SESUG Paper ###-year

Breaking Down Monolithic ETLs: Best Practices to Optimize Code Generation in SAS DI Studio and Enhance Oracle Database Performance

Osmel Brito Bigott; Angye Rivero Rodríguez, DATANALITICA

ABSTRACT

For over 15 years, we have relied on SAS Data Integration Studio to develop and maintain robust ETL and data pipeline solutions across industries. As data volumes have grown and performance demands increased, we have encountered the natural evolution of DI Studio-generated jobs becoming monolithic, complex, and increasingly difficult to maintain and optimize—especially when integrating with high-performance databases like Oracle.

This paper shares our journey in breaking down monolithic ETL jobs into modular, reusable components using macro-based programming and design patterns that complement SAS DI Studio's metadata-driven environment. We present a set of best practices that improve code maintainability, reduce redundancy, and enhance troubleshooting and reusability. Additionally, we explore how certain code-generation techniques can negatively impact Oracle database performance and show how simple architectural decisions—such as leveraging pushdown optimization, proper index usage, and transactional control—can drastically improve runtime and scalability.

The session will provide real-world examples, lessons learned, and actionable strategies that attendees can adopt to modernize and optimize their own DI Studio implementations. Whether you're inheriting legacy DI code or starting fresh, this paper will equip you with the tools and perspective to transition from fragile ETLs to efficient, scalable, and maintainable data integration pipelines.

INTRODUCTION

In a recent project involving the migration of ETL processes from a legacy tool to SAS Data Integration Studio, we encountered a significant challenge. During the load into target tables, the final transformation responsible for this activity did not execute properly. The large data volumes and the high number of record updates resulted in extremely poor performance, leading to severe slowdowns. In some cases, the process would not even complete—it simply entered a state of indefinite waiting.

This situation prompted us to investigate and implement a series of SAS macros and make targeted adjustments to the auto-generated code produced by SAS Data Integration Studio, ensuring that the ETLs could run both efficiently and correctly.

It is important to emphasize, however, that SAS Data Integration Studio remains a powerful and reliable platform for designing and managing ETL processes and data pipelines. When configured and optimized properly, it continues to deliver robustness, scalability, and maintainability, making it a strong choice for organizations facing complex data integration challenges.

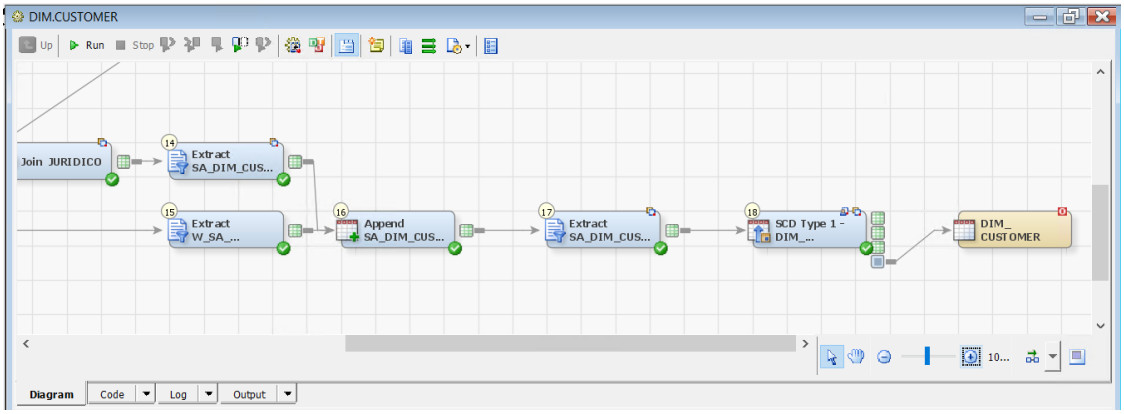
Such challenges are far from uncommon. As data volumes continue to grow and performance requirements increase, organizations face mounting pressure to deliver rapid insights in an ever-changing world. In this paper, we will share, in a practical manner, how this challenge was resolved. Our goal is to provide SAS Data Integration Studio users with concrete strategies to handle similar scenarios effectively and to optimize their ETL processes—delivering better performance, faster results, and improved client experiences.

PRACTICAL CASE

We will work with a real and practical example based on the SCD Type 1 transformation in SAS Data Integration Studio (SAS DIS).

In this case, we present an ETL job designed to extract Customer Data from the database of a core system, apply the necessary transformations, and then load both the new customers and the updated data of existing customers into the company's Business Intelligence database.

