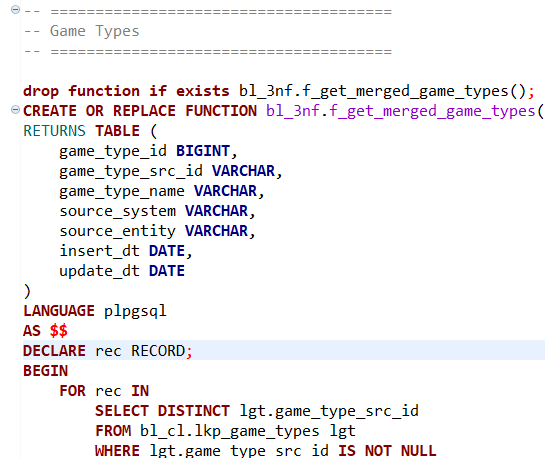
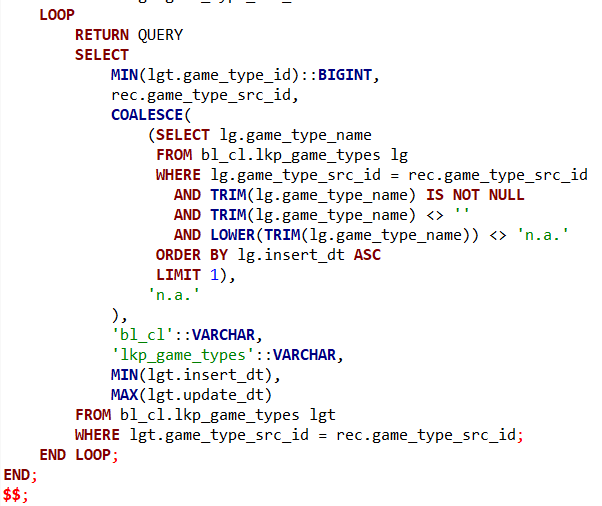
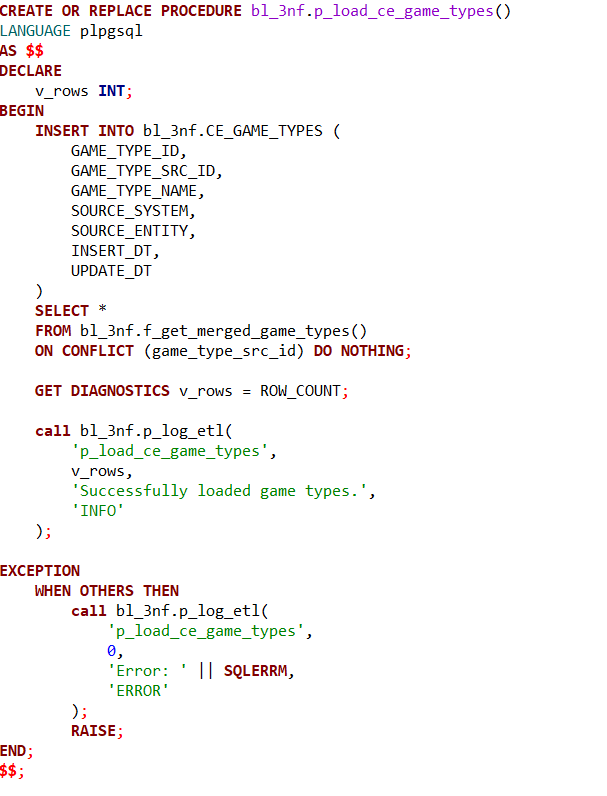


| Business Template  **Loading 3NF objects** |
| --- |
|  |



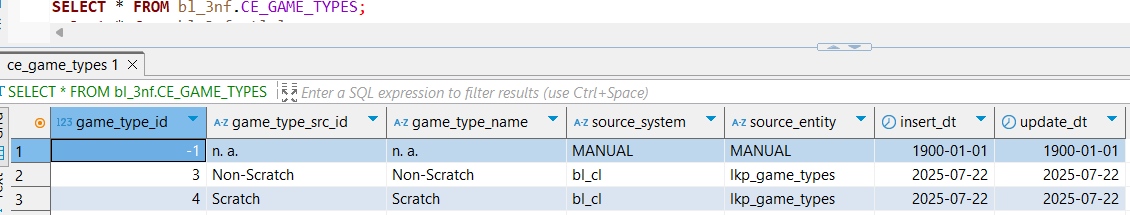




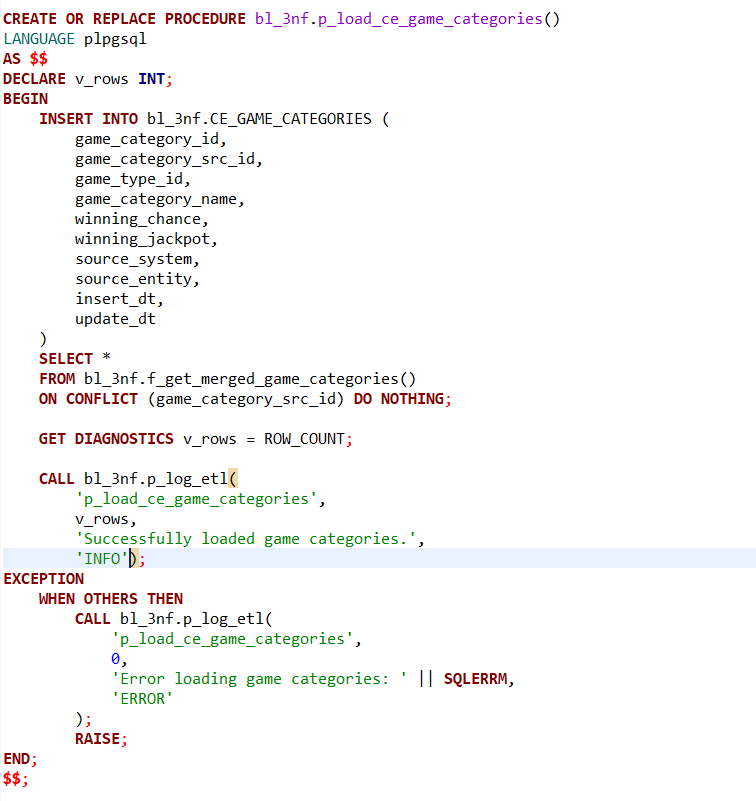
The logic behind this code is designed to consolidate game type data from a lookup table into a clean, deduplicated form suitable for further use. A key point to understand is that the source lookup table, bl\_cl.lkp\_game\_types, already implements Slowly Changing Dimension (SCD) Type 1 behavior. This means that when game type attributes change, the old values are overwritten with new ones rather than being preserved as historical versions. As a result, the source table always holds the most current and relevant information for each game type, identified uniquely by game\_type\_src\_id.

Given this setup, the function f\_get\_merged\_game\_types() focuses on extracting and merging this data efficiently. It loops through each distinct game\_type\_src\_id in the source, selecting the minimum game\_type\_id as a canonical identifier, and carefully picks the best available game type name by filtering out nulls, blanks, or placeholders like "n.a." It also gathers the earliest insertion date and latest update date to provide temporal context.

Because the source already applies SCD Type 1 logic, the downstream process does not need to handle complex versioning or track historical changes. Instead, it can simply insert the consolidated data into the target table, CE\_GAME\_TYPES, using a procedure that ensures no duplicates are created by relying on the game\_type\_src\_id uniqueness constraint. This procedure also includes robust logging for monitoring success or failure.







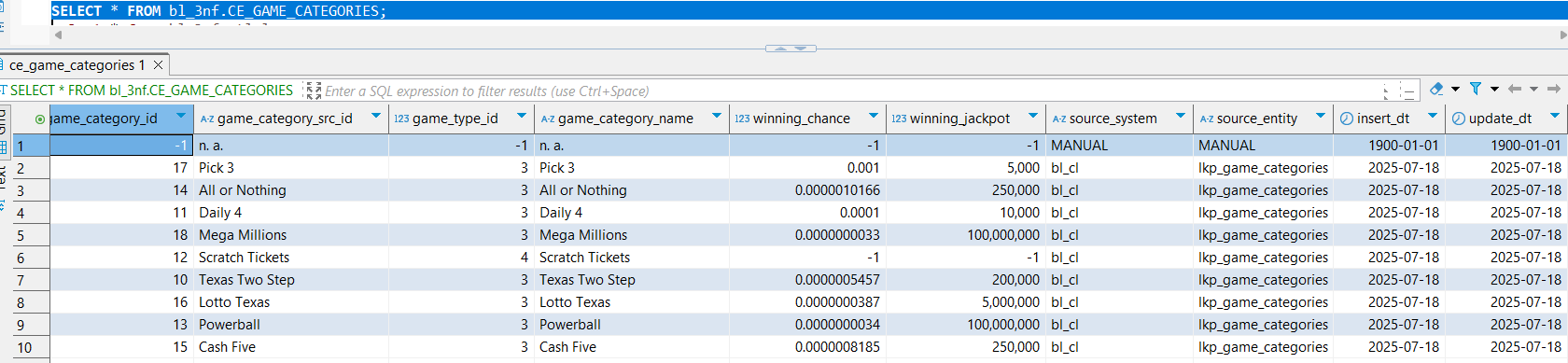
This logic focuses on consolidating and loading game category data from the source lookup table into the target table for further analysis or reporting. Alike the previous game types function, this logic uses 2 procedures to first aggregate data and then load it efficiently.

The source table, bl\_cl.lkp\_game\_categories, holds game category records with attributes such as game\_category\_src\_id, game\_category\_name, winning\_chance, and winning\_jackpot. To enrich and normalize this data, the function joins the categories with the already consolidated game types from bl\_3nf.CE\_GAME\_TYPES based on the game type name, thereby associating each category with the correct game\_type\_id.

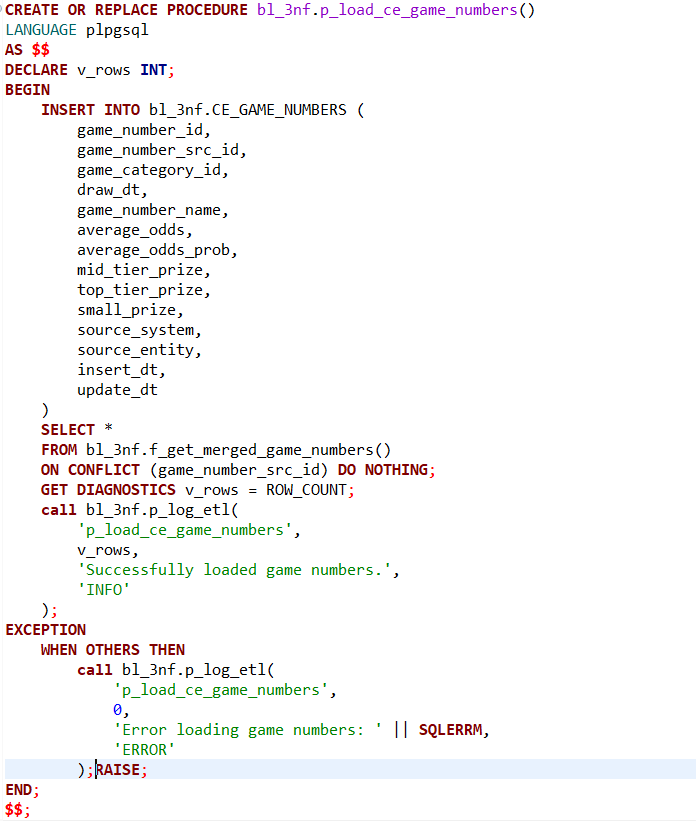
The aggregation step plays a crucial role in deduplication and quality control. For each unique game\_category\_src\_id and corresponding game\_type\_id, the function selects the maximum available game\_category\_id and filters out invalid or placeholder values for game\_category\_name (ignoring 'n.a.'), winning\_chance, and winning\_jackpot (ignoring -1 which likely represents missing data). If no valid values exist, it defaults to placeholders such as 'n.a.' or -1. The function also captures the earliest insertion date and latest update date, providing useful temporal context.

This approach assumes that the source data either implements SCD Type 1 logic or is otherwise maintained so that the latest valid snapshot is available. Because of this, the downstream loading procedure, p\_load\_ce\_game\_categories(), simply inserts the merged data into the target table, bl\_3nf.CE\_GAME\_CATEGORIES, avoiding duplicates by using the ON CONFLICT DO NOTHING clause keyed on game\_category\_src\_id.

Additionally, the procedure includes error handling and logging to monitor the ETL process, recording the number of rows inserted and any errors encountered. This ensures the process is both robust and transparent.







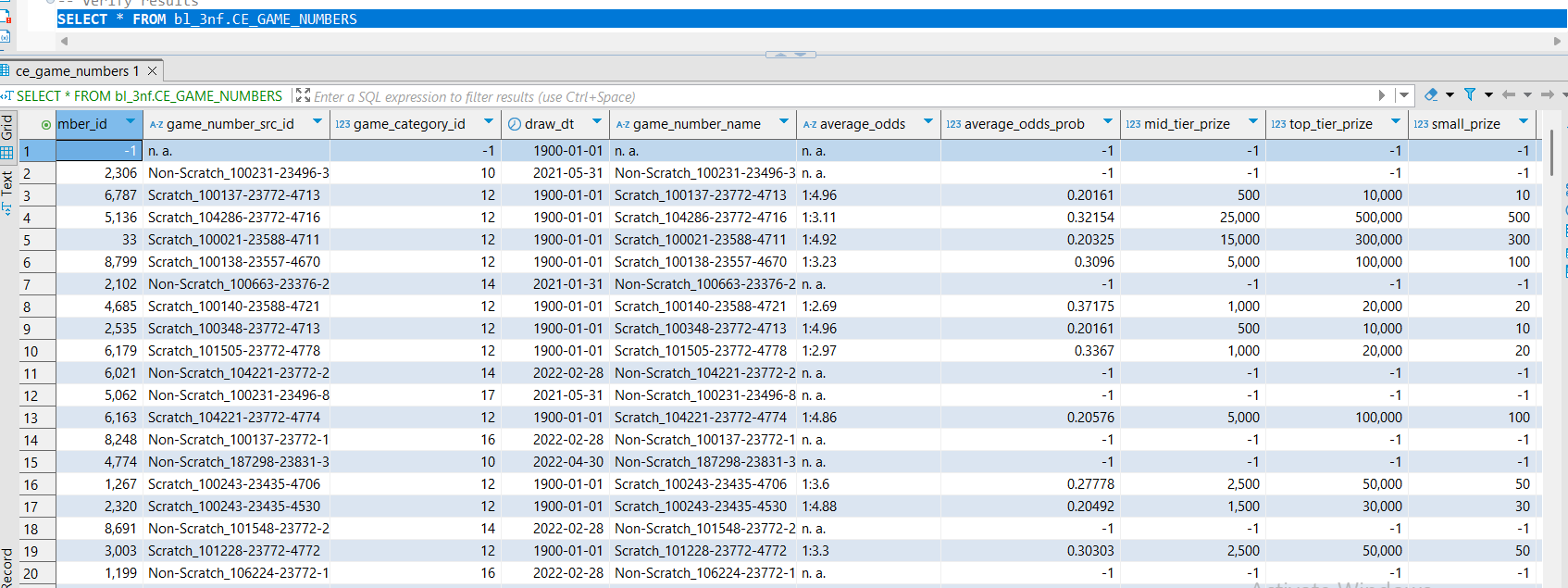
This segment of the ETL process focuses on consolidating and loading game number data from the source lookup table into a clean, normalized target table. The source table, bl\_cl.lkp\_game\_numbers, contains detailed attributes related to game numbers, including identifiers, draw dates, prize amounts, and odds information.

