats, especially for analytical workloads.
- Distributed Computing: Leverage distributed computing for processing large datasets.

1.12 How do you approach data pipeline testing?

Here are several key approaches to testing data pipelines:

- Unit Testing: Focuses on verifying individual components in isolation.
- Integration Testing: Ensures that different components of the pipeline function correctly when combined.

- End-to-End Testing: Validates the entire pipeline from beginning to end.
- Data Validation Testing: Confirms the accuracy, completeness, and consistency of the data.
- Performance Testing: Evaluates the pipeline's behavior and responsiveness under various data loads.
- Fault Injection Testing: Deliberately introduces errors to check the pipeline's error handling and recovery mechanisms.
- Regression Testing: Conducted after modifications to ensure that existing functionalities remain intact and no new issues have been introduced.

1.13 What is your experience with data versioning and how do you implement it?

Data versioning is the process of tracking alterations to datasets over time. Key implementation strategies include:

- Version Control Systems: Utilizing these systems for managing code and configuration files.
- Slowly Changing Dimensions (SCDs): Implementing SCDs within data warehouses.
- Data Lake Technologies: Employing data lake solutions that inherently support versioning, such as Delta Lake.
- Metadata Management: Maintaining comprehensive metadata for different dataset versions.
- Backup and Restore: Establishing a strong backup and restore strategy.

1.14 How do you handle data skew in distributed processing systems?

Effectively addressing data skew involves several strategies:

- Identification and Analysis: Pinpointing and understanding the skewed keys.
- Even Distribution: Employing salting or hashing to spread data more uniformly.
- Optimized Joins: Utilizing broadcast joins for smaller datasets.
- Partition Management: Modifying partition sizes or using custom partitioners.
- Aggregations: Implementing two-phase aggregation for skewed aggregations.
- Data Model Considerations: Exploring alternative data models or schema designs.

1.15 Explain the concept of data lineage and why it's important?

Data lineage tracks the entire lifecycle of data, detailing its origins, movements, transformations, and effects. Its significance lies in several key areas:

- Understanding Data: It helps in comprehending data provenance and assessing data quality.
- Impact Analysis: It facilitates analyzing the potential impacts of proposed changes.
- Compliance and Auditing: It assists with regulatory compliance and auditing processes.
- Troubleshooting: It supports the troubleshooting and debugging of data-related issues.
- Data Governance: It enhances overall data governance and metadata management.

1.16 How do you approach capacity planning for data infrastructure?

Capacity planning encompasses several key aspects:

- Resource Analysis and Forecasting: This includes analyzing current resource utilization, identifying growth trends, and forecasting future data volumes and processing requirements.
- Scalability and Performance: Evaluating different scaling options (vertical vs. horizontal) and considering peak load scenarios and seasonality are crucial for ensuring system performance.
- Infrastructure and Cost: Assessing costs and budget constraints, along with considering cloud versus on-premises infrastructure options, are vital for efficient planning.
- Resilience: Planning for redundancy and fault tolerance is essential to ensure system reliability and availability.

1.17 What is your experience with data catalogs and metadata management?

Data catalogs and metadata management encompass several key areas:

- Documentation: Implementing tools to document datasets, their schemas, and relationships.
- Processes: Establishing clear processes for metadata creation and ongoing maintenance.
- Integration: Ensuring seamless integration of metadata across various systems and tools.
- Discovery: Implementing robust data discovery and search capabilities.
- Governance: Supporting data governance and compliance initiatives.
- Self-Service: Facilitating self-service analytics for business users.

1.18 How do you handle schema evolution in data pipelines?

Managing schema evolution involves several key approaches:

- Utilizing schema-on-read formats: Employing formats such as Parquet or Avro, where the schema is applied at the time of data retrieval.
- Designing for compatibility: Ensuring both backward and forward compatibility in schema designs.
- Schema versioning: Implementing a system for versioning schemas and maintaining compatibility across different versions.
- Centralized schema management: Leveraging schema registries for a centralized approach to schema management.
- Strategic data migration: Developing and executing data migration strategies for significant schema alterations.
- Rigorous testing: Thoroughly testing all schema changes before their deployment.

1.19 What is your approach to monitoring and alerting in data engineering systems?

Effective monitoring and alerting requires a multi-faceted approach:

- Comprehensive Logging: Implement robust logging across all system components.
- Real-time Visibility: Set up real-time monitoring dashboards for immediate insights.
- Performance Metrics: Define key performance indicators (KPIs) and service level objectives (SLOs).
- Proactive Alerting: Establish proactive alerts for potential issues.
- Anomaly Detection: Utilize anomaly detection techniques to identify unusual patterns.
- Incident Management: Develop a clear incident response process.
- Regular Audits: Conduct regular system health checks and audits.

1.20 How do you ensure data consistency in distributed systems?

Data Consistency Strategies:

- Consistency Models: Employ strong consistency where crucial, and eventual consistency for performance gains in specific situations.
- Distributed Transactions: Utilize distributed transactions as required, leveraging methods like two-phase commit or the saga pattern for intricate operations.
- Idempotency: Design idempotent operations to effectively manage duplicate requests.
- Conflict Resolution: Implement robust conflict resolution mechanisms, especially in multi-master system architectures.

1.21 What is your experience with data modeling for NoSQL databases?

Data modeling for NoSQL databases requires a distinct approach, focusing on several key aspects:

- NoSQL Database Type: A thorough understanding of the specific NoSQL database type (e.g., document, key-value, column-family, graph) is fundamental.
- Query-Driven Design: Design efforts should prioritize anticipated query patterns over traditional normalized data structures.
- Performance Optimization: Strategic denormalization and data duplication are crucial considerations for enhancing performance.
- Scalability and Partitioning: Effective planning for scalability and data partitioning is essential for growth.
- Indexing Strategies: Implementing appropriate indexing strategies is vital for efficient data retrieval.
- Schema Management: Addressing schema flexibility and managing its evolution are important for adaptability.

1.22 How do you approach data quality assurance in ETL processes?

Effective data quality assurance in ETL processes encompasses several key practices:

- Data Validation: Implementing robust data validation rules at both the source and target systems.
- Data Profiling: Conducting thorough data profiling to gain insights into data characteristics.
- Data Cleansing & Standardization: Establishing processes for data cleansing and standardization to ensure consistency and accuracy.
- Performance Monitoring: Utilizing data quality scorecards to monitor and track improvements over time.
- Reconciliation Checks: Performing data reconciliation checks between source and target to verify data integrity.
- Issue Resolution: Defining a clear process for managing and resolving any identified data quality issues.

1.23 What strategies do you use for managing technical debt in data engineering projects?

Managing technical debt effectively involves a multi-faceted approach, encompassing:

- Proactive Code Management: This includes conducting regular code reviews and refactoring sessions to maintain code quality and implementing automated testing to promptly identify and address regressions.
- Streamlined Development Practices: Embracing CI/CD (Continuous Integration/Continuous Deployment) practices ensures consistent and reliable deployments.
- Strategic Planning and Prioritization: Dedicating time for system improvements during project planning, prioritizing critical updates and migrations, and conducting periodic architecture reviews are crucial for long-term health.
- Comprehensive Documentation: Maintaining thorough documentation is essential for understanding and managing existing systems.

1.24 How do you handle data privacy and compliance requirements in your projects?

Approaches to handling data privacy and compliance involve:

- Data Governance:
  - Implementing data classification and tagging.
  - Implementing data retention and deletion policies.
- Security Measures:
  - Applying appropriate data masking and encryption techniques.
  - Implementing role-based access control (RBAC).
- Monitoring and Auditing:
  - Maintaining audit logs for data access and modifications.
- Risk Management & Compliance:
  - Conducting regular privacy impact assessments.
  - Staying updated with relevant regulations (e.g., GDPR, CCPA).

Chapter 2
Database Systems and SQL

2.1 What is a relational database?

A database that organizes data into tables with predefined relationships is known as a relational database. SQL (Structured Query Language) is used for managing and querying the data within these databases.

