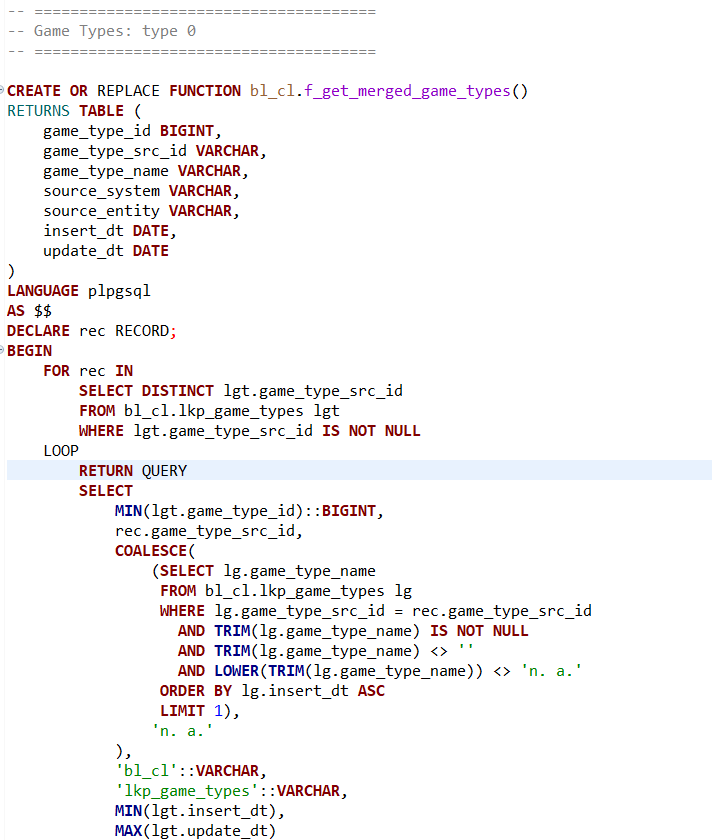
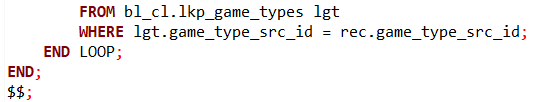
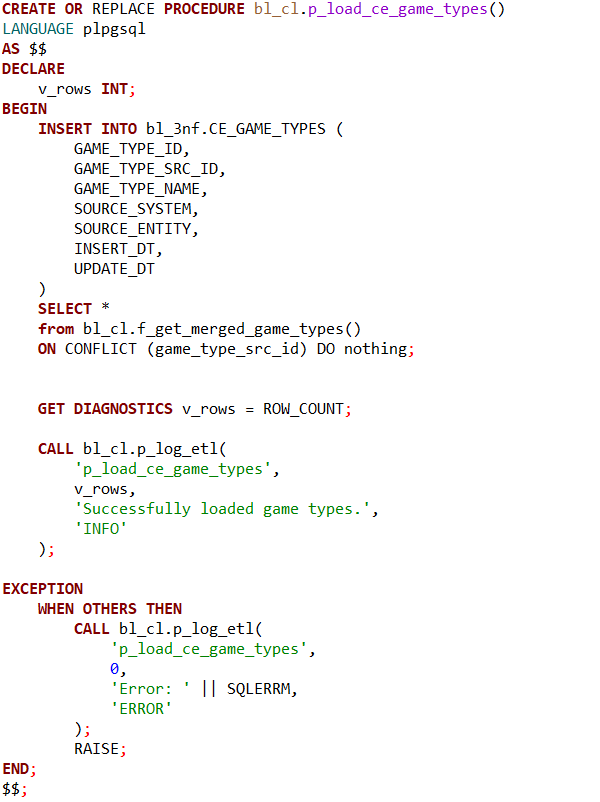


