**Advanced SQL for Data Engineer (Honor Module)**

This module covers some advanced SQL techniques that will be useful for Data Engineers. In this module, you will learn how to build more powerful queries with advanced SQL techniques like views, transactions, stored procedures, and joins. If you are following the Data Engineering track, you must complete this module. Completion of this module is not required for those completing the Data Science or Data Analyst tracks.

**Learning Objectives**

* Generate joins to query data from multiple tables
* Create and query a view
* Create stored procedures and invoke them from other code
* Describe the significance of ACID transactions
* Implement transactions in SQL statements
* Compare and contrast the benefits and disadvantages of using stored procedures
* Define views and describe the benefits they provide
* Compare and contrast the different JOIN operators

**About this Honors Module**

This module covers some advanced SQL techniques that will be useful for Data Engineers. In this module, you will learn how to build more powerful queries with advanced SQL techniques like views, transactions, stored procedures, and joins.

If you are following the Data Engineering track, you must complete this module. Completing this module is not required for those completing the Data Science or Data Analyst tracks.

# **Views, Stored Procedures, and Transactions**

## **Views**

**Understanding Views in SQL**

1. **Definition of a View**:
   * A view is an alternative way of representing data from one or more tables or existing views.
   * It can include all or some columns from the base tables.
2. **Purpose of Views**:
   * **Data Representation**: Views create a named specification of a results table that can be queried like a regular table.
   * **Data Modification**: You can run INSERT, UPDATE, and DELETE queries against a view to change data in the base tables.
   * **Data Security**: Views can be used to show a selection of data while omitting sensitive information (e.g., salaries, birth dates).
   * **Data Combination**: They allow combining data from multiple tables in meaningful ways.
   * **Simplified Access**: Grant access to a view without exposing the underlying tables.
3. **Creating a View**:
   * Use the CREATE VIEW statement.
   * Assign a name to the view (up to 128 characters).
   * Specify the columns to include, optionally using aliases.
   * Use the AS SELECT clause to define the columns and the FROM clause to specify the base table.
   * An optional WHERE clause can refine the rows included in the view.
4. **Example of Creating a View**:
   * A view named EMPINFO can be created based on the Employees table, showing only non-sensitive data like:
     + Employee ID
     + Name
     + Address
     + Job ID
     + Manager ID
     + Department ID
   * Sensitive data like salary or birthdate is omitted.
5. **Dynamic Nature of Views**:
   * Views are dynamic; they consist of data returned from the SELECT statement used to create them.
   * When used in another SQL statement, a view behaves like a SELECT statement that returns its content.
6. **Limitations**:
   * The SELECT statement used to create a view cannot use the ORDER BY clause or name a host variable.
7. **Removing a View**:
   * To remove a view completely, use the DROP VIEW statement.

**Key Takeaways**:

* Views are a powerful tool for data representation and security in SQL.
* They allow for simplified data access and can help in managing sensitive information effectively.

## **Stored Procedures**

Stored Procedures Overview

* **Definition**: A stored procedure is a set of SQL statements stored and executed on the database server. Instead of sending multiple SQL statements from the client to the server, you encapsulate them in a stored procedure and send one statement to execute them.

Key Features

* **Languages**: Stored procedures can be written in various languages, including:
  + SQL
  + PL/SQL
  + Java
  + C
  + Others (depending on the database system)

Functionality

* **Parameters**: Stored procedures can accept parameters, allowing for dynamic execution.
* **CRUD Operations**: They can perform Create, Read, Update, and Delete operations.
* **Return Results**: Stored procedures can return results to the client application.

Benefits of Using Stored Procedures

1. **Reduction in Network Traffic**: Only one call is needed to execute multiple statements, minimizing the amount of data sent over the network.
2. **Performance Improvement**: Processing occurs on the server where the data is stored, with only the final result sent back to the client.
3. **Code Reusability**: Multiple applications can use the same stored procedure, promoting code reuse.
4. **Increased Security**:
   * Reduces exposure of table and column information to client-side developers.
   * Allows for server-side logic to validate data before it is accepted into the system.

Important Considerations

* **Limitations**: SQL is not a fully-fledged programming language; avoid writing all business logic in stored procedures.