2.2 What are the main differences between SQL and NoSQL databases?

Here are the key distinctions:

- Structure: SQL databases adhere to a predefined schema, whereas NoSQL databases offer a flexible or schema-less approach.
- Scalability: NoSQL databases typically excel at horizontal scaling, while SQL databases commonly scale vertically.
- Data Model: SQL databases organize data into tables and rows. In contrast, NoSQL databases employ diverse models such as document, key-value, or graph.
- ACID Compliance: SQL databases generally uphold ACID properties. NoSQL databases, however, may compromise some ACID guarantees to optimize performance and scalability.

2.3 What is normalization in database design?

Normalization is a systematic approach to database design that aims to optimize data storage, minimize data duplication, and enhance data consistency and reliability. At its core, it's about structuring tables and their columns in a way that eliminates redundant data and ensures that data dependencies are logical. This process involves a series of guidelines, or "normal forms," which dictate how a database should be structured to achieve these goals.
The primary objectives of normalization include:

- Reducing Data Redundancy: By breaking down large tables into smaller, more specific tables and establishing relationships between them, normalization ensures that each piece of data is stored only once. This eliminates the need to update the same information in multiple places, which can lead to inconsistencies.
- Improving Data Integrity: Data integrity refers to the accuracy and consistency of data over its entire lifecycle. Normalization helps maintain data integrity by ensuring that changes to data are applied consistently and that the database adheres to defined rules and constraints. For example, if a customer's address changes, it only needs to be updated in one place (the customer table), rather than in every table that might reference that customer.
- Enhancing Data Maintainability and Flexibility: A normalized database is generally easier to maintain and modify. When data is organized logically, it becomes simpler to add new features, update existing information, or delete outdated records without affecting other parts of the database. This also makes the database more flexible and adaptable to future changes in business requirements.
- Optimizing Query Performance (in some cases): While excessive normalization can sometimes lead to more complex queries due to the need for multiple table joins, a well-normalized database can improve query performance by reducing the amount of data that needs to be scanned and improving the efficiency of data retrieval.

The process of normalization typically involves progressing through various normal forms, with each form addressing specific types of data anomalies:

- First Normal Form (1NF): This is the most basic form and requires that all columns contain atomic (indivisible) values and that there are no repeating groups of columns. Each column should hold a single value, and each row should be unique.
- Second Normal Form (2NF): To be in 2NF, a table must first be in 1NF, and all non-key attributes must be fully functionally dependent on the primary key. This means that no non-key attribute should depend on only a part of a composite primary key.
- Third Normal Form (3NF): Building on 2NF, 3NF requires that all non-key attributes are directly dependent on the primary key and not on other non-key attributes (i.e., there are no transitive dependencies).
- Boyce-Codd Normal Form (BCNF): BCNF is a stricter version of 3NF. It requires that every determinant (an attribute or set of attributes that determines the value of another attribute) is a candidate key.
- Fourth Normal Form (4NF) and Fifth Normal Form (5NF): These higher normal forms address more complex data dependencies, such as multi-valued dependencies and join dependencies, further reducing redundancy and improving data integrity in specific scenarios.

While normalization offers significant benefits, it's important to note that sometimes de-normalization (intentionally introducing redundancy) might be performed for performance reasons in specific applications, especially in data warehousing or analytical systems where read performance is prioritized over write performance. However, for most transactional databases, a well-normalized design is crucial for data consistency and long-term maintainability.

2.4 Explain the concept of database indexing.

Database indexing is a fundamental technique employed in database management systems to drastically enhance the efficiency of data retrieval operations. In essence, it involves the creation of a specialized data structure, often a B-tree or hash index, that acts as a lookup table for the data within a database table. This structure allows the database system to swiftly pinpoint the physical location of specific rows based on the values in one or more designated columns, referred to as “indexed columns”.

Without an index, when a query requests data based on certain criteria, the database typically has to perform a full table scan. This means it meticulously examines every single row in the table, one by one, to see if it matches the query’s conditions. For small tables, this might be negligible, but for large tables containing millions or billions of rows, a full table scan can be an incredibly time-consuming and resource-intensive process, leading to significant performance bottlenecks.

An index, however, functions much like the index in a book. Instead of flipping through every page to find a particular topic, you consult the index, which provides the page number where that topic can be found. Similarly, when a query targets an indexed column, the database first consults the index. The index then quickly directs the database to the exact disk location of the relevant data rows, bypassing the need to read through the entire table. This dramatically reduces the number of disk I/O operations required, which is often the slowest component of data retrieval, thereby accelerating query execution times.

While indexes primarily boost read performance, they do come with trade-offs. Each index consumes additional disk space, and more importantly, every time data in an indexed column is inserted, updated, or deleted, the corresponding index must also be updated. This overhead can slightly slow down write operations. Therefore, the decision to index a column involves a careful balance between improving read performance and potentially impacting write performance. Effective indexing strategies involve identifying frequently queried columns, especially those used in ‘WHERE‘ clauses, ‘JOIN‘ conditions, and ‘ORDER BY‘ clauses, to maximize their benefits.

2.5 What is a stored procedure?

Stored procedures are precompiled sets of SQL statements saved in a database, allowing for execution with a single call. They enhance performance and code reusability by accepting parameters, performing complex operations, and returning results.

Chapter 3
Apache Hadoop, MapReduce and Hive

3.1 What is Hadoop?

Hadoop is an open-source framework that has revolutionized the way large datasets are handled. At its core, Hadoop provides a robust and scalable solution for both distributed storage and processing of massive amounts of data across clusters of commodity hardware. This distributed nature is crucial for handling Big Data, as traditional single-machine approaches quickly become bottlenecks.
The framework is fundamentally comprised of two principal components:

- Hadoop Distributed File System (HDFS): This is Hadoop's storage layer. HDFS is designed to store very large files reliably across machines in a large cluster. It achieves reliability by replicating data blocks across multiple nodes, ensuring data availability even if some nodes fail. HDFS is optimized for high-throughput access to data and is well-suited for batch processing, rather than interactive access. Its architecture includes a single NameNode, which manages the file system metadata and namespace, and multiple DataNodes, which store the actual data blocks.
- MapReduce: This is Hadoop's processing engine. MapReduce is a programming model and an associated execution framework for processing large datasets with a parallel, distributed algorithm on a cluster. It works by dividing a complex data processing task into smaller, independent sub-tasks. The "Map" phase processes input data records in parallel and generates intermediate key-value pairs. The "Reduce" phase then aggregates and combines these intermediate results to produce the final output. MapReduce handles job scheduling, task execution, and fault tolerance, making it easier for developers to write distributed applications without worrying about the underlying complexities of parallel processing.

Beyond these two core components, the Hadoop ecosystem has grown significantly to include various other tools and technologies that complement its functionalities, such as Yet Another Resource Negotiator (YARN) for resource management, Hive for data warehousing, Pig for high-level data flow programming, and HBase for NoSQL database capabilities, among others. Together, these components form a powerful platform for Big Data analytics.

3.2 Explain the concept of MapReduce.

MapReduce is a fundamental programming model and associated processing technique designed for handling and analyzing vast datasets across distributed computing environments. At its core, MapReduce simplifies the complexities of parallel processing by breaking down large computational tasks into smaller, manageable units that can be executed concurrently on multiple machines. This approach is particularly effective for big data applications where traditional single-machine processing would be impractical or impossible.

The MapReduce paradigm operates in two distinct, yet interconnected, phases:

- Map Phase: This is the initial stage where the input data, often massive in size and unstructured, is first divided into independent, smaller chunks. Each of these chunks is then processed by a "mapper" function. The primary role of the mapper is to take a piece of input data and transform it into a set of intermediate key-value pairs. During this transformation, the mapper often filters out irrelevant data, sorts the data, and performs initial aggregations or transformations relevant to the overall computation. The key aspect here is the parallel execution: multiple mappers can work simultaneously on different data chunks, significantly speeding up the initial processing.
- Reduce Phase: Following the completion of the Map phase, the intermediate key-value pairs generated by all mappers are shuffled and sorted. All values associated with the same key are then grouped together and sent to a single "reducer" function. The reducer's responsibility is to aggregate, combine, or further process these grouped values to produce the final desired output. This often involves performing calculations like sums, counts, averages, or more complex data transformations. Similar to the Map phase, multiple reducers can operate in parallel, each handling a different set of keys, thereby maintaining the distributed nature of the computation.