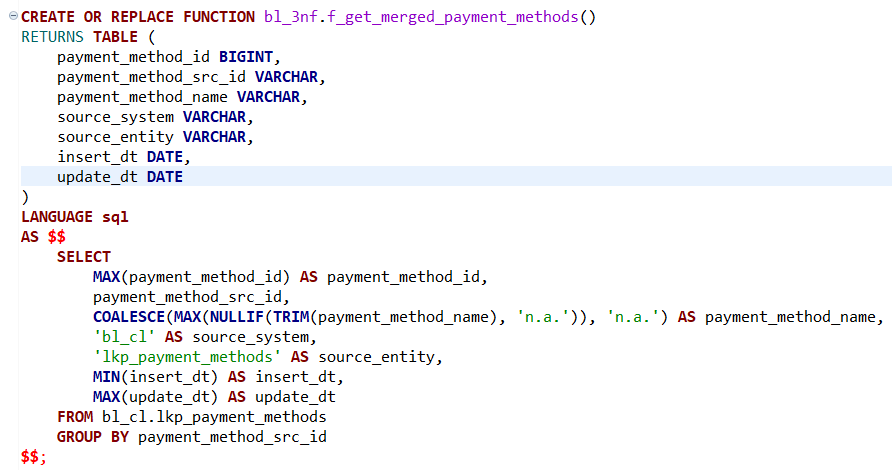
The function f\_get\_merged\_game\_numbers() plays a key role in merging and cleaning this data. It joins the game numbers with the previously consolidated game categories from bl\_3nf.CE\_GAME\_CATEGORIES by matching on the game category name. This enriches the game number records with the corresponding game\_category\_id, ensuring consistent linkage within the data model.

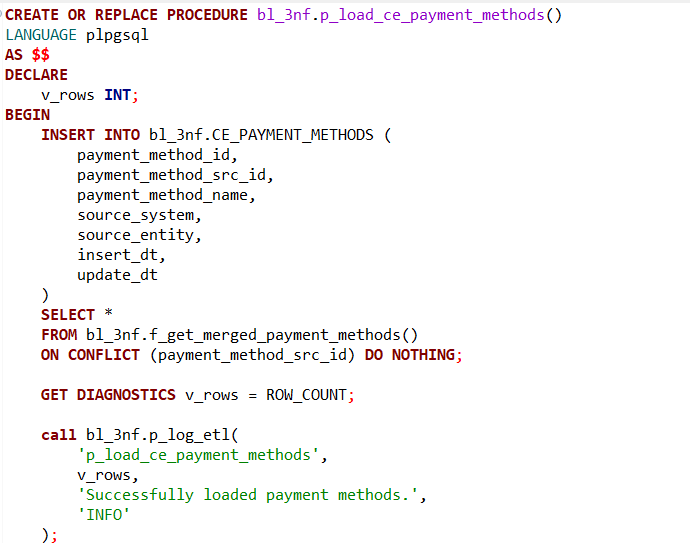
To handle data quality and duplicates, the function aggregates records by game\_number\_src\_id and game\_category\_id, selecting the maximum game\_number\_id as a canonical identifier. It filters out invalid or placeholder values—such as the default date 1900-01-01, the string 'n.a.' for names and odds, and -1 for numeric prize or probability fields—and replaces them with meaningful defaults when necessary. It also captures the earliest insertion date and latest update date to maintain temporal accuracy.

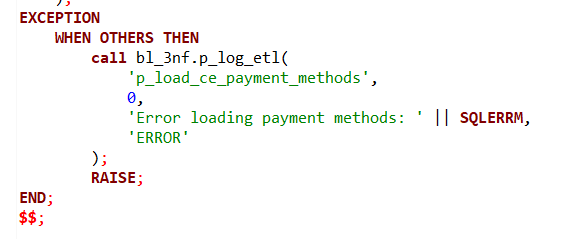
This logic assumes the upstream source data is either maintained with SCD Type 1 behavior or is otherwise updated so that only the most current snapshot of data exists. Therefore, the downstream loading procedure, p\_load\_ce\_game\_numbers(), can simply insert these merged records into the target table bl\_3nf.CE\_GAME\_NUMBERS, using the ON CONFLICT (game\_number\_src\_id) DO NOTHING clause to prevent duplication.

The procedure also includes robust error handling and logging. After the insert operation, it records the number of rows affected and logs success or failure messages via the centralized ETL logging procedure. This ensures that the ETL process is transparent, traceable, and resilient.









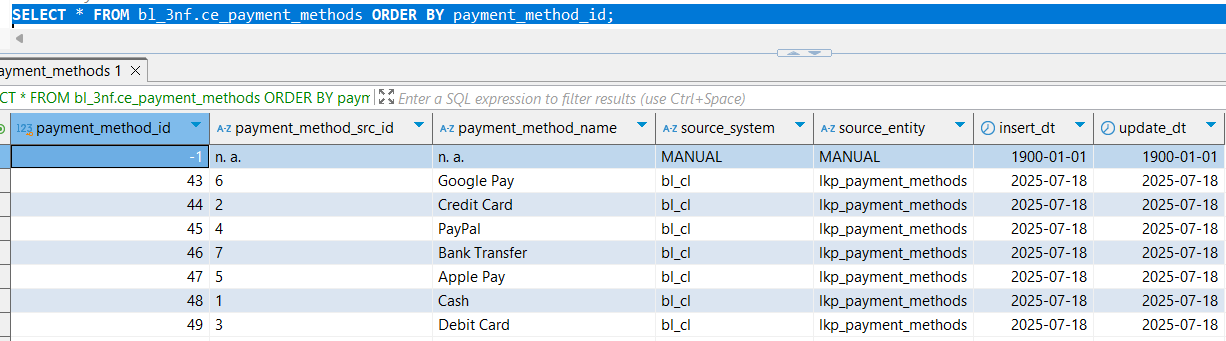
This part of the ETL pipeline is designed to extract, clean, and consolidate payment method data from the source lookup table into a standardized target table. The source data resides in bl\_cl.lkp\_payment\_methods, which contains records identified by payment\_method\_src\_id, along with descriptive names and metadata such as insert and update timestamps.

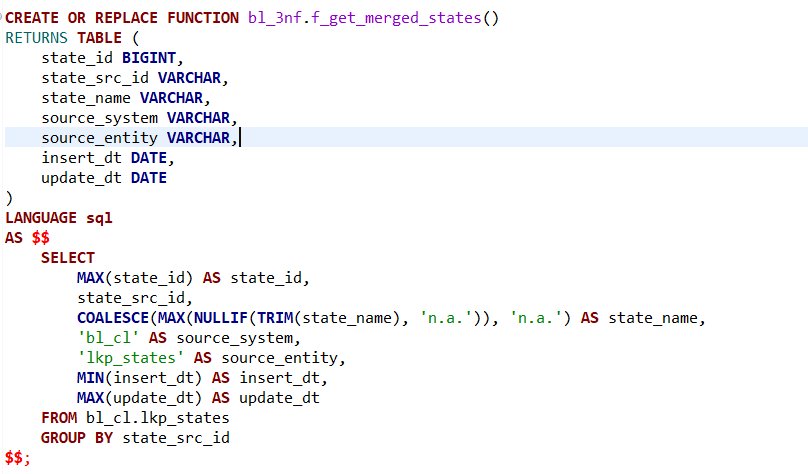
The function f\_get\_merged\_payment\_methods() handles the merging of this data. For each distinct payment\_method\_src\_id, it selects the highest payment\_method\_id as a representative identifier. The function also applies a data quality filter to the payment method names by removing values that are either blank or marked as placeholders like 'n.a.'. If a valid name cannot be found, it safely defaults back to 'n.a.'. The earliest insert date and latest update date are retained for reference, preserving the lifecycle timeline of the data.

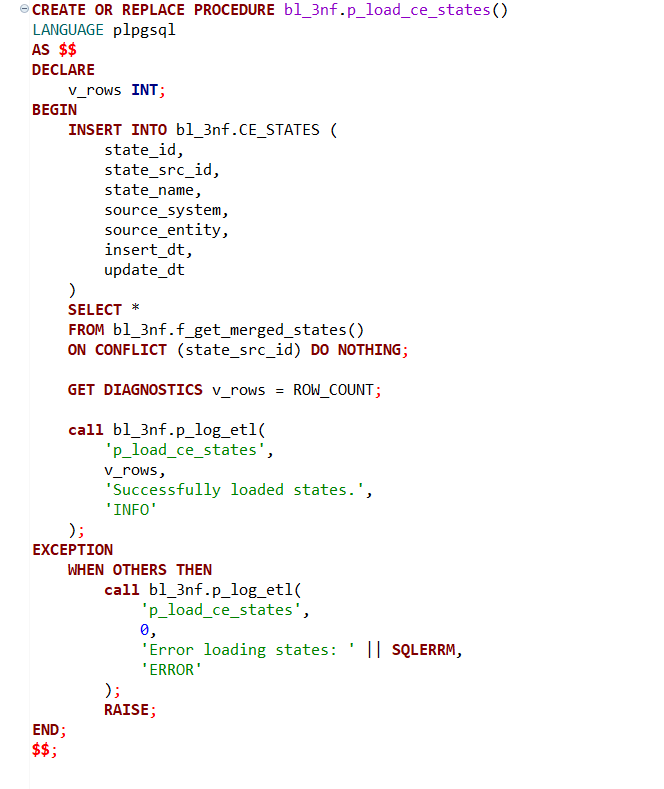
Since the source table is assumed to be managed under **Slowly Changing Dimension Type 1** (SCD Type 1) behavior, any changes to a payment method (such as a name correction) are simply overwritten in place, rather than tracked over time. This simplifies the logic: there’s no need to manage historical versions or perform change detection. The data in the source is treated as the latest known snapshot of truth.

The corresponding procedure p\_load\_ce\_payment\_methods() is responsible for loading this cleaned and deduplicated data into the target table bl\_3nf.CE\_PAYMENT\_METHODS. It ensures no duplicate entries are inserted by using the ON CONFLICT (payment\_method\_src\_id) DO NOTHING clause, relying on the assumption that payment\_method\_src\_id uniquely identifies each payment method. This keeps the target table consistent and avoids redundancy.

Additionally, the procedure includes diagnostics and logging. After the insert, it captures the number of rows affected and records this in an ETL log table using the centralized logging procedure. If an error occurs, it logs the failure along with the error message and rethrows the exception, supporting both traceability and operational monitoring.







This section of the ETL logic is responsible for integrating and standardizing geographic **state** information from a source lookup table into a clean, deduplicated target structure within the 3NF (Third Normal Form) schema.

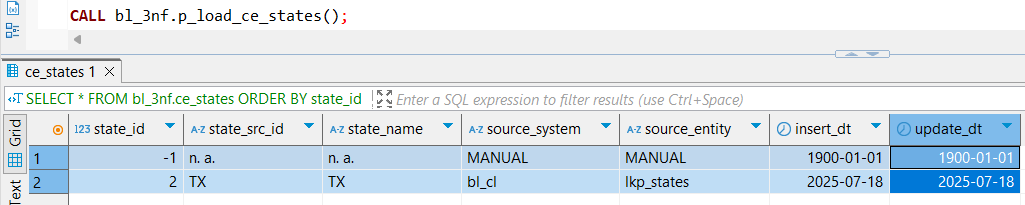
The function f\_get\_merged\_states() retrieves data from bl\_cl.lkp\_states, which contains raw entries for various states, each identified by a state\_src\_id and accompanied by descriptive metadata like state\_name, insert\_dt, and update\_dt. The function performs aggregation and quality checks to prepare the data for centralized storage.

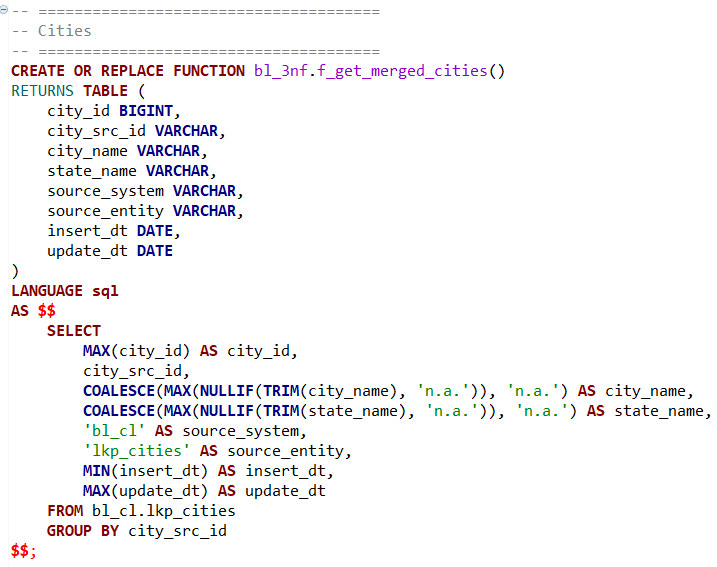
For each unique state\_src\_id, the function selects the maximum available state\_id as the representative key. It also ensures that state\_name values are meaningful by stripping out placeholder values such as 'n.a.' or empty entries via the NULLIF(TRIM(...), 'n.a.') construct. If no valid name is found for a state, the logic safely defaults to 'n.a.'. Additionally, the function captures the earliest insert\_dt and the most recent update\_dt, providing useful historical context for each record.