The process flow is illustrated in the following diagram:

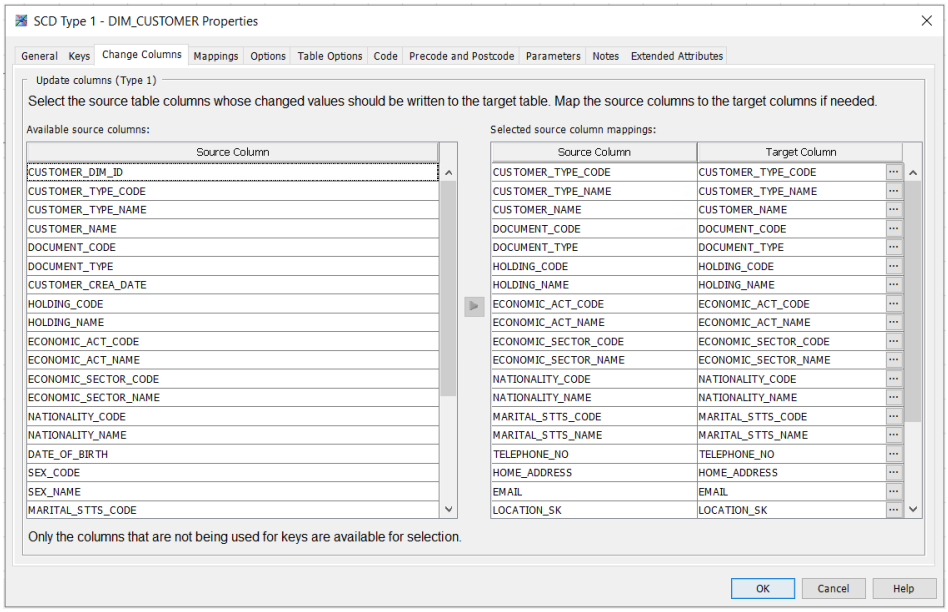


Display 1. ETL de caso práctico.

CHALLENGE OR CRITICAL POINT IDENTIFIED

The critical issue emerged in the **SCD Type 1 transformation**. In this example, the target table **DIM_CUSTOMER** resides in an **Oracle database**, and the customer entity contains many columns subject to frequent changes. The customer dataset is also quite large—approximately **200,000 records**.

These conditions created a **high-processing workload**, which caused the SCD Type 1 transformation to perform poorly. Execution times became excessively long, and in some cases, the process did not complete at all, remaining in a state of indefinite execution.



Display 2. Pestaña de Columnas Cambiantes de transformación SCD Type 1.

IMPLEMENTED SOLUTION TO THE IDENTIFIED CHALLENGE

To address the critical issue of inefficient execution in the **SCD Type 1 transformation**, a strategy was implemented with the goal of improving performance and ensuring successful process completion. This required analyzing the **auto-generated code produced by SAS Data Integration Studio (SAS DI)**, which is accessible through the transformation's properties, specifically under the **Code** tab.

It is important to note that transformations in SAS DI Studio support three different **code generation modes**:

- **Automatic:** The default option, where SAS DI generates the code automatically.
- **User Written Body:** This option allows users to modify a specific set of code instructions while leaving the rest intact.
- **All User Written:** With this option, any line of code generated by SAS DI can be modified or replaced.

After reviewing the generated code, adjustments and improvements were introduced by switching the code generation mode from **Automatic** to **User Written Body**. This gave us the flexibility to implement a more efficient solution, which consisted of:

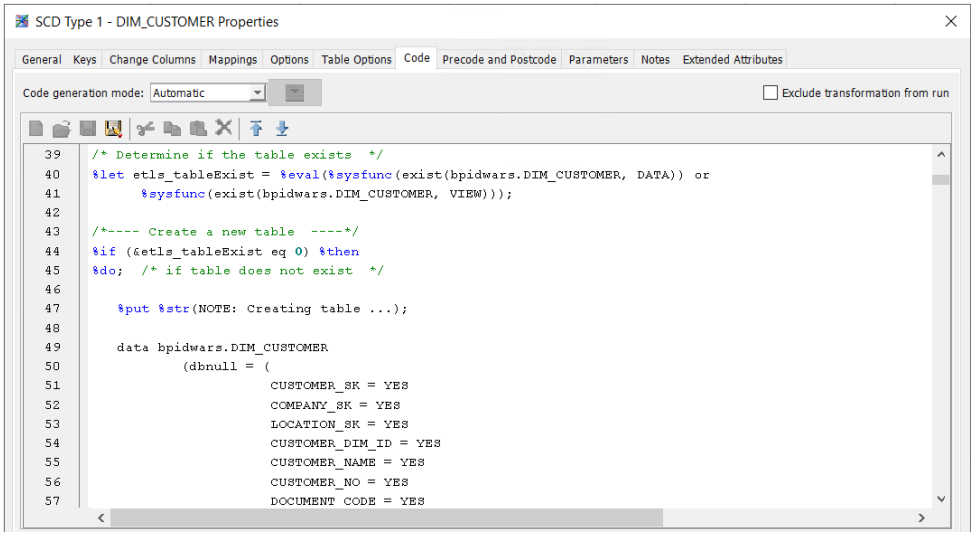
ELIMINATING REDUNDANT CODE

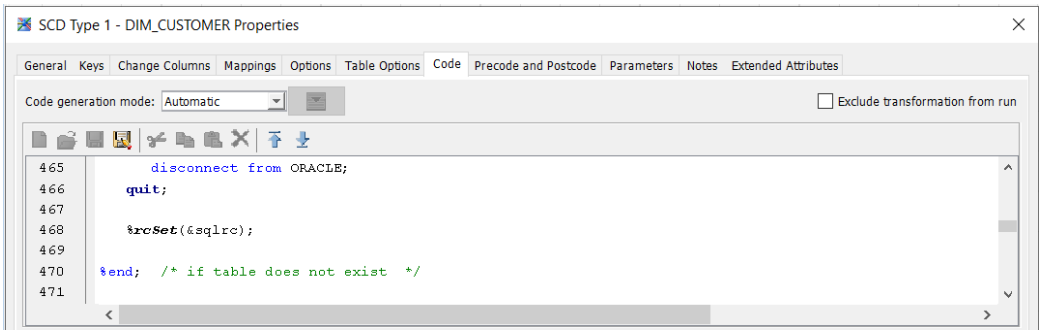
By default, SAS DI validates whether the target table exists and also generates the code to create the table if it does not. However, in a production environment—or in any controlled environment where we know the table already exists—this block of code is unnecessary.

In our example, this redundant block spanned from **line 39 to line 470**, covering the entire condition and logic related to table existence checks. Removing this code significantly reduced overhead and improved execution time.

The specific steps included:

1. Switching the **Code Generation Mode** from *Automatic* to *User Written Body*.
2. Deleting the redundant lines of code that checked for and created the table.



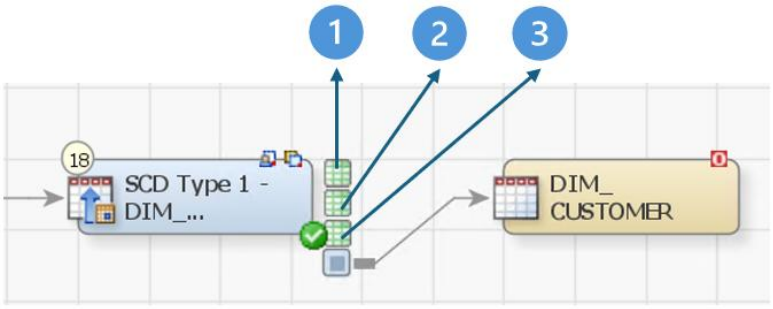


BREAKING THE MONOLITHIC

By default, SAS DI Studio deletes at the beginning of execution the temporary tables used in the **SCD Type 1 process**, such as reference tables, changed-records tables, and new-records tables.

To break away from this **monolithic program structure** and make the process more efficient through code reuse, we encapsulated this logic into a **macro**. This approach requires maintaining a **standard naming convention** for these tables across all jobs where an SCD Type 1 transformation is used.

