Introduction to Relational Databases

KOENIG
step forward

Introduction to Relational Databases

- Overview of relational database systems
- Key concepts: Instances and Schemas, Tuples/Tables, Rows, Columns, and Relationships

Overview of relational database systems

- What is a Relational Database System?
  - A **Relational Database Management System (RDBMS)** is software that manages relational databases using **structured tables, relationships, and SQL (Structured Query Language)** for data manipulation.
  - It ensures **data integrity, consistency, and security**, making it a preferred choice for modern applications.

Overview of relational database systems

- Key Features of RDBMS
  - **Data Stored in Tables:** Data is structured in rows (records) and columns (fields).
  - **Relationships:** Establishes connections between tables using primary and foreign keys.
  - **SQL for Data Operations:** Standard language for querying and managing data.
  - **ACID Compliance:** Ensures **Atomicity, Consistency, Isolation, and Durability** in transactions.
  - **Data Integrity & Constraints:** Enforces rules like **Primary Key, Foreign Key, NOT NULL, UNIQUE, and DEFAULT** constraints.
  - **Scalability & Performance:** Supports indexing, partitioning, and replication for performance optimization.

Overview of relational database systems

- **Relational vs. non-relational databases**
  - The main difference between relational and non-relational databases (NoSQL databases) is how data is stored and organized.
  - Non-relational databases do not store data in a rule-based, tabular way.
  - Instead, they store data as individual, unconnected files and can be used for complex, unstructured data types, such as documents or rich media files.
  - Unlike relational databases, NoSQL databases follow a flexible data model, making them ideal for storing data that changes frequently or for applications that handle diverse types of data.

Overview of relational database systems

- Popular Relational Database Systems

RDBMS	Description
MySQL	Open-source, widely used for web applications (LAMP stack).
PostgreSQL	Advanced, open-source RDBMS with strong support for complex queries.
Microsoft SQL Server	Enterprise-grade, developed by Microsoft for business applications.
Oracle Database	High-performance, scalable, used for large enterprises.
MariaDB	Open-source fork of MySQL with additional performance features.

Key concepts : Instances and Schemas

- The collection of information stored in the database at a particular moment is called an **instance** of the database.
- The overall design of the database is called the database **schema**.
- The physical schema describes the database design at the physical level, while the logical schema describes the database design at the logical level.

Key concepts : Tuples/Tables

A table is a structured collection of related data stored in rows and columns. Each table represents an entity (e.g., Customers, Orders).

- Example Table: Customers

```
CREATE TABLE Customers (
   CustomerID INT PRIMARY KEY,
   Name VARCHAR(100),
   Email VARCHAR(255) UNIQUE,
   City VARCHAR(50)
);
```

  - CustomerID: Unique identifier (Primary Key).
  - Name: Stores the customer’s name.
  - Email: Stores unique customer emails.
  - City: Stores the customer’s city.

Key concepts: Rows/Records

- Rows (Records)
  - A row represents a single record in a table. Each row contains specific values for each column.
- Inserting Rows

```
INSERT INTO Customers (CustomerID, Name, Email, City)
VALUES (101, 'Alice', 'alice@email.com', 'New York');
```

