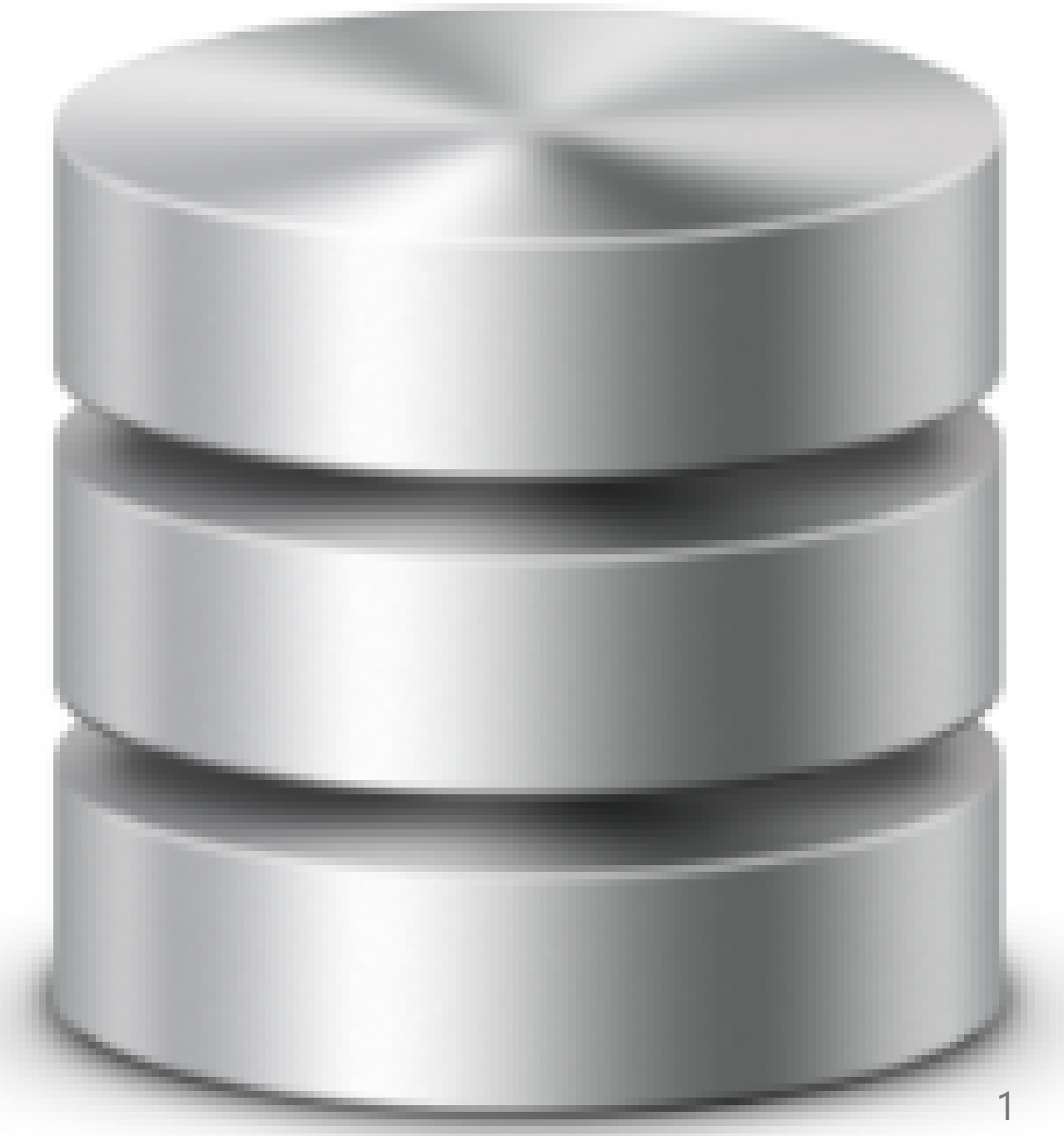
GCSE

Relational Databases



GCSE

Contents

GCSE

Key Concepts

Objectives

- Explain the concept of a database.
- Explain the concept of a relational database.
- Understand key database concepts: tables, records, fields, data types, primary keys, foreign keys.
- Learn how relational databases help eliminate data inconsistency and redundancy.

What is a Database?

- A database is an organized collection of data that allows for easy access, management, and updating.
- **Examples:**
  - Schools store student information (e.g., grades).
  - Online stores like Amazon track products, orders, and inventory.

Relational Databases

- A **relational database** stores data in tables and links them using common fields.
- **Key Features:**
  - **Tables** store data in rows and columns.
  - **Relationships** connect tables using keys.