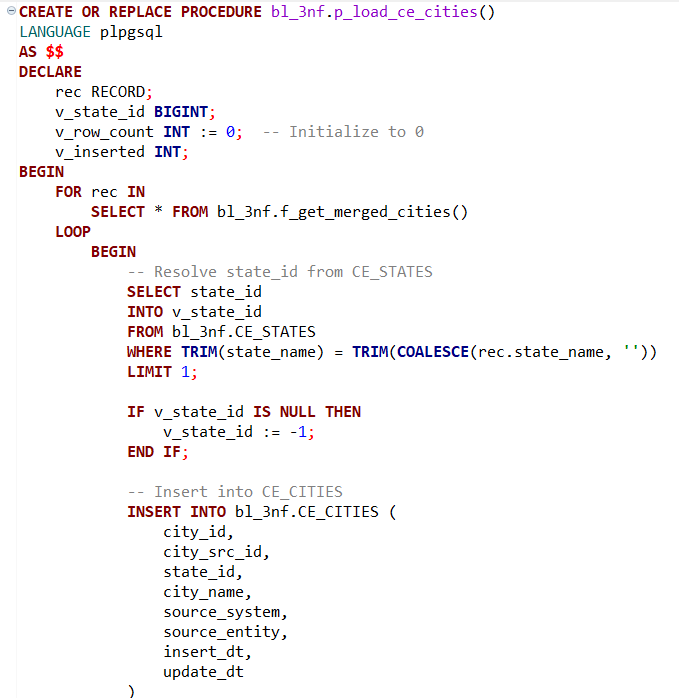
Because the source table is assumed to follow **Slowly Changing Dimension Type 1 (SCD Type 1)** practices, any changes to state names or identifiers overwrite previous values. This simplifies the downstream processing: the data is treated as the latest known version of the truth, without the need for tracking historical changes or applying delta logic.

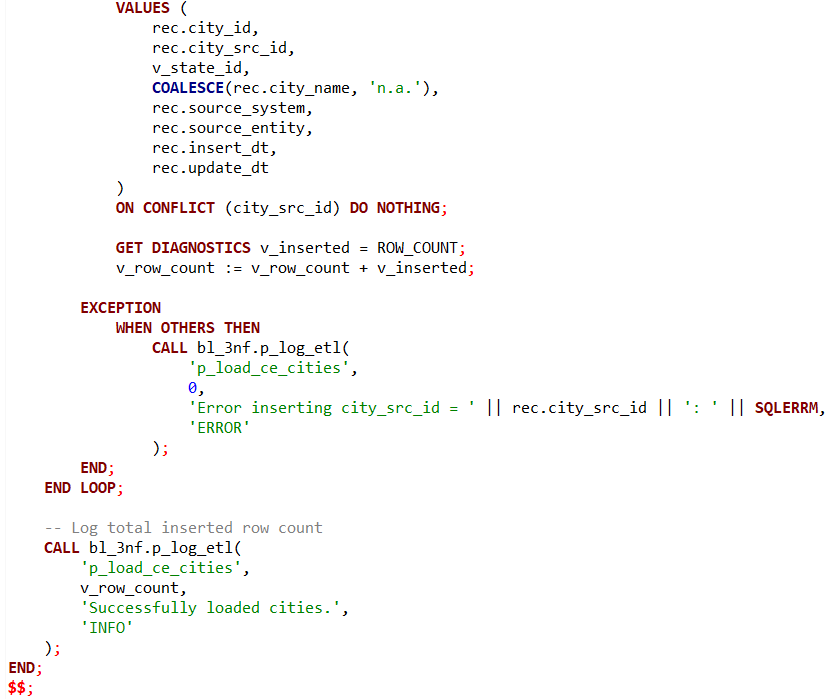
The procedure p\_load\_ce\_states() inserts the output of the function into the target table bl\_3nf.CE\_STATES. The ON CONFLICT (state\_src\_id) DO NOTHING clause ensures that duplicate entries based on state\_src\_id are ignored during the load. This prevents redundancy and maintains a one-to-one mapping between the source and target representations of each state.

To enhance traceability and monitoring, the procedure includes diagnostics and error handling. After the insert operation, the number of rows affected is recorded using GET DIAGNOSTICS, and an ETL log entry is written via the centralized p\_log\_etl procedure. In case of failure, the procedure captures and logs the error message before re-raising the exception, ensuring transparency and operational resilience.









This part of the ETL pipeline handles the extraction, transformation, and loading of **city-level** reference data into a clean, standardized structure. It draws from the raw source table bl\_cl.lkp\_cities, consolidating city records and linking them to their corresponding state records in the target dimension table.

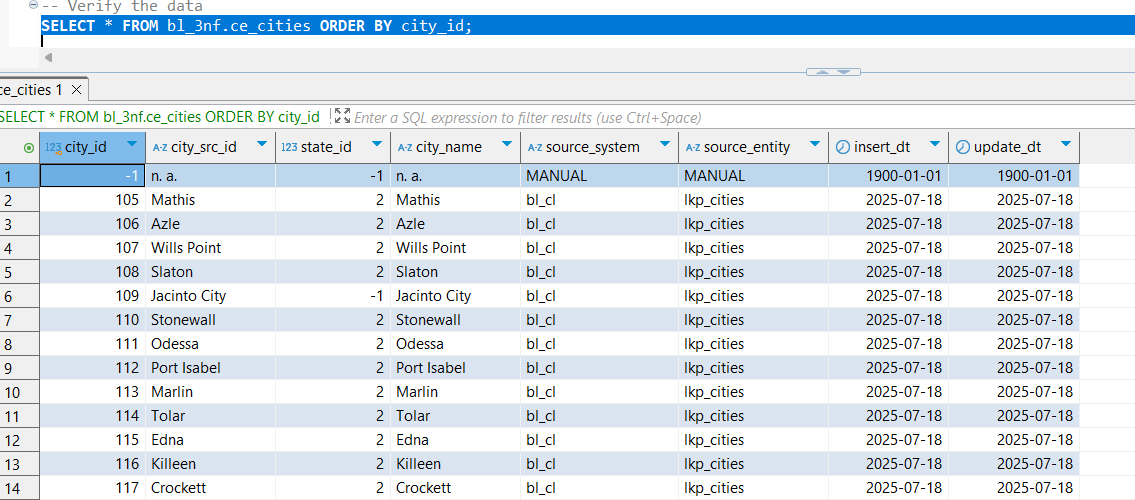
The function f\_get\_merged\_cities() performs a deduplication and cleanup process over the raw lkp\_cities data. For each unique city\_src\_id, it selects the maximum city\_id as the representative identifier. Additionally, it uses COALESCE(MAX(NULLIF(...))) logic to derive the best available values for city\_name and state\_name, discarding meaningless placeholders such as 'n.a.'. It also captures metadata such as insert\_dt and update\_dt, pulling the earliest and latest values respectively for each record.

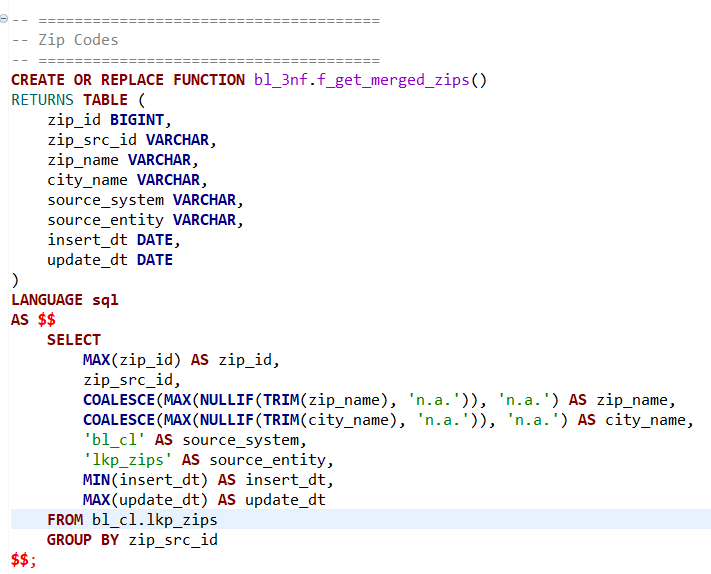
The transformation is based on the assumption that the lookup data in the source system adheres to **Slowly Changing Dimension Type 1 (SCD Type 1)** principles. This means any updates to the descriptive fields of a city (such as name or state name) overwrite previous values in-place, which eliminates the need to track historical changes explicitly. As a result, the ETL logic only needs to work with the current, most up-to-date snapshot of the data.

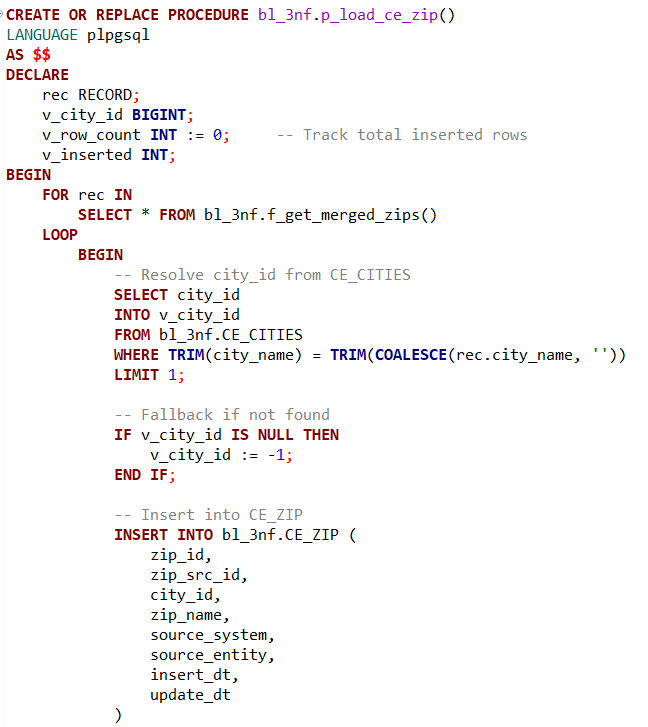
The procedure p\_load\_ce\_cities() loads the transformed data into the bl\_3nf.CE\_CITIES table. It processes each row individually to handle a special requirement: resolving the correct state\_id from the CE\_STATES table. This mapping is essential for establishing a foreign key relationship between cities and states in the 3NF model. The procedure attempts to match on state\_name, using a case-trimmed comparison. If no match is found, a default value of -1 is assigned to indicate an unresolved or unknown state.

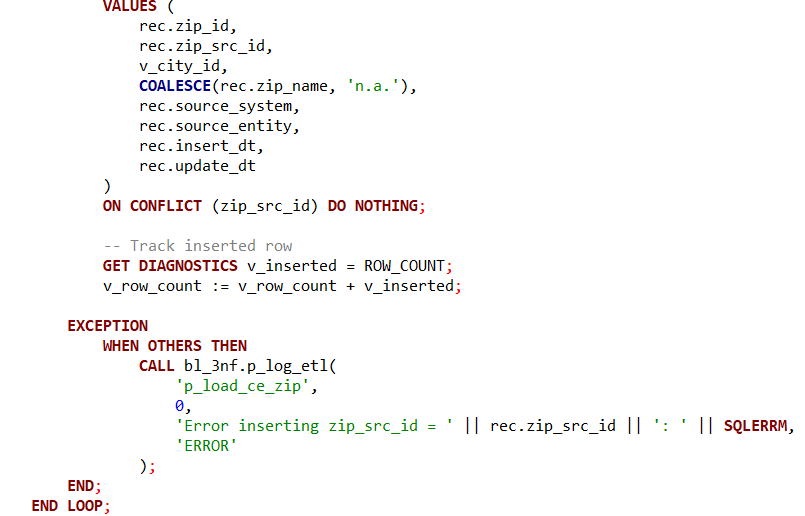
Each insertion is wrapped in a BEGIN ... EXCEPTION ... END block to gracefully handle any potential issues on a per-row basis. If an error occurs while inserting a particular city, it is logged using p\_log\_etl, capturing the failed city\_src\_id and the error message. This ensures that the overall ETL process continues even in the presence of individual record issues, improving robustness.

To maintain uniqueness, the insert statement includes ON CONFLICT (city\_src\_id) DO NOTHING, which skips any duplicate city entries already loaded. After all cities have been processed, the total number of successfully inserted records is logged.









This section of the ETL pipeline is designed to manage the integration and loading of ZIP code reference data into a centralized and structured format within the data warehouse. The logic focuses on standardizing ZIP records and establishing referential integrity with cities.

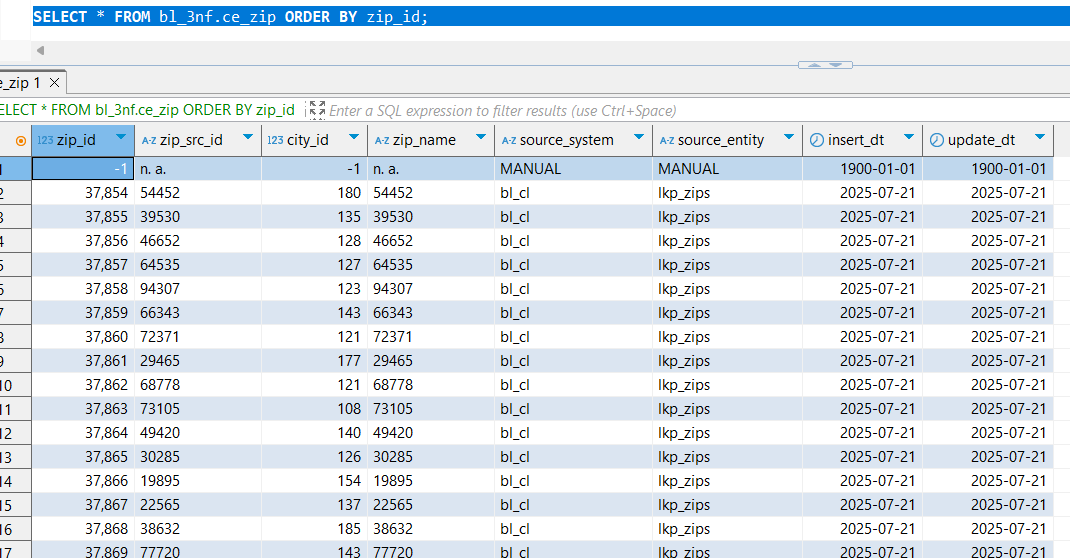
The function f\_get\_merged\_zips() performs the initial data consolidation. It extracts ZIP code records from the raw source table bl\_cl.lkp\_zips and groups them by their zip\_src\_id. For each group, it selects the maximum zip\_id as the primary identifier and uses conditional aggregation (COALESCE(MAX(NULLIF(...)))) to ensure that invalid placeholders such as 'n.a.' are ignored in favor of actual values when available. Additionally, the function derives insert\_dt and update\_dt by selecting the earliest and latest timestamps, respectively, to maintain basic audit tracking. The result of this function is a clean, deduplicated, and enriched snapshot of ZIP code data.