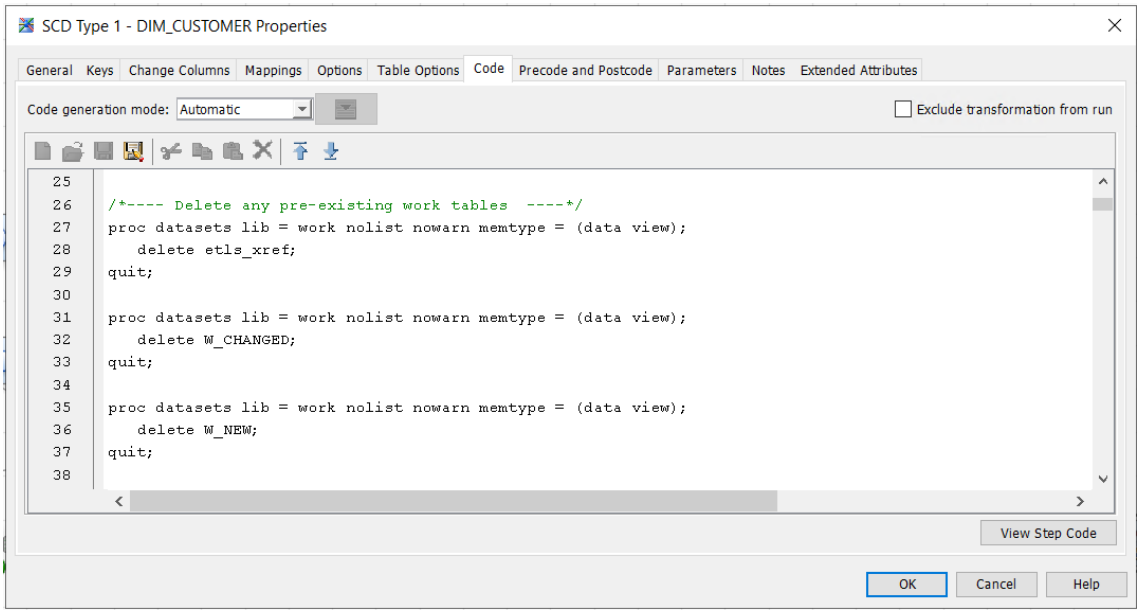
Since SAS DI Studio automatically generates random names for these tables, it becomes necessary to **rename them** and assign the predefined standardized names. For example:



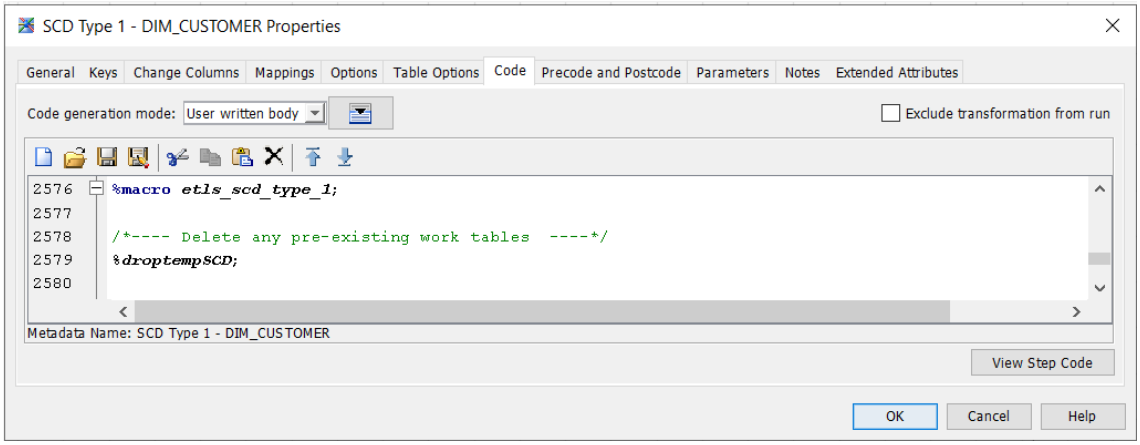
1. **Cross-reference table**, which we will always call **W_CROSS**
2. **Changed records table**, which we will always call **W_CHANGED**
3. New records table, which we will always call **W_NEW**

In this way, we replace the entire block of code generated by SAS DI Studio with the macro defined as **droptempSCD**, resulting in the following:

