Data Analyst SQL Knowledge Checklist

As a data analyst, mastering SQL is essential for querying databases, analyzing data, and generating insights. Here's what you need to know in SQL and how you can use it

1. Basic SQL Queries

Purpose: Retrieve data from databases.

- **`SELECT`**: The most fundamental command for extracting data from a database.
- **`WHERE`**: Filter rows based on conditions.
- **`ORDER BY`**: Sort the results based on one or more columns.
- **`LIMIT`**: Limit the number of rows returned.
- **`DISTINCT`**: Remove duplicates from the result set.

Use: These commands help you extract relevant data for analysis (e.g., getting sales data for a specific year or product category).

Example:

```
SELECT product_name, sales
FROM Products
WHERE category = 'Electronics'
ORDER BY sales DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

— This query selects the top 10 electronics products sorted by sales.

2. Joins

Purpose: Combine data from multiple tables.

- **`INNER JOIN`**: Returns rows with matching values in both tables.
- **`LEFT JOIN`**: Returns all rows from the left table and matching rows from the right table. Non-matching rows from the right table result in `NULL`.
- **`RIGHT JOIN`**: Returns all rows from the right table and matching rows from the left table.
- **`FULL OUTER JOIN`**: Returns all rows when there is a match in either table.
- **`CROSS JOIN`**: Returns the Cartesian product of two tables.

Use: Joins are crucial for bringing data together from different sources (e.g., combining customer and sales data).

Example:

```
SELECT C.CustomerName, O.OrderDate, O.TotalAmount
FROM Customers C
INNER JOIN Orders O ON C.CustomerID = O.CustomerID
WHERE O.OrderDate >= '2024-01-01';
```

— This `INNER JOIN` combines customers and orders, showing orders made after January 1, 2024.

3. Aggregate Functions

Purpose: Summarize and group data.

- **`COUNT()`**: Returns the number of rows.
- **`SUM()`**: Calculates the sum of a numeric column.
- **`AVG()`**: Returns the average value of a numeric column.
- **`MAX()` / `MIN()`**: Returns the maximum or minimum value in a column.

Use: Aggregates help in producing summary statistics from raw data.

Example:

```
SELECT Category, COUNT(*) AS TotalProducts, SUM(Sales) AS
TotalSales
FROM Products
GROUP BY Category;
```

— This query groups products by category and returns the total number of products and their total sales for each category.

4. Grouping and Filtering Aggregated Data

Purpose: Group data and filter aggregated results.

- **`GROUP BY`**: Groups rows sharing a property and applies aggregate functions to each group.
- **`HAVING`**: Filters groups based on aggregate function results (similar to `WHERE`, but for aggregated data).

Use: Use this when analyzing data based on categories or segments.

Example:

```
SELECT Category, AVG(Sales) AS AvgSales
FROM Products
GROUP BY Category
HAVING AVG(Sales) > 1000;
```

— This query shows product categories where the average sales exceed $1,000.

5. Subqueries

Purpose: Use the result of one query inside another query.

- **Subquery in SELECT**: Calculating values based on other values in the same table.

- **Subquery in FROM**: Creating a temporary table for further analysis.
- **Subquery in WHERE**: Use the result of a subquery to filter the main query.

Use: Subqueries allow more complex querying logic and can break down tasks into manageable parts.

Example:

```
SELECT ProductName, Sales
FROM Products
WHERE Sales > (SELECT AVG(Sales) FROM Products);
```

— This query selects products where sales exceed the average sales.

6. Common Table Expressions (CTEs)

Purpose: Create temporary result sets that can be referenced within the main query.

Use: CTEs help make complex queries easier to read and maintain.

Example