Since the staging layer is implemented with **Slowly Changing Dimension Type 1 (SCD Type 1)** semantics, it always reflects the most recent and accurate values for each source record. This simplifies the downstream loading process, as no history tracking or versioning logic is required—data can simply be inserted or ignored based on uniqueness.

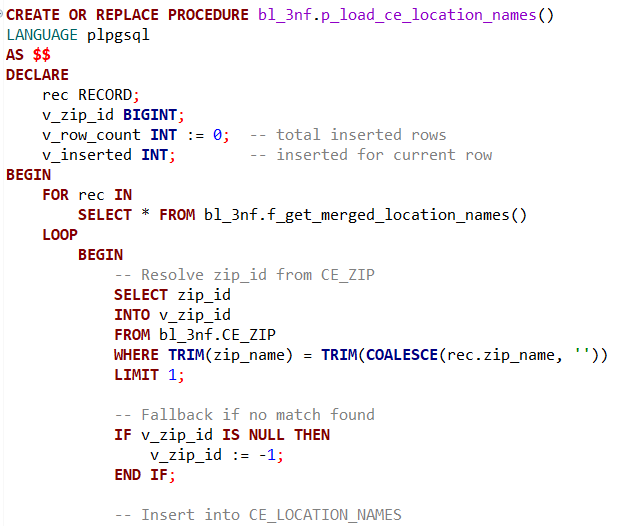
The procedure p\_load\_ce\_zip() iterates over the cleaned ZIP records returned by the function and handles the logic necessary to load them into the CE\_ZIP dimension table. A critical part of this process is resolving the city\_id from the CE\_CITIES table based on the city\_name. This lookup ensures that each ZIP code is correctly linked to a city, preserving referential integrity in the 3NF model. If no corresponding city is found, the logic assigns a default value of -1 to city\_id, indicating that the relationship could not be resolved.

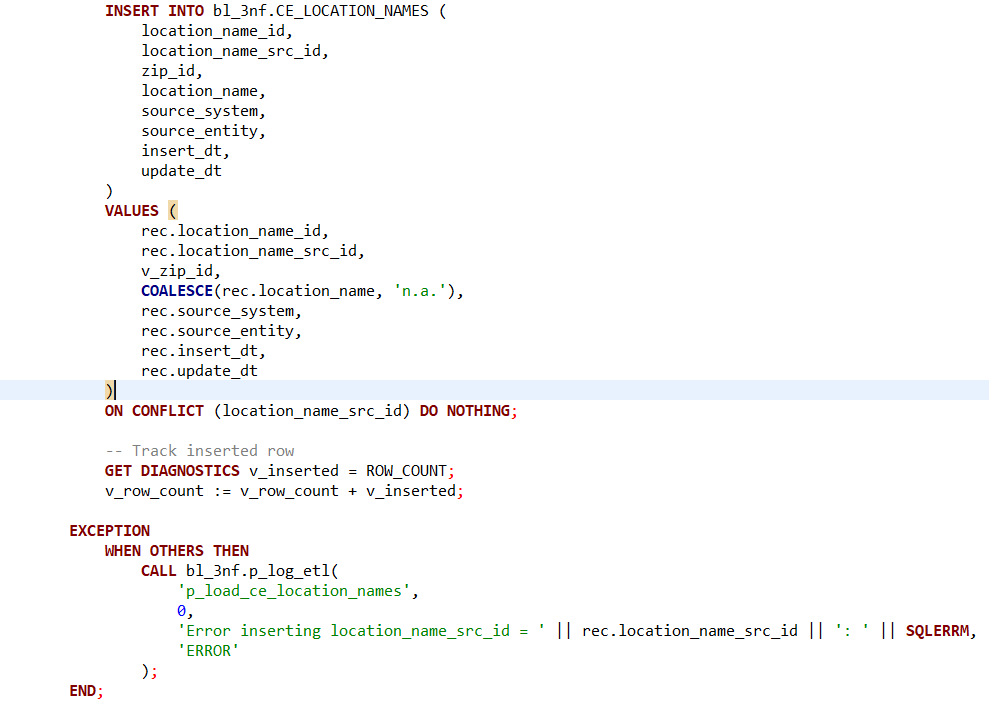
The procedure inserts the ZIP records one by one using a FOR loop and tracks the number of successful insertions. It employs a BEGIN...EXCEPTION...END block within the loop to ensure that errors with individual records do not halt the entire load process. Any insertion failures are logged through p\_log\_etl, including the offending zip\_src\_id and a detailed error message.

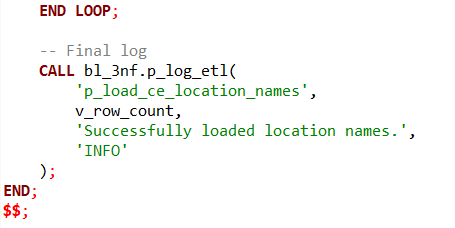
To maintain idempotency and prevent duplicate data, the insertion is guarded with ON CONFLICT (zip\_src\_id) DO NOTHING, which ensures that only new ZIP codes are added. After processing all records, the total number of inserted rows is logged using p\_log\_etl, providing visibility into the ETL run's outcome.









This portion of the ETL framework is responsible for transforming and loading **location name** reference data into the third normal form (3NF) model. The logic ensures that location names are clean, unique, and relationally connected to ZIP codes, creating a well-structured geographic hierarchy within the enterprise data warehouse.

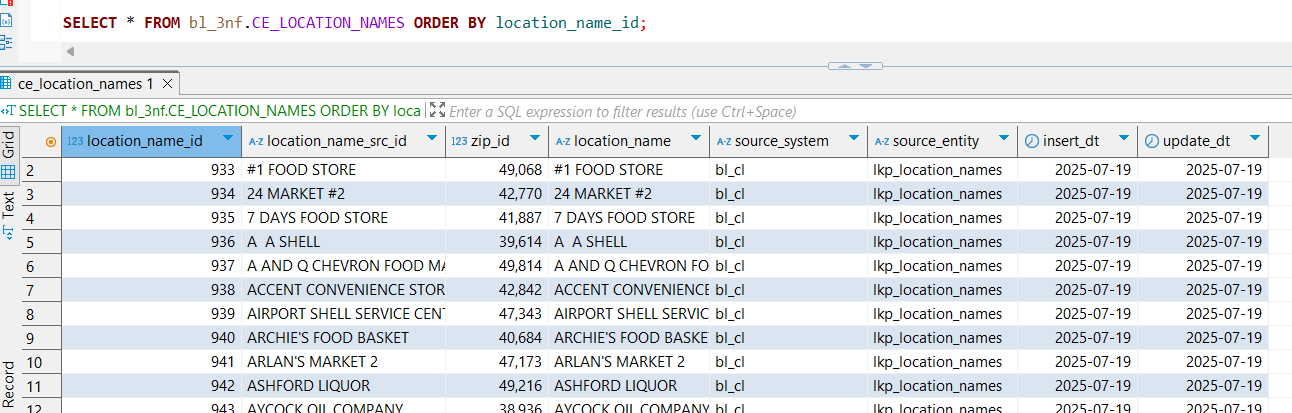
The process begins with the SQL function f\_get\_merged\_location\_names(), which retrieves raw location data from bl\_cl.lkp\_location\_names. The function performs basic data cleansing—trimming whitespace from location\_name and zip\_name—and assigns metadata such as the source system (bl\_cl) and source entity (lkp\_location\_names). Unlike other functions in the pipeline, this function does not use MAX() or COALESCE() for deduplication because the input data is assumed to be clean and already uniquely grouped by location\_name\_src\_id. As a result, the GROUP BY clause matches all fields from the SELECT, acting more as a syntactic requirement than a transformation mechanism.

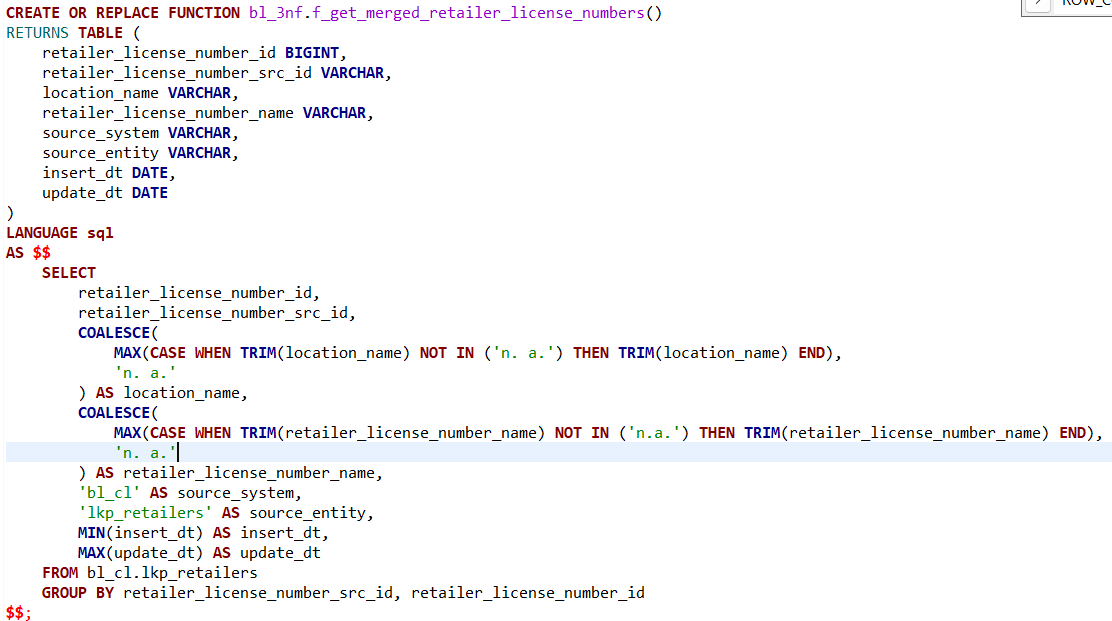
The corresponding procedure p\_load\_ce\_location\_names() is implemented in PL/pgSQL and iterates over the rows returned by the function. For each record, it attempts to resolve the corresponding zip\_id from the CE\_ZIP table using a match on zip\_name. This is a key step in enforcing referential integrity, ensuring that every location name is associated with a known ZIP code. If a match is not found, the logic defaults zip\_id to -1, signaling a missing relationship while still allowing the load to proceed.

Each location record is inserted into the CE\_LOCATION\_NAMES table using a parameterized INSERT statement. To maintain idempotency and avoid duplicates, the operation includes the clause ON CONFLICT (location\_name\_src\_id) DO NOTHING, which skips inserts for records that already exist. The procedure tracks how many rows were successfully inserted using the GET DIAGNOSTICS mechanism and aggregates this count across the loop for final logging.

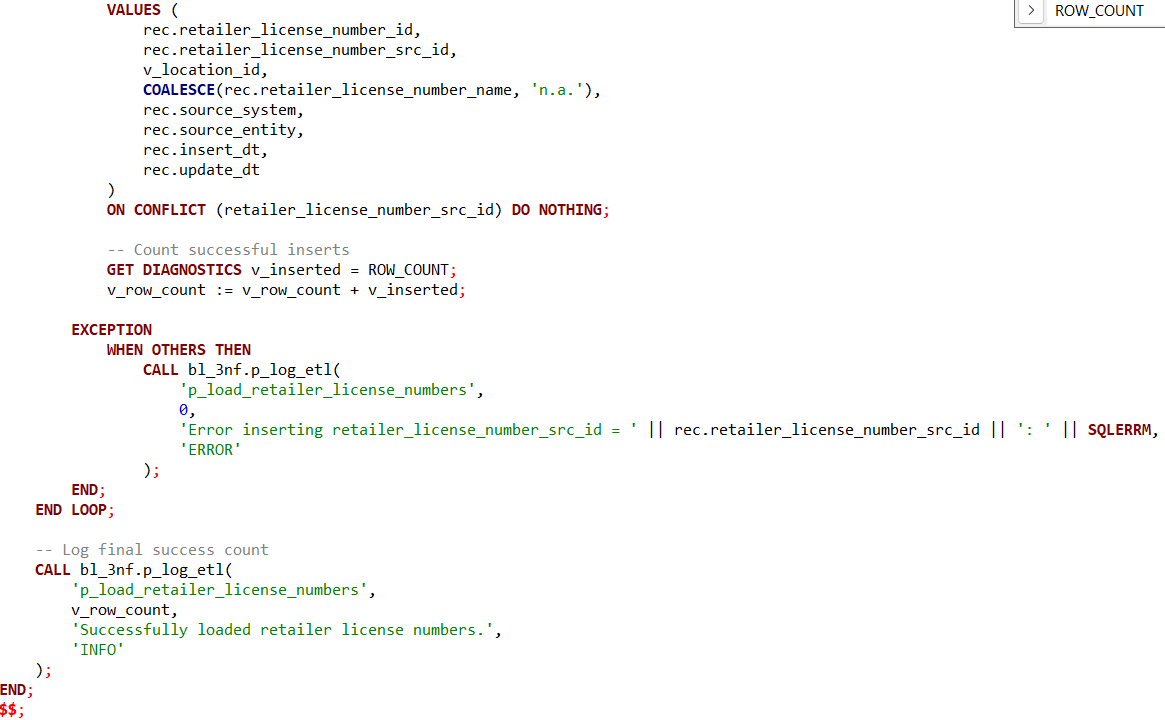
The procedure also includes robust error handling. Any errors that occur during the insertion of a specific record are caught and logged via the p\_log\_etl procedure. This ensures that faulty records do not interrupt the overall ETL process and that all errors are traceable by source identifier (location\_name\_src\_id) and message.

At the conclusion of the load, a final summary log is written indicating the total number of inserted records. This provides transparency and accountability for the ETL run, supporting auditability and operational monitoring.









This ETL routine is designed to extract, transform, and load **retailer license number** reference data into the 3NF CE\_RETAILER\_LICENSE\_NUMBERS table. It plays a critical role in establishing a clean and normalized structure that links retailer license records to their corresponding geographic locations, enabling traceability, compliance validation, and advanced reporting.