| Business Template  **Loading 3NF objects** |
| --- |
|  |



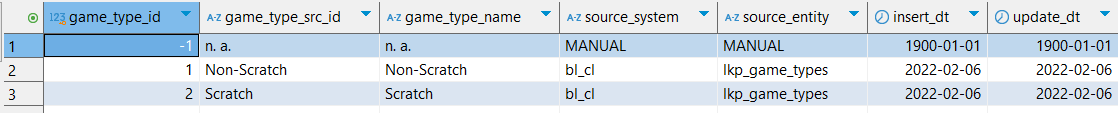




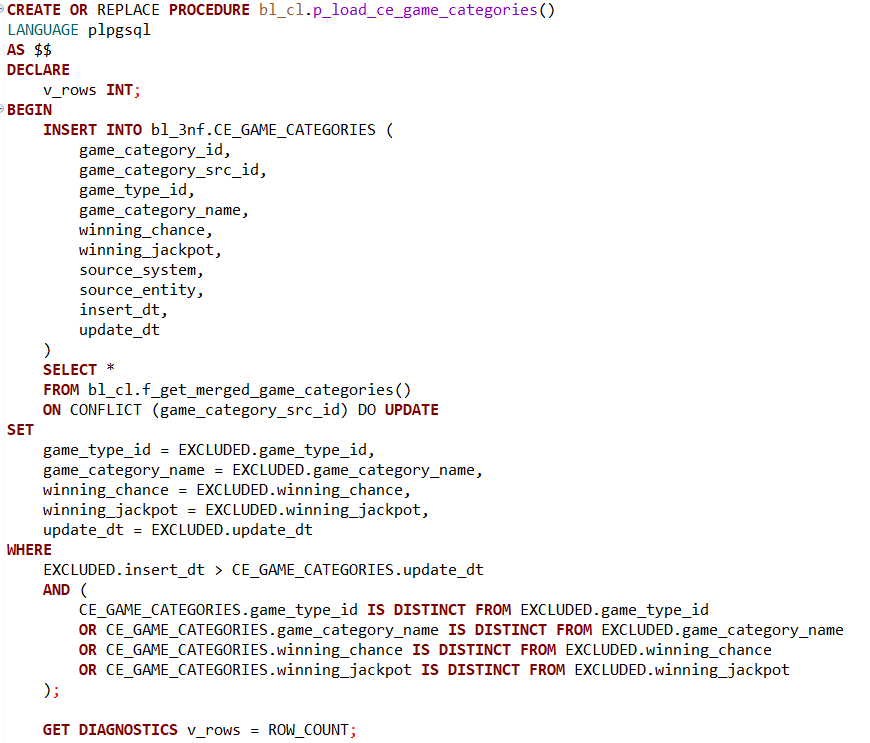
The logic behind this code is designed to consolidate game type data from a lookup table into a clean, deduplicated form suitable for downstream use. The key aspect to understand is that the source lookup table, bl\_cl.lkp\_game\_types, follows Slowly Changing Dimension (SCD) Type 0 behavior. In this model, once a record is inserted, its attributes remain static and are never updated. Any changes in game type attributes result in the creation of a new entry with a new game\_type\_src\_id, rather than modifying existing records.

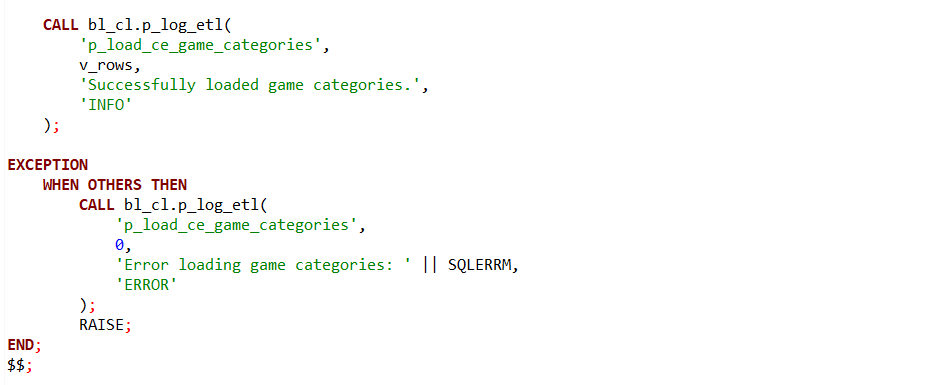
As a result, the source table maintains a complete and unchanging set of entries, where each game\_type\_src\_id represents a fixed snapshot of a game type at the time of insertion. The function f\_get\_merged\_game\_types() operates under this assumption and focuses on extracting and consolidating this data efficiently. For each distinct game\_type\_src\_id, it selects the minimum game\_type\_id as a canonical reference and determines the most meaningful game type name by excluding null, blank, or placeholder values like "n.a." It also captures the earliest insertion date to preserve the original temporal context.

