Geek Squad SQL Best Practices

10-10-2018

Overview

Queries, especially long queries, are notoriously hard to understand (even for the authors) and can be even harder to debug. However, best practices exist that, if followed, allow for easier understanding and debugging.

The purpose of this document is to outline Geek Squad Best Practices for data extraction from various data sources via SQL. In general, this document describes best practices for

- SQL Coding Style Standards
- Quality Control

SQL Coding Style Standards

Coding style principles:

- Readability – consistent indentation and use of white space allow for reader to easily identify constructs within the query code to more easily understand what the query is doing. This also helps when trying to debug the query code.
- Limit multiple nested select statements.
- Use spaces instead of tabs for indentation.
- Use a code beautifier, such as poorsql.com.
- Code for performance if the query will be rerun.
- Version Control - Every extract needs to be accompanied by a link to a repository with the correct version of the query committed.
- File Naming Structure- A consistent naming structure is needed. After every query run, the date should be appended to the file name. If multiple runs are completed in the same day, date-time should be appended.
- Data Dictionary - Every extract should be accompanied by a data dictionary. A data dictionary is composed of: column names, descriptions of what information is intended to be in the column, source table and an explanation of the manipulations applied to the source data. Ideally this would be saved in a shareable location.
- Commenting Practices- All queries should have comments throughout describing the action taking place. Extra detail in the comments is necessary for

queries over 50 lines.

- Delimiter Consistency - Each Version of the file needs to have the same delimitator as is established at the beginning of a project.

Coding style specifics:

- Do not use SELECT * in your queries. Specify the column names. This technique results in reduced disk I/O and better performance.
- Always use table aliases when using more than one table. This makes the code more human readable.
- Do not use column numbers in the ORDER BY clause. Using column names makes the query more human readable.
- Always use a column list in INSERT statements.
- Every nested statement must begin with new level of indentation.
- All database reserved words will be in uppercase.
- Make column and constraint names lowercase.
- The length of lines should not exceed 80 characters.
- Comments should be used in the SQL to describe functionality. The comments must be in the form of /* <text> */.
- Do not use dynamic SQL. SQL injection is one of the most common attacks out there. This type of attack leads to severe data breaches that expose millions of records to an attacker. Extremely strategic SQL injection attacks can even elevate permissions for the attacker to give them administrative rights on the database server.
- Use unique constraints when a column is not expected to hold duplicate values in any case. For example, a unique constraint should be defined on party ID.
- Use ANSI Style Joins - With ANSI joins, the WHERE clause is used only for filtering data. Whereas with older style joins, the WHERE clause handles both the join condition and filtering data. Furthermore, ANSI join syntax supports the full outer join. The first of the following two queries shows the old-style join, while the second one shows the new ANSI join syntax:

```
-- old style join
SELECT a.Author_id
       t.Title
FROM   TITLES t,
       AUTHORS a,
       TITLEAUTHOR ta
WHERE  a.Au_id = ta.Au_id
       AND ta.Title id = t.Title id
       AND t.Title LIKE '%Computer%'

/* ANSI join syntax */
SELECT a.Authuor_id,
       t.Title
FROM   AUTHORS a
       INNER JOIN TITLEAUTHOR ta
         ON a.Au id = ta.Au id
       INNER JOIN TITLES t
         ON ta.Title id = t.Title id
WHERE  t.Title LIKE '%Computer%'
```

- Avoid hard coding scalars – instead use constants

  A good first step is to define constants:

```
WITH constants AS (
        SELECT 600 AS SHORT_DUR
            ,1800 AS LONG_DUR
            ,['Sun', 'Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat'] AS
daysofweek
        )
```

- Do not use ORDER BY clause unnecessarily in queries until the result set is strictly required to be sorted in a particular order.
- Do not use DISTINCT keyword when GROUP BY clause is present in the same query because GROUP BY clause itself results in an only unique combination of columns. Thus DISTINCT is not of meaningful when GROUP BY clause is present.
- Avoid using LIKE operator as much as possible. Using like operator is not efficient and degrades the query performance. LIKE operator should be avoided in WHERE clause when the first character(s) is specified with a wildcard character.
- Avoid unnecessary use of CAST() and CONVERT() functions to convert between the data types. Also, minimize the use of other scalar functions as they degrade the performance of the queries.
- When possible, use primary or surrogate key(s) in joins instead of the natural key(s). Typically, the PK construct or a numeric valued surrogate key will result in a lower query cost than the same key represented as a string.