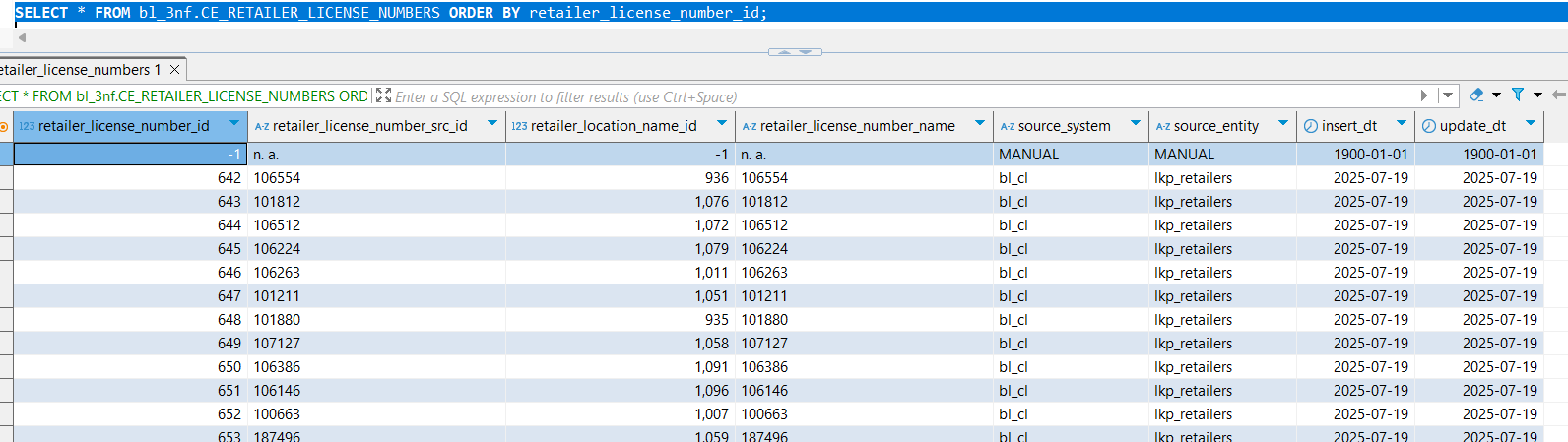
The process begins with the SQL function f\_get\_merged\_retailer\_license\_numbers(), which queries the source bl\_cl.lkp\_retailers lookup table. The function applies transformation logic to clean and consolidate data. It uses MAX(CASE WHEN ...) to extract the most meaningful non-null and non-placeholder ('n.a.' or 'n. a.') values for both location\_name and retailer\_license\_number\_name. This approach ensures that each license record is associated with its most valid descriptive metadata, while defaulting to 'n. a.' when necessary. The function also groups the data by both retailer\_license\_number\_src\_id and retailer\_license\_number\_id, assuming the source may contain multiple rows per license that need consolidation.

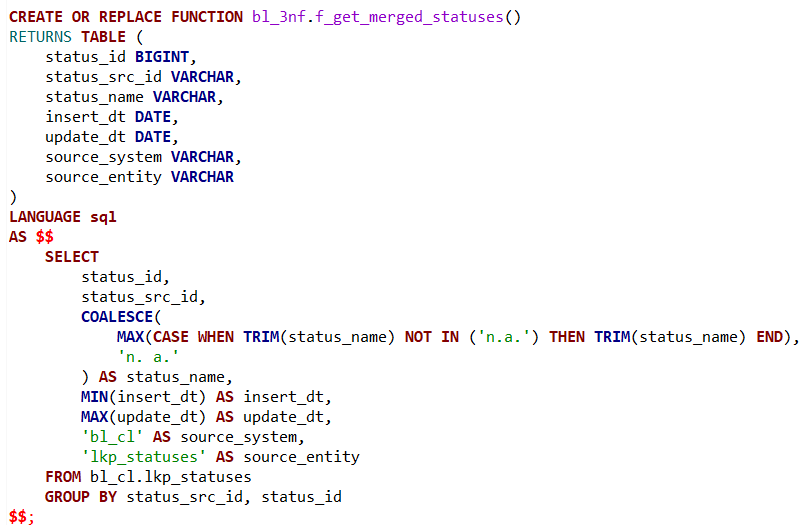
The associated procedure, p\_load\_retailer\_license\_numbers(), iterates over the cleaned dataset and attempts to resolve a foreign key relationship with CE\_LOCATION\_NAMES. It looks up the location\_name\_id by matching the trimmed location name from the current record with the entries in the location names dimension. If no match is found, it assigns a default value of -1, preserving the referential structure while signaling an unresolved dependency.

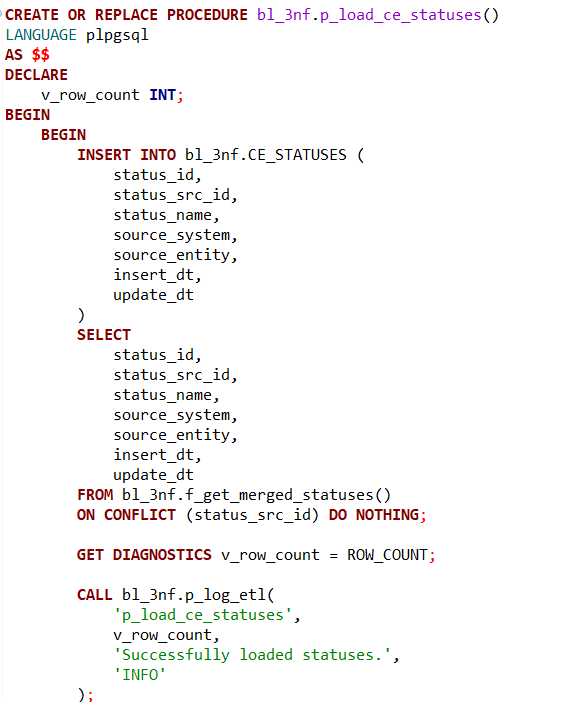
The core of the loop is an INSERT statement that loads each retailer license record into the CE\_RETAILER\_LICENSE\_NUMBERS table. The fields include both identifiers (id, src\_id), descriptive name, location foreign key, and audit metadata. The use of ON CONFLICT (retailer\_license\_number\_src\_id) DO NOTHING enforces idempotency by preventing duplication on repeated loads.

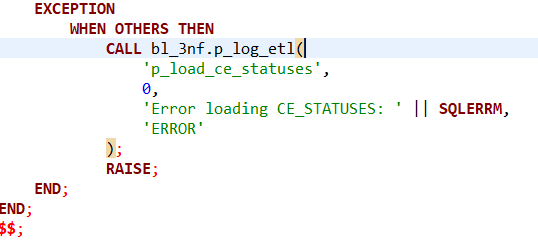
For each row processed, the procedure tracks the number of successful insertions using GET DIAGNOSTICS. This value is aggregated in v\_row\_count to produce a final load count. Any exceptions during individual insert operations are caught and logged via the p\_log\_etl procedure, capturing the source ID and detailed error message. This fine-grained error logging allows the ETL to continue running despite isolated failures, improving resilience and observability.

At the end of the procedure, a summary log entry is recorded to confirm the number of successfully inserted retailer license number records. This entry is stored using the centralized p\_log\_etl mechanism, which supports ETL auditing, monitoring, and troubleshooting.









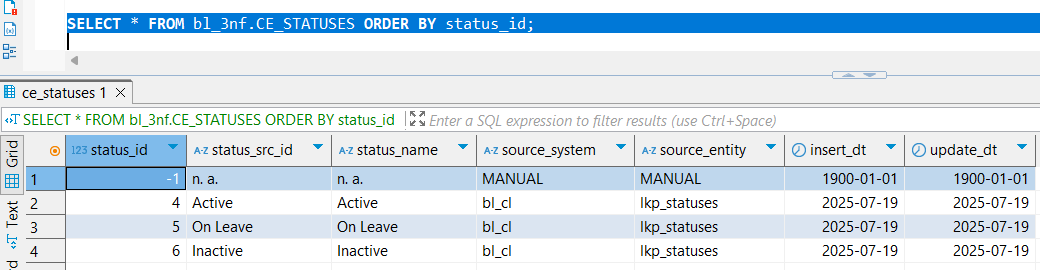
The ETL routine for **Statuses** in the bl\_3nf schema facilitates the normalization and consolidation of status reference data into the central entity table CE\_STATUSES. This process supports consistent status tracking across the data model and enhances downstream analytics, particularly in areas like lifecycle state management or compliance reporting.

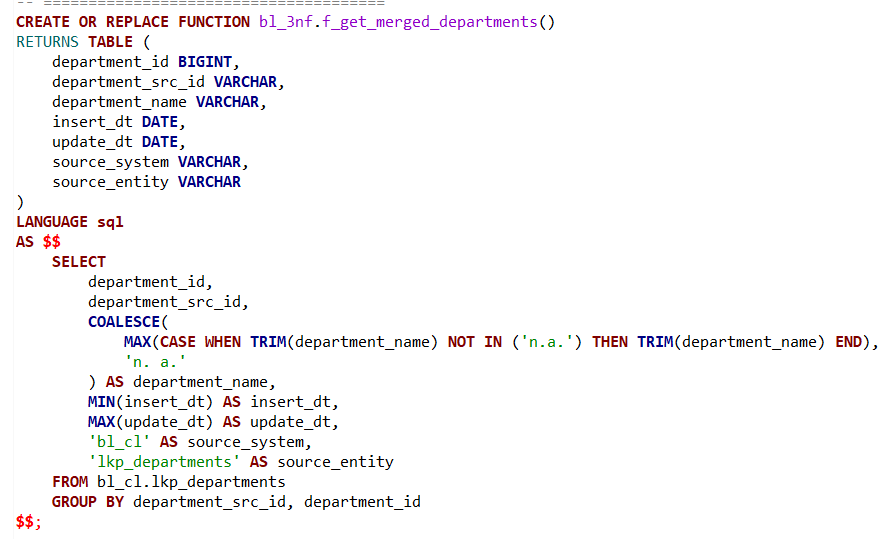
The function f\_get\_merged\_statuses() is responsible for extracting and preparing the raw status data from the bl\_cl.lkp\_statuses source table. Within this function, each record is grouped by its source and internal identifiers—status\_src\_id and status\_id. This grouping enables deduplication and consolidation of redundant entries.

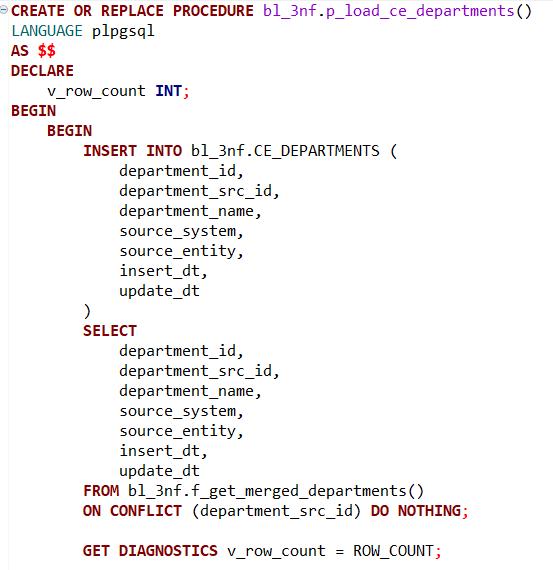
For the transformation step, the function applies a MAX(CASE WHEN ...) pattern to ensure that placeholder values such as 'n.a.' are ignored when selecting the most accurate status\_name. If no valid name is found, it defaults to 'n. a.', preserving data completeness. Timestamps are also normalized: MIN(insert\_dt) and MAX(update\_dt) are used to retain the earliest and most recent known activity for each logical status. Metadata fields such as source\_system and source\_entity are hardcoded to reflect the origin ('bl\_cl') and the staging table ('lkp\_statuses'), enabling traceability in a multi-source environment.

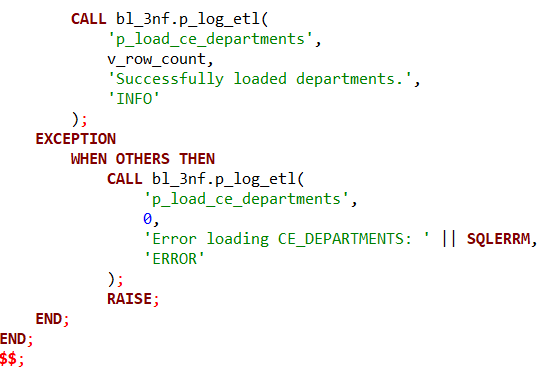
The corresponding procedure p\_load\_ce\_statuses() orchestrates the loading of this curated dataset into the target CE\_STATUSES table. It begins by invoking the function and attempting to insert each record. The use of ON CONFLICT (status\_src\_id) DO NOTHING enforces uniqueness based on the source ID and guarantees idempotency—subsequent ETL runs won’t duplicate existing entries.

Insert success is measured using the GET DIAGNOSTICS command, which captures the number of rows successfully inserted into the table. This count is passed into a logging mechanism via p\_log\_etl, which provides a standardized audit trail and supports operational monitoring of the ETL process. Should any exceptions occur, the procedure catches them and logs an appropriate error message, including the error details via SQLERRM, before re-raising the exception to halt further execution. This ensures both visibility and robustness in the face of unexpected failures.









The ETL process for **Departments** in the bl\_3nf schema ensures that department reference data is standardized, de-duplicated, and reliably inserted into the CE\_DEPARTMENTS central entity table. This step is critical for maintaining a clean dimensional model, supporting operations such as organizational reporting, compliance validation, and cross-entity mappings.