Since the source adheres strictly to SCD Type 0 logic, the downstream process does not need to handle overwrites or version tracking. Instead, it simply inserts the consolidated and validated records into the target table, CE\_GAME\_TYPES. This insertion process ensures no duplication occurs by relying on the uniqueness of game\_type\_src\_id. Additionally, the procedure includes robust logging to monitor successful operations and capture any failures, ensuring transparency and traceability throughout the data pipeline.









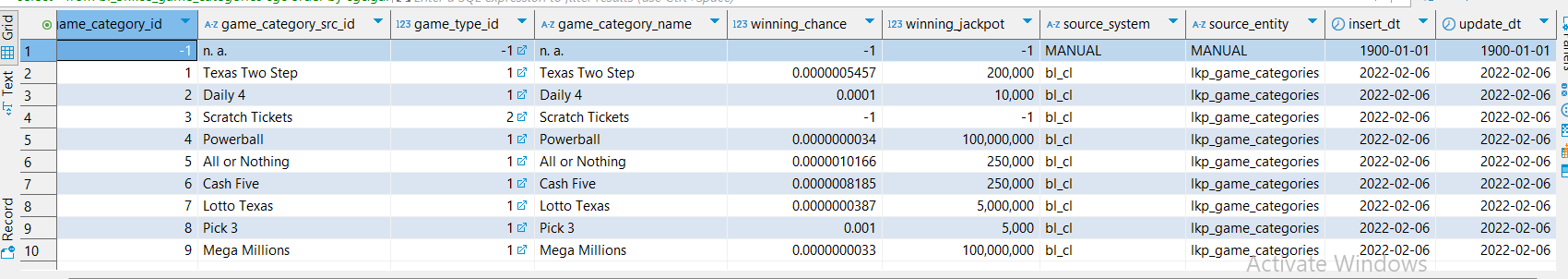
This logic focuses on consolidating and loading game category data from the source lookup table into the target table for further analysis or reporting. Alike the previous game types function, this logic uses 2 procedures to first aggregate data and then load it efficiently.

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.

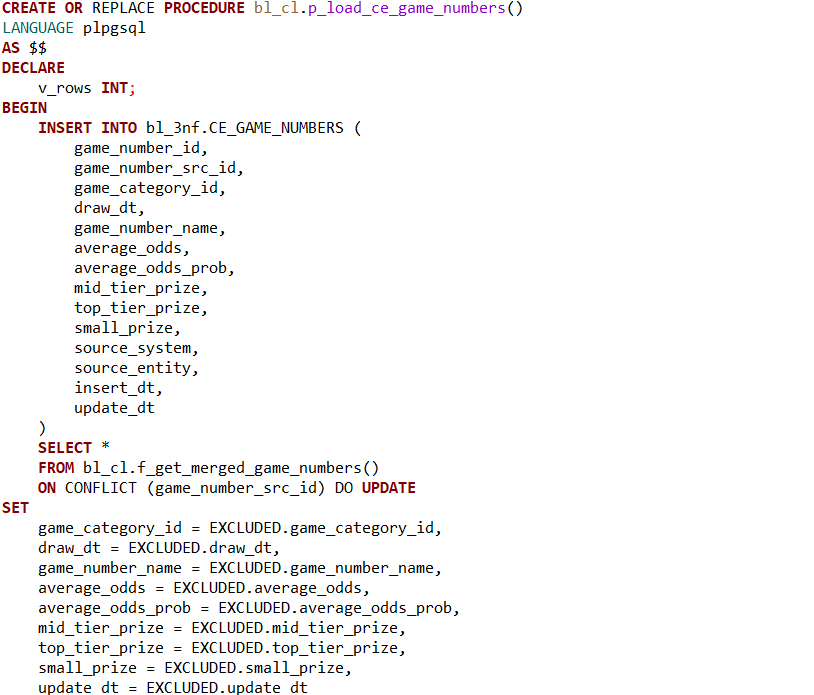
The aggregation step plays a crucial role in deduplication and quality control. For each unique game\_category\_src\_id and corresponding game\_type\_id, the function selects the maximum available game\_category\_id and filters out invalid or placeholder values for game\_category\_name (ignoring 'n.a.'), 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 .

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.









