DM-01 - Definitions
Definitions of technical terms -> look in the slides
Relational database -> Uses primary and foreign keys to enforce the relationship constraints

Types of Databases and Their Use Cases

1. Relational Databases
Definition:

- Relational databases store data in tables (rows and columns).
- Data is organized into relations (tables) which can be linked via foreign keys.
- They use Structured Query Language (SQL) for defining and manipulating the data.

Examples:

- MySQL
- PostgreSQL
- Oracle Database
- Microsoft SQL Server
- SQLite

Use Cases:

- **Structured Data:** When the data is highly structured and requires complex querying, relational databases are ideal.
- **ACID Compliance:** Applications needing strong consistency, atomicity, isolation, and durability (ACID properties) benefit from relational databases.
- **Transactional Systems:** Suitable for financial applications, e-commerce systems, and enterprise applications where transactions are crucial.
- **Data Integrity:** When data integrity and normalization are priorities, relational databases are preferred.

Pros:

- Robust and mature technology.
- Powerful querying capabilities with SQL.
- Strong data integrity and ACID compliance.
- Excellent support for joins and complex transactions.

Cons:

- Less efficient for unstructured or semi-structured data.
- Scaling horizontally can be challenging and expensive.
- Schema changes can be cumbersome.

2. Document-Based Databases
Definition:

- Document-based databases store data in document formats, typically JSON or BSON.
- Each document contains data in key-value pairs and can have a flexible schema.

Examples:

- MongoDB
- CouchDB
- Amazon DocumentDB
- ArangoDB

Use Cases:

- **Flexible Schema:** Ideal for applications where the schema can evolve over time, such as content management systems or data that doesn't fit neatly into tables.
- **Unstructured Data:** Suitable for storing unstructured or semi-structured data, like JSON documents.
- **Rapid Development:** When rapid development and iteration are required, as schema changes are easier to handle.
- **Data Aggregation:** Efficient for applications needing to aggregate large volumes of data, like logging and real-time analytics.

Pros:

- Flexible schema allows for easy changes.
- Good performance for read and write operations.
- Scalability is often easier compared to relational databases.
- Suitable for hierarchical data storage.

Cons:

- Limited support for complex transactions.
- Querying capabilities can be less powerful compared to SQL.
- Data consistency may not be as strong as in relational databases.

3. Graph Databases

Definition:

- Graph databases store data as nodes (entities) and edges (relationships).
- They use graph structures for semantic queries, representing and storing data in graphs.

Examples:

- Neo4j
- Amazon Neptune
- OrientDB
- ArangoDB (multi-model, also supports graph)

Use Cases:

- **Relationship-Heavy Data:** Ideal for applications where the relationships between data points are as important as the data itself, such as social networks, fraud detection, and recommendation engines.
- **Complex Queries:** Suitable for scenarios requiring complex queries involving traversals and pathfinding.
- **Network Analysis:** Useful for network analysis, bioinformatics, and scenarios involving interconnected data.

Pros:

- Efficient for querying and managing relationships.
- Excellent performance for complex, interconnected data queries.
- Intuitive for representing graph structures and networks.
- Schema-less nature provides flexibility.

Cons:

- Learning curve for graph query languages (like Cypher for Neo4j).
- Not suitable for all types of data (e.g., highly structured, transactional data).
- Limited support for large-scale transaction processing.

4. Key-Value Databases

Definition:

- Key-value databases store data as a collection of key-value pairs, where each key is unique and is used to retrieve the corresponding value.

Examples:

- Redis
- Amazon DynamoDB
- Riak
- Couchbase (also supports document model)

Use Cases:

- **Caching:** Ideal for caching sessions, user profiles, and other temporary data.
- **High-Speed Transactions:** Suitable for applications needing fast read and write operations.
- **Simple Data Models:** When the data structure is simple and doesn't require complex querying.

Pros:

- Extremely fast read/write operations.
- Simple and easy to use.
- Highly scalable and can handle high traffic loads.
- Flexible schema design.

Cons:

- Limited querying capabilities.
- Not suitable for complex transactions.
- Data modeling can be challenging for more complex use cases.

5. Column-Family Databases

Definition:

- Column-family databases store data in columns rather than rows, which allows for efficient read and write operations for certain types of queries.

Examples:

- Apache Cassandra
- HBase
- ScyllaDB
- Amazon DynamoDB (also supports key-value model)

Use Cases:

- **Time-Series Data:** Ideal for time-series data, logging, and real-time analytics.
- **Wide-Column Data Models:** Suitable for use cases involving wide-column data structures, like user profiles and sensor data.
- **Scalability:** When horizontal scalability and high availability are critical.

Pros:

- High write and read performance.
- Can handle large volumes of data across many servers.
- Good for real-time analytics and big data applications.
- Flexible schema design.

Cons:

- Complex setup and maintenance.
- Querying can be less intuitive compared to relational databases.
- Limited support for ACID transactions.