- When possible, run an explain plan (Teradata, Oracle) or execution plan (SQL Server). These are effectively the same thing but referred to with different terminology depending on DBMS vendor. The plan will show the relative cost of each step of the query and any indexes or partitions being used by the SQL statement and can help identify potential issues.

- Use positive logic in CASE or other conditional statements whenever possible over negative logic. Stated differently, this means defining something by what conditions it satisfies as opposed to what conditions it doesn't satisfy. Negative logic is more likely to result in potential unexpected behavior.

Commented [AR1]: Examples for positive vs. negative?

Common Table Expressions

Use common table descriptions rather than nested select statements for readability purposes.

Consider the following description is from: https://facility9.com/2008/12/a-quick-introduction-to-common-table-expressions-3/

> *Essentially, a common table expression (CTE) is a temporary result set. In this regard, a CTE is similar to a view, temporary table, or derived table. There are some important ways that a CTE is different from a view, temporary table, or a dervied table:*
>
> *CTEs are not persisted, views are. If you only need a particular result set in a single query/stored procedure, creating a view will only cloud up the metadata that is being stored in the database. Using a CTE encapsulates this logic and stores it with the relevant query. If you don't have the ability to create views, a CTE is also a great way around the lack of permissions.*
>
> *A CTE can be referenced multiple times in the same statement. How is this better than referring to the same view/result set multiple times in a single query? For starters, every time you want to refer to the same result set from a view or query it's necessary to repeat the query. This isn't so bad if your query is as simple as 'SELECT x, y, z FROM small_view' but when you are creating a complicated result set that contains multiple aggregates, joins, and groupings your query will become cluttered very quickly. Not to mention the maintenance nightmare that this will cause when you have to change your query due to changes in the underlying table structure.*
>
> *Every time you make use of a derived table, that query is going to be executed. When using a CTE, that result set is pulled back once and only once within a single query. Here's a recent example I wrote: data is pulled back for multiple bills within a set of accounts. Within each account the total cost is aggregated by bill. In addition, four separate aggregations are made based on a subset of the result set for each bill. If I had done this with derived tables, that would come out to 5 hits to the underlying database. By using a CTE, only one main hit to disk is made, the result set is held in memory, the query is processed and returned to the user, and then the result set is dropped.*
>
> *CTEs have limited scope. A CTE is only resident within a single query. Unlike temporary tables which will persist until the user disconnects from the server, a CTE disappears*

once the query has completed. This makes memory and table management a lot easier on both developers and DBAs.

CTEs can be recursive. A recursive CTE is a very powerful piece of functionality and can be used to retrieve complex hierarchies from a single table that might otherwise require multiple queries to retrieve.

CTEs offer better aggregation possibilities. Within a single query, it normally isn't possible to produce aggregations on the output of non-deterministic functions. When the output of a non-deterministic function is included in a CTE, you can group by the output. Of course, you could also do this by including the function in a derived table, however the T-SQL to create the CTE ends up being much cleaner to read than using derived tables.

The main advantage of the Common Table Expression (when not using it for recursive queries) is encapsulation, instead of having to declare the sub-query in every place you wish to use it, you are able to define it once, but have multiple references to it.

Metadata for Data Extracts

- Column Names,
- Description of the INTENDED information in each column,
- Aggregation method applied to each column
- Any calculations or transformations included
- Raw query

Quality Control

- Primary Key Uniqueness – validate that the expected Validation

- Aggregation Checks - any 'group by' functions should be tested by comparing aggregated results vs the unaggregated results (EX: the sum of column 'X' in non-group by table = 100; after applying the GROUP-BY does the sum of column 'X' still equal 100?)

- Table Joins - when joining tables, validate the primary keys are behaving as expected and no duplication or leakage has occurs. This can be done by counting both records that are NOT joined, as well as records that were joined. Do those quantities of each group match expectations? Terradata has a 'minus' function to enable this:
- https://forgetcode.com/teradata/1591-minus

- Verifying Granularity - Determine the "grain" or level of detail that the data needs to be at in the final form. Do the values of each column make sense at said level of detail?

- Inductive Testing - Test examples of each data transformation by reconciling with upstream systems or prebuilt reporting structures.

- Peer Review - As with any engineering discipline, it is a best practice to have other skilled peers view your work. This is a simple process to implement and should be considered standard procedure before any significant delivery.