- Fetching Rows

```
SELECT * FROM Customers;
```

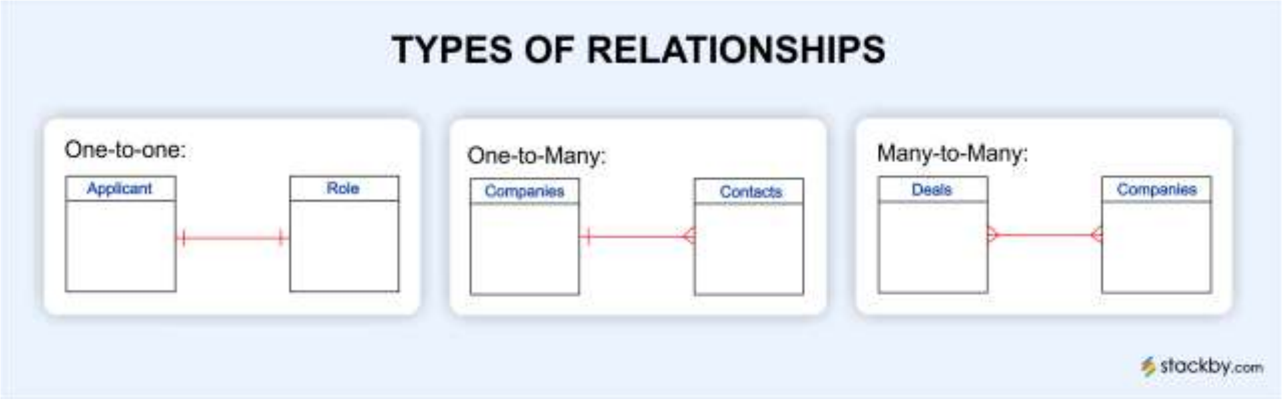
Key concepts: Columns/Attributes

- Columns (Fields/Attributes)
  - A column defines the type of data stored in a table. Each column has a name and a data type.
- Common Data Types in MS SQL Server

Data Type	Description
INT	Integer values
VARCHAR(n)	Variable-length text (n chars)
DATE	Date values
DECIMAL(p,s)	Fixed precision decimal

Key concepts: Relationships

- Relationships
  - Tables are linked using primary keys and foreign keys to establish relationships.
- Types of Relationships
  - One-to-One (1:1) – A row in Table A corresponds to only one row in Table B.
  - One-to-Many (1:M) – A row in Table A links to multiple rows in Table B.
  - Many-to-Many (M:N) – Multiple rows in Table A relate to multiple rows in Table B (via a junction table).



Data Integrity Principles

Data Integrity Principles

- Understanding data consistency, accuracy, and reliability
- Primary keys, foreign keys, and their role in maintaining integrity
- Referential integrity and cascading rules

Understanding data consistency, accuracy, and reliability

What is Data Integrity?

- Data integrity ensures that the data in a database is accurate, consistent, and reliable over its lifecycle.
- It prevents errors, duplication, and inconsistencies through constraints and rules.

Understanding data consistency, accuracy, and reliability

Key Principles of Data Integrity

Data Consistency

  - Ensures that data is uniformly stored across the database.
  - Maintains correct relationships between tables using foreign keys.
  - Example: If a customer places an order, the system ensures the CustomerID exists in the Customers table.
- Enforced by:
  - Foreign Keys
  - Transactions (ACID properties)

Understanding data consistency, accuracy, and reliability

Key Principles of Data Integrity

Data Accuracy

- Ensures that data is correct and follows business rules.
- Example: The email column should contain only valid and unique email addresses.

- Enforced by:
  - Constraints (e.g., UNIQUE, CHECK)
  - Data validation rules
- Example: Enforcing Accuracy with Constraints

```
ALTER TABLE Customers ADD CONSTRAINT chk_Email CHECK (Email LIKE '%@%.%');
```

  - Ensures every email follows a valid format.

Understanding data consistency, accuracy, and reliability

Key Principles of Data Integrity

Data Reliability

- Ensures that stored data remains trustworthy and free from corruption.
- Uses transactions to prevent partial updates in case of failures.

- Enforced by:
  - ACID (Atomicity, Consistency, Isolation, Durability)
  - Transactions (BEGIN TRANSACTION, COMMIT, ROLLBACK)
- Example: Using Transactions for Reliability

```
BEGIN TRANSACTION;
    INSERT INTO Customers (CustomerID, Name, Email) VALUES (104, 'David', 'david@email.com');
    INSERT INTO Orders (OrderID, CustomerID, OrderDate, Amount) VALUES (5, 104, '2025-03-15', 200.00);
COMMIT; -- Ensures both queries succeed together
```

If one query fails, no changes are saved, maintaining data reliability.