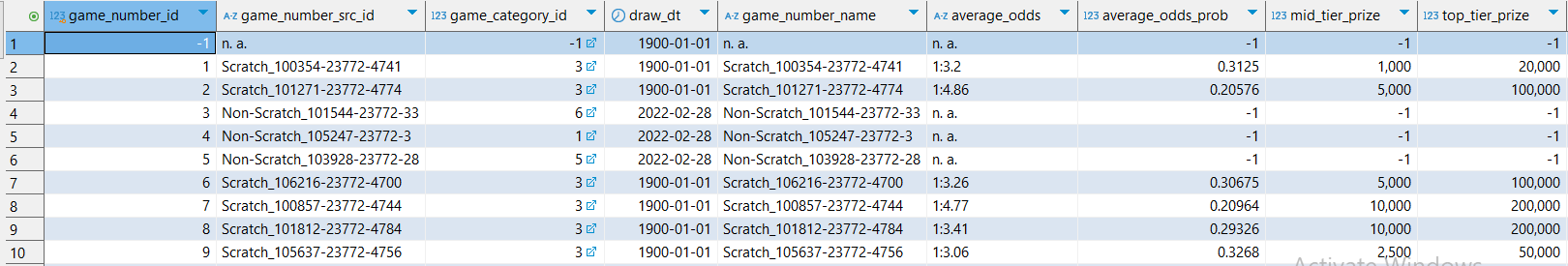
This segment of the ETL process focuses on consolidating and loading game number data from the source lookup table into a clean, normalized target table. The source table, bl\_cl.lkp\_game\_numbers, contains detailed attributes related to game numbers, including identifiers, draw dates, prize amounts,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

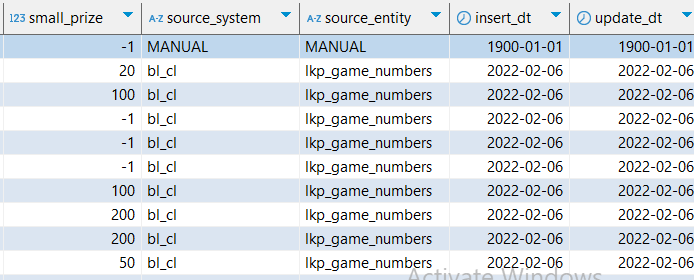
The function f\_get\_merged\_game\_numbers() plays a key role in merging and cleaning this data. It joins the game numbers with the previously consolidated game categories from bl\_3nf.CE\_GAME\_CATEGORIES by matching on the game category name. This enriches the game number records with the corresponding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

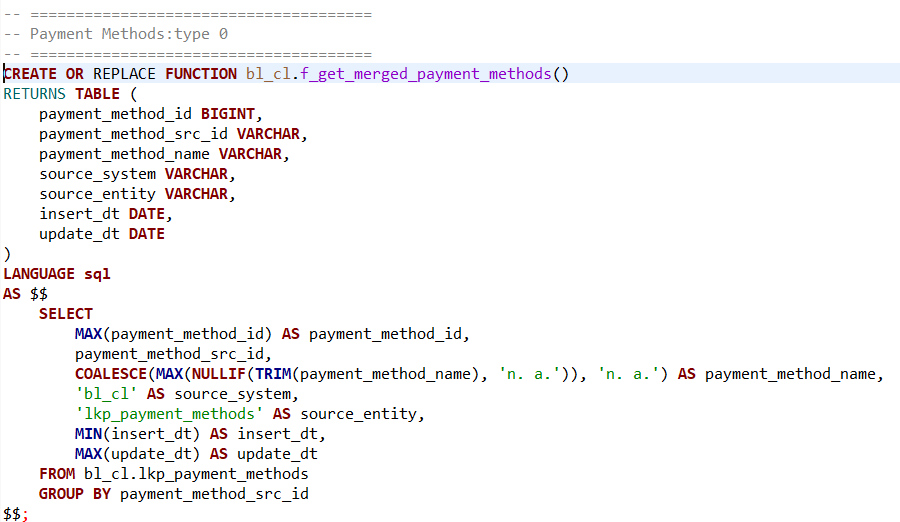
To handle data quality and duplicates, the function aggregates records by game\_number\_src\_id and game\_category\_id, selecting the maximum game\_number\_id as a canonical identifier. It filters out invalid or placeholder values—such as the default date 1900-01-01, the string 'n.a.' for names and odds, 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.