The power of MapReduce lies in its ability to abstract away the complexities of distributed programming, fault tolerance, and data distribution. Developers can focus on writing the specific map and reduce functions, while the underlying MapReduce framework handles the orchestration of tasks, data movement, and error recovery. This makes it a highly scalable and robust solution for various data processing tasks, including log analysis, web indexing, data warehousing, and machine learning model training.

Chapter 4
Apache Spark

4.1 What is Apache Spark?

Apache Spark is an open-source, distributed processing system designed for big data workloads. It stands out for its speed, primarily due to its in-memory computation capabilities, which significantly accelerate iterative algorithms and interactive data mining.

Key Features and Capabilities:

- Fast, In-Memory Data Processing: Spark's core strength lies in its ability to perform computations in memory, dramatically reducing disk I/O and making it much faster than traditional disk-based systems like Hadoop MapReduce for certain workloads. This is particularly beneficial for machine learning algorithms that require multiple passes over data.
- Elegant and Expressive Development APIs: Spark offers high-level APIs in various programming languages, including Scala, Java, Python, and R. These APIs simplify complex data processing tasks, allowing data workers to write more concise and readable code compared to lower-level frameworks.
- Versatile Workload Support: Spark is not limited to a single type of data processing. It provides modules for:
  – Spark SQL: For structured data processing using SQL queries, integrating with various data sources like Hive, JSON, and Parquet.
  – Spark Streaming: For real-time processing of live data streams, enabling applications like fraud detection and continuous monitoring.
  – MLlib (Machine Learning Library): A rich library of machine learning algorithms for tasks such as classification, regression, clustering, and collaborative filtering.
  – GraphX: For graph-parallel computation, allowing analysis of relationships within data.
- Efficient Execution for Iterative Access to Datasets: Many analytical and machine learning algorithms require repeated operations on the same dataset. Spark's in-memory caching and optimized execution engine make these iterative processes highly efficient. This direct access to datasets, without constantly writing to and reading from disk, significantly boosts performance.
- Fault Tolerance: Spark is designed to be fault-tolerant, meaning it can recover from failures of individual nodes in a cluster. This is achieved through its Resilient Distributed Datasets (RDDs), which are immutable, distributed collections of objects that can be rebuilt in case of a node failure.
- Scalability: Spark can scale out to hundreds or even thousands of nodes, allowing it to process petabytes of data. Its distributed architecture enables parallel execution of tasks across a cluster.

In essence, Apache Spark provides a powerful and flexible framework for handling diverse big data challenges, offering a significant performance advantage over older technologies, particularly for tasks demanding rapid, iterative access to large datasets.

4.2 How does Spark differ from Hadoop MapReduce?

Key differences include:

- Speed: Apache Spark significantly outperforms Hadoop MapReduce in terms of processing speed, primarily due to its in-memory processing capabilities. While MapReduce writes intermediate results to disk, Spark keeps data in RAM whenever possible, drastically reducing I/O operations and leading to faster computations, especially for iterative tasks.
- Ease of use: Spark offers more user-friendly and expressive APIs compared to Hadoop MapReduce. Developers can write applications in multiple popular languages, including Scala, Java, Python, and R, making it accessible to a broader range of programmers. MapReduce, on the other hand, primarily relies on Java and can be more verbose and complex for certain operations.
- Versatility: Spark is a more versatile unified analytics engine that supports a wider array of workloads beyond traditional batch processing. It integrates seamlessly with various components for streaming data (Spark Streaming), interactive queries (Spark SQL), graph processing (GraphX), and machine learning (MLlib), whereas MapReduce is primarily designed for batch processing.
- Iterative processing: Spark excels in iterative algorithms, which are common in machine learning and graph computations. Because Spark can cache data in memory across multiple iterations, it avoids the overhead of reading and writing data to disk in each iteration, a limitation of MapReduce. This makes Spark significantly more efficient for algorithms that repeatedly process the same dataset.

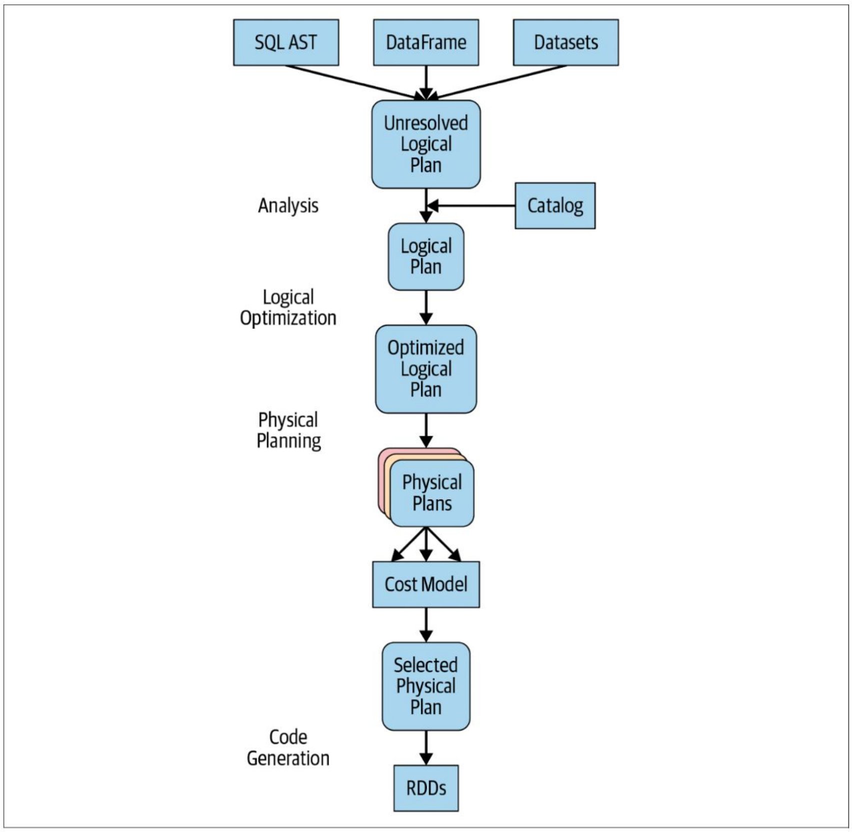

Fig. 4.1: ....

4.3 What is Catalyst Optimizer? How does Spark Execute a Query?

The Catalyst optimizer takes a computational query and converts it into an execution plan. It goes through four transformational phases, as shown in Figure 4.1:

Phase 1: Analysis
The Spark SQL engine begins by generating an abstract syntax tree (AST) for the SQL or DataFrame query. In this initial phase, any columns or table names will be resolved by consulting an internal Catalog, a programmatic interface to Spark SQL that holds a list of names of columns, data types, functions, tables, databases, etc. Once they've all successfully resolved, the query proceeds to the next phase.

Phase 2: Logical optimization
As the above figure shows, this phase comprises two internal stages. Applying a standard rule based optimization approach, the Catalyst optimizer will first construct a set of multiple plans and then, using its cost-based optimizer (CBO), assign costs to each plan. These plans are laid out as operator trees (like in Figure 4.1); they may include, for example, the process of constant folding, predicate pushdown, projection pruning, Boolean expression simplification, etc. This logical plan is the input into the physical plan.

Phase 3: Physical planning
In this phase, Spark SQL generates an optimal physical plan for the selected logical plan, using physical operators that match those available in the Spark execution engine.

Phase 4: Code generation
The final phase of query optimization involves generating efficient Java bytecode to run on each machine. Because Spark SQL can operate on data sets loaded in memory, Spark can use state-of-the-art compiler technology for code generation to speed up execution. In other words, it acts as a compiler. Project Tungsten, which facilitates whole-stage code generation, plays a role here.