Example:

- `Students` and `Classes` tables linked by a common student ID.

Key Concepts: Table

- A **Table** is a collection of data organized in rows and columns.
- **Example:**
  The `Student` table might include columns like StudentID, Name, Age, and Address.

Key Concepts: Record (Row)

- A **Record** is a single set of information in a table.
- **Example:**
  One row in the `Student` table could be "John Doe, Age 16, Address: 123 Street."

Key Concepts: Field (Column)

- A **Field** is a specific attribute in a table.
- **Example:**
  Fields in `Student` could be Name, Age, and Address.

Key Concepts: Data Types

- **Data Types** specify the kind of data stored in a field.

Common Types:

- **Text:** Names, addresses.
- **Integer:** Whole numbers (e.g., Age).
- **Decimal:** Numbers with decimals (e.g., prices).
- **Date/Time:** Dates and times.

Key Concepts: Primary Key

- A **Primary Key** is a unique identifier for each record in a table.
- **Example:**
  `StudentID` in the `Student` table.

Key Concepts: Foreign Key

- A **Foreign Key** links tables together by referencing the primary key of another table.
- **Example:**
  `StudentID` in `Classes` table links to the `Student` table.

Data Redundancy & Inconsistency

- **Data Redundancy:** Storing the same data in multiple places.
- **Data Inconsistency:** Conflicting copies of the same data.

Example:

- Storing a student's contact info in both `Student` and `Classes` can cause errors if updated inconsistently.

Eliminating Redundancy & Inconsistency

- Relational databases store each piece of data once, reducing redundancy.
- Changes to a record (e.g., address) need only be updated in one table.

Example:
The `Student` table holds the address, while `Classes` stores a foreign key linking the student.

Summary

- **Databases** organize data for easy access and management.
- **Relational databases** link data across tables using relationships.
- Concepts like **tables, records, fields, primary keys**, and **foreign keys** are essential.
- They help eliminate **redundancy** and prevent **inconsistency**.

Worked Example

GCSE

Objectives

- Understand relational databases.
- Convert flat-file data into a relational database.

Understanding Flat-File Systems

A flat-file system stores all data in one table (e.g., a spreadsheet). It can lead to redundancy (duplicate data) and inconsistency (conflicting data).

Example Spreadsheet:

StudentID	Name	Age	ClassID	Class Name	Teacher	Room
1	John	16	C001	Maths	Mr. Smith	101
1	John	16	C002	Science	Mrs. Johnson	102
2	Jane	15	C001	Maths	Mr. Smith	101

Problems with Flat-File Systems

- **Redundancy:** Duplicate data (e.g., student info repeated for each class).
- **Inconsistency:** Conflicting data if changes aren't applied uniformly.
- **Limited Scalability:** Difficult to manage as data grows.

Converting to a Relational Database

To solve these problems, split data into **related tables**:

- **Student Table:** Stores student details.
- **Class Table:** Stores class information.

NB. It is recommended to name the tables using a singular form

Example Tables

`Student` Table:

StudentID (PK)	Name	Age
1	John	16
2	Jane	15

`Class` Table:

ClassID (PK)	Class Name	Teacher	Room
C001	Maths	Mr. Smith	101
C002	Science	Mrs. Johnson	102

Creating Relationships

But, now we've lost the connection between the two entities!

We need to put that back by using a **linking table** to represent the **Many-to-Many** relationship between students and classes.

`Enrollment` **Table:**

StudentID (FK)	ClassID (FK)
1	C001
1	C002
2	C001

Degrees of relationship

- **One-to-One**
  Each record in Table A relates to **one** record in Table B.
- **One-to-Many**
  One record in Table A relates to **many** records in Table B.
- **Many-to-Many**
  Many records in Table A relate to many records in Table B. Requires a linking table.

Database Relationships

Degrees of relationships:

- **One-to-One**: Rare, usually an attribute of one table.
- **One-to-Many**: Example: One teacher teaches many subjects.
- **Many-to-Many**: Example: Many students enroll in many classes (resolved with a linking table).

How to Determine the Relationship

Ask two questions:

1. **One record in Table A relates to how many records in Table B?**
2. **One record in Table B relates to how many records in Table A?**

The answers will guide you to either **One-to-One**, **One-to-Many**, or **Many-to-Many**.

Data Redundancy & Inconsistency Solved

- **Reduced Redundancy:** Data is only stored once (e.g., student details in `Student` table).
- **Prevented Inconsistency:** Changes are made in one place, reflected in all related records.