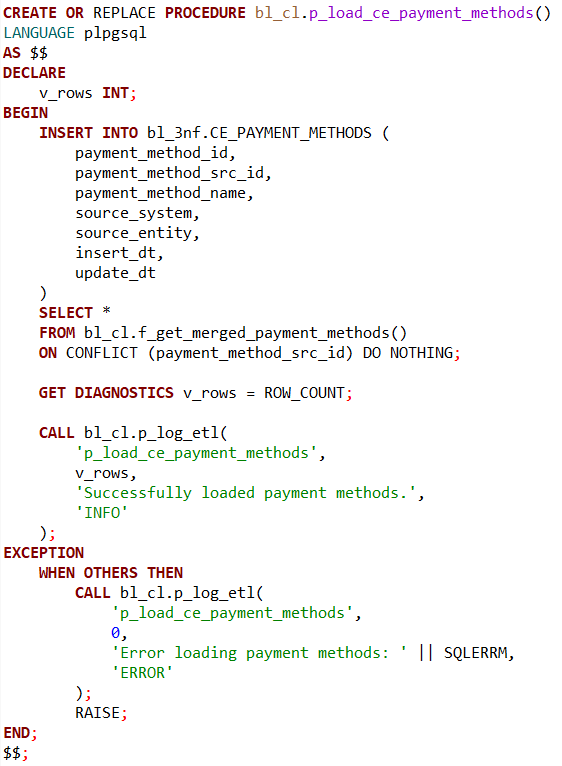
This logic assumes the upstream source data is either maintained with SCD Type 1 behavior or is otherwise updated so that only the most current snapshot of data exists. Therefore, the downstream loading procedure, p\_load\_ce\_game\_numbers(), can simply insert these merged records into the target table 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.









### 

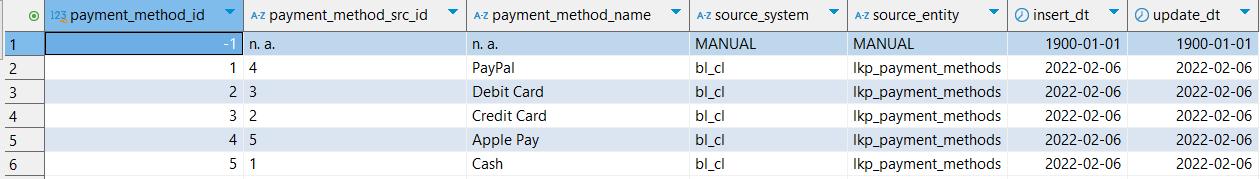
This part of the ETL pipeline is designed to extract, clean, and consolidate payment method data from the source lookup table into a standardized target table. The source data resides in bl\_cl.lkp\_payment\_methods, which contains records uniquely identified by payment\_method\_src\_id, along with descriptive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

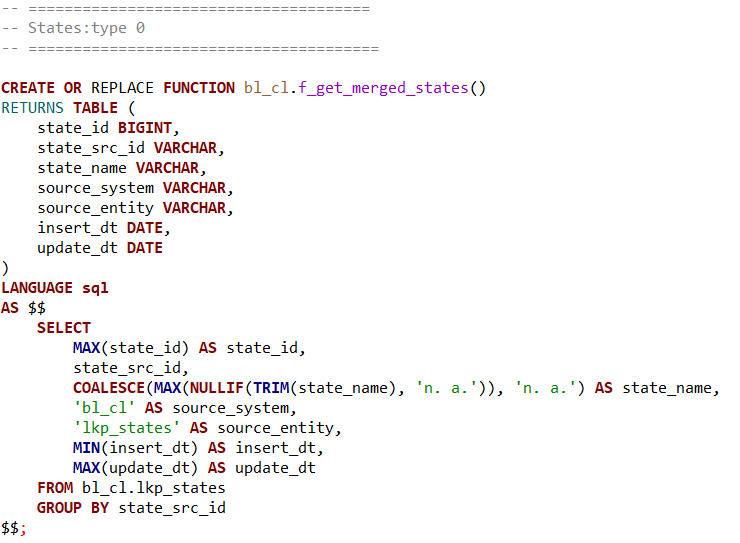
The function f\_get\_merged\_payment\_methods() is responsible for merging this data in a way that respects the Slowly Changing Dimension Type 0 (SCD Type 0) behavior of the source. Under Type 0 logic, once a payment method record is created, its attributes are considered static and are never modified. If any attribute (e.g., name) changes, a new entry with a new payment\_method\_src\_id is added to the source table — ensuring that historical records remain unchanged.