Primary keys, foreign keys, and their role in maintaining integrity

What is a Primary Key?

- A Primary Key (PK) uniquely identifies each record in a table.
- Must be unique for every row.
- Cannot contain NULL values.
- Ensures that no duplicate records exist in the table.

Example: Creating a Primary Key

```
CREATE TABLE Customers (
   CustomerID INT PRIMARY KEY,  -- Ensures uniqueness
   Name VARCHAR(100) NOT NULL,
   Email VARCHAR(255) UNIQUE NOT NULL
);
```

CustomerID is a Primary Key, ensuring each customer has a unique ID.

Primary keys, foreign keys, and their role in maintaining integrity

What is a Foreign Key?

- A Foreign Key (FK) establishes a relationship between two tables by referencing a primary key in another table.
- Enforces referential integrity.
- Prevents orphaned records (records that refer to non-existent values in another table).

Primary keys, foreign keys, and their role in maintaining integrity

Example: Foreign Key for Data Integrity

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,  -- Foreign Key
    OrderDate DATE NOT NULL,
    Amount DECIMAL(10,2) CHECK (Amount > 0),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

The CustomerID in Orders references the Primary Key CustomerID in Customers.

- Ensures that Orders cannot exist without a valid Customer.
- Prevents accidental deletion of Customers with existing Orders.

Primary keys, foreign keys, and their role in maintaining integrity

- Role of Primary and Foreign Keys in Maintaining Data Integrity

Integrity Type	Enforced By	Example
Uniqueness	Primary Key	Prevents duplicate CustomerID values
Referential Integrity	Foreign Key	Ensures every order has a valid customer
Consistency	Primary + Foreign Key	Prevents orphaned records
Accuracy	Foreign Key Constraints	Restricts invalid data entry

Referential integrity and cascading rules

What is Referential Integrity?

- Referential Integrity ensures that relationships between tables remain valid and consistent by enforcing foreign key constraints.

Why is it important?

- Prevents orphaned records (e.g., an order referencing a non-existent customer).
- Ensures data consistency across related tables.
- Controls deletion and updates of referenced records.

Referential integrity and cascading rules

- Example: Enforcing Referential Integrity

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Email VARCHAR(255) UNIQUE NOT NULL
);
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE NOT NULL,
    Amount DECIMAL(10,2) CHECK (Amount > 0),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

  - Orders cannot have a CustomerID that doesn’t exist in Customers.

Referential integrity and cascading rules

- Cascading Rules in MS SQL Server
  - Cascading rules define what happens when a referenced record in the parent table is updated or deleted.
- Types of Cascading Actions

Action	Effect
CASCADE	Automatically updates or deletes related records.
SET NULL	Sets foreign key column to NULL when the referenced record is deleted/updated.
SET DEFAULT	Sets foreign key column to a default value.
NO ACTION	Prevents update or delete if related records exist.

Referential integrity and cascading rules

- Cascade Delete: Automatically Remove Related Records
  - If a Customer is deleted, all related Orders are also deleted.
  - Example: ON DELETE CASCADE

```
ALTER TABLE Orders
ADD CONSTRAINT FK_Customer
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE CASCADE;
```

  - Now, deleting a customer automatically deletes their orders:

```
DELETE FROM Customers WHERE CustomerID = 101;
```

  - Effect: All orders of CustomerID = 101 are also deleted.

Database Schema Artifacts

Database Schema Artifacts

- Tables and views
- Indexes, constraints, and triggers
- Entity-relationship (ER) diagrams: visualizing schema design

Tables and views

- What are Database Schema Artifacts?
  - Database schema artifacts are structures that define and organize data in a relational database. The two primary schema artifacts are:
  - Tables → Store structured data.
  - Views → Virtual tables derived from queries.
- Tables
  - A table is a database object that stores data in rows and columns.
- Key Features of Tables
  - Columns define the attributes (e.g., CustomerID, Name).
  - Rows store individual records.
  - Primary Key ensures unique identification.
  - Foreign Key establishes relationships.