Before - Code generation mode: *Automatic*



After - Code generation mode: *User written body*

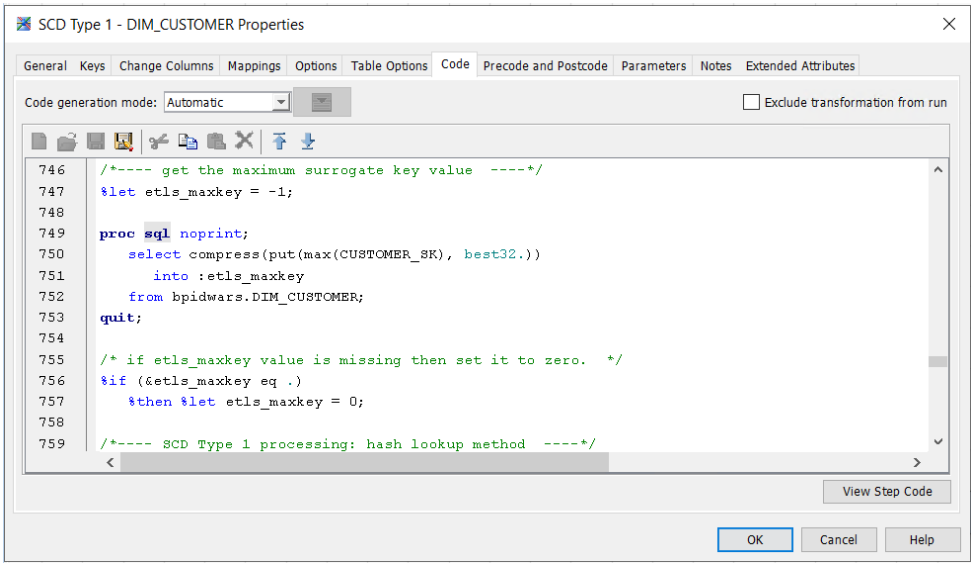


Macro code `droptempSCD`:

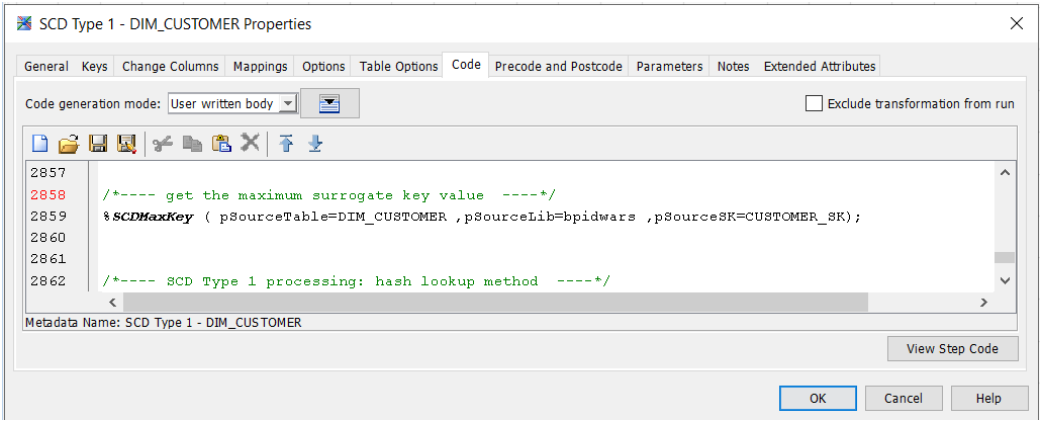
```
%macro droptempSCD (pLibname=WORK);

    proc datasets lib = WORK nolist nowarn memtype = (data view);
       delete ETLS_XREF ;
    quit;

    proc datasets lib = &pLibname nolist nowarn memtype = (data view);
       delete W_CHANGED W_NEW W_CROSS W_MAPPING;
    quit;

%mend;
```

In the same way, another block of code generated to obtain the **maximum surrogate key value** from the target table was replaced by a macro. This macro was designed with parameters that allow it to be reused across multiple jobs wherever applicable. The macro was defined as **`SCDMaxKey`**.

Before - Code generation mode: *Automatic*



After - Code generation mode: *User written body*



Código de la macro `SCDMaxKey`:

```
%macro SCDMaxKey ( pSourceTable=
                  ,pSourceLib=
                  ,pSourceSK=);


    %let etls_maxkey = -1;

    proc sql noprint noerrorstop;
        select compress(put(max(&pSourceSK), best32.))
          into :etls_maxkey
        from &pSourceLib..&pSourceTable;
    quit;

    /* if etls_maxkey value is missing then set it to zero.  */
    %if (&etls_maxkey eq .) %then %let etls_maxkey = 0;

%mend;
```

DIRECT EXECUTION IN THE DATABASE

SAS Data Integration Studio provides the ability to run processes **directly inside the database** using **SQL pass-through execution**. By leveraging this capability with Oracle, we achieved substantial performance improvements, reduced network traffic, and fully exploited the power of the Oracle database—an environment specifically designed to handle large-scale data processing.

Implementing pass-through execution corrected the deficiencies we observed with the standard SCD Type 1 transformation, which previously consumed excessive execution time and, in some cases, never completed, remaining stuck in an indefinite running state.