4.4 Explain Spark Memory Management

Resources: [1] [2]

[1] ▶ Deep Dive: Apache Spark Memory Management
[2] ▶ Deep Dive into Project Tungsten Bringing Spark Closer to Bare Metal

4.5 What is PySpark?

PySpark is the Python API for Apache Spark, an open-source, distributed computing system designed for fast and large-scale data processing. It allows developers and data scientists to leverage the power of Spark's distributed architecture while working within the familiar and user-friendly Python programming language. This combination provides a powerful toolset for big data analytics, machine learning, and ETL (Extract, Transform, Load) operations.
At its core, PySpark enables you to write Spark applications using Python. This means you can define transformations and actions on large datasets, and Spark will then distribute these computations across a cluster of machines. The "simplicity of Python" refers to its straightforward syntax, extensive libraries, and large community, which make it a popular choice for data manipulation and analysis. The "power of Spark" lies in its ability to handle massive datasets by parallelizing operations and processing data in-memory whenever possible, leading to significantly faster execution compared to traditional disk-based systems.
PySpark offers access to all of Spark's components, including:

- Spark SQL: For working with structured data using SQL queries or DataFrames. DataFrames provide a powerful way to organize data into named columns, making it easy to perform operations like filtering, aggregation, and joining.
- Spark Streaming: For processing real-time data streams from various sources like Kafka, Flume, or HDFS.
- MLlib: Spark's machine learning library, offering a wide range of algorithms for classification, regression, clustering, and more, all optimized for distributed execution.
- GraphX: For graph-parallel computation, enabling analysis of interconnected data.

By bridging Python and Spark, PySpark facilitates the development of scalable data applications. It allows users to prototype quickly in Python and then deploy those applications on a Spark cluster for production-level big data processing, making it an indispensable tool in the modern data ecosystem.

4.6 What are some key features of Scala for data engineering?

Scala offers several key features beneficial for data engineering:

- Seamless Java Interoperability: Compatible with existing Java libraries and frameworks.
- Robust Type Safety: Strong static typing helps identify errors during compilation, leading to more reliable code.
- Concise Functional Programming: Supports a compact syntax for functional programming paradigms.
- Apache Spark's Native Language: Serves as the primary language for Apache Spark, optimizing big data operations.
- High Performance: Delivers excellent performance for processing large datasets.

Chapter 5
Apache Kafka

5.1 What is Apache Kafka?

Apache Kafka is an open-source, distributed streaming platform designed to handle vast amounts of data. It serves three core functions:

- Publishing and Subscribing to Streams of Records: Kafka acts as a central nervous system for data, allowing different applications to publish data (producers) and other applications to consume that data (consumers) in real-time. This decoupled architecture enables high scalability and flexibility, as applications can operate independently without directly communicating.
- Storing Streams of Records in a Fault-Tolerant Way: Kafka persists all published messages to disk, ensuring data durability. Its distributed nature, with data replicated across multiple servers (brokers), provides fault tolerance. If one server fails, the data remains accessible from other replicas, preventing data loss and ensuring continuous operation.
- Processing Streams of Records as They Occur: While not a full-fledged stream processing engine itself, Kafka integrates seamlessly with various stream processing frameworks (like Apache Flink, Apache Spark Streaming, or Kafka Streams). This allows for real-time analysis, transformations, and computations on the incoming data streams, enabling immediate insights and reactions to events as they unfold.

Essentially, Kafka provides a high-throughput, low-latency platform for handling real-time data feeds, making it a critical component for building scalable and resilient data pipelines in modern distributed systems.

Chapter 6
ETL/ELT, Pipelines and Orchestration

6.1 Explain the ETL process.

ETL, or Extract, Transform, Load, is a process designed for managing data. It involves collecting data from diverse sources, modifying it to meet operational requirements, and then loading it into a designated target, typically a data warehouse.
The ETL process consists of three key steps:

- Extract: Data is retrieved from its original source systems.
- Transform: The extracted data undergoes cleaning, validation, and conversion into an appropriate format.
- Load: The newly transformed data is then inserted into the target system.

Chapter 7
Data Warehouse, Data Lake, Data Lakehouse and Open Table Formats

7.1 What is a data warehouse?

A centralized repository, a data warehouse stores vast quantities of structured data from an organization's diverse sources. Its primary function is for query and analysis, rather than transaction processing.

7.2 What is the difference between a data lake and a data warehouse?

Here are the key differences:

- Data Structure: Data warehouses primarily store structured data, whereas data lakes accommodate structured, semi-structured, and unstructured data.
- Purpose: Data warehouses are optimized for analytical purposes, while data lakes function as a repository for raw data.
- Schema: Data warehouses utilize a schema-on-write approach, in contrast to data lakes which employ schema-on-read.
- Users: Business analysts are the typical users of data warehouses, while data scientists frequently use data lakes.

Chapter 8
Data Modeling, Architecture and Design

8.1 What is data modeling?

Data modeling is a crucial IT process that visually represents data structures and relationships, offering a clear understanding of data organization. Its primary purpose is to help stakeholders grasp data complexities by translating abstract requirements into diagrams like ERDs, which depict entities, attributes, and their relationships. Beyond understanding, data modeling standardizes data elements, ensuring quality and integrity by establishing consistent naming conventions and data types. It also acts as a blueprint for database design, guiding developers in creating efficient databases by defining tables, columns, and keys. Ultimately, data modeling is a strategic process that transforms raw data into a structured asset, enabling effective data management and informed decision-making.

8.2 What are the three main types of data models?

Data models are essential for organizing information, defining how data is represented, accessed, and updated. There are three main types:

- Conceptual Data Model: This is a high-level, abstract view focusing on business needs ("what" data is needed). It identifies entities, attributes, and relationships, facilitating communication between business and technical teams.
- Logical Data Model: Building on the conceptual model, this provides a more detailed, yet DBMS-independent, view of data structures ("how" data will be organized). It defines tables, columns, primary and foreign keys, and relationships, serving as a bridge to physical implementation.
- Physical Data Model: This is the implementation-specific representation of the data model within a chosen database system ("how" data is stored and accessed). It includes technical details like specific data types, indexing, and storage mechanisms, optimizing performance and facilitating database creation.

These models progress from a broad business understanding to a detailed, implementable database structure, ensuring efficiency and meeting business requirements.

8.3 What is a STAR schema?

A star schema is a data warehousing design with a central fact table (numerical business events) linked directly to surrounding dimension tables (descriptive attributes like time, product, customer). This de-normalized structure prioritizes fast data retrieval and analytical queries due to its simplicity, faster query performance, reduced join complexity, and optimization for aggregation. While offering ease of maintenance, its limitations include potential data redundancy and less flexibility for complex ad-hoc queries involving joins between dimension tables. Overall, it's a fundamental and efficient framework for business intelligence.

8.4 What is a SNOWFLAKE schema?

The Snowflake schema is an advanced version of the Star schema that normalizes dimension tables, breaking them down into smaller, related tables. This structure, visually resembling a snowflake, has a central fact table connected to primary dimension tables, which are further normalized into more specific dimension tables, creating a multi-layered hierarchy. For instance, a single "Product" dimension in a Star schema could be split into separate tables for "Product," "Product Category," "Brand," etc., in a Snowflake schema. The main advantages of the Snowflake schema are reduced data redundancy, improved data integrity, more efficient storage, easier maintenance, and benefits for hierarchical dimensions with independent query needs.

8.5 What are the advantages and disadvantages of denormalization?

Denormalization: Pros and Cons
Advantages:

- Faster Queries: Denormalization leads to improved query performance.
- Simplified Queries: It simplifies complex queries.
- Reduced Joins: The need for multiple table joins is minimized.

Disadvantages:

- Data Redundancy: Denormalization results in increased data redundancy.
- Complex Updates: Data updates and inserts become more complex.
- Inconsistencies: There is a higher potential for data inconsistencies.

