JOINS in MySQL

Joins in SQL are used to combine rows from two or more tables based on a related column between them. Here's a breakdown of the different types of joins with examples:

1. INNER JOIN

Explanation:
An INNER JOIN returns only the rows where there is a match in both tables. If there is no match, the row is not returned.

Example:

Consider two tables:

Students Table:

StudentID	Name	Class
1	John	10
2	Jane	12
3	Tom	10
4	Lucy	11

Grades Table:

GradeID	StudentID	Grade
1	1	A
2	3	B
3	4	A+
4	5	C

Query:

sql   Copy code

```sql
SELECT Students.Name, Grades.Grade
FROM Students
INNER JOIN Grades ON Students.StudentID = Grades.StudentID;
```

Result:

Name	Grade
John	A
Tom	B
Lucy	A+

Only the students who have grades in the Grades table are returned.

2. LEFT JOIN (or LEFT OUTER JOIN)

Explanation:
A LEFT JOIN returns all rows from the left table (Students), and the matched rows from the right table (Grades). If there is no match, NULL values are returned for columns from the right table.

Query:

```sql
SELECT Students.Name, Grades.Grade
FROM Students
LEFT JOIN Grades ON Students.StudentID = Grades.StudentID;
```

Result:

Name	Grade
John	A
Jane	NULL
Tom	B
Lucy	A+

All students are returned, but Jane has no grade, so NULL is shown.

3. RIGHT JOIN (or RIGHT OUTER JOIN)

Explanation:
A RIGHT JOIN returns all rows from the right table (Grades), and the matched rows from the left table (Students). If there is no match, NULL values are returned for columns from the left table.

Query:

```sql
SELECT Students.Name, Grades.Grade
FROM Students
RIGHT JOIN Grades ON Students.StudentID = Grades.StudentID;
```

Result:

Name	Grade
John	A
Tom	B
Lucy	A+
NULL	C

All grades are returned, but the grade C has no corresponding student, so NULL is shown.

4. OUTER JOIN (FULL OUTER JOIN)

Explanation:
A FULL OUTER JOIN returns all rows when there is a match in either the left or right table. Rows without a match in one of the tables will have NULL in the columns from the table with no match.

Note: MySQL does not directly support FULL OUTER JOIN, but you can achieve the same using a combination of LEFT JOIN and RIGHT JOIN.

Query:

```sql
SELECT Students.Name, Grades.Grade
FROM Students
LEFT JOIN Grades ON Students.StudentID = Grades.StudentID
UNION
SELECT Students.Name, Grades.Grade
FROM Students
RIGHT JOIN Grades ON Students.StudentID = Grades.StudentID;
```

Result:

Name	Grade
John	A
Jane	NULL
Tom	B
Lucy	A+
NULL	C

All rows from both tables are returned. The UNION operator combines the results of the LEFT JOIN and RIGHT JOIN.

Normalisation in Mysql

Normalization is a process in database design used to organize data to reduce redundancy and improve data integrity. It involves dividing a database into two or more tables and defining relationships between them to remove redundancy and dependency. There are several forms of normalization, each with specific rules and goals.

1NF (First Normal Form)

Rule:

- A table is in 1NF if all the columns contain atomic (indivisible) values and each column contains values of a single type. Additionally, each row must have a unique identifier (primary key).

Example:

Consider a table that records orders:

Orders Table (Unnormalized):

OrderID	Customer	Items
1	John Doe	Pen, Notebook
2	Jane Smith	Pencil, Eraser, Ruler
3	Sam Brown	Pen, Pencil

This table is not in 1NF because the Items column contains multiple values.

Orders Table (1NF):

OrderID	Customer	Item
1	John Doe	Pen
1	John Doe	Notebook
2	Jane Smith	Pencil
2	Jane Smith	Eraser
2	Jane Smith	Ruler
3	Sam Brown	Pen
3	Sam Brown	Pencil

Now, each field contains atomic values, and the table is in 1NF.

2NF (Second Normal Form)

Rule:

- A table is in 2NF if it is in 1NF and all non-key columns are fully dependent on the primary key. In other words, there should be no partial dependency of any column on the primary key (no column should depend on only a part of a composite primary key).

Example:

Consider a table that records student enrollments:

Enrollments Table (1NF):

StudentID	CourseID	StudentName	CourseName
1	101	John Doe	Math
1	102	John Doe	Science
2	101	Jane Smith	Math
2	103	Jane Smith	History

This table is in 1NF but not in 2NF because StudentName depends only on StudentID, and CourseName depends only on CourseID, which are part of the composite key (StudentID, CourseID).

