Essential SQL Techniques for Data Quality Management

Data quality is the cornerstone of effective decision-making and SQL, as a powerful language for data manipulation, is instrumental in assessing and improving data quality. This series of SQL statement templates provides a foundation for identifying and addressing common data quality issues.

These templates can be adapted to fit specific database structures and business requirements, serving as a starting point for comprehensive data quality analysis and remediation efforts.

> *Note: Substitute your table name for Customers and your columns/fields for Email, FirstName, LastName, etc.)*

Sample SQL Statements

Checking for NULLs in a Specific Column

- ```
  SELECT *
  FROM Customers
  WHERE Email IS NULL;
  ```
- This query is used to identify rows in the Customers table where the Email column has no value (NULL). In SQL, NULL represents missing or unknown data. The SELECT * statement retrieves all columns from the Customers table. The WHERE clause filters the results to only those rows where the Email column is NULL. This can help you find incomplete records that may need further attention or correction.

Counting NULLs in a Specific Column

- ```
  SELECT COUNT(*) AS NullCount
  FROM Customers
  WHERE Email IS NULL;
  ```
- This query is similar to the first but focuses on counting how many rows have NULL values in the Email column. The COUNT(*) function counts the total number of rows that meet the WHERE condition. The result is returned as a single value, labeled as NullCount. This is useful for assessing the extent of missing data in a particular column.

Finding Rows with NULLs in Any Column

- ```
  SELECT *
  FROM Customers
  WHERE (Email IS NULL
         OR FirstName IS NULL
         OR LastName IS NULL);
  ```
- In this query, the goal is to identify rows where any of the specified columns (Email, FirstName, or LastName) contain NULL values. The OR operator is used to check each column individually. If any of the conditions are true, the row is included in the result set. This helps you find records that may be partially incomplete, enabling targeted data cleanup.
  ```

Checking for Empty Strings (or NULLs) in a Column

- ```
  SELECT *
  FROM Customers
  WHERE (Email IS NULL
         OR Email = '');
  ```
- This query identifies rows where the Email column is either NULL or an empty string (''). While NULL represents the absence of data, an empty string indicates a value that is technically present but contains no characters. This is a common issue in data entry systems, and this query helps in finding such cases where the Email column needs to be corrected or filled in.

Checking for Duplicate Records

- ```
  SELECT Email, COUNT(*) AS DuplicateCount
  FROM Customers
  GROUP BY Email
  HAVING COUNT(*) > 1;
  ```
- This query identifies duplicate entries based on the Email column. The GROUP BY clause groups rows with the same Email together, and the COUNT(*) function counts the number of rows in each group. The HAVING clause then filters the groups to only those with more than one row, indicating duplicates. This is crucial for maintaining data integrity, especially in cases where unique identifiers like email addresses should not be duplicated.

Detecting Outliers or Unusual Values

- **Negative ages**
  ```
  -- Negative ages
  SELECT *
  FROM Customers
  WHERE Age < 0;
  ```
- **Future birth dates**
  ```
  -- Future birth dates
  SELECT *
  FROM Customers
  WHERE BirthDate > GETDATE();
  ```
- These queries are used to identify data entries that fall outside of expected or valid ranges. The first query checks for Age values that are less than zero, which is logically impossible and likely indicates a data entry error. The second query finds birth dates that are in the future, which is also an error unless dealing with fictional or test data. These checks help ensure that the data conforms to real-world constraints.

Checking for Inconsistent Formatting

- ```
  SELECT *
  FROM Customers
  WHERE LEN(PhoneNumber) != 10;
  ```
- This query checks for inconsistencies in the format of phone numbers. The LEN() function returns the length of the PhoneNumber column, and the WHERE clause filters for rows where this length is not equal to 10 (assuming a standard 10-digit phone number format). This helps identify records that may have missing digits or extra characters, which could lead to communication issues or invalid data.
  ```

Ensuring Referential Integrity

- ```
  SELECT *
  FROM Orders o
  LEFT JOIN Customers c
         ON o.CustomerID = c.CustomerID
  WHERE c.CustomerID IS NULL;
  ```
- This query checks for referential integrity by ensuring that every Order in the Orders table has a corresponding customer in the Customers table. The LEFT JOIN returns all rows from the Orders table and matches them with rows in the Customers table based on the CustomerID. If no match is found, the columns from the Customers table will be NULL, which is then filtered by the WHERE clause. This identifies "orphaned" orders that are associated with non-existent customers, which can indicate data corruption or entry errors.

Checking for Invalid Email Addresses

- ```
  SELECT *
  FROM Customers
  WHERE Email NOT LIKE '%@%';
  ```
- This query looks for email addresses that do not contain the "@" symbol, which is a basic requirement for valid email addresses. The LIKE operator is used with the % wildcard to search for the "@" symbol anywhere within the Email column. The NOT LIKE clause filters out all valid email addresses, leaving only those that are likely invalid. This helps in identifying and correcting email addresses that are incorrectly formatted.

Validating Numeric Ranges

- ```
  SELECT *
  FROM Customers
  WHERE Age < 0 OR Age > 120;
  ```
- This query checks for values in the Age column that fall outside a reasonable range (0 to 120 years). The OR operator is used to filter for ages that are either less than zero or greater than 120, both of which are unusual and likely indicative of data errors. By identifying these outliers, you can ensure that the data more accurately reflects real-world scenarios.
  ```