8.6 What is the slowly changing dimension (SCD)?

Slowly Changing Dimensions (SCDs) are vital in data warehousing for managing changes in dimensional data over time, ensuring accurate historical analysis. Unlike transactional data, dimension data describes business attributes (e.g., customer details, product information) that can change.
Common SCD types include:

- Type 1 (Overwrite): Updates the existing value, losing historical data. Suitable for error correction or when historical accuracy isn't critical.
- Type 2 (New Row): Creates a new row for each change, preserving a complete history. This is the most common and robust type, allowing for accurate historical reporting by using effective and end dates to track validity periods.
- Type 3 (New Column): Adds a new column to the same row to track a limited history of changes for a specific attribute, typically the immediate previous value.

The selection of an SCD type depends on business requirements for historical data tracking, reporting, and analysis, impacting data storage and query complexity.

8.7 What is data mart?

A data mart is a specialized, smaller version of a data warehouse designed to meet the specific analytical needs of a particular department or business function within an organization. Unlike a comprehensive enterprise data warehouse, a data mart focuses on relevant data for a specific area, such as sales or marketing. Key advantages include improved performance due to smaller datasets, enhanced usability for targeted users, cost-effectiveness in development and maintenance, granular data security, and faster development and deployment. Essentially, data marts provide focused, relevant data to empower specific user groups for informed decision-making and reporting, bridging the gap between broad enterprise data and immediate departmental analytical requirements.

8.8 Explain the concept of data partitioning.

Data partitioning is a core technique in database management and distributed computing, dividing large datasets into smaller, manageable segments called partitions. This improves query performance, facilitates parallel processing, and simplifies data management.
Key partitioning strategies include:

- Range Partitioning: Divides data based on predefined value ranges in a column (e.g., by date). It's effective for range-based queries but requires careful range definition to avoid "hot spots" (uneven data distribution).
- Hash Partitioning: Distributes data using a hash function applied to a specified column. It achieves even data distribution, ideal for exact match queries, and balances computational load. Challenges include selecting the right hash function and rebalancing data if the number of partitions changes.
- List Partitioning: Partitions data based on an explicit list of discrete values in a column (e.g., by product category). It's straightforward for naturally categorized data but may require modification if new categories are introduced.

More advanced approaches include:

- Composite Partitioning: Combines two or more core strategies (e.g., range by date, then hash by customer ID) for multi-layered organization.
- Interval Partitioning: An automated variation of range partitioning that creates new partitions based on predefined intervals, often time-based.

Choosing the right partitioning strategy depends on data characteristics, query types, and system performance goals. Effective data partitioning is crucial for scalable database architectures, enhanced query performance in big data environments, and improved maintainability of large datasets.

Chapter 9
Cloud Computing

9.1 What are the main advantages of cloud computing for data engineering?

- Scalability: Resources can be easily scaled up or down to meet demand.
- Cost-effectiveness: Users only pay for the resources they consume.
- Flexibility: A wide array of services and tools are readily available.
- Reliability: The system incorporates built-in redundancy and disaster recovery capabilities.
- Global Reach: Resources can be deployed across various geographic regions.

9.2 AWS

9.2.1 What is Amazon S3?

Amazon S3 (Simple Storage Service), an object storage service by Amazon Web Services (AWS), offers scalable, durable, and highly available storage. It's a popular choice for data lakes and backup solutions, accommodating diverse data types.

9.3 Azure

9.3.1 What is Azure Synapse Analytics?

Azure Synapse Analytics is a comprehensive, limitless analytics service that combines data integration, enterprise data warehousing, and big data analytics capabilities. It offers flexible data querying options, allowing users to utilize serverless or dedicated resources to operate at scale.

Chapter 10
Python

10.1 Is Python a compiled language or an interpreted language?

Please remember one thing, whether a language is compiled or interpreted or both is not defined in the language standard. In other words, it is not a proper programming language. Different Python distributions (or implementations) choose to do different things (compile or interpret or both). However the most common implementations like CPython do both compile and interpret, but in different stages of its execution process.
Compilation: When you write Python code and run it, the source code (.py files) is first compiled into an intermediate form called bytecode (.pyc files). This bytecode is a lower-level representation of your code, but it is still not directly machine code. It's something that the Python Virtual Machine (PVM) can understand and execute.
Interpretation: After Python code is compiled into bytecode, it is executed by the Python Virtual Machine (PVM), which is an interpreter. The PVM reads the bytecode and executes it line-by-line at runtime, which is why Python is considered an interpreted language in practice.
Some implementations, like PyPy, use Just-In-Time (JIT) compilation, where Python code is compiled into machine code at runtime for faster execution, blurring the lines between interpretation and compilation.

10.2 Difference between for loop and while loop in Python

- For loop: Used when we know how many times to repeat, often with lists, tuples, sets, or dictionaries.
- While loop: Used when we only have an end condition and don't know exactly how many times it will repeat.

10.3 What is a dynamically typed language?

- In a dynamically typed language, the data type of a variable is determined at runtime, not at compile time.
- No need to declare data types manually; Python automatically detects it based on the assigned value.
- Examples of dynamically typed languages: Python, JavaScript.
- Examples of statically typed languages: C, C++, Java.
- Dynamically typed languages are easier and faster to code.
- Statically typed languages are usually faster to execute due to type checking at compile time.

10.4 How are arguments passed by value or by reference in Python?

- Python's argument-passing[3] model is neither "Pass by Value" nor "Pass by Reference" but it is "Pass by Object Reference".
- Depending on the type of object you pass in the function, the function behaves differently. Immutable objects show "pass by value" whereas mutable objects show "pass by reference"

[3] https://www.geeksforgeeks.org/pass-by-reference-vs-value-in-python/

Chapter 11
Batch and Stream Processing

11.1 What is batch processing?

Batch processing is an efficient method for handling large data volumes when immediate results aren't critical. It works by collecting and grouping transactions over time, then processing them in a single, continuous run. This approach is highly beneficial for repetitive, high-volume tasks like payroll or financial reconciliation, optimizing resource utilization by dedicating system resources to uninterrupted processing. Its efficiency comes from minimizing overhead, optimizing system resources through streamlined data handling, and providing natural checkpointing for recovery. While it doesn't provide immediate results, this is a deliberate design choice that prioritizes predictability, throughput, and cost-effectiveness, making it ideal for tasks like analyzing sales trends or updating inventory, which can be done without overwhelming real-time operational systems.

11.2 What is stream processing?

Stream processing is a vital method for handling continuous data flows in real-time, unlike traditional batch processing. Its core principle is to minimize latency, allowing immediate analysis and action on incoming data. Key characteristics include continuous data flow, low latency, immutability, event-driven architecture, and scalability. Stream processing has wide-ranging applications across industries such as financial services (fraud detection), IoT (sensor monitoring), e-commerce (personalized recommendations), telecommunications (network monitoring), healthcare (patient monitoring), and logistics (shipment tracking). Popular technologies include Apache Kafka, Apache Flink, Apache Spark Streaming, and Google Cloud Dataflow. Ultimately, stream processing enables organizations to transform raw data into actionable intelligence with speed and proactivity.

11.3 What is the Lambda architecture?

The Lambda architecture is a data processing approach for big data environments that combines batch and stream processing to offer both historical accuracy and real-time insights.
It consists of three layers:

1. Batch Layer: Manages an immutable master dataset, storing all raw data for a complete historical record. It pre-computes "batch views" from large datasets, offering high accuracy but with eventual consistency. This layer is scalable and fault-tolerant.
2. Speed Layer: Processes incoming data streams in real-time, providing immediate, though potentially approximate, "real-time views." It performs incremental computations to offer low-latency insights and compensates for the batch layer's latency.
3. Serving Layer: Acts as a unified interface for queries, combining results from both batch and speed layers to provide comprehensive and up-to-date answers. It is optimized for fast read access and merges historical context with the latest updates.

Advantages of the Lambda architecture include robustness, fault tolerance, scalability, comprehensive data views, and data consistency. However, challenges include architectural complexity, potential code duplication, resource intensity, and state management in the speed layer. Despite these complexities, it's a powerful pattern for resilient and scalable data processing in big data environments.