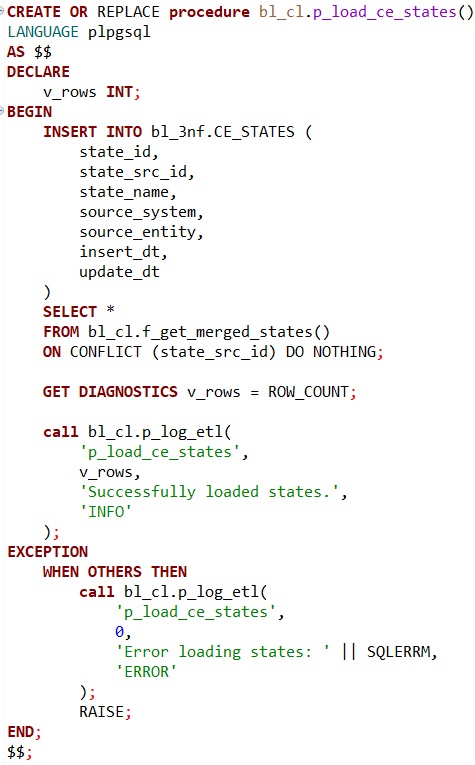
Working with this static data model, the function processes each distinct payment\_method\_src\_id, selecting the highest payment\_method\_id as a representative identifier. It also applies a data quality filter to ensure the payment method name is meaningful, removing values that are blank or placeholder entries like 'n.a.'. If no valid name is found, the function defaults safely to 'n.a.'. The earliest insert date is retained to preserve the original context of each record, but since records are never updated, the update timestamp is not considered significant.

Because the source data is guaranteed to be stable and version-free, the downstream procedure p\_load\_ce\_payment\_methods() simply inserts the cleaned, deduplicated data into the target table bl\_3nf.CE\_PAYMENT\_METHODS. The process uses the ON CONFLICT (payment\_method\_src\_id) DO NOTHING clause to avoid inserting duplicates, relying on the uniqueness of payment\_method\_src\_id to ensure consistency.

Additionally, the procedure incorporates diagnostics and logging. After attempting the insert, it logs the number of rows affected into a centralized ETL log table. If an error occurs, it captures the error message, logs the failure for traceability, and rethrows the exception to support robust 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.

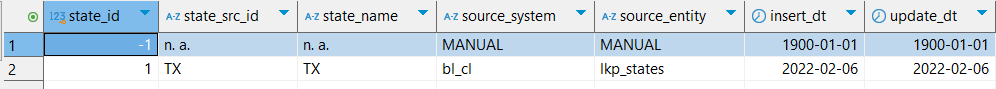
The function f\_get\_merged\_states() retrieves data from bl\_cl.lkp\_states, which contains raw entries for various states, each uniquely identified by a state\_src\_id, along with descriptive metadata such as state\_name, insert\_dt, and update\_dt.

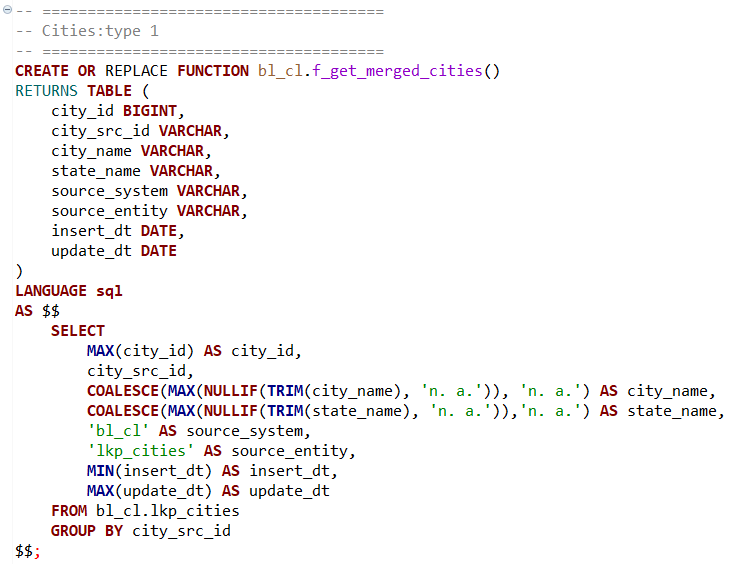
The key assumption here is that the source table follows **Slowly Changing Dimension Type 0 (SCD Type 0)** principles. Under this model, once a state record is inserted, its attributes remain fixed — they are never updated. If any changes occur to a state's attributes (e.g., name correction), a new entry with a new state\_src\_id is created, while the original record remains untouched. This ensures that historical data remains preserved in its original form.