Tables and views

- Views
  - A view is a virtual table based on a query.
  - Does not store data physically.
  - Provides a customized view of data from one or more tables.
  - Enhances security by restricting access to certain columns.
- Key Features of Views
  - Simplifies complex queries.
  - Restricts access to sensitive data.
  - Improves readability and maintainability.

Tables and views

- Example: Creating a View for Active Customers

```
CREATE VIEW ActiveCustomers AS
SELECT CustomerID, Name, Email
FROM Customers
WHERE CreatedAt >= DATEADD(MONTH, -6, GETDATE());
```

  - This view shows customers added in the last 6 months.
- Using Views in Queries
  - You can use a view just like a table in SELECT queries:

```
SELECT * FROM ActiveCustomers;
```

  - Views act like tables but fetch data dynamically.

Indexes, constraints, and triggers

- Indexes in MS SQL Server
  - Indexes speed up query performance by allowing the database to find data faster.
- Types of Indexes

Index Type	Description	Example
Clustered Index	Sorts and stores rows in physical order (one per table).	PRIMARY KEY creates a Clustered Index automatically.
Non-Clustered Index	Stores a separate data structure for fast lookups.	CREATE INDEX idx_email ON Customers (Email);
Unique Index	Ensures no duplicate values in a column.	CREATE UNIQUE INDEX idx_unique_email ON Customers (Email);
Full-Text Index	Improves text searches.	CREATE FULLTEXT INDEX ON Products(ProductDescription);

Indexes, constraints, and triggers

- Example: Creating an Index

```
CREATE INDEX idx_customer_name ON Customers (Name);
```

- Improves performance for queries like:

```
SELECT * FROM Customers WHERE Name = 'Alice';
```

Indexes, constraints, and triggers

- Constraints in MS SQL Server
  - Constraints enforce data integrity by restricting invalid data.
- Types of Constraints

Constraint	Description	Example
Primary Key (PK)	Ensures a unique identifier for each row.	PRIMARY KEY (CustomerID)
Foreign Key (FK)	Enforces referential integrity between tables.	FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
Unique	Prevents duplicate values in a column.	UNIQUE (Email)
Not Null	Ensures a column cannot have NULL values.	Name VARCHAR(100) NOT NULL
Check	Validates data based on a condition.	CHECK (Amount > 0)
Default	Assigns a default value if none is provided.	CreatedAt DATETIME DEFAULT GETDATE()

Indexes, constraints, and triggers

- Example: Creating Constraints

```
CREATE TABLE Orders (
   OrderID INT PRIMARY KEY,
   CustomerID INT,
   Amount DECIMAL(10,2) CHECK (Amount > 0),
   CreatedAt DATETIME DEFAULT GETDATE(),
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE CASCADE
);
```

Indexes, constraints, and triggers

- Triggers in MS SQL Server
  - Triggers are automated actions that execute before or after INSERT, UPDATE, or DELETE operations.
- Types of Triggers

Trigger Type	Description	Example
AFTER INSERT	Runs after inserting a row.	Logs new customers in an audit table.
AFTER UPDATE	Runs after an update operation.	Tracks changes to product prices.
AFTER DELETE	Runs after deleting a row.	Archives deleted orders.
INSTEAD OF	Replaces an operation with custom logic.	Prevents deletion of VIP customers.

Indexes, constraints, and triggers

- Example: Trigger to Log Deleted Orders

```
CREATE TABLE OrderHistory (
   OrderID INT,
   CustomerID INT,
   DeletedAt DATETIME DEFAULT GETDATE()
);
CREATE TRIGGER trg_AfterDelete ON Orders
AFTER DELETE
AS
BEGIN
   INSERT INTO OrderHistory (OrderID, CustomerID, DeletedAt)
   SELECT OrderID, CustomerID, GETDATE() FROM deleted;
END;
```

- Automatically logs deleted orders into OrderHistory.

