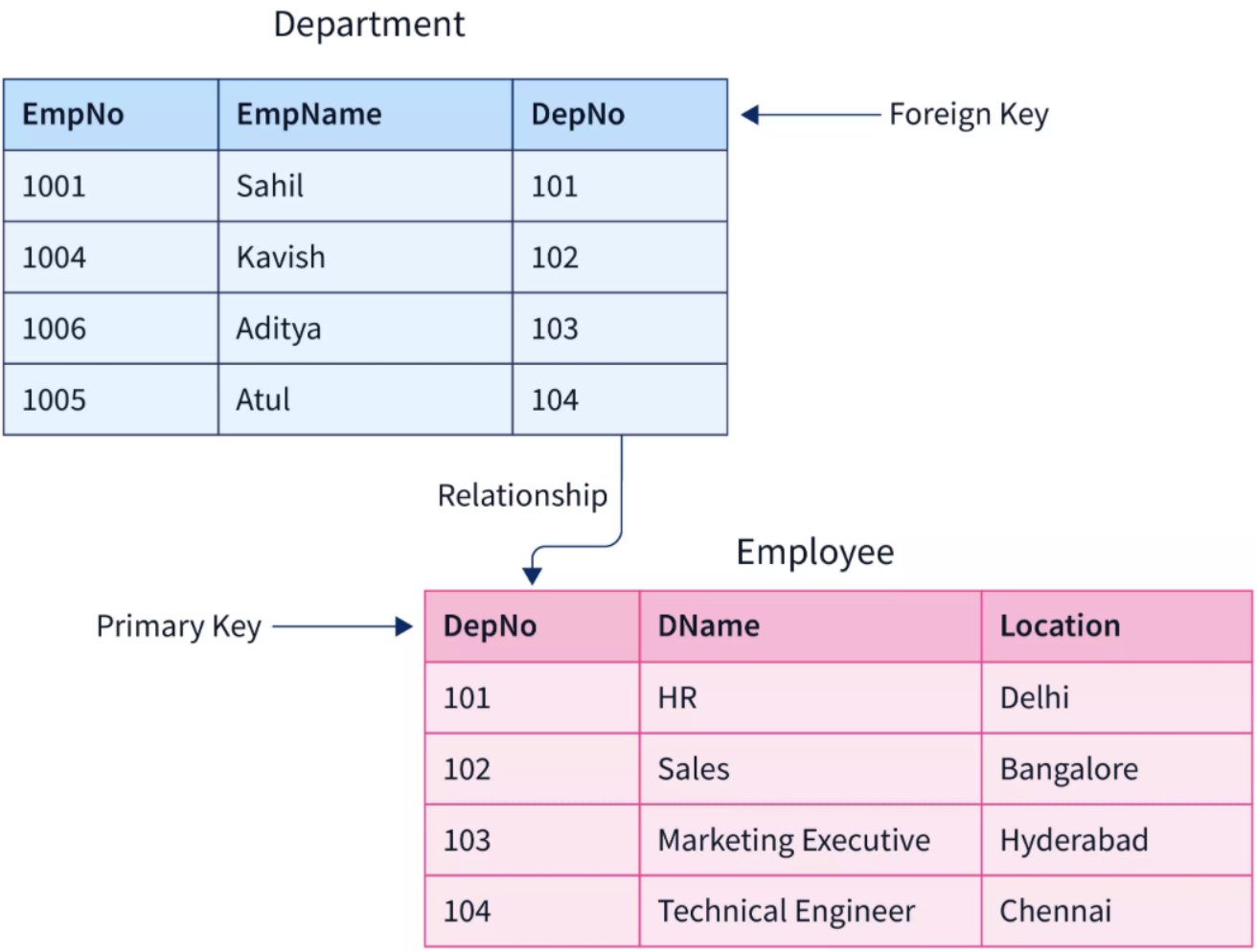
8-Tables Keys & Relationships

Author: Vincent Lau

Learning Objectives

- Understand different keys in table, such as Primary Key, Foreign Key & Unique.
- Understand constraints in table, such as DEFAULT, NOT NULL, UNIQUE, etc.
- Understand Constraint in table.
- Understand 3 types of table relationships.