Example Code for a Stored Procedure

DELIMITER $$

CREATE PROCEDURE update\_sal(IN emp\_id INT, IN emp\_rating INT)

BEGIN

IF emp\_rating = 1 THEN

UPDATE employees SET salary = salary \* 1.10 WHERE employee\_id = emp\_id;

ELSE

UPDATE employees SET salary = salary \* 1.05 WHERE employee\_id = emp\_id;

END IF;

END $$

DELIMITER ;

Explanation of the Code

1. **Change the Delimiter**:
   * DELIMITER $$: Changes the statement delimiter from the default semicolon (;) to $$ to avoid confusion with semicolons in the procedure.
2. **Creating the Procedure**:
   * CREATE PROCEDURE update\_sal(IN emp\_id INT, IN emp\_rating INT): Defines a new stored procedure named update\_sal with two input parameters: emp\_id (employee ID) and emp\_rating (employee rating).
3. **Begin and End Block**:
   * BEGIN ... END: Contains the procedural logic that will be executed when the procedure is called.
4. **Conditional Logic**:
   * IF emp\_rating = 1 THEN: Checks if the employee's rating is 1.
     + If true, executes:

UPDATE employees SET salary = salary \* 1.10 WHERE employee\_id = emp\_id;

This increases the employee's salary by 10%.

* + ELSE: If the rating is not 1, executes:

UPDATE employees SET salary = salary \* 1.05 WHERE employee\_id = emp\_id;

This increases the salary by 5%.

1. **End of Procedure**:
   * END $$: Marks the end of the stored procedure definition.
2. **Resetting the Delimiter**:
   * DELIMITER ;: Resets the delimiter back to the default semicolon.

Calling the Stored Procedure

To call this stored procedure, you would use the following SQL command:

CALL update\_sal(101, 1);

Explanation of the Call

* CALL update\_sal(101, 1);: Calls the update\_sal procedure, passing 101 as the employee ID and 1 as the employee rating. The procedure updates the salary of the employee with ID 101 by 10%.

Summary

* Stored procedures are powerful tools for executing SQL statements on the server, offering benefits like reduced network traffic, improved performance, code reuse, and enhanced security.

## **ACID Transactions**

1. **ACID** stands for:
   * **Atomic**: All changes must be performed successfully or not at all.
   * **Consistent**: Data must be in a consistent state before and after the transaction.
   * **Isolated**: No other process can change the data while the transaction is running.
   * **Durable**: The changes made by the transaction must persist.
2. **Transaction**: An indivisible unit of work that can consist of one or more SQL statements. It must either complete fully or not at all.
3. **Commands**:
   * **BEGIN**: Starts a transaction (implicit in some databases).
   * **COMMIT**: Saves all changes made during the transaction.
   * **ROLLBACK**: Undoes all changes if any command fails.

Example Scenario

Let's consider a scenario where a customer, Rose, buys boots for $200. The following SQL statements would be executed:

1. **Update Rose's account balance**:

UPDATE accounts SET balance = balance - 200 WHERE customer\_id = 'rose\_id';

1. **Update the Shoe Shop's balance**:

UPDATE accounts SET balance = balance + 200 WHERE shop\_id = 'shoe\_shop\_id';

1. **Update the inventory**:

UPDATE inventory SET stock = stock - 1 WHERE product\_id = 'boots\_id';

Explanation of the Code

* **Atomicity**: If any of the above statements fail (e.g., Rose doesn't have enough money), the entire transaction should fail. This is where the **ROLLBACK** command comes into play.
* **Consistency**: Before and after the transaction, the database must remain in a consistent state. If Rose's account balance goes negative, that would violate consistency.
* **Isolation**: While this transaction is being processed, no other transactions should be able to modify the accounts or inventory involved.
* **Durability**: Once the **COMMIT** command is issued, the changes are saved permanently, even if the system crashes afterward.

Complete Transaction Example

Here’s how you might implement this in SQL:

BEGIN;

-- Update Rose's account

UPDATE accounts SET balance = balance - 200 WHERE customer\_id = 'rose\_id';

-- Update Shoe Shop's account

UPDATE accounts SET balance = balance + 200 WHERE shop\_id = 'shoe\_shop\_id';

-- Update inventory

UPDATE inventory SET stock = stock - 1 WHERE product\_id = 'boots\_id';

-- Check if all updates were successful

IF (/\* check for errors \*/) THEN

ROLLBACK; -- Undo all changes if there was an error

ELSE

COMMIT; -- Save all changes if everything is successful

END IF;

Summary

* **ACID transactions** ensure that database operations are reliable and maintain data integrity.
* Use **BEGIN**, **COMMIT**, and **ROLLBACK** to manage transactions effectively.
* Always check for errors to decide whether to commit or roll back changes.