Choosing the Right Database

- **Relational Databases:** Choose for structured data, strong ACID compliance, and applications needing complex queries and transactions.
- **Document-Based Databases:** Opt for flexibility, rapid development, unstructured/semi-structured data, and applications that don't require complex transactions.
- **Graph Databases:** Use for relationship-heavy data, complex graph queries, and applications involving networked or interconnected data.
- **Key-Value Databases:** Ideal for simple data models, caching, and high-speed transactions.
- **Column-Family Databases:** Suitable for time-series data, real-time analytics, and applications needing horizontal scalability.

Each type of database has its strengths and is suited to different scenarios. The choice depends on the specific requirements of the application, including the nature of the data, the need for flexibility, and the complexity of the queries.

Horizontal Splitting vs. Vertical Splitting of a Data Structure

Horizontal Splitting (Sharding)

Definition:

- Horizontal splitting, also known as sharding, involves dividing a dataset into smaller, more manageable pieces called shards. Each shard contains a subset of the rows in the database but all the columns.

How it Works:

- In horizontal splitting, the data is distributed across multiple tables or databases based on specific criteria, such as ranges of values, hash functions, or geographic regions.
- Each shard contains the same schema as the original table but only a portion of the data.

Example:

- Consider a user database for a social media application. To shard the data horizontally, users with IDs from 1 to 100,000 might be stored in one shard, those with IDs from 100,001 to 200,000 in another, and so on.

Use Cases:

- **Scalability:** When the dataset grows large and needs to be distributed across multiple servers to handle the load.
- **Performance:** To improve performance by reducing the amount of data each query needs to process.
- **Geographical Distribution:** When data needs to be located closer to users to reduce latency.

Pros:

- Can handle very large datasets by distributing the load across multiple servers.
- Improves query performance and reduces latency.
- Enables easier scaling by adding more shards as data grows.

Cons:

- Adds complexity in terms of data distribution and management.
- Can lead to uneven distribution if the sharding key is not chosen carefully.
- Requires handling cross-shard queries and transactions.

Vertical Splitting (Vertical Partitioning)

Definition:

- Vertical splitting, or vertical partitioning, involves dividing a dataset by splitting the columns of a table into multiple tables. Each table contains a subset of the columns but all the rows.

How it Works:

- In vertical splitting, the original table is partitioned into smaller tables, each containing a subset of the columns from the original table. The tables are linked by a common key, typically the primary key of the original table.

Example:

- Consider a user profile table with columns for user ID, name, email, address, and preferences. This table could be split vertically into two tables: one with columns for user ID, name, and email, and another with columns for user ID, address, and preferences.

Use Cases:

- **Optimization:** To optimize performance by separating frequently accessed columns from less frequently accessed ones.
- **Security:** To improve security by isolating sensitive data in a separate table with restricted access.
- **Data Management:** To make data management and maintenance more efficient by grouping related columns together.

Pros:

- Improves query performance by reducing the amount of data scanned for specific queries.
- Enhances security by isolating sensitive data.
- Simplifies data management and maintenance.

Cons:

- Requires joining tables to reconstruct the original dataset for certain queries.
- Can complicate database design and require careful planning.
- May introduce additional overhead for maintaining consistency between tables.

Json example

```
[
  {
    "Address": "123 Maple Street",
    "Zip": "12345",
    "City": "Springfield",
    "Occupants": [
      {
        "FirstName": "John",
        "LastName": "Doe",
        "Spouse": {
          "FirstName": "Jane",
          "LastName": "Doe",
          "Age": 35
        },
        "Age": 37
      },
      {
        "FirstName": "Jane",
        "LastName": "Doe",
        "Spouse": {
          "FirstName": "John",
          "LastName": "Doe",
          "Age": 37
        },
        "Age": 35
      }
    ],
    "Owners": [
      {
        "FirstName": "Alice",
        "LastName": "Smith",
        "Spouse": {
          "FirstName": "Bob",
          "LastName": "Smith",
          "Age": 45
        },
        "Age": 42
      },
      {
        "FirstName": "Bob",
        "LastName": "Smith",
        "Spouse": {
          "FirstName": "Alice",
          "LastName": "Smith",
          "Age": 42
        },
        "Age": 45
```

```
      }
    ]
  },
  {
    "Address": "456 Oak Avenue",
    "Zip": "67890",
    "City": "Greendale",
    "Occupants": [
      {
        "FirstName": "Michael",
        "LastName": "Johnson",
        "Spouse": {
          "FirstName": "Sarah",
          "LastName": "Johnson",
          "Age": 32
        },
        "Age": 34
      },
      {
        "FirstName": "Sarah",
        "LastName": "Johnson",
        "Spouse": {
          "FirstName": "Michael",
          "LastName": "Johnson",
          "Age": 34
        },
        "Age": 32
      }
    ],
    "Owners": [
      {
        "FirstName": "David",
        "LastName": "Brown",
        "Spouse": {
          "FirstName": "Emma",
          "LastName": "Brown",
          "Age": 50
        },
        "Age": 48
      },
      {
        "FirstName": "Emma",
        "LastName": "Brown",
        "Spouse": {
          "FirstName": "David",
          "LastName": "Brown",
          "Age": 48
```

```
        },
        "Age": 50
      }
    ]
  }
]
```