11.4 What is Apache Flink?

Apache Flink is an open-source framework for distributed data streaming. It enables high-performing, always-available, and accurate stream processing applications by offering precise control over time and state. This ensures consistent and accurate results, even when data arrives out-of-order or late.

Chapter 12
Data Security and Governance

12.1 What is data governance?

Data governance encompasses the frameworks, roles, policies, standards, and performance indicators critical for an organization's effective and efficient utilization of information to achieve its objectives. It defines the processes and responsibilities necessary for maintaining data quality, security, and compliance.

12.2 What is data encryption?

Data encryption converts data into an unreadable code (ciphertext) using an algorithm. This process prevents unauthorized access by transforming original data (plaintext) so it can only be decrypted with a specific key.

12.3 What is GDPR and how does it affect data engineering?

The General Data Protection Regulation (GDPR) is an EU law focusing on data protection and privacy. Its implications for data engineering include:

- Data Practices: Affects how data is collected, stored, processed, and utilized.
- Data Subject Rights: Addresses rights of individuals concerning their data, such as the "right to be forgotten."
- Breach Notifications: Establishes requirements for notifying data breaches.
- International Transfers: Governs data transfers across borders.

12.4 What is data masking?

Data masking creates a structurally similar yet inauthentic version of an organization's data. This technique safeguards sensitive information by providing a functional substitute for uses like software testing and user training.

12.5 What is role-based access control (RBAC)?

Role-based access control (RBAC) streamlines the management of user rights by regulating access to computer or network resources. This method assigns permissions to specific roles, and individual users are then assigned to these roles based on their function within an organization.

Part II
Probability & Statistics

References:

1. ...

Contents

Chapter 13
Fundamentals

13.1 Define probability. Explain the difference between marginal, conditional, and joint probability with a concise example for each.

13.2 What is expectation (mean)? Define variance and standard deviation; explain how they behave under linear transformations of a random variable.

13.3 Explain measures of central tendency and dispersion. (mean, median, mode, variance, standard deviation, interquartile range, five-number summary) — why they matter and when to prefer one over another.

13.4 What is a probability distribution? Describe common distributions (Bernoulli, Binomial, Poisson, Uniform, Exponential, Normal/Gaussian) and give typical use-cases for each.

13.5 Different types of distributions: contrast discrete vs. continuous distributions and describe what "heavy-tailed" / "long-tailed" means; give three real-world examples of long-tailed phenomena and explain their practical consequences.

13.6 What is skewness and kurtosis? Define them and describe two methods to measure skewness.

Chapter 14
Laws & Limit theorems

14.1 State the Law of Large Numbers (LLN). How is it used in practice?

14.2 State and interpret the Central Limit Theorem (CLT). What general conditions are required for the CLT to hold and give examples of when you can (and cannot) safely apply it.

Chapter 15
Inference—Estimation & Hypothesis Testing

15.1 Describe point estimation methods. Explain Method of Moments, Maximum Likelihood Estimation (MLE), and Bayesian estimation (MAP/posterior summaries).

15.2 What is an unbiased estimator? Define bias, consistency, efficiency (Cramér–Rao concept) and asymptotic normality.

15.3 How do you estimate parameters for a Uniform(0, d) sample? (state estimators and discuss bias/consistency)

15.4 Explain hypothesis testing. Define null vs. alternative hypothesis, p-value, significance level (α), Type I and Type II errors, and statistical power.

15.5 What is the relationship between significance level and confidence level? (interpretation and numeric examples)

15.6 What is a confidence interval? How do you interpret a 95% confidence interval? Contrast confidence intervals with prediction intervals and state how each is calculated.

15.7 When to use a z-test vs. a t-test? Explain the differences and conditions for each, and how to compute the test statistic from a sample.

15.8 Describe Chi-square tests and ANOVA. When is each appropriate (goodness-of-fit, independence, group-mean comparison)?

15.9 Multiple hypothesis testing: what problems arise when testing many hypotheses and which corrections/approaches would you consider (Bonferroni, Holm, Benjamini–Hochberg / FDR, permutation-based methods)?

15.10 How to compute pooled/blended mean and standard deviation given two (or K) subsets with known sizes, means and variances? (Show formulas and describe use-cases.)

Chapter 16
Bayesian vs Frequentist & Likelihood

16.1 Explain the difference between frequentist and Bayesian probability approaches. Give examples where each is preferable.

16.2 Define prior, likelihood, and posterior. What is a conjugate prior?

16.3 What is the difference between probability and likelihood? Provide clarifying examples.

Chapter 17
Resampling, Simulation & Nonparametric Methods

17.1 Explain the bootstrap. Describe basic bootstrap steps, when it's useful, and limitations.

17.2 Describe permutation tests and when you would use them.

17.3 Outline Monte Carlo simulation and typical diagnostics used to validate simulation results (e.g., convergence, variance reduction ideas).

Chapter 18
Regression-related & Diagnostics (statistical aspects only)

18.1 State the assumptions of ordinary least squares (OLS) linear regression. What diagnostics do you run to check them?

18.2 What is multicollinearity? Define Variance Inflation Factor (VIF) and how to interpret it.

18.3 Explain heteroscedasticity. How can it be detected and what remedies can be used (e.g., robust SEs, transforms)?

18.4 Bias–variance trade-off (statistical view). How does it relate to estimator choice and model complexity?

Chapter 19
Experimental Design & A/B Testing (statistics focus)

19.1 What is A/B testing? Describe the standard workflow from hypothesis to conclusion (including metric selection).

19.2 Common pitfalls in A/B testing. Discuss issues such as peeking/sequential testing, underpowered experiments, multiple comparisons, biased assignment, metric ambiguity, and sample contamination.

19.3 Practical experiment rules. Explain pre-registration, stopping rules, randomization principles, and guardrails (secondary metrics).

Chapter 20
Advanced & Applied Statistical Topics

20.1 Time series essentials (basic concepts only): stationarity, autocorrelation, and seasonality detection.

20.2 Survival analysis basics and what censoring means (Kaplan–Meier intuition).

20.3 EM algorithm (high-level): describe the E and M steps and a common use-case (mixture models).

20.4 Intro to causal inference (statistical tools): confounding, randomized experiments, difference-in-differences, instrumental variables, propensity scores — when each is used and core assumptions.

20.5 Information measures: define entropy and Kullback–Leibler divergence and explain one statistical/estimation application.

20.6 Data quality concerns: selection bias, sampling bias, missing-data mechanisms (MCAR, MAR, MNAR) and common imputation approaches.

20.7 Model monitoring & drift detection (statistical checks): outline basic statistical tests/metrics to monitor model behavior and dataset shifts.

20.8 Ethics & fairness (statistical checks): list basic statistical analyses to surface bias (group-wise metrics, calibration checks, disparate impact measures).

Part III
Data Science

References:

1. ...

Contents

Chapter 21
Data Cleaning, EDA & Feature Engineering

21.1 What are the common steps in data cleaning and preprocessing? (e.g., deduplication, missing-value handling, encoding categorical variables, scaling).

21.2 How do you handle missing data? Compare deletion, mean/median/mode imputation, hot-deck, KNN or model-based imputation, and multiple imputation.

21.3 Describe the types of missingness (MCAR, MAR, MNAR). How does each affect your handling strategy?

21.4 What are outliers? How do you detect outliers (z-score, IQR, robust methods, visualization) and what strategies can you use to handle them (cap/Winsorize, remove, transform, model-based)?

21.5 Mention three ways to make your model robust to outliers.

21.6 What is skewness? How can you detect skew and which transformations (log, Box–Cox, Yeo–Johnson) can correct skewed distributions?

21.7 Explain feature scaling: normalization vs standardization. When is scaling necessary?

21.8 What are common categorical encoding techniques (one-hot, ordinal, target/mean encoding, hashing) and what are their tradeoffs?