**[Optional] Hands-on Labs Using IBM Db2**

**Advanced SQL for Data Engineering**

You can practice the module's concepts using the IBM Db2 system. Here are links to resources that will guide you in using Db2 for using advanced SQL concepts like Views, Stored Procedures, Transactions, and Joins.

[Using Views](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DB0201EN-SkillsNetwork/labs/Labs_Coursera_V5/labs/Lab%20-%20Using%20Views/instructional-labs.md.html?origin=www.coursera.org)

[Stored Procedures](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DB0201EN-SkillsNetwork/labs/Labs_Coursera_V5/labs/Lab%20-%20Stored%20Procedures/instructional-labs.md.html?origin=www.coursera.org)

[Committing and Rolling Back a transaction](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DB0201EN-SkillsNetwork/labs/Labs_Coursera_V5/labs/Lab%20-%20Committing%20and%20Rolling%20back%20a%20Transaction/instructional-labs.md.html?origin=www.coursera.org)

[Joins](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DB0201EN-SkillsNetwork/labs/BonusModule_Coursera_v5/Hands-on-lab-Joins.md.html?origin=www.coursera.org)

[Final Project: Advanced SQL for Data Engineers](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBMDeveloperSkillsNetwork-DB0201EN-SkillsNetwork/labs/BonusModule_Coursera_v5/FinalProject_BonusModule.md.html?origin=www.coursera.org)

**Summary: Views, Stored Procedures, and Transactions**

Congratulations! You have completed this lesson. At this point in the course, you know:

* Views are a dynamic mechanism for presenting data from one or more tables.A transaction represents a complete unit of work, which can be one or more SQL statements.
* An ACID transaction is one where all the SQL statements must complete successfully, or none at all.
* A stored procedure is a set of SQL statements that are stored and executed on the database server, allowing you to send one statement as an alternative to sending multiple statements.
* You can write stored procedures in many different languages like SQL PL, PL/SQL, Java, and C.

# **JOIN statements**

## **JOIN Overview**

Key Points on JOIN Operator

1. **Definition of JOIN**:
   * The **JOIN** operator is used to combine rows from two or more tables based on a related column between them.
2. **Primary Keys and Foreign Keys**:
   * **Primary Key**: A unique identifier for each row in a table.
   * **Foreign Key**: A column that creates a link between two tables, referring to the primary key of another table.
3. **Example Scenario**:
   * In a library database, you may have tables for **authors**, **books**, **borrowers**, **loans**, and **copies**. To find out which borrower has which copy of a book on loan, you need to join the **borrower**, **loan**, and **copy** tables.
4. **Types of JOINs**:
   * **Inner Join**: Returns only the rows with matching values in both tables.
   * **Outer Join**: Returns matching rows and also the rows from one or both tables that do not match.

Example Code

Here’s a simple SQL example demonstrating an **Inner Join**:

SELECT borrowers.name, books.title

FROM borrowers

INNER JOIN loans ON borrowers.borrower\_id = loans.borrower\_id

INNER JOIN copies ON loans.copy\_id = copies.copy\_id

INNER JOIN books ON copies.book\_id = books.book\_id;

Explanation of the Code:

* **SELECT**: Specifies the columns to retrieve (borrower names and book titles).
* **FROM**: Indicates the primary table to select data from (borrowers).
* **INNER JOIN**: Combines rows from the **borrowers** table with the **loans** table where the borrower\_id matches.
* Additional **INNER JOINs** are used to connect the **loans** table with the **copies** table and then with the **books** table, ensuring that all relevant data is retrieved.

Summary

* The JOIN operator is essential for retrieving related data from multiple tables.
* Understanding primary and foreign keys is crucial for effectively using JOINs.
* Different types of JOINs allow for flexibility in how data is combined.

## **Inner join**

Key Concepts of Inner Join

1. **Definition**:
   * An **inner join** combines rows from two or more tables based on a related column between them. It returns only the rows that have matching values in both tables.
2. **When to Use**:
   * Use an inner join when you want to retrieve records that have corresponding entries in both tables.
3. **Syntax**:
   * The basic syntax for an inner join is as follows:
   * SELECT column1, column2, ...
   * FROM table1 AS alias1
   * INNER JOIN table2 AS alias2

ON alias1.common\_column = alias2.common\_column;

Example Scenario

Imagine you have two tables:

* **Borrower Table**: Contains information about borrowers.
* **Loan Table**: Contains information about loans.