Example - Teacher & Subject Relationship

`Teacher` Table:

TeacherID (PK)	Name	SubjectID (FK)
1	Mr. Smith	1
2	Mrs. Johnson	2

`Subject` Table:

SubjectID (PK)	Subject
1	Maths
2	Science

Data Types

Each field has a **data type**:

- **StudentID:** Integer.
- **Teacher Name:** String.
- **ClassID:** Alphanumeric.

Summary

1. **Reduces Redundancy:** Data is stored once.
2. **Prevents Inconsistency:** Updates are easy and uniform.
3. **Efficient Management:** Easy to scale and manage.

Key Terms

Term	Definition
Table	Collection of records (rows).
Record	A single row of related data.
Field	An individual column of data.
Primary Key	Unique identifier in a table.
Foreign Key	A field linking to a primary key in another table.

Worked Example Recap

- Start with a flat-file (spreadsheet).
- Identify redundancy and inconsistency issues.
- Break data into relational tables.
- Link data using foreign keys.

SQL

Objectives

- Learn how to use `SELECT`, `FROM`, and `WHERE` to retrieve and filter data.
- Use `ORDER BY` to sort results.
- Query multiple tables using joins or table references.
- Use `INSERT INTO` to add new data.
- Use `UPDATE` and `DELETE` to modify or remove records.

What is SQL?

SQL (Structured Query Language) is used to manage and interact with relational databases.

Key SQL operations:

- **Retrieve** data
- **Insert** new records
- **Update** existing data
- **Delete** data

SELECT Statement

Retrieve data from a database.

Basic Syntax:

```
SELECT column1, column2, ...
FROM table_name;
```

Example:

```
SELECT StudentName, Age
FROM Student;
```

This retrieves the `StudentName` and `Age` columns from the `Student` table.

WHERE Clause

Filter records based on a condition.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example:

```
SELECT StudentName, Age
FROM Student
WHERE Age > 15;
```

This retrieves students older than 15.

ORDER BY Clause

Sort the result set in ascending (`ASC`) or descending (`DESC`) order.

Syntax:

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC | DESC];
```

Example:

```
SELECT StudentName, Age
FROM Student
ORDER BY Age DESC;
```

This retrieves students sorted by age in descending order.

Querying Multiple Tables

Extract data from more than one table using joins or references.

Example:

```
SELECT Student.StudentName, Classes.ClassName
FROM Student, Enrollment, Classes
WHERE Student.StudentID = Enrollment.StudentID
AND Classes.ClassID = Enrollment.ClassID;
```

This retrieves `StudentName` and `ClassName` from related tables.

INSERT INTO Statement

Add new records to a table.

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO Student (StudentID, StudentName, Age)
VALUES (4, 'Emily Green', 17);
```

This inserts a new student into the `Student` table.

UPDATE Statement

Change existing data in a table.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:

```
UPDATE Student
SET Age = 18
WHERE StudentName = 'John Doe';
```

This updates the age of John Doe.

DELETE Statement

Remove records from a table.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example:

```
DELETE FROM Student
WHERE StudentID = 4;
```

This deletes the student with ID 4.

Summary of Key SQL Commands

Command	Purpose	Example
`SELECT`	Retrieve data	`SELECT StudentName FROM Students;`
`FROM`	Specifies the table	`SELECT * FROM Students;`
`WHERE`	Filters records	`SELECT * FROM Students WHERE Age = 16;`
`ORDER BY`	Sorts data	`SELECT * FROM Students ORDER BY Age DESC;`
`INSERT`	Adds new data	`INSERT INTO Students (StudentID,`

Here’s the entire slide deck based on the provided notes, formatted in markdown for MARP:

Python and SQL

Objectives"

- Set up and interact with an SQLite3 database using Python
- Create tables (`Students` and `Classes`), insert data, and establish relationships.
- Query the database to retrieve, update, and manage data.

Prerequisites

- Install Python 3 or higher
- Ensure `sqlite3` is available (comes pre-installed with Python)
- Basic knowledge of Python syntax

Part 1: Setting Up the Database

Step 1: Importing SQLite3 in Python

```
import sqlite3
```

- `import sqlite3` allows you to use SQLite3 in Python, which is a lightweight, file-based database system.

Step 2: Creating a Connection

```
conn = sqlite3.connect('school.db')
cursor = conn.cursor()
```

- `sqlite3.connect('school.db')` creates or connects to a SQLite database file.
- `cursor = conn.cursor()` creates a cursor to interact with the database.