By moving the heavy-lifting logic from SAS to Oracle via pass-through SQL, we allowed the database engine to optimize query execution plans, parallelize operations, and use its own indexing strategies. This approach ensured that only the minimal amount of data needed to flow between SAS and Oracle, significantly streamlining the ETL process.

The following sections detail the optimizations applied through pass-through execution and how they improved runtime efficiency and overall stability of the SCD Type 1 transformation.

- **Create the changes table in the Oracle database** and insert the records identified as changed by the SAS process. This table is created and dropped each time the process runs, which made it necessary to develop supporting macros:
    - **pushChangetoDBMS**, to insert the identified changes into the Oracle table.
    - **dropChangeDBMS**, to remove the table at the end of execution.
    - **connORCL**, to establish and manage the connection to the Oracle database.

Code of the macro `dropChangeDBMS`:

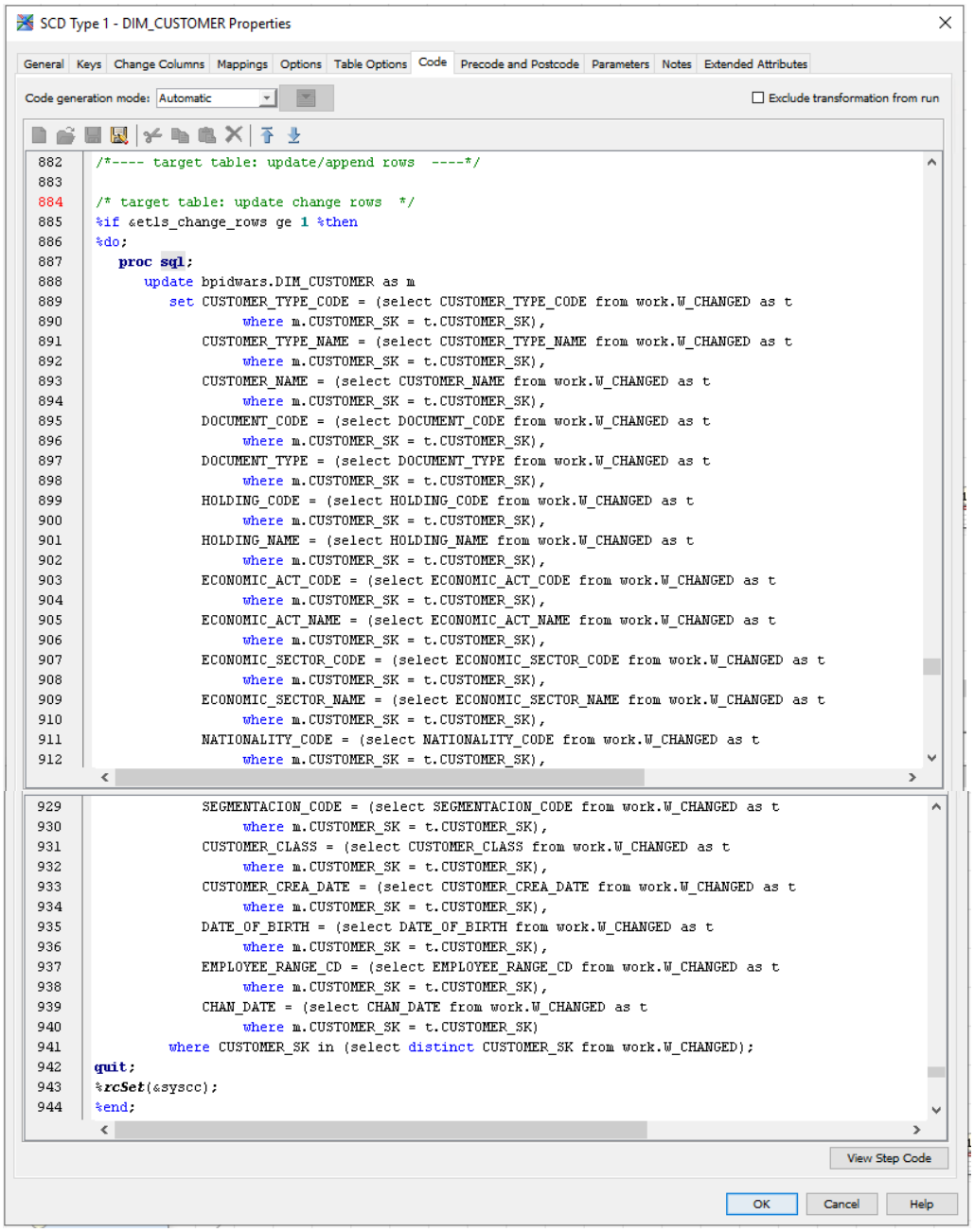
```
%macro dropChangeDBMS( pTNSPath=&_pathDWH
                      ,pSchema=
                      ,pLibname=
                      ,pConn=STAGE);

    %let vSchema = %str(%')&pSchema.%str(%');
    %let vtable = %str(%')W_CHANGED%str(%');


    %let etls_tableExist = %eval(%sysfunc(exist(&pLibname..W_CHANGE, DATA));

    %if (&etls_tableExist ne 0) %then %do;

        %put %str(NOTE: Drop SCD from DBMS...);
        proc sql noerrorstop;

                %connORCL ( pTNSPath=&pTNSPath ,pConn=&pConn);

                execute (drop table &pSchema..W_CHANGED purge) by ORACLE;

                disconnect from ORACLE;

        quit;

    %end;

    proc sql noerrorstop;

            %connORCL ( pTNSPath=&pTNSPath ,pConn=&pConn);

            execute (exec pbi_orcl.pkg_utilities.drop_table_if_exists(&vtable,
&vSchema)) by ORACLE;

            disconnect from ORACLE;

    quit;

%mend;
```