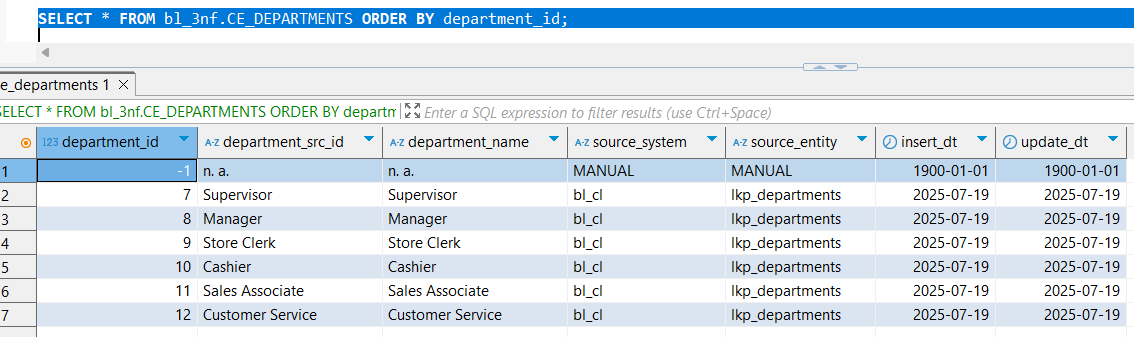
The process starts with the function f\_get\_merged\_departments(), which performs the extraction and transformation of data from the raw source table bl\_cl.lkp\_departments. This function handles potential inconsistencies in naming by applying a MAX(CASE WHEN ...) expression to select the most meaningful, non-placeholder department name. Specifically, it filters out values like 'n.a.' and ensures trimmed values are used for consistency. In the absence of a valid department name, a default value 'n. a.' is assigned to maintain data integrity.

To accurately track changes and lineage, the function includes timestamp management via MIN(insert\_dt) and MAX(update\_dt), giving visibility into the temporal scope of the data. The fields source\_system and source\_entity are explicitly defined as 'bl\_cl' and 'lkp\_departments' respectively, providing downstream users with metadata about the data’s origin—vital for audit and debugging.

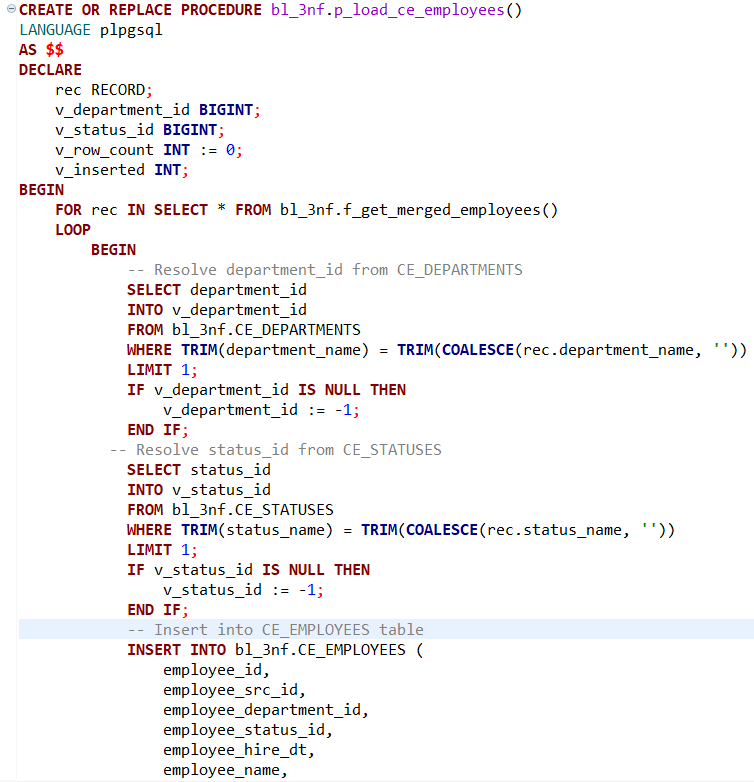
The loading logic is encapsulated in the stored procedure p\_load\_ce\_departments(). This procedure calls the function and inserts the resulting dataset into the target table CE\_DEPARTMENTS. The ON CONFLICT (department\_src\_id) DO NOTHING clause ensures the process is idempotent, preventing duplicates on repeated executions by using the source system's ID as a natural key.

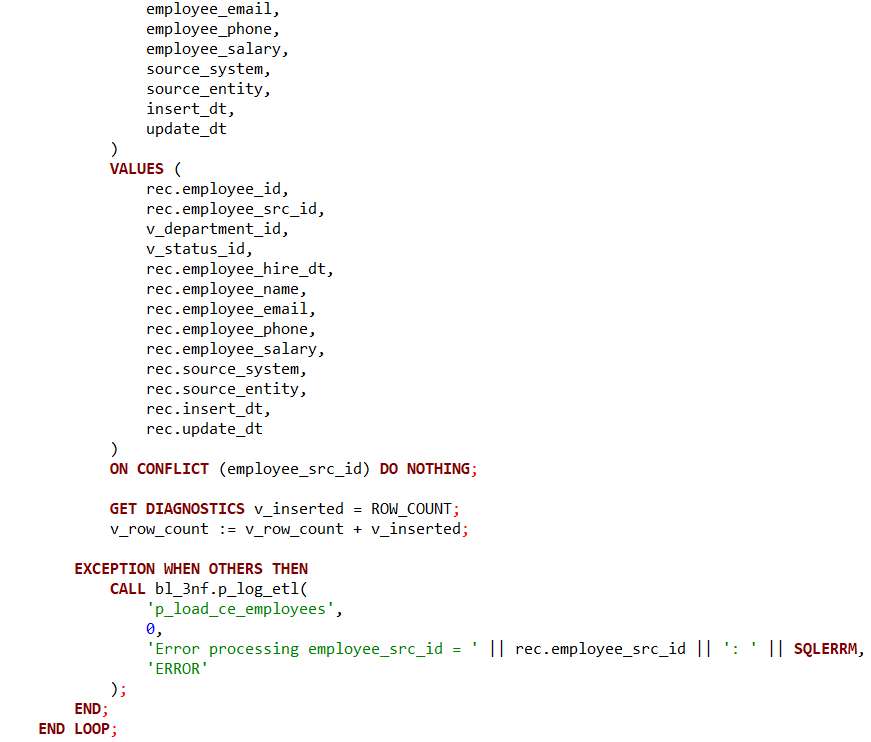
The procedure uses GET DIAGNOSTICS to determine how many rows were successfully inserted and logs this information through the p\_log\_etl procedure. This logging mechanism enhances operational monitoring and traceability by capturing both success metrics and error conditions.

Error handling is implemented through an exception block, which captures unexpected failures and logs them accordingly using SQLERRM. This practice improves robustness and enables proactive issue resolution.









### ETL Logic for Employees – Centralized Entity Load

The ETL (Extract, Transform, Load) logic for the **Employees** domain in the bl\_3nf schema is designed to construct a clean, de-duplicated, and referentially linked dataset for employee information. This dataset is ingested from the source table bl\_cl.lkp\_employees and loaded into the canonical table bl\_3nf.CE\_EMPLOYEES. This process ensures that employee data is normalized, joined with related dimensions, and stored in a central entity (CE) layer suitable for analytical and operational use cases.

### 1. Transformation Phase – f\_get\_merged\_employees()

At the heart of the transformation layer is the SQL function f\_get\_merged\_employees(). This function serves as a data refinement step, consolidating potentially messy or duplicated source entries into a single, authoritative record per employee.

#### **Key Features of the Function:**

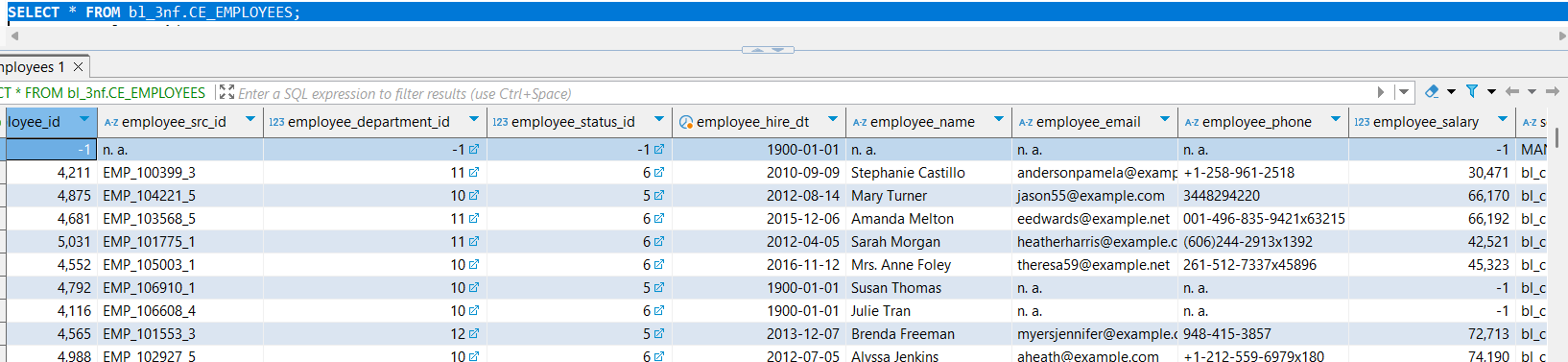
* **Coalesced Attribute Selection:** Several fields are populated using COALESCE(MAX(CASE...)) constructs. This ensures that fields such as employee\_name, employee\_email, and employee\_phone return only the most relevant, non-placeholder values. Common invalid entries such as 'n. a.' are explicitly excluded.
* **Data Aggregation Logic:** Numeric and date fields such as employee\_salary and employee\_hire\_dt are aggregated using MAX() to ensure the latest values are retained. Meanwhile, insert\_dt and update\_dt track the range of the data’s lifecycle using MIN() and MAX() respectively.
* **Dimensional Metadata Enrichment:** The source system (bl\_cl) and entity (lkp\_employees) are appended to every record for lineage and traceability, essential for audits and cross-system debugging.
* **Deduplication and Grouping:** The function groups records by employee\_id and employee\_src\_id, ensuring that every unique employee is processed only once—critical in environments where duplicate or partial records might exist.

### 2. Load Phase – p\_load\_ce\_employees()

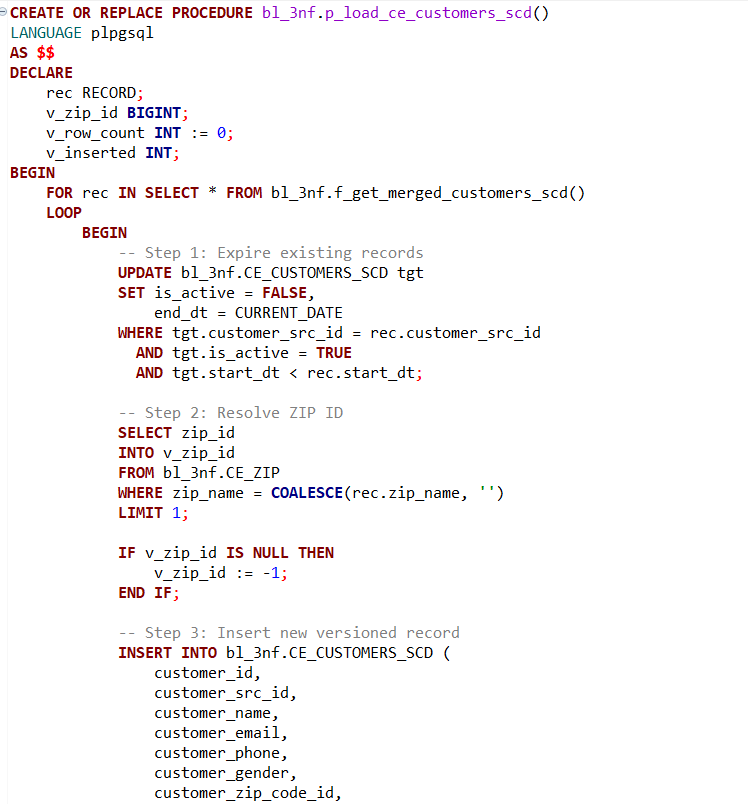
The loading of cleaned employee data is handled by the PL/pgSQL stored procedure p\_load\_ce\_employees(). This procedural logic is structured for both robustness and clarity, with several built-in safeguards.

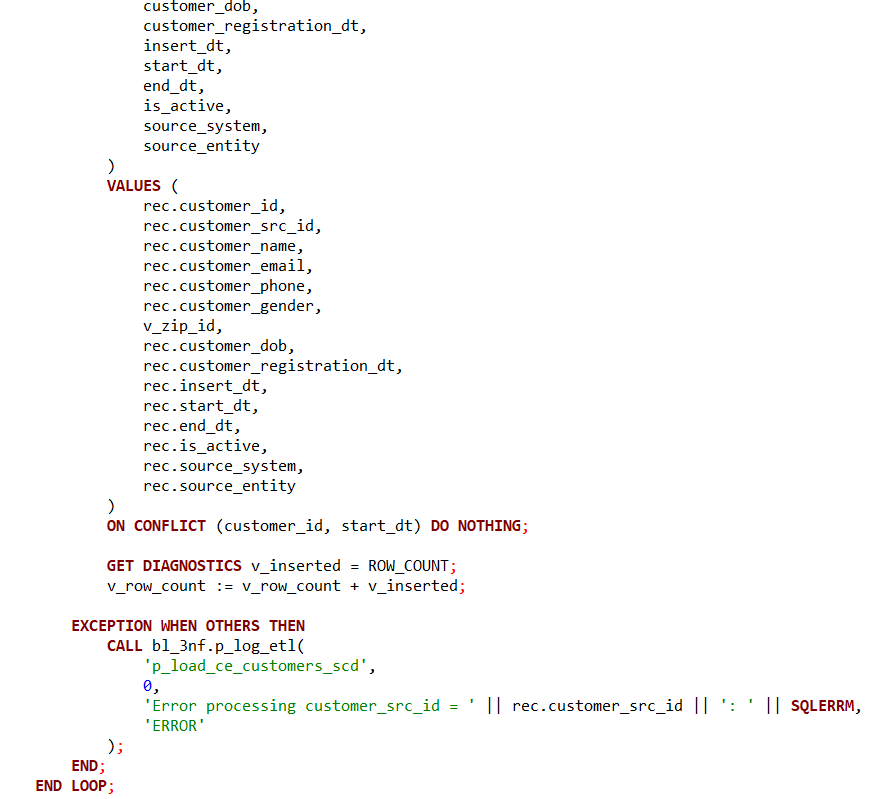
#### **Process Overview:**

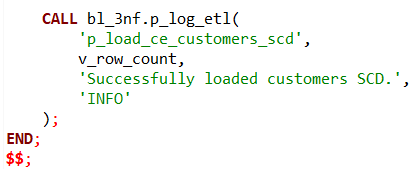
* **Looping Over Transformed Records:** The procedure iterates through the result set from f\_get\_merged\_employees(). Each record is processed independently, allowing for fine-grained exception handling and partial successes.
* **Foreign Key Resolution:**
  + The employee's department is resolved from the CE\_DEPARTMENTS table based on the trimmed department\_name.
  + Similarly, the employee’s status is resolved from CE\_STATUSES.
  + In both cases, if a match is not found, a placeholder value of -1 is assigned. This helps maintain referential integrity without halting the ETL process due to missing lookups.
* **Idempotent Insert Logic:** The final insert statement uses ON CONFLICT (employee\_src\_id) DO NOTHING. This ensures the process can be safely re-run without causing duplication or violating constraints, which is a hallmark of well-architected ETL systems.
* **Operational Logging and Error Handling:**
  + A diagnostic counter (v\_row\_count) tallies the number of successfully inserted records.
  + Any errors encountered during the loop are captured using EXCEPTION WHEN OTHERS, and a descriptive error is written to the ETL log via p\_log\_etl().
  + After the loop, a summary log entry is written to indicate the total number of successfully loaded records.