21.9 What is feature engineering? Give concrete examples for time-series, text, and categorical features.

21.10 What is feature selection? Describe filter, wrapper and embedded methods.

21.11 What is the five-number summary and why is it useful?

21.12 How do you detect and prevent target leakage? Give examples.

Chapter 22
Exploratory Data Analysis & Diagnostics

22.1 Provide a practical pre-analysis checklist you run before modeling (missingness, duplicates, distributions, sample size, etc.).

22.2 How do you visualize relationships between variables (scatterplots, correlation matrices, pairplots, partial dependence)?

22.3 How do you check regression assumptions: linearity, independence, homoscedasticity, normality of residuals?

22.4 What is calibration of predicted probabilities? How do you evaluate and improve calibration?

22.5 How do you evaluate class imbalance and what techniques address it (resampling, class weights, SMOTE, threshold tuning)?

Chapter 23
Probability & Statistics Essentials (for modeling)

23.1 Explain the bias–variance trade-off and how to detect underfitting vs overfitting.

23.2 Define Type I/Type II errors, p-value, significance level, and statistical power.

23.3 What is a confidence interval and how do you interpret a 95% confidence interval?

23.4 What is the difference between parametric and non-parametric models? Give two examples of each.

23.5 How would you perform a power/sample-size calculation given expected effect size, variance, α and desired power?

Chapter 24
Supervised Learning — Classic Models

24.1 Linear regression: State the model form, assumptions, key evaluation metrics (MSE, RMSE, R^2) and when it is appropriate.

24.2 Logistic regression: How does it differ from linear regression? How do you interpret coefficients (odds ratios)? Give an example of usage.

24.3 K-Nearest Neighbors (KNN): How does KNN work for classification/regression? Pros and cons (compute cost, curse of dimensionality).

24.4 Decision trees: How are they built? Define entropy, information gain and Gini impurity; what splitting methods exist?

24.5 Random forests: Explain their motivation and list two reasons they generally outperform single decision trees.

24.6 Gradient boosting: Describe gradient-boosting conceptually and name popular libraries (XGBoost, LightGBM, CatBoost).

24.7 Compare gradient boosting vs random forest: similarities, differences, advantages and disadvantages of each.

24.8 Support Vector Machines (SVM): Explain margins, the kernel trick, and guidance for kernel selection. Do you need to scale inputs for SVM? Explain.

24.9 Naive Bayes: Why is it “naive”? Where does it perform well?

24.10 What is hard voting vs soft voting in ensemble classifiers?

24.11 What is boosting and name two famous boosting methods?

24.12 KNN, SVM, tree-based model selection: when would you pick each?

Chapter 25
Model Evaluation & Selection

25.1 Define precision, recall, specificity, F1, accuracy; discuss trade-offs and when each metric is appropriate.

25.2 What is the ROC curve and AUC? When should you prefer PR curve over ROC?

25.3 For regression, compare MSE vs MAE and discuss when each is preferred.

25.4 What is cross-validation (k-fold)? Explain stratified CV and time-series CV; what is nested CV and why use it?

25.5 How do you use validation & learning curves to diagnose model issues?

25.6 How do you compare models reliably (statistical tests, repeated CV, confidence intervals across folds)?

25.7 What metrics can be used for multi-label classification and how do you choose among them?

25.8 For clustering, what evaluation metrics exist (silhouette, Davies–Bouldin, inertia, adjusted rand index) and how do you choose?

25.9 How would you evaluate and compare dimensionality reduction algorithms on a dataset?

Chapter 26
Unsupervised Learning & Dimensionality Reduction

26.1 K-means: How does it work and how to choose k (Elbow, silhouette score)? Limitations?

26.2 Hierarchical clustering: Agglomerative vs divisive — when to use hierarchical over K-means?

26.3 DBSCAN: What is density-based clustering and how do ε and minPts affect results?

26.4 PCA: What does PCA do; what are eigenvectors/eigenvalues in this context? When use PCA vs Kernel PCA vs Incremental PCA vs Randomized PCA?

26.5 What is the curse of dimensionality and how do you address it?

26.6 Mention two clustering algorithms that scale to large datasets.

26.7 How do you evaluate the performance of a dimensionality-reduction algorithm?

Chapter 27
Recommendation Systems & Special Topics

27.1 What are different approaches to building recommendation systems (collaborative filtering, content-based, hybrid)?

27.2 What is active learning and describe one active-learning strategy.

27.3 Explain association rules (Apriori): support, confidence, lift — give a use-case.

Chapter 28
Optimization, Regularization & Training Dynamics

28.1 What are loss functions and cost functions and how do they differ?

28.2 Explain L1 and L2 regularization: how they work and their differences.

28.3 How does L1/L2 regularization affect neural networks?

28.4 Explain batch, mini-batch and stochastic gradient descent; pros/cons of each.

28.5 What is momentum and what happens if momentum is set too close to 1 (e.g., 0.9999)?

28.6 Describe common optimizers (SGD, SGD+momentum, Adam, RMSProp) and when you'd use each.

28.7 What are learning-rate schedules and why are they useful?

28.8 What hyperparameters would you tune for neural networks and classic models? Methods: grid search, random search, Bayesian optimization, Hyperband.

28.9 What is early stopping and how does it regularize training?

28.10 What is gradient clipping and when would you use it to control exploding gradients?

Chapter 29
Neural Networks — Fundamentals & Practicalities

29.1 Describe the structure of a neural network (layers, activations, weights) and high-level backpropagation.

29.2 What are common activation functions (ReLU, sigmoid, tanh, softmax) and where to use them?

29.3 What is vanishing/exploding gradient and how do ReLU, residual connections, batch normalization or careful initialization help?

29.4 Why use batch normalization? Which hyperparameters affect a batch norm layer?

29.5 What is dropout and what is its effect on training and inference speed?

29.6 What is parameter sharing and why is it useful (e.g., in CNNs)?

29.7 What are epochs, batches, and iterations?

29.8 How can you tell a neural network is overfitting and what are remedies (regularization, dropout, data augmentation, early stopping)?

29.9 If a model's loss does not decrease in the first few epochs, what could be wrong (learning rate, initialization, data issues, bug)?

29.10 Why are sigmoid/tanh less preferred in deep hidden layers compared to ReLU?

29.11 What is a depthwise separable convolution and why use it?

Chapter 30
Convolutional & Sequence Models (Vision & Time/Sequence)

30.1 Describe a typical CNN architecture (convolution, activation, pooling, fully-connected).

30.2 How does data augmentation help in computer vision? Give common techniques.

30.3 What are tasks in vision (classification, detection, segmentation) and relevant metrics (accuracy, IoU, mAP)?

30.4 Explain object detection vs image classification; explain semantic vs instance segmentation.

30.5 What popular vision architectures exist (ResNet, EfficientNet, YOLO, Mask R-CNN) and the key idea behind each?

30.6 Can CNNs be used for 1D signals? When does that make sense?

30.7 What is feature extraction in vision and how is transfer learning typically applied (ImageNet pre-trained models)?

30.8 Describe RNNs and their limitations; how do LSTM and GRU address vanishing gradients?

30.9 What are the main gates in an LSTM and their roles?

Chapter 31
Autoencoders, GANs & Generative Models

31.1 What are autoencoders and their architecture; list three use cases (dimensionality reduction, denoising, anomaly detection).

31.2 What are variational autoencoders (VAE) conceptually?

31.3 What are Generative Adversarial Networks (GANs)? Describe generator and discriminator and one application.

31.4 How do you evaluate generative models qualitatively and quantitatively?

Chapter 32
Transformers, Attention & LLMs

32.1 Explain the Transformer architecture: self-attention, multi-head attention, positional encoding.

32.2 What is self-attention and why does it help capture long-range dependencies?

32.3 How do encoder-only (BERT), decoder-only (GPT) and encoder–decoder (T5) models differ in typical uses?

32.4 What is the difference between masked LM (BERT) and autoregressive LM (GPT)?

32.5 Explain positional encoding and why it is needed.

32.6 How do you fine-tune a large language model for classification, QA or summarization? Outline practical steps and pitfalls.