```
WITH SalesSummary AS (
    SELECT CustomerID, SUM(TotalAmount) AS TotalSales
    FROM Orders
    GROUP BY CustomerID
)
SELECT C.CustomerName, S.TotalSales
FROM Customers C
JOIN SalesSummary S ON C.CustomerID = S.CustomerID
ORDER BY S.TotalSales DESC;
```

— This CTE creates a sales summary for each customer and then ranks customers by total sales.

7. Window Functions

Purpose: Perform calculations across a set of table rows related to the current row, without collapsing rows.

- **ROW_NUMBER()**: Assigns a unique number to each row.
- **RANK()**: Similar to **ROW_NUMBER()**, but handles ties by assigning the same rank.
- **LEAD() / LAG()**: Access the value of a column from a previous or subsequent row.
- **SUM() / AVG()** (with OVER clause): Calculate running totals or averages.

Use: Window functions are useful for performing calculations like running totals, rankings, or comparisons between rows.

Example:

```
SELECT ProductName, Sales,
    SUM(Sales) OVER (ORDER BY Sales DESC) AS RunningTotal
FROM Products;
```
— This query calculates a running total of sales, sorted in descending order by sales.

8. Indexes

Purpose: Improve query performance by allowing faster data retrieval.

Use: Understanding indexing is crucial for optimizing large datasets, especially when querying large tables or frequently accessed columns.

Example:

```
CREATE INDEX idx_CustomerName ON Customers
(CustomerName);
```
—This creates an index on the `CustomerName` column to speed up queries involving that field.

9. Transactions

Purpose: Ensure data integrity by grouping multiple operations as a single unit of work.

- **BEGIN**, **COMMIT**, **ROLLBACK**: Manage transactions to ensure all queries either succeed together or fail together.

Use: Transactions are important when performing operations that need atomicity (e.g., multiple updates that should either all happen or none at all).

Example:

```
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 1;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 2;

COMMIT TRANSACTION;
```
— This transaction ensures that both account updates succeed or fail together.

10. Data Manipulation

Purpose: Insert, update, and delete data in a database.

- **INSERT INTO**: Adds new records to a table.
- **UPDATE**: Modifies existing records.
- **DELETE**: Removes records from a table.

Use: These commands allow you to maintain and update your database records.

Insert Example:

```
INSERT INTO Customers (CustomerName, Email)
VALUES ('John Doe', 'john@example.com');
```

Update Example:

```
UPDATE Customers
SET Email = 'john.newemail@example.com'
WHERE CustomerID = 1;
```

Delete Example

```
DELETE FROM Customers
WHERE CustomerID = 1;
```

— These commands demonstrate adding, updating, and deleting data in the `Customers` table.

11. SQL Data Types

Purpose: Define the types of data that can be stored in a table.

- Common data types: INTEGER, FLOAT, VARCHAR, BOOLEAN, DATE, TIME, etc.

Use: Understanding the appropriate data type ensures data is stored correctly, leading to better query performance and accuracy.

Example:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName NVARCHAR(100),
    Email NVARCHAR(100),
    SignupDate DATE
);
```

— This defines a `Customers` table with `INT`, `NVARCHAR`, and `DATE` data types.

12. Normalization

Purpose: Organize data to reduce redundancy and improve integrity.

- **First Normal Form (1NF)**: Ensure that the values in each column are atomic (no repeating groups).
- **Second Normal Form (2NF)**: Ensure that each non-primary-key column is fully functionally dependent on the primary key.
- **Third Normal Form (3NF)**: Ensure that no transitive dependencies exist.

Use: Helps in structuring databases efficiently to avoid anomalies when inserting, updating, or deleting data.