### ETL Logic for Customers SCD – Managing Historical Changes with Versioning

This ETL process handles the loading and historical tracking of customer data using a **Slowly Changing Dimension Type 2 (SCD Type 2)** strategy. Its purpose is to capture and preserve changes in customer attributes over time while maintaining older versions for accurate historical analysis and auditability. The design is tailored to manage **multi-source inputs**, ensuring that even if only one system reports a change, the updated data is merged appropriately while preserving stable data from other sources.

### 1. Transformation Function – f\_get\_merged\_customers\_scd()

This function performs the data consolidation, cleansing, and business rule enforcement prior to loading the SCD target.

#### **Key Responsibilities:**

* **Active Record Filtering:** Only records marked as is\_active = TRUE in the source table (bl\_cl.lkp\_customers) are selected, avoiding redundant or obsolete data.
* **Cleansing and Deduplication:** Fields such as customer\_name, customer\_email, and zip\_name are aggregated using MAX(CASE WHEN ...) logic to ignore placeholder values like 'n.a.'. The result is a merged "best version" of each customer across sources.
* **SCD Metadata Handling:**
  + insert\_dt is set to the current ETL run date.
  + end\_dt is initialized to '9999-12-31', signifying that the record is currently valid.
  + is\_active is set to TRUE, following the SCD Type 2 convention of marking the latest version as active.
* **Multi-Source Consideration:** While historical tracking was already implemented at the **source system level**, this function introduces an additional layer of **consolidation across sources**. It ensures that if one source sends a change while others remain the same, the resulting merged record reflects the update, **preserving stable data from the non-updating source**.

### 2. Load Procedure – p\_load\_ce\_customers\_scd()

This PL/pgSQL procedure performs the actual SCD logic — identifying changes, expiring old records, and inserting new versions — while resolving relationships and logging metadata.

#### **Step-by-Step Process:**

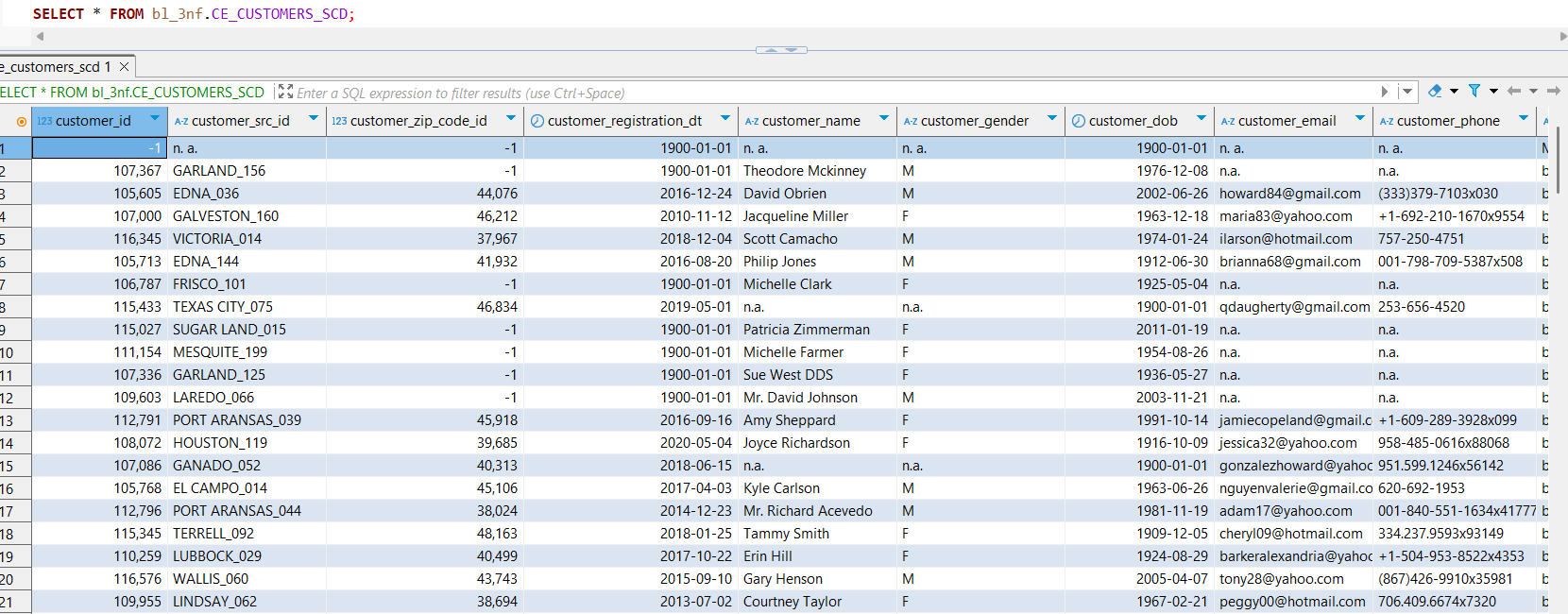
* **Step 1: Expire Old Versions:** For each incoming customer\_src\_id, the procedure finds existing **active** records in CE\_CUSTOMERS\_SCD with an earlier start\_dt and updates them:  
  + Sets is\_active = FALSE
  + Sets end\_dt = CURRENT\_DATE
* This aligns with SCD Type 2 behavior, ensuring no overlapping active periods for the same customer.
* **Step 2: Resolve Dimension Key (ZIP):** Customer ZIP codes are normalized via a lookup to CE\_ZIP, retrieving a surrogate key (zip\_id). A fallback value of -1 is used if no match is found, preserving referential integrity.
* **Step 3: Insert New Versioned Record:** A new customer version is inserted into the SCD table with the updated values. The unique key (customer\_id, start\_dt) ensures version uniqueness while supporting multiple historical entries for the same customer.
* **Step 4: Error Handling and Logging:** Robust exception handling wraps each iteration. Any processing error for a customer record triggers an error log entry using p\_log\_etl, without halting the loop. Upon completion, the total number of inserted records is logged as an ETL success event.

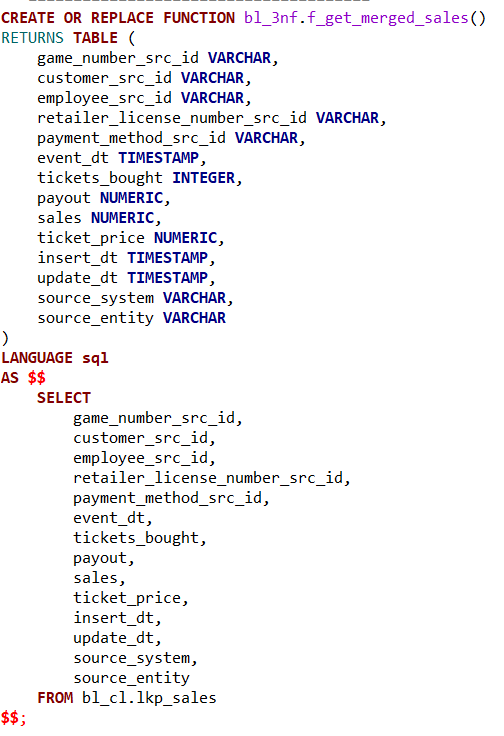
### Special Consideration: Cross-Source Updates

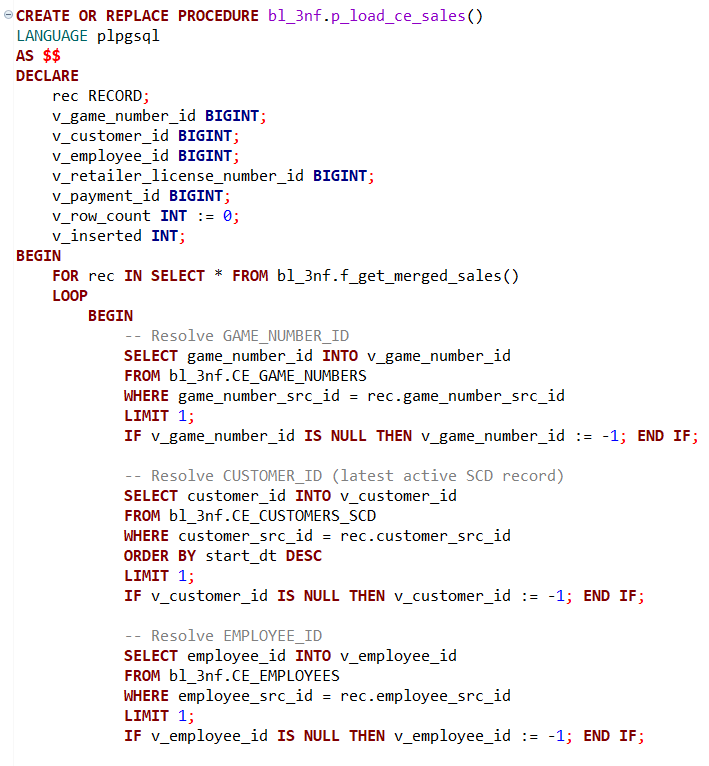
A notable enhancement in this logic is the ability to **merge updates from one source system while retaining attributes from another**, enabling partial updates. For example:

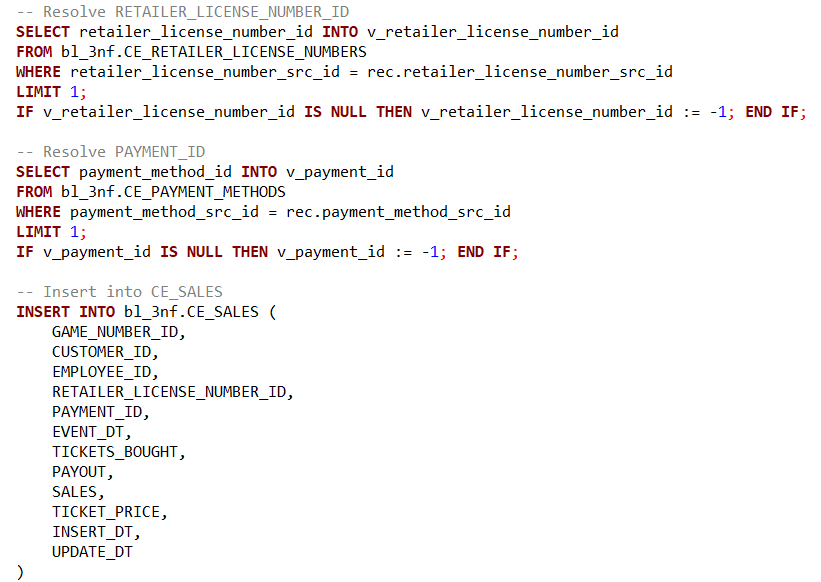
* Suppose **source A** updates the email while **source B** retains the same name and phone.
* The ETL logic identifies and merges these changes, updating only the modified attribute while retaining the rest — **creating a new active version** that combines the latest values across sources.

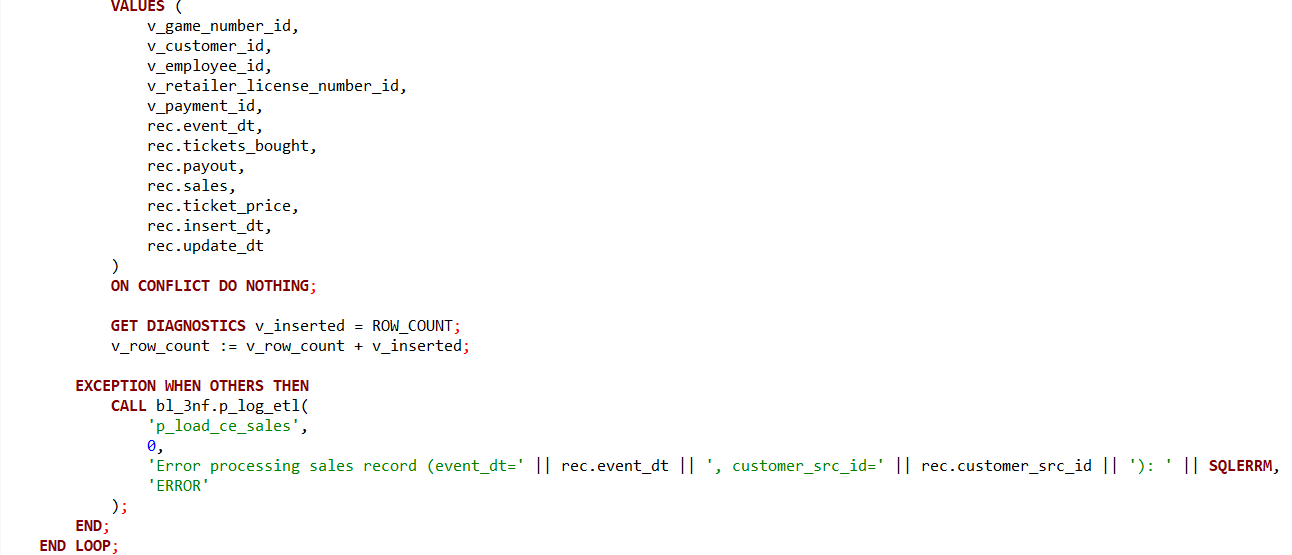
This pattern is essential in decentralized environments where different systems manage separate slices of the customer profile (e.g., CRM, billing, support). It ensures data completeness and integrity without requiring every source to resubmit unchanged fields.

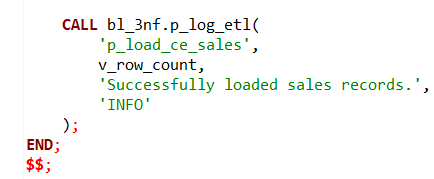












### ETL Logic for Sales – Transactional Fact Loading with Dimension Resolution

The **Sales** ETL process is designed to populate a **fact table** that captures transactional events related to ticket sales. This logic emphasizes **referential integrity**, **dimension key resolution**, and **error-resilient loading**. While this is not a Slowly Changing Dimension (SCD), it plays a critical role in the data warehouse by tying together multiple dimension tables and enabling analytics on customer behavior, game activity, and payment trends.