Enrollments Table (2NF):

We split the table into two:

Students Table:

StudentID	StudentName
1	John Doe

StudentID	StudentName
2	Jane Smith

Courses Table:

CourseID	CourseName
101	Math
102	Science
103	History

Enrollments Table (New):

StudentID	CourseID
1	101
1	102
2	101
2	103

Now, all non-key columns are fully dependent on the composite key, and the tables are in 2NF.

3NF (Third Normal Form)

Rule:

- A table is in 3NF if it is in 2NF and all the columns are not only dependent on the primary key but also non-transitively dependent (no transitive dependency). In other words, no non-key column should depend on another non-key column.

Example:

Consider a table recording student information:

StudentInfo Table (2NF):

StudentID	StudentName	CourseID	InstructorID	InstructorName
1	John Doe	101	10	Mr. Smith
1	John Doe	102	11	Mrs. Johnson
2	Jane Smith	101	10	Mr. Smith

This table is in 2NF but not in 3NF because InstructorName depends on InstructorID, which is not a key.

StudentInfo Table (3NF):

We split the table into:

Instructors Table:

InstructorID	InstructorName
10	Mr. Smith
11	Mrs. Johnson

StudentInfo Table (New):

StudentID	StudentName	CourseID	InstructorID
1	John Doe	101	10
1	John Doe	102	11
2	Jane Smith	101	10

Now, all columns are non-transitively dependent on the primary key, and the tables are in 3NF.

BCNF (Boyce-Codd Normal Form)

Rule:

- A table is in BCNF if it is in 3NF and for every functional dependency X -> Y, X is a superkey (a set of columns that uniquely identifies a row).

Example:

Consider a table that records course assignments:

Assignments Table (3NF):

CourseID	InstructorID	InstructorExperience
101	10	10 years
102	11	5 years

Here, both CourseID and InstructorID form a composite key, but InstructorExperience depends only on InstructorID.

Assignments Table (BCNF):

We split the table into:

Courses Table:

CourseID	InstructorID
101	10
102	11

Instructors Table:

InstructorID	InstructorExperience
10	10 years
11	5 years

Now, each functional dependency has a superkey on the left side, and the tables are in BCNF.

This progression from 1NF to BCNF helps to systematically reduce redundancy and improve data integrity in database design.

ACID Properties

The **ACID** properties are a set of principles that ensure reliable processing of database transactions. These properties are crucial for maintaining the integrity and consistency of data in a Database Management System (DBMS), especially in environments where multiple transactions occur concurrently.

1. Atomicity

Definition:

- Atomicity ensures that each transaction is treated as a single "unit of work." This means that either all of the operations within the transaction are completed successfully, or none of them are. If any part of the transaction fails, the entire transaction is rolled back, and the database is left unchanged.

Example: Imagine a banking system where you are transferring $100 from Account A to Account B. This transaction involves two steps:

1. Deduct $100 from Account A.
2. Add $100 to Account B.

If the system crashes after deducting $100 from Account A but before adding it to Account B, Atomicity ensures that the transaction is rolled back, and no money is lost or partially transferred.

2. Consistency

Definition:

- Consistency ensures that a transaction brings the database from one valid state to another valid state. Any transaction will preserve the integrity constraints of the database. The data should always be in a consistent state before and after the transaction.

Example: In a university database, a rule might state that the total number of students enrolled in a course must not exceed the maximum capacity. Consistency ensures that after any transaction, such as enrolling a new student, the total number of students does not exceed this limit.

3. Isolation

Definition:

- Isolation ensures that the execution of one transaction is isolated from the others. In other words, the operations of one transaction cannot interfere with those of another transaction. The result of the transaction should not be affected by any other concurrently running transactions.

Example: Suppose two customers try to purchase the last available unit of a product online at the same time. Isolation ensures that only one of the transactions will succeed in purchasing the product, while the other transaction will be blocked until the first transaction is complete, thus preventing any data anomalies like overselling the product.

4. Durability

Definition:

- Durability ensures that once a transaction has been committed, it will remain in the system even in the event of a system crash. The changes made by the transaction are permanent and will not be lost.

Example: If a bank transaction confirming a withdrawal has been successfully completed and the system crashes immediately afterward, Durability guarantees that the withdrawal will not be lost, and the changes will be reflected in the account balance once the system is back up.

Summary of ACID Properties:

- **Atomicity:** Ensures all-or-nothing execution.
- **Consistency:** Guarantees that database constraints are not violated.
- **Isolation:** Protects concurrent transactions from interfering with each other.
- **Durability:** Ensures that committed transactions persist even in the case of a system failure.

Together, these properties help maintain the reliability, consistency, and integrity of a database, especially in environments with concurrent access and potential system failures.