Compact Regex Cheat Sheet

Basic Characters

- . : Any character except newline
- \d : Digit (0-9)
- \D : Non-digit
- \w : Word character (a-z, A-Z, 0-9, _)
- \W : Non-word character
- \s : Whitespace (space, tab, newline)
- \S : Non-whitespace

Anchors

- ^ : Start of string or line
- $: End of string or line
- \b : Word boundary
- \B : Non-word boundary

Quantifiers

- * : 0 or more
- + : 1 or more
- ? : 0 or 1
- {n} : Exactly n times
- {n,} : n or more times
- {n,m} : Between n and m times

Groups and Ranges

- [] : Character set
    - [a-z] : a to z
    - [A-Z] : A to Z
    - [0-9] : 0 to 9
    - [^...] : Negation (not in the set)
- () : Group
    - | : OR (alternation)
    - (abc|def) : abc or def

Escape Sequences

- \ : Escape special character
- \n : Newline
- \t : Tab

- \r : Carriage return

Special Constructs

- (?:...) : Non-capturing group
- (?=...) : Positive lookahead
- (?!...) : Negative lookahead
- (?<=...) : Positive lookbehind
- (?<!...) : Negative lookbehind

Examples

- Email: ^[\w.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$
- URL: ^https?:\/\/[^\s/$.?#].[^\s]*$
- IP Address: ^(?:\d{1,3}\.){3}\d{1,3}$
- Phone Number: ^\+?(\d{1,3})?[-.\s]?(\d{1,4})[-.\s]?(\d{1,4})[-.\s]?(\d{1,9})$

Flags

- i : Case-insensitive
- m : Multi-line mode
- s : Dotall mode (dot matches newline)
- x : Ignore whitespace in pattern

1. Finding the Word “cat”:
   \bcat\b

2. Matching Sentences Containing the Word “error”:
   [^.?!]*\berror\b[^.?!]*[.?!]

3. Finding Words Ending with “ing”:
   \b\w+ing\b

4. Matching Lines Starting with a Number:
   ^\d.*$

5. Finding the Word “regex” Case-Insensitive:
   \bregex\b

6. Matching Any Word Containing “123”:
   \b\w*123\w*\b

7. Finding All Words With Only Uppercase Letters:
   \b[A-Z]+\b

8. Matching Sentences Ending with a Question Mark:
   [^?]*\?

9. Finding Words That Start with “pre”:
   \bpre\w*\b

10. Matching Lines Containing an Email Address:
    ^.*[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}.*$

11. Finding the Word “important” Regardless of Case:
    \bimportant\b

12. Matching Sentences That Contain a Number:
    [^.?!]*\d+[^.?!]*[.?!]

13. Finding Words Longer Than 10 Characters:
    \b\w{11,}\b

14. Matching Lines That End with a Period:
    ^.*\.$

15. Finding Words That Start and End with the Same Letter:
    \b(\w)\w*\1\b

16. Finding Words That Contain Double Letters:
    \b\w*(\w)\1\w*\b

17. Matching Sentences That Begin with “The”:
    ^The\b[^.?!]*[.?!]

18. Finding Words with Hyphens:
    \b\w+-\w+\b

19. Matching Lines with a URL:
    ^.*(https?|ftp)://[^\s/$.?#].[^\s]*.*$

20. Matching Sentences That Contain an Exclamation Point:
    [^.?!]*![^.?!]*[.?!]

21. Finding Words That Are Palindromes:
    \b(\w)(\w)?(\w)?\w?\3\2\1\b

Why index fast

Indexes are faster because they reduce the amount of data PostgreSQL needs to scan. Instead of performing a slow sequential scan on the entire table, PostgreSQL can use efficient data structures like B-trees or hash tables to quickly locate the relevant rows. Indexes store data in a sorted order, making range queries much faster, and can sometimes fulfill queries entirely without accessing the

actual table rows. This all leads to significantly improved query performance, especially for large tables and selective queries.

Shorthand

TableName1(PK Column1, Column2, Column3)
TableName2(PK Column1, Column2, FK Column3)

TableName1.Column1 -> TableName2.Column3 TableName1.Column1 -> TableName3.Column2

Employee(PK EmployeeId, FirstName, LastName, FK DepartmentId) Department(PK DepartmentId, Code, Name)

TimeOnProject(Pk TimeOnProjectId, Time, FK EmployeeId, FK ProjectId)

Project(PK ProjectId, ProjectNumber)

EmployeeId -> TimeOnProject.EmployeeId ProjectId -> TimeOnProject.ProjectId DepartmentId -> Employee.DepartmentId

SomeTable(PK(FK MyId, FK HerId), Amount)