Introduction



In a relational database, table relationships define how tables are connected or related to each other based on their data. There are three common types of table relationships: one-to-many, one-to-one, and many-to-many.

Table Keys

In the context of relational databases, primary keys, foreign keys, super keys, and candidate keys are important concepts that play roles in ensuring data integrity, establishing relationships, and uniquely identifying records within tables. Here's an introduction to each of these concepts:

Super Key

A super key is a set of one or more columns that can be used to uniquely identify records within a table. It may include more columns than the minimum required for uniqueness. Any subset of a super key is also a super key. Super keys help identify candidate keys.

In a `Customers` table, both `(customer_id)` and `(customer_id, email)` are super keys. **The second one is a super key because it uniquely identifies records, even though it contains more than the minimal columns needed for uniqueness.**

Candidate Key

A candidate key is a minimal set of columns that can be used to uniquely identify each record in a table. It is a subset of a super key that does not have any unnecessary attributes. A table can have multiple candidate keys. Also, candidate keys are the potential primary keys for the tables.

In a `Customers` table, `(customer_id)` is a candidate key, as it uniquely identifies each customer. **If `(email)` was also unique for each customer, then it could also be a candidate key.**

These concepts are fundamental to designing well-structured databases that ensure data integrity, enforce relationships, and facilitate efficient data retrieval.

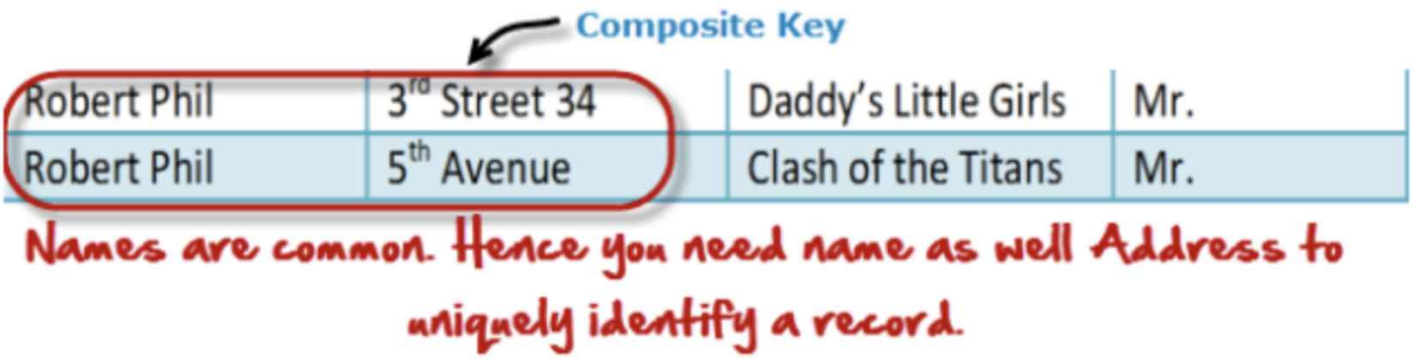
Primary Key

A primary key is a column or set of columns in a table that uniquely identifies each row (record) in the table. It ensures that each record has a unique identifier, and it also enforces data integrity by preventing duplicate or null values.

In this example, the `student_id` column is the primary key of the `Students` table. Each student record is uniquely identified by their `student_id`.

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);
```

A **composite key** is a primary key composed of **multiple columns used to identify a record uniquely.**



Foreign Key

A foreign key is a column in a table that establishes a link between data in two tables. It creates a relationship between tables by referencing the primary key of another table. This allows you to enforce referential integrity, ensuring that values in the foreign key column correspond to values in the primary key column of the referenced table.

In this example, the `student_id` and `course_id` columns in the `Enrollments` table are foreign keys that reference the `student_id` column in the `Students` table and the `course_id` column in the `Courses` table, respectively.