### 1. Transformation Function – f\_get\_merged\_sales()

This SQL function performs the first stage of the ETL pipeline by sourcing and structuring the raw sales data from the lookup table bl\_cl.lkp\_sales.

#### **Highlights:**

* **Direct Pass-Through:** The function does not perform transformations but returns sales records as-is from the source view. This implies that upstream processes have already cleaned, deduplicated, and possibly merged records prior to loading.
* **Comprehensive Structure:** It includes all required foreign keys in their raw form (e.g., customer\_src\_id, employee\_src\_id), alongside core transaction metrics like:  
  + tickets\_bought
  + sales
  + payout
  + ticket\_price
  + event\_dt (when the sale occurred)
* **Provenance Columns:** Fields like source\_system and source\_entity are retained for **lineage tracking** and troubleshooting.

### 2. Load Procedure – p\_load\_ce\_sales()

This stored procedure executes the core ETL logic — resolving dimension keys, handling missing references, and loading the data into the CE\_SALES fact table.

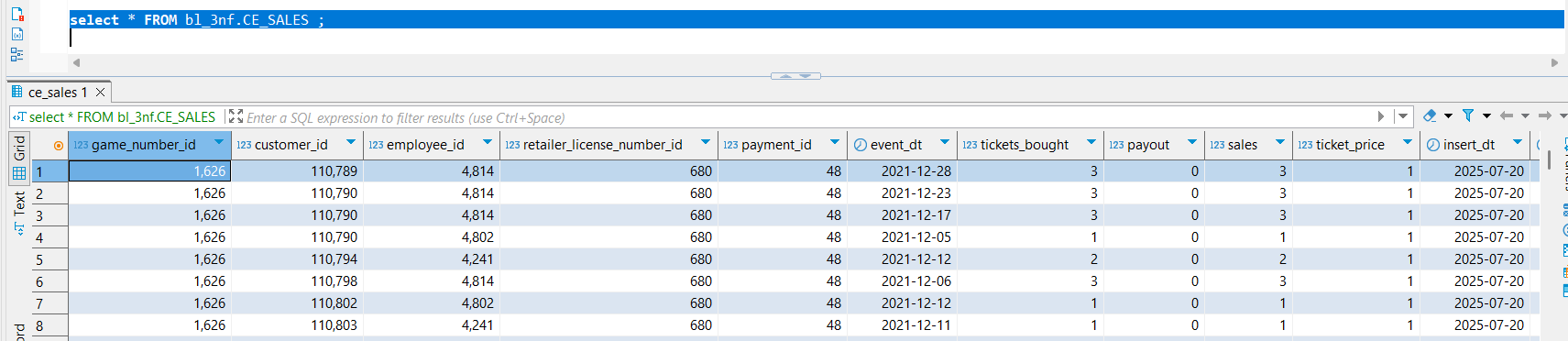
#### **Step-by-Step Mechanics:**

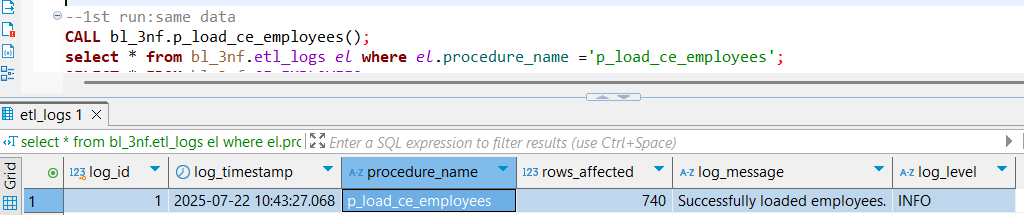
* **Step 1: Resolve Surrogate Keys** For every incoming sales record, the procedure attempts to map each **source ID** to its corresponding **surrogate key** (primary key) in the dimension tables:  
  + game\_number\_src\_id → CE\_GAME\_NUMBERS
  + customer\_src\_id → CE\_CUSTOMERS\_SCD (only the **latest active** customer version is considered)
  + employee\_src\_id → CE\_EMPLOYEES
  + retailer\_license\_number\_src\_id → CE\_RETAILER\_LICENSE\_NUMBERS
  + payment\_method\_src\_id → CE\_PAYMENT\_METHODS
* This ensures that fact records maintain strict referential integrity to the dimensional model.
* **Step 2: Default for Unmatched Keys** If a dimension lookup fails, a fallback ID of -1 is assigned. This is a common best practice that:  
  + Flags the record as unmatched
  + Allows the load to proceed
  + Enables post-load analysis or reprocessing without data loss
* **Step 3: Insert into Fact Table** The resolved keys, along with transaction details, are inserted into the bl\_3nf.CE\_SALES table.  
  + ON CONFLICT DO NOTHING ensures **idempotency**, preventing duplicate insertions (e.g., during reruns).
* **Step 4: Logging and Error Handling**
  + A structured TRY...CATCH block wraps each record load. This isolates errors to a single record, logs the issue using p\_log\_etl, and allows the job to continue.
  + After the loop, the total successfully inserted rows are logged as an informational event.

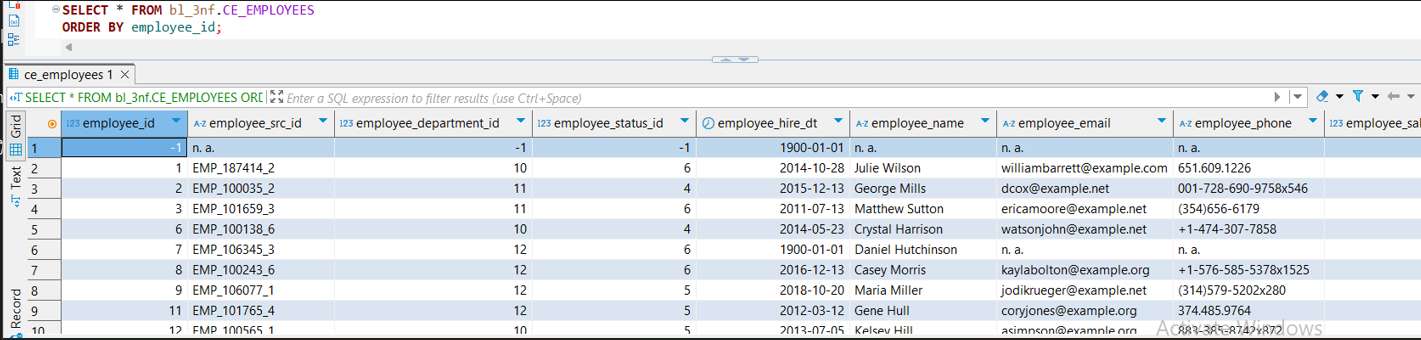
### Integration with the Customer SCD

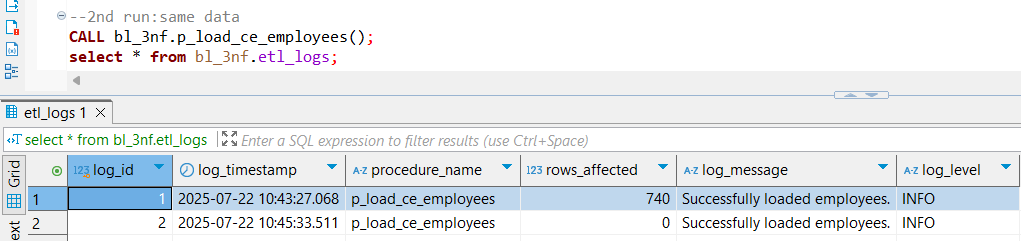
A key part of this sales ETL logic is its **tight coupling to the CE\_CUSTOMERS\_SCD dimension**:

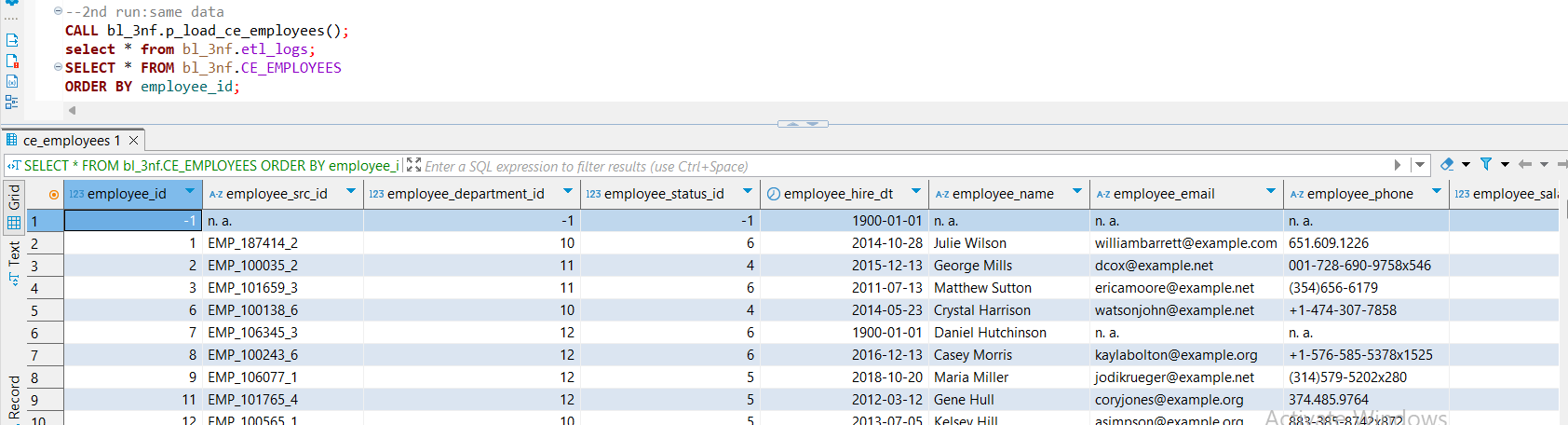
* It queries only the **latest active customer version** based on customer\_src\_id, ensuring the sale is linked to the correct customer profile at the time of the transaction.
* This highlights a classic **type-2 dimension lookup pattern** — anchoring fact data to time-accurate dimension values, even as the dimension evolves.











As you can see, the procedure call completes successfully in both cases. However, in the second run, no rows are affected because we are attempting to insert the same records again.

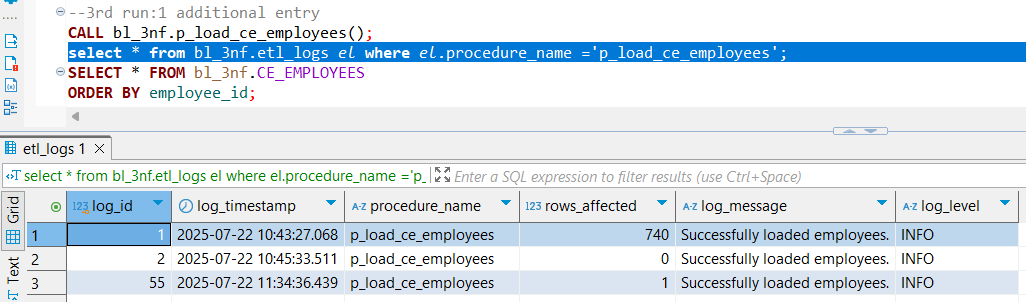
Next, I will add one additional row to the source data, re-run all cleansing procedures, and then execute the upsert procedure again. Logically, we expect that only one row will be affected, since we’ve added just a single new transaction within the same daily batch. Let’s verify that.

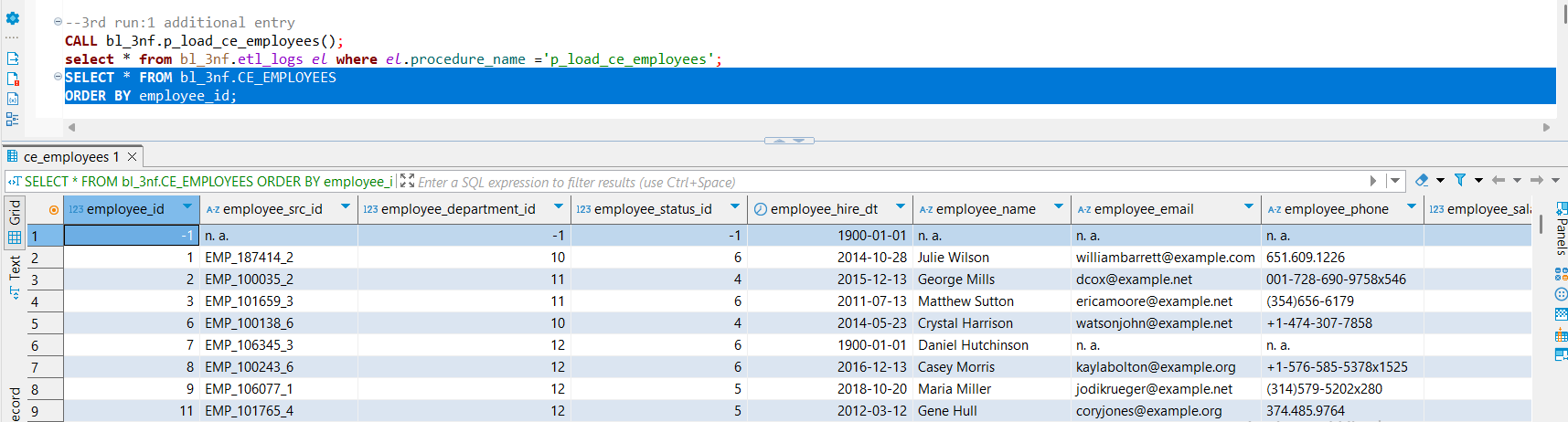
It's also important to remember that in a real-world scenario, we typically process only one batch per day, assuming that each entity is updated at most once per day. Therefore, after processing each batch, the working and staging tables (e.g., wrk\_ or intermediate cleansing tables) should be truncated.

This ensures that any updates in the next daily batch — even for previously processed entities — can be correctly detected and applied, especially in the lookup on bl\_cl or core entity tables on 3nf









Our expectations were confirmed: when reprocessing the same batch with one additional transaction in the staging area, the cleansing procedures updated the bl\_cl lookup as expected. Then, during the 3NF loading step, exactly one row was affected — as verified through our centralized ETL log. This confirms that the pipeline correctly handles incremental changes from staging through to the 3NF layer.