Entity-relationship (ER) diagrams: visualizing schema design

- What is an ER Diagram?
  - An Entity-Relationship (ER) diagram is a visual representation of a database schema, showing entities (tables), attributes (columns), and relationships (foreign keys).
- Key Uses:
  - Design database schemas before implementation.
  - Understand relationships between tables.
  - Ensure data integrity with proper constraints.

Entity-relationship (ER) diagrams: visualizing schema design

- Key Components of ER Diagrams

ER Component	Description	Example
Entity (Table)	Represents a real-world object with attributes.	Customers, Orders
Attribute (Column)	A property of an entity.	CustomerID, Name, Email
Primary Key (PK)	Uniquely identifies each record.	CustomerID (PK)
Foreign Key (FK)	Links tables by referencing PKs.	CustomerID in Orders
Relationship	Defines associations between entities.	"Customer places Order"

Entity-relationship (ER) diagrams: visualizing schema design

- Generating an ER Diagram in SQL Server Management Studio (SSMS)
  - Steps to Generate ER Diagram in SSMS:
  1. Open SSMS and connect to your database.
  2. Navigate to Databases → YourDatabase → Database Diagrams.
  3. Right-click and select New Database Diagram.
  4. Add Tables to visualize relationships.
  5. Save the diagram.

Schema Design Best Practices

Schema Design Best Practices

- Normalization: achieving optimal data structure
- Denormalization: balancing performance and storage needs
- Handling complex relationships with joins and constraints

Normalization: achieving optimal data structure

What is Normalization?

- Normalization is the process of structuring a database to reduce redundancy and improve data integrity by organizing data into multiple tables with relationships.

- Key Benefits:
  - Eliminates duplicate data → Saves storage.
  - Ensures data consistency → Avoids anomalies.
  - Improves query performance → Efficient joins and indexing.

Normalization: achieving optimal data structure

- 1st Normal Form (1NF) – Remove Duplicates & Atomicity
  - Rules:
    - Each column has atomic values (no multiple values in one field).
    - Each row has a unique identifier (Primary Key).
- Bad Example (Repeating Columns):

CustomerID	Name	Phone1	Phone2
1	Alice	123-456789	987-654321

- Fixed (Separate Table for Phones):

Customers Table

CustomerID	Name
1	Alice

CustomerPhones Table

PhoneID	CustomerID	Phone
1	1	123-456789
2	1	987-654321

Normalization: achieving optimal data structure

- 2nd Normal Form (2NF) – Eliminate Partial Dependencies
  - Rules:
    - Meet 1NF conditions.
    - No partial dependency (all non-key attributes must depend on the full primary key).
  - Bad Example (OrderDetails contains ProductName, which depends only on ProductID, not OrderID):

OrderID	ProductID	ProductName	Quantity
101	P01	Laptop	2

- Fixed (Separate Products Table):
  - OrderDetails Table:

OrderID	ProductID	Quantity
101	P01	2

Products Table

ProductID	ProductName
P01	Laptop

Normalization: achieving optimal data structure

- 3rd Normal Form (3NF) – Eliminate Transitive Dependencies
  - Rules:
    - Meet 2NF conditions.
    - No transitive dependency (non-key attributes should depend only on the primary key).
  - Bad Example (City depends on ZipCode, not CustomerID):

CustomerID	Name	ZipCode	City
1	Alice	10001	New York

- Fixed (Separate ZipCodes Table):
  - Customers Table:

CustomerID	Name	ZipCode
1	Alice	10001

ZipCodes Table:

ZipCode	City
10001	New York

Denormalization: balancing performance and storage needs

- What is Denormalization?
  - Denormalization is the process of combining tables to reduce joins and improve query performance at the cost of some data redundancy.
  - It is often used when read performance is more critical than write efficiency.
- Why Use Denormalization?
  - Faster reads (fewer joins).
  - Simplifies complex queries.
  - Reduces expensive join operations in large datasets.
  - Increases redundancy.
  - More complex updates (data inconsistency risks).