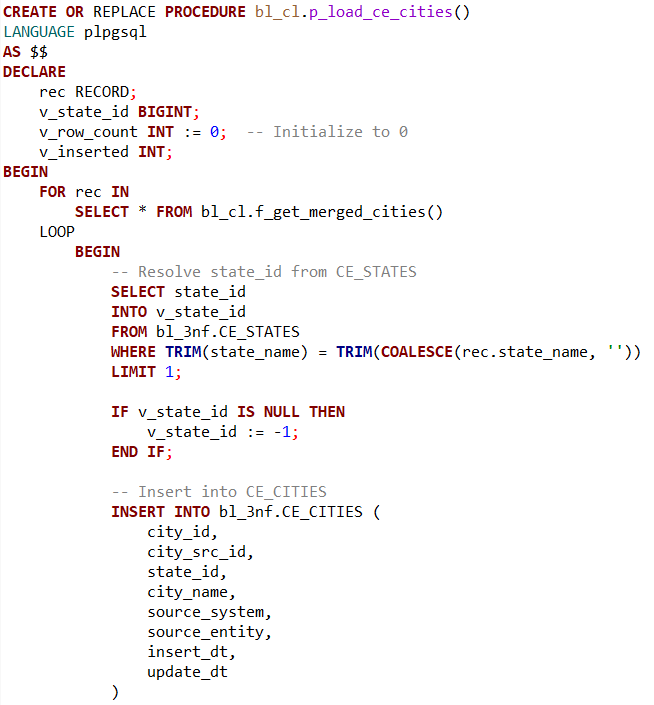
Working with this static dataset, the function processes each unique state\_src\_id, selecting the maximum state\_id as the representative key. It applies data quality checks to the state\_name, filtering out invalid or placeholder values such as 'n.a.' or blanks using the NULLIF(TRIM(...), 'n.a.') construct. If no valid name is available, it defaults to 'n.a.'. Since updates are not expected under SCD Type 0, the function primarily uses the insert\_dt as the key temporal indicator, while update\_dt is retained only for completeness.

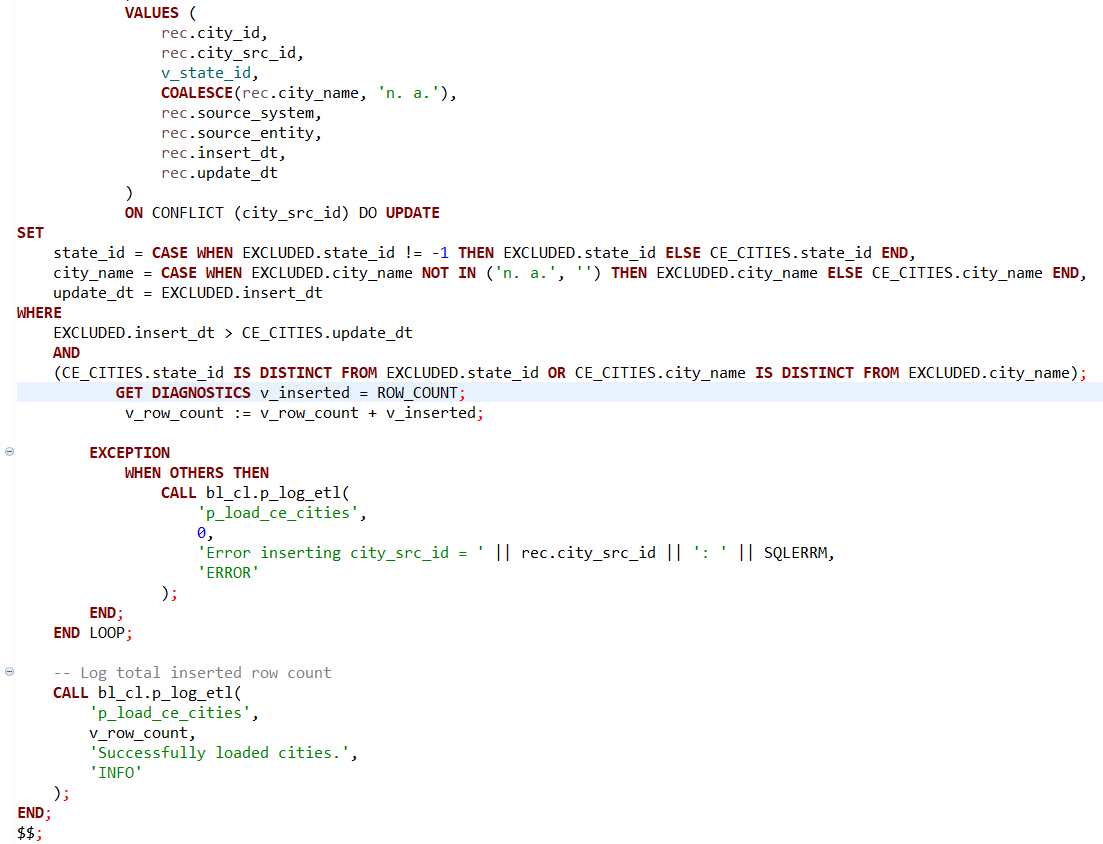
Because the source table is immutable, the downstream procedure p\_load\_ce\_states() does not need to detect or manage changes. It simply inserts the clean, validated data into the target table bl\_3nf.CE\_STATES. The use of the ON CONFLICT (state\_src\_id) DO NOTHING clause ensures that duplicate entries are ignored, relying on the assumption that each state\_src\_id represents a unique, static record.