Code of the macro pushChangetoDBMS:

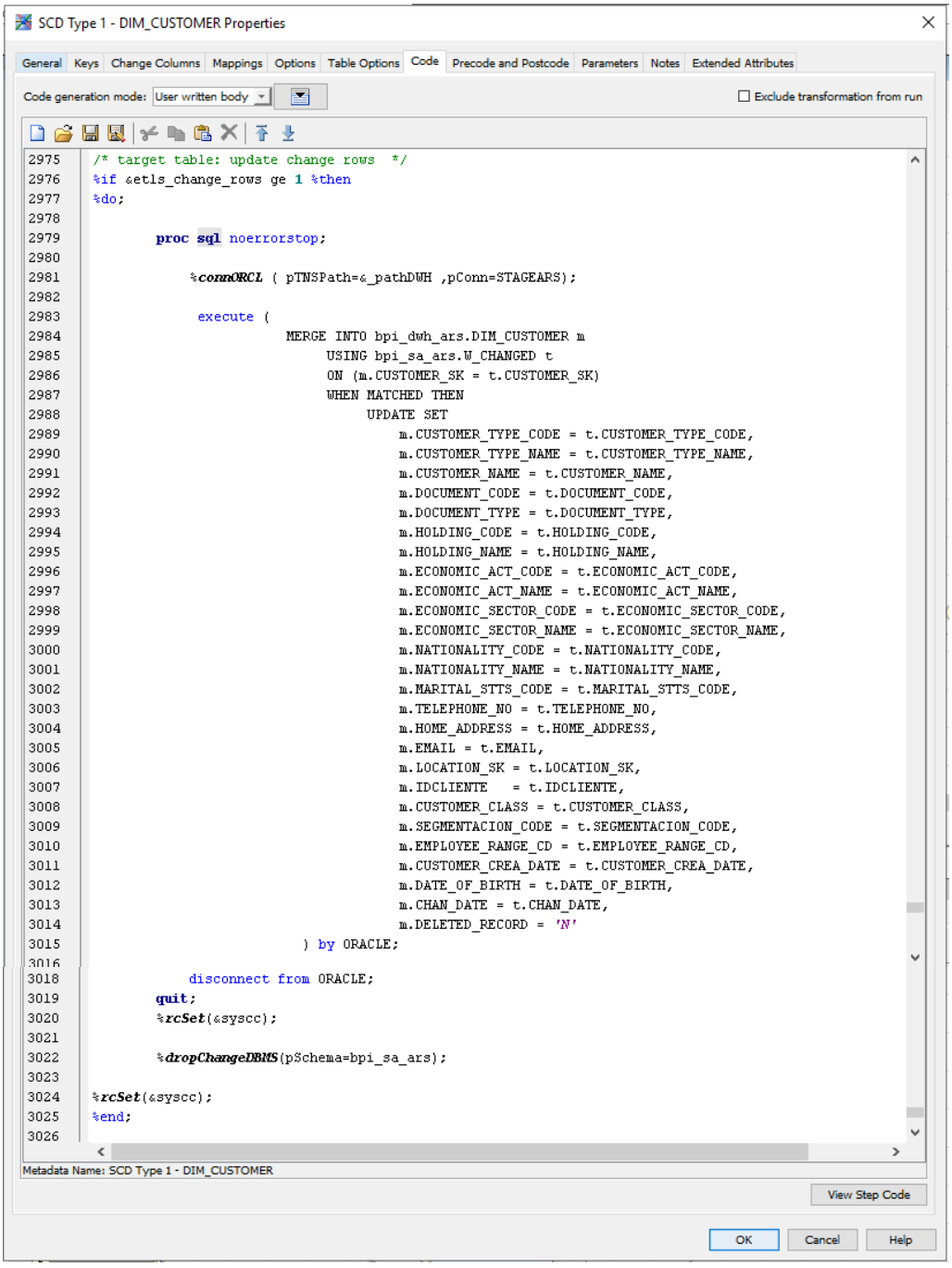
```
%macro pushChangetoDBMS(pTNSPath=&_pathDWH);

    %dropChangeDBMS(pTNSPath=&pTNSPath, pSchema=&saSchema,
pLibname=&saLibname);

    data &saLibname..W_CHANGED;
        set work.W_CHANGED;
        stop;
    run;

    proc append base=&saLibname..W_CHANGED data=WORK.W_CHANGED force nowarn;
    run;

%mend;
```