Part 2: Creating Tables

Step 3: Create the `Students` Table

```
create_students_table = '''
CREATE TABLE IF NOT EXISTS Student (
    StudentID INTEGER PRIMARY KEY,
    StudentName TEXT NOT NULL,
    Age INTEGER
);
'''
cursor.execute(create_students_table)
conn.commit()
```

- Creates a `Student` table with `StudentID`, `StudentName`, and `Age`.

Step 4: Create the `Class` Table

```
create_class_table = '''
CREATE TABLE IF NOT EXISTS Class (
    ClassID TEXT PRIMARY KEY,
    ClassName TEXT NOT NULL,
    Teacher TEXT,
    RoomNumber TEXT
);
'''
cursor.execute(create_class_table)
conn.commit()
```

- Creates a `Class` table with `ClassID`, `ClassName`, `Teacher`, and `RoomNumber`.

Step 5: Create the `Enrollment` Table

```
create_enrollment_table = '''
CREATE TABLE IF NOT EXISTS Enrollment (
    StudentID INTEGER,
    ClassID TEXT,
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (ClassID) REFERENCES Class(ClassID)
);
'''
cursor.execute(create_enrollment_table)
conn.commit()
```

- `Enrollment` links students and classes using foreign keys.

Part 3: Inserting Data

Step 6: Insert Data into `Student` table

```
students = [
    (1, 'John Doe', 16),
    (2, 'Jane Roe', 15),
    (3, 'Sam White', 16)
]
cursor.executemany('INSERT INTO Student (StudentID, StudentName, Age) VALUES (?, ?, ?)', students)
conn.commit()
```

- Inserts sample students into the `Student` table.

Step 7: Insert Data into `Classes`

```
classes = [
    ('C001', 'Maths', 'Mr. Smith', '101'),
    ('C002', 'Science', 'Mrs. Johnson', '102'),
    ('C003', 'History', 'Mr. Brown', '201')
]
cursor.executemany('INSERT INTO Class (ClassID, ClassName, Teacher, RoomNumber) VALUES (?, ?, ?, ?)', classes)
conn.commit()
```

- Inserts sample data into the `Class` table.

Step 8: Insert Data into `Enrollment`

```
enrollments = [
    (1, 'C001'),
    (1, 'C002'),
    (2, 'C001'),
    (2, 'C003'),
    (3, 'C002')
]
cursor.executemany('INSERT INTO Enrollment (StudentID, ClassID) VALUES (?, ?)', enrollments)
conn.commit()
```

- Links students to classes using the `Enrollment` table.

Part 4: Querying the Database

Step 9: Retrieve Students and Classes

```
query = '''
SELECT Student.StudentName, Student.Age, Class.ClassName, Class.Teacher
FROM Enrollment
JOIN Student ON Enrollment.StudentID = Student.StudentID
JOIN Class ON Enrollment.ClassID = Class.ClassID;
'''
cursor.execute(query)
results = cursor.fetchall()

for row in results:
    print(f"Student: {row[0]}, Age: {row[1]}, Class: {row[2]}, Teacher: {row[3]}")
```

- Retrieves students and their enrolled classes.

Step 10: Retrieve Students in a Specific Class

```
query = '''
SELECT Student.StudentName, Student.Age
FROM Enrollment
JOIN Student ON Enrollment.StudentID = Student.StudentID
WHERE Enrollment.ClassID = 'C001';
'''
cursor.execute(query)
results = cursor.fetchall()

print("Students enrolled in Maths:")
for row in results:
    print(f"Student: {row[0]}, Age: {row[1]}")
```

- Retrieves students enrolled in a specific class (`Maths` in this case).

Part 5: Updating and Deleting Data

Step 11: Update Student Information

```
cursor.execute('UPDATE Student SET Age = 16 WHERE StudentName = "Jane Roe"')
conn.commit()

cursor.execute('SELECT * FROM Student WHERE StudentName = "Jane Roe"')
print(cursor.fetchone())
```

- Updates the age of a student (`Jane Roe`).

Step 12: Delete a Student

```
cursor.execute('DELETE FROM Student WHERE StudentName = "Sam White"')
conn.commit()

cursor.execute('SELECT * FROM Student')
print(cursor.fetchall())
```

- Deletes a student (`Sam White`).

Part 6: Closing the Database Connection

Close the connection

```
conn.close()
```

- Closes the database connection.

Summary

- Set up and managed an SQLite3 database using Python.
- Created `Student`, `Class`, and `Enrollment` tables.
- Inserted, retrieved, updated, and deleted data using SQL.