32.7 What are tokenization schemes (WordPiece, BPE, SentencePiece) and why does tokenization matter?

32.8 What is chain-of-thought (CoT) prompting and when might it help reasoning?

32.9 What is prompt engineering basics for GPT-like models and when to prefer prompting vs fine-tuning?

32.10 Describe RAG (Retrieval-Augmented Generation): components (retriever, vector DB, reader/generator) and why RAG improves factuality.

32.11 What are typical failure modes of LLMs (hallucination, bias, context-window limits) and common mitigations?

Chapter 33
NLP Practicalities & Evaluation

33.1 What are word embeddings (Word2Vec, GloVe) vs contextual embeddings (BERT)? When to use each?

33.2 Outline an NLP pipeline (tokenization, normalization, embedding, modeling, post-processing).

33.3 How do you convert text for classification (TF-IDF, bag-of-words, embeddings)? Pros/cons.

33.4 What is seq2seq modeling and give an example (machine translation).

33.5 What are decoding strategies (greedy, beam search, sampling) and trade-offs?

33.6 How do you evaluate NLP models (BLEU, ROUGE, perplexity) and limitations of automatic metrics?

Chapter 34
LLM Ops, Safety & Advanced LLM Use-Cases

34.1 What is RAG in production and give an example use-case (chatbot with documents).

34.2 How do you evaluate retrieval (recall@k, MRR) and tune retrieval vs generation?

34.3 How to monitor LLMs in production for hallucination and factual drift? What metrics or tooling would you use?

34.4 What are ethical and legal considerations when training/fine-tuning LLMs on external corpora (copyright, privacy)?

Chapter 35
Training at Scale & Systems

35.1 How do you train models on very large datasets (mini-batch, data generators, sharding, distributed training)?

35.2 Describe data-parallel vs model-parallel training and practical considerations (gradient sync, all-reduce).

35.3 How to optimize GPU/TPU utilization (mixed precision, gradient accumulation, data pipeline optimization)?

35.4 What are limitations of Spark for deep learning and when to use Spark MLlib vs specialized DL frameworks?

35.5 What is Spark MLlib and how do you construct an ML pipeline with Transformer/Estimator API?

35.6 How does Spark support streaming machine learning (incremental updates)?

Chapter 36
Deployment, MLOps & Production Practices

36.1 How do you deploy ML models to production: containerization, REST/gRPC APIs, serverless endpoints, batch jobs?

36.2 What is a feature store and why is it important for production ML?

36.3 How do you ensure offline/online feature parity and consistency between training and serving?

36.4 What is model monitoring and how do you detect data drift, concept drift, and model degradation?

36.5 What are safe rollout strategies (canary, blue–green, shadow) and rollback procedures?

36.6 What is CI/CD for ML and how does it differ from traditional CI/CD? What do you version (code, data, models)?

36.7 Name MLOps tools and their roles (MLflow, Kubeflow, DVC, Airflow/Prefect, Feast).

36.8 How do you perform experiment tracking and reproducibility (seed management, artifact storage, dataset lineage)?

36.9 How do you log and alert on production model performance?

Chapter 37
Cloud Services, Managed Platforms & Big Data

37.1 Which cloud ML services do you know (AWS SageMaker, GCP Vertex AI, Azure ML, Databricks)? What components do they provide?

37.2 How do serverless or SQL-native ML services (BigQuery ML, Synapse) enable modeling without separate hosting? Tradeoffs?

37.3 What are benefits of managed ML platforms (autoscaling, monitoring, model registry)?

37.4 Give an example of AWS SageMaker components (notebook, training job, model registry, endpoint).

37.5 How do you incorporate BigQuery ML or similar services into a data-product pipeline?

Chapter 38
Software Engineering, Tools & Best Practices

38.1 Which libraries/frameworks do you commonly use (Pandas, scikit-learn, TensorFlow, PyTorch, Hugging Face)?

38.2 What is scikit-learn used for? Name built-in models or utilities you've used.

38.3 Compare TensorFlow vs PyTorch: practical differences and when to choose one.

38.4 How do you productionize code from Jupyter Notebooks (refactor, package, unit tests, CI)?

38.5 What is DVC or Delta Lake and how do they help with data versioning and reproducible pipelines?

38.6 What is concurrency in Python; how does the GIL affect multithreaded code and when to use multiprocessing vs threading vs async?

38.7 Give an example of a Python design pattern relevant to ML engineering (factory, dependency injection, singleton).

38.8 What tools do you use for containerization and orchestration (Docker, Kubernetes) in ML workflows?

Chapter 39
Model Robustness, Privacy & Ethics

39.1 What is adversarial robustness and how might you test or harden models against adversarial inputs?

39.2 What are techniques for privacy-preserving ML (differential privacy, federated learning, secure aggregation)? When to use them?

39.3 How do you assess fairness, accountability and transparency in models? Give three statistical checks or mitigation strategies (disparate impact, equalized odds, calibration).

39.4 What is model interpretability and which tools/methods help (SHAP, LIME, partial dependence)?

Chapter 40
Time Series, ARIMA & Forecasting

40.1 Explain the ARIMA model and its components (AR, I, MA).

40.2 What assumptions does ARIMA make and how do you check them?

40.3 How do you handle seasonality, trend, and stationarity in time series?

40.4 What evaluation metrics and cross-validation strategies are used for forecast models?

Chapter 41
Advanced / Research-style & System Design

41.1 Derive the E and M steps for a two-component Gaussian mixture (outline EM algorithm).

41.2 Outline an approach for causal inference from observational data (propensity scores, DID, IV) and state identification assumptions.

41.3 What is continual (online) learning and how do you mitigate catastrophic forgetting?

41.4 Describe federated learning and its production challenges (heterogeneity, communication, privacy).

41.5 How would you design a RAG system end-to-end for a company knowledge base? Include retriever, vector DB, generator, indexing and evaluation.

Chapter 42
Behavioral / Practical Prompts

42.1 Describe a production ML system you built or maintained: problem, approach, architecture, monitoring, and lessons learned.

42.2 Give an example where you improved model performance without changing architecture — what did you change and why?

42.3 Describe a time you diagnosed a production model failure — how did you identify and remediate it?

Part IV
Behavioural

References:

1. Tech Interview Handbook: Behavioral Interview Questions
2. ▶ Behavioral Interview Discussion with Ex-Meta Hiring Committee Member

Contents

Chapter 43
Personal Questions:

43.1 Tell me about yourself.

43.2 What project(s) are you currently/previously working on?

43.3 What is the most challenging aspect of your project?

43.4 Talk about a project you are most passionate about, or one where you did your best work.

43.5 Why do you want to work for this company?

43.6 Why do you want to leave your current/last company?

43.7 What are you looking for in your next role?

43.8 What are your strengths/weaknesses? What strengths do you think are most important for your job position?

43.9 What words would your colleagues use to describe you?

43.10 What would you hope to achieve in the first six months after being hired?

43.11 Tell me why you will be a good fit for the position.

43.12 How do you tackle challenges? Name a difficult challenge you faced while working on a project, how you overcame it, and what you learned.

43.13 What are you excited about and what frustrates you?

43.14 Imagine it is your first day here at the company. What do you want to work on? What features would you improve on?

43.15 What does your best day of work look like?

43.16 Time management has become a necessary factor in productivity. Give an example of a time-management skill you’ve learned and applied at work.

43.17 What aspects of your work are most often criticized?

Chapter 44
Common Behavioural Questions (STAR Pattern)

44.1 Tell me about a time when you had a conflict with a co-worker.

44.2 Tell me about a problem you’ve had getting along with a work associate.

44.3 Tell me about a time in which you had a conflict and needed to influence somebody else.

44.4 Tell me about a time you had a disagreement with your manager.

44.5 What is something that you had to push for in your previous projects?

44.6 What is the most constructive feedback you have received in your career?

44.7 Tell me about a time you met a tight deadline.

44.8 How have you handled criticism of your work?