```
CREATE TABLE Enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES Students(student_id),
    FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
```

If you insert a record with a value (Foreign Key Column) not existing in reference table, the database will throw an SQL error.

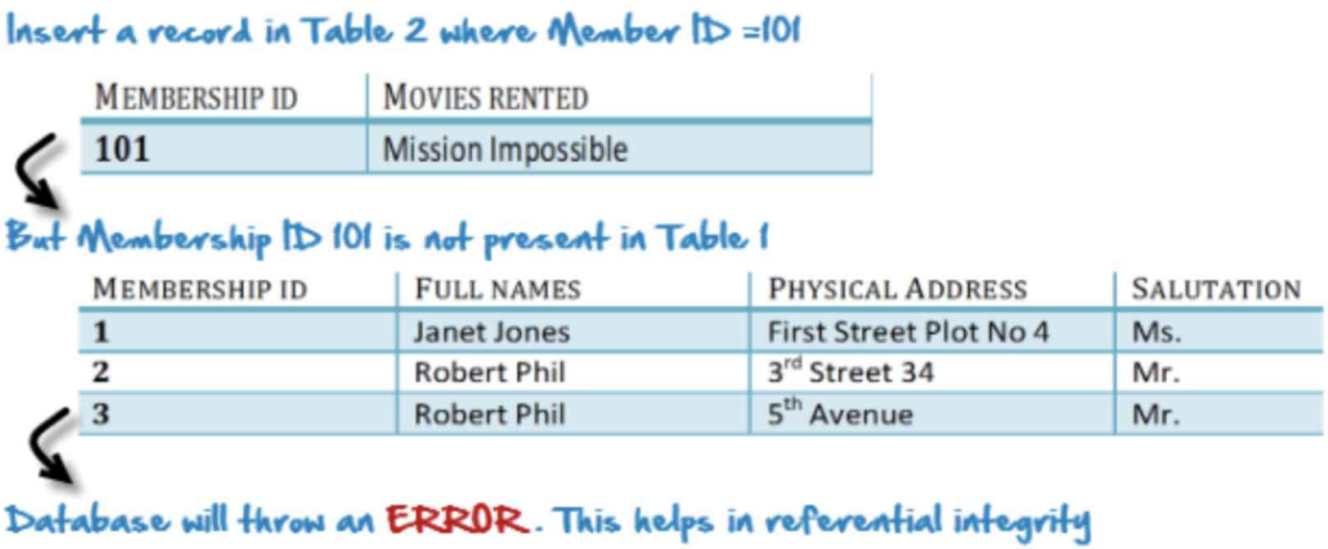


Table Constraints

Constraints are rules that you define to enforce data integrity and maintain the validity and consistency of your database. Constraints are applied to columns in database tables to ensure that certain conditions are met.

Here are some common types of constraints along with code examples:

NOT NULL

The `NOT NULL` constraint ensures that a column cannot contain a `NULL` value.

In this example, the `first_name` , `last_name` , and `hire_date` columns must have non-null values.

```sql
-- Add a NOT NULL constraint during create table
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY, -- primary key has Not null constraint
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    hire_date DATE NOT NULL
);
```

UNIQUE

The `UNIQUE` constraint ensures that values in a column are unique across all rows in the table.

In this example, both the `email` and `student_code` columns must contain unique values.

```sql
-- Add a UNIQUE constraint during create table
CREATE TABLE Students (
    student_id INT PRIMARY KEY, -- UNIQUE + NOT NULL + Indexing
    email VARCHAR(100) UNIQUE,
    student_code VARCHAR(20) UNIQUE
);

-- Add a UNIQUE constraint by ALTER
ALTER TABLE Students ADD CONSTRAINT unique_email UNIQUE (email);
```

PRIMARY KEY

The `PRIMARY KEY` constraint defines the primary key for a table. It enforces uniqueness and non-null values for the specified column(s).

In this example, the `product_id` column is designated as the primary key.

```sql
-- Add a PRIMARY KEY constraint during create table
CREATE TABLE Products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    price DECIMAL(10, 2)
);

-- Add a PRIMARY KEY constraint by ALTER
ALTER TABLE Products ADD CONSTRAINT pk_product_id PRIMARY KEY (product_id);
```

FOREIGN KEY

The `FOREIGN KEY` constraint establishes a link between data in two tables, enforcing referential integrity.

In this example, the `customer_id` column references the `customer_id` column in the `Customers` table.