Handling complex relationships with joins and constraints

- When working with relational databases, complex relationships between tables require joins for querying and constraints for data integrity.
- Types of Joins

Join Type	Description
INNER JOIN	Returns only matching records from both tables.
LEFT JOIN	Returns all records from the left table and matching records from the right table.
RIGHT JOIN	Returns all records from the right table and matching records from the left table.
FULL JOIN	Returns all records when there is a match in either table.
CROSS JOIN	Returns a Cartesian product of both tables.

Hands-on Activities

Hands-on Activities

- Designing a database schema for a sample use case

- Implementing primary and foreign key constraints
- Validating referential integrity using triggers and rules

Designing a database schema for a sample use case

- Scenario: E-Commerce System
- You need to design a relational database schema for an e-commerce platform that manages:
  - Customers
  - Products
  - Orders
  - Payments
  - Shipping

Designing a database schema for a sample use case

- Identifying Entities and Relationships
  - Entities & Attributes
    - Customers (CustomerID, Name, Email, Phone, Address)
    - Products (ProductID, Name, Price, StockQuantity)
    - Orders (OrderID, CustomerID, OrderDate, Status)
    - OrderDetails (OrderDetailID, OrderID, ProductID, Quantity, Subtotal)
    - Payments (PaymentID, OrderID, PaymentMethod, Amount, PaymentDate)
    - Shipping (ShippingID, OrderID, ShippingDate, TrackingNumber)
  - Relationships
    - Customers place Orders (1-to-Many)
    - Orders have multiple Products (Many-to-Many → OrderDetails table)
    - Orders have Payments (1-to-1)
    - Orders have Shipping (1-to-1)

Implementing primary and foreign key constraints

- Primary Key (PK)
  - A Primary Key uniquely identifies each record in a table.
  - It must be unique and not NULL.
  - Example:
    - CustomerID INT PRIMARY KEY IDENTITY(1,1)
- Foreign Key (FK)
  - A Foreign Key ensures referential integrity by linking a column to another table's Primary Key.
  - Example:
    - CONSTRAINT FK_Orders_Customers FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE CASCADE
  - If a Customer is deleted, their related Orders are also deleted.

Validating referential integrity using triggers and rules

- Referential integrity ensures that foreign keys reference valid primary key values in related tables.
- While FOREIGN KEY constraints enforce this automatically, triggers provide additional validation when complex logic is required.
- Enforcing Referential Integrity Using Triggers
- Triggers can be used to enforce referential integrity when:
  - You need custom validation beyond standard constraints.
  - You want to log violations instead of rejecting the transaction.
  - You need to prevent cascading deletes or updates under specific conditions.

Common Challenges and Solutions

Common Challenges and Solutions

- Avoiding redundancy and anomalies
- Strategies for evolving schemas without compromising data integrity

Avoiding redundancy and anomalies

- Problem: Data Redundancy and Anomalies
  - Redundancy: Storing duplicate data across multiple tables.
  - Anomalies: Issues with insertion, update, and deletion due to poor schema design.
- Solution: Normalization
  - Normalization eliminates redundancy and ensures data integrity by breaking data into smaller, related tables.
    - 1NF (First Normal Form): Remove duplicate columns and ensure atomicity.
    - 2NF (Second Normal Form): Ensure each table stores data related to a single entity.
    - 3NF (Third Normal Form): Remove transitive dependencies (non-key attributes should not depend on other non-key attributes).

Strategies for evolving schemas without compromising data integrity

- Problem: Schema Changes Affecting Data Integrity
  - Adding new columns without breaking existing applications.
  - Modifying column types while preserving data.
- Solution: Schema Evolution Strategies
  - Use ALTER TABLE for Small Changes
  - Using DEFAULT Values to Prevent NULL Issues
  - Creating Views for Backward Compatibility
  - Using Triggers for Data Transformation