Tables Structure

* **Borrower Table**:
  + Borrower\_ID (Primary Key)
  + Lastname
  + Country
* **Loan Table**:
  + Borrower\_ID (Foreign Key)
  + Loan\_Date

SQL Query Example

To retrieve a list of all people who are borrowing books along with the date of the loan, you would write the following SQL query:

SELECT B.Borrower\_ID, B.Lastname, B.Country, L.Loan\_Date

FROM Borrower AS B

INNER JOIN Loan AS L

ON B.Borrower\_ID = L.Borrower\_ID;

Explanation of the Code

* **SELECT**: Specifies the columns you want to retrieve. Here, we are selecting Borrower\_ID, Lastname, and Country from the Borrower table (aliased as B) and Loan\_Date from the Loan table (aliased as L).
* **FROM**: Indicates the primary table from which to retrieve data. In this case, it's the Borrower table.
* **INNER JOIN**: This keyword is used to combine rows from the Borrower table and the Loan table based on the condition specified in the ON clause.
* **ON**: This clause specifies the condition for the join. Here, it states that the Borrower\_ID in the Borrower table must match the Borrower\_ID in the Loan table.

Result Set

The result set will only include rows where there is a match between the Borrower\_ID in both tables. If a borrower has no loans, they will not appear in the result.

Summary

* **Inner joins** are essential for retrieving related data from multiple tables.
* The syntax involves specifying the tables, the join condition, and the columns to select.
* Only matching rows from both tables are returned in the result set.

## **Outer joins**

Outer Joins Overview

Outer joins return rows from both tables involved in the join, including rows that do not have matching values in the join columns. There are three types of outer joins:

1. **Left Outer Join**
2. **Right Outer Join**
3. **Full Outer Join**

1. Left Outer Join

* **Description**: Returns all rows from the left table and the matching rows from the right table. If there is no match, NULL values are returned for columns from the right table.
* **Syntax**:
* SELECT Borrower.BorrowerID, Borrower.LastName, Borrower.Country, Loan.LoanDate
* FROM Borrower

LEFT JOIN Loan ON Borrower.BorrowerID = Loan.BorrowerID;

2. Right Outer Join

* **Description**: Returns all rows from the right table and the matching rows from the left table. If there is no match, NULL values are returned for columns from the left table.
* **Syntax**:
* SELECT Loan.Borrower\_ID, Loan.LoanDate, Borrower.LastName, Borrower.Country
* FROM Loan

RIGHT JOIN Borrower ON Loan.Borrower\_ID = Borrower.BorrowerID;

3. Full Outer Join

* **Description**: Returns all rows from both tables. If there is no match, NULL values are returned for columns from the table that does not have a match.
* **Syntax**:
* SELECT Borrower.BorrowerID, Loan.LoanDate
* FROM Borrower

FULL OUTER JOIN Loan ON Borrower.BorrowerID = Loan.BorrowerID;

Key Points

* **Left Outer Join**: All rows from the left table + matching rows from the right table.
* **Right Outer Join**: All rows from the right table + matching rows from the left table.
* **Full Outer Join**: All rows from both tables, with NULLs where there are no matches.

Example Explanation

* In a **Left Outer Join**, if a borrower has not taken any loans, their information will still appear in the result set, but the loan date will show as NULL.
* In a **Right Outer Join**, if a loan exists without a corresponding borrower, the loan information will appear, but borrower details will show as NULL.
* In a **Full Outer Join**, you will see all borrowers and all loans, with NULLs where there are no matches.

Conclusion

Understanding outer joins is crucial for data retrieval in SQL, especially when dealing with incomplete datasets. Practice writing these queries to reinforce your understanding.

**Summary: JOIN Statements**

Congratulations! You have completed this lesson. At this point in the course, you know:

* A join combines the rows from two or more tables based on a relationship between certain columns in these tables.
* To combine data from three or more different tables, you simply add new joins to the SQL statement.
* There are two types of joins: inner join and outer join; and three types of outer joins: left outer join, right outer join, and full outer join.
* The most common type of join is the inner join, which matches the results from two tables and returns the selected elements from each table, only where corresponding elements in both the tables are the same.
* You can use an alias as shorthand for a table or column name.