Example (First Normal Form):

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    ProductID INT,
    Quantity INT
);

CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName NVARCHAR(100),
    Category NVARCHAR(50)
);
```

— These normalized tables avoid redundancy, keeping product details separate from order details.

13. SQL for Data Cleaning

Purpose: Use SQL to identify and clean dirty or inconsistent data.

- **Handling NULLs**: Use IS NULL or IS NOT NULL to filter or handle missing data.
- **String Functions**: Use functions like TRIM(), LOWER(), UPPER() to clean string data.
- **Date Functions**: Use functions like DATE(), EXTRACT() to manipulate date fields.
- **CASE and COALESCE()**: Useful for transforming and filling missing data.

Example:

```
SELECT COALESCE(CustomerName, 'Unknown') AS CustomerName
FROM Customers;
```

—This replaces **`NULL`** values in the **`CustomerName`** column with the string "**Unknown**."

14. Stored Procedures

Purpose: Encapsulate SQL logic for reuse, improve performance, and enhance security.

- **Definition**: A stored procedure is a precompiled collection of SQL statements stored under a name, which can be executed as a single unit.
- Advantages:
  - **Modularity**: Simplifies complex logic into reusable code blocks.
  - **Performance**: Executes faster due to precompilation.
  - **Security**: Restricts direct access to tables, enhancing data protection.
  - **Maintainability**: Changes in business logic can be managed in one location without altering application code.
  - **Parameterization**: Allows customization of execution by passing parameters.

Use: Stored procedures help streamline repetitive tasks, enforce business rules, and manage complex queries efficiently.

Example:

```
CREATE PROCEDURE GetSalesByProduct
    @ProductID INT
AS
BEGIN
    SELECT
        Sales.SaleID,
        Sales.SalesAmount,
        Sales.SaleDate
    FROM
        Sales
    WHERE
        Sales.ProductID = @ProductID;
END;
```

— This example creates a stored procedure named `GetSalesByProduct` that retrieves sales records for a specific product.

Executing a Stored Procedure:

```
EXEC GetSalesByProduct @ProductID = 1;
```

— This command executes the `GetSalesByProduct` procedure for the product with ID 1.

15. Functions

Purpose: Encapsulate logic to return a value and can be used in SQL statements.

- Types of Functions:
  - **Scalar Functions**: Return a single value (e.g., string manipulation functions, mathematical functions).
  - **Table-Valued Functions**: Return a table and can be used like a table in queries.

Use: Functions provide reusable code for calculations and data transformations, allowing for cleaner and more maintainable SQL.

Example of a Scalar Function:

```
CREATE FUNCTION GetDiscountedPrice
(
    @OriginalPrice DECIMAL(10, 2),
    @DiscountRate DECIMAL(5, 2)
)
RETURNS DECIMAL(10, 2)
AS
BEGIN
    RETURN @OriginalPrice * (1 - @DiscountRate);
END;
```

Using Scalar Functions:

```
SELECT ProductName, dbo.GetDiscountedPrice(SalesAmount,
0.1) AS DiscountedPrice
FROM Products;
```

— This query applies the `GetDiscountedPrice` function to calculate discounted prices for products.

Example of a Table-Valued function:

```
CREATE FUNCTION GetSalesByCustomer(@CustomerID INT)
RETURNS TABLE
AS
RETURN
(
    SELECT SaleID, SalesAmount
    FROM Sales
    WHERE CustomerID = @CustomerID
);
```

— This function returns a table of sales records for a specific customer.

Using the Table valued Function

```
SELECT *
FROM GetSalesByCustomer(1) AS SalesForCustomer;
```

— This query retrieves all sales records for the customer with ID 1.

Footnote: *This document serves as a foundational guide for learning SQL, covering essential concepts, commands, and best practices. Whether you're a beginner seeking to understand the basics or an intermediate learner aiming to refine your skills, mastering SQL is crucial for effective data management and analysis. For a deeper understanding, consider exploring additional resources such as online courses, tutorials, and hands-on practice with real-world datasets. Continuous practice and real-life application of these concepts will significantly enhance your proficiency in SQL.*

Credit: *Compiled by MC Guevara, a passionate data analyst dedicated to sharing knowledge and empowering others in their SQL journey.*

Help Line: *For questions, clarifications, or further assistance, feel free to reach out at career@mcguevara.com or visit online forums and communities like Stack Overflow, SQLServerCentral, or the Microsoft SQL Server forums.*