To update the changed records more efficiently, the process was executed directly in the Oracle database using its native `MERGE` operation. This approach updates the target table with data from the previously created changes table in Oracle, eliminating the inefficient `UPDATE` logic generated by SAS DI Studio. In the default SAS DIS implementation, for each record and for each field, the system executes a separate SELECT statement—an approach that significantly increases execution time. By replacing this with a single `MERGE` operation, the update process became far more streamlined, leveraging Oracle's built-in optimization and reducing overall processing overhead, as illustrated below:

Before - Code generation mode: *Automatic*



After - Code generation mode: *User written body*



ADDRESSING WEAKNESSES TO IMPROVE DATA CONSISTENCY.

The **SCD Type 1 transformation** does not detect records deleted in the source system. As a result, it lacks the ability to logically flag them as deleted in the target. Although in general it is not considered best practice to physically delete records, in this particular case records are removed at the source when a customer is deactivated or discontinued. Over time, this behavior leads to **inconsistencies and "orphaned" data** in the target table **DIM_CUSTOMER**.

To maintain consistency, a straightforward update is applied to **DIM_CUSTOMER** each time the process runs. This update flags records that no longer exist in the source system as logically deleted in the target, ensuring data accuracy and preventing the accumulation of stale or misleading entries.

The logic is implemented through a simple update statement, as shown in the following example:

```
/* Mark deleted records */
proc sql noerrorstop;

   update bpidwars.DIM_CUSTOMER cust
   set DELETED_RECORD = 'S'
   where cust.CUSTOMER_DIM_ID not in (select distinct c.CUSTOMER_DIM_ID from
work.etls_xref c)
   ;

quit;
```

CONCLUSION

The experience presented in this paper demonstrates that **monolithic ETL processes** developed in SAS Data Integration Studio can become a barrier to performance and scalability, particularly when working with high-volume databases such as Oracle. However, this does not diminish the value of SAS DIS. On the contrary, **SAS Data Integration Studio remains a robust and powerful platform for designing and orchestrating ETL and data pipeline processes**, offering a level of structure, governance, and maintainability that is often difficult to achieve when coding entirely from scratch.

Through the identification of critical bottlenecks, the analysis of auto-generated code, and the use of **modular, reusable macros**, we were able to transform a slow and inefficient process into a streamlined, sustainable, and high-performing solution. Modularization proved to be a key success factor, allowing us to reduce redundancies, simplify maintenance, and replicate best practices across projects.

Strategies such as eliminating unnecessary code, leveraging **pass-through execution** in the database, using Oracle's native functions like MERGE, and improving data consistency not only resolved the immediate performance challenges but also established a **replicable methodology** for future ETL projects.

For practitioners working with SAS DIS, we recommend adopting a **proactive approach**: review and optimize auto-generated code, embrace the flexibility provided by macros, and treat the database architecture as an ally rather than a constraint. By doing so, ETLs built with SAS DIS can evolve from monolithic, rigid flows into **modular, scalable, and efficient data pipelines** that meet modern performance demands while preserving the strengths of the platform.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Angye Rivero Rodríguez (angye.r@datanalitica.com)
Osmel Brito Bigott (osmel.b@datanalitica.com)
DatAnalítica
info@datanalitica.com