```
-- Add a FOREIGN KEY constraint during create table
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);

-- Add a FOREIGN KEY constraint by ALTER
ALTER TABLE Orders ADD CONSTRAINT fk_customer_id FOREIGN KEY (customer_id)
REFERENCES Customers(customer_id);
```

CHECK

The `CHECK` constraint defines a condition that values in a column must satisfy.

In this example, the `salary` column must have a positive value due to the `CHECK` constraint.

```
-- Add a CHECK constraint during create table
CREATE TABLE Employees (
    employee_id INT PRIMARY KEY,
    salary DECIMAL(10, 2) CHECK (salary > 0),
    employment_status ENUM('Full-Time', 'Part-Time', 'Contract')
);

-- Add a CHECK constraint
ALTER TABLE Employees ADD CONSTRAINT positive_salary CHECK (salary > 0);
```

DEFAULT

The `DEFAULT` constraint assigns a default value to a column when no value is provided during an `INSERT` operation.

In this example, if no due date is provided, the default value of '2023-12-31' will be used.

```
-- Add a DEFAULT constraint during create table
CREATE TABLE Tasks (
```

```
    task_id INT PRIMARY KEY,
    task_name VARCHAR(100),
    due_date DATE DEFAULT '2023-12-31'
);

-- Add a DEFAULT constraint
ALTER TABLE Tasks ALTER COLUMN due_date SET DEFAULT '2023-12-31';
```

Constraints play a crucial role in maintaining data quality and integrity in a relational database. By using constraints, you ensure that the data in your tables adheres to specific rules and conditions, reducing the chances of errors and inconsistencies.

Table Relationships

One-to-One

In a one-to-one relationship, each record in the first table is related to exactly one record in the second table, and vice versa.

For example, you might have a `User` table and a `Profile` table. Each user has a unique profile, and each profile belongs to a single user.

```
CREATE TABLE User (
    user_id INT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100)
);

CREATE TABLE Profile (
    profile_id INT PRIMARY KEY,
    user_id INT UNIQUE,
    full_name VARCHAR(100),
    FOREIGN KEY (user_id) REFERENCES User(user_id)
);
```

One-to-Many

In a one-to-many relationship, one record in the first table is related to multiple records in the second table, but each record in the second table is related to only one record in the first table.

For example, consider a scenario where you have a `Department` table and an `Employee` table. Each department can have multiple employees, but each employee belongs to only one department.

```sql
CREATE TABLE Department (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(50)
);

CREATE TABLE Employee (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES Department(department_id)
);
```

Many-to-Many

In a many-to-many relationship, multiple records in the first table are related to multiple records in the second table, and vice versa.

Consider a `Student` table and a `Course` table. A student can be enrolled in multiple courses, and each course can have multiple students.

In this scenario, a student can be enrolled in multiple courses, and each course can have multiple students.

```sql
CREATE TABLE Student (
    student_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);

CREATE TABLE Course (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100)
);

CREATE TABLE Student_Course (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES Student(student_id),
    FOREIGN KEY (course_id) REFERENCES Course(course_id)
);
```

Implementing Relationships

To implement these relationships in a relational database, you typically use primary and foreign keys:

One-to-One

One of the tables includes a foreign key that references the primary key of the other table. For instance, the `Profile` table might have a `user_id` column that references the `User` table's primary key.

One-to-Many

The "many" side table includes a foreign key that references the primary key of the "one" side table. For example, the `Employee` table would include a `department_id` column that references the `Department` table's primary key.

Many-to-Many

To implement a many-to-many relationship, you use a junction table (also known as a bridge or linking table). This table holds the foreign keys that reference the primary keys of the two related tables. For the `Student` and `Course` example, you would have a `Student_Course` junction table with `student_id` and `course_id` columns.

Understanding and correctly implementing these relationships is crucial for maintaining data integrity and efficiently querying and retrieving related data in a relational database.