To support traceability and operational monitoring, the procedure includes diagnostic logging. After the insert operation, the number of rows affected is captured using GET DIAGNOSTICS, and an ETL log record is written via the centralized p\_log\_etl procedure. If any error occurs during the process, the error is logged and rethrown to ensure clear visibility into failures and maintain ETL robustness.









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.

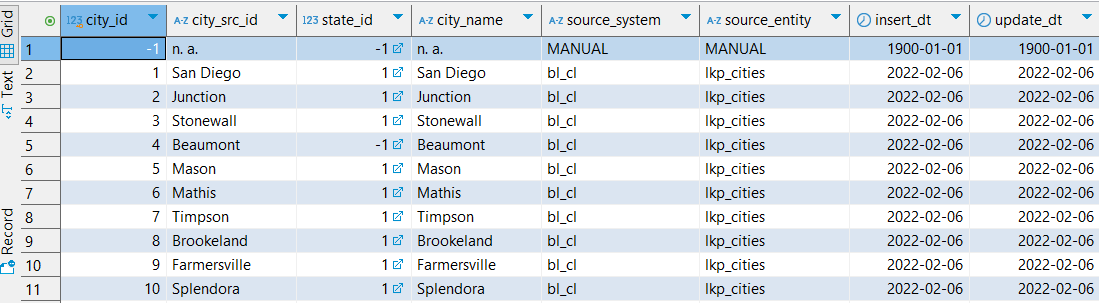
The function f\_get\_merged\_cities() performs a deduplication and cleanup process over the raw lkp\_cities data. For each unique city\_src\_id, it selects the maximum city\_id as the representative identifier. Additionally, it uses COALESCE(MAX(NULLIF(...))) logic to derive the best available values 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.

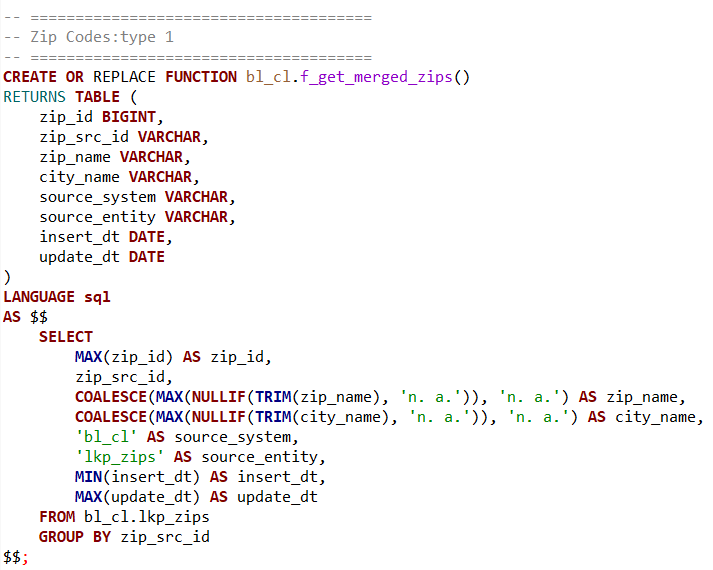
The transformation is based on the assumption that the lookup data in the source system adheres to **Slowly Changing Dimension Type 1 (SCD Type 1)** principles. This means any updates to the descriptive fields of a city (such as name or state name) overwrite previous values in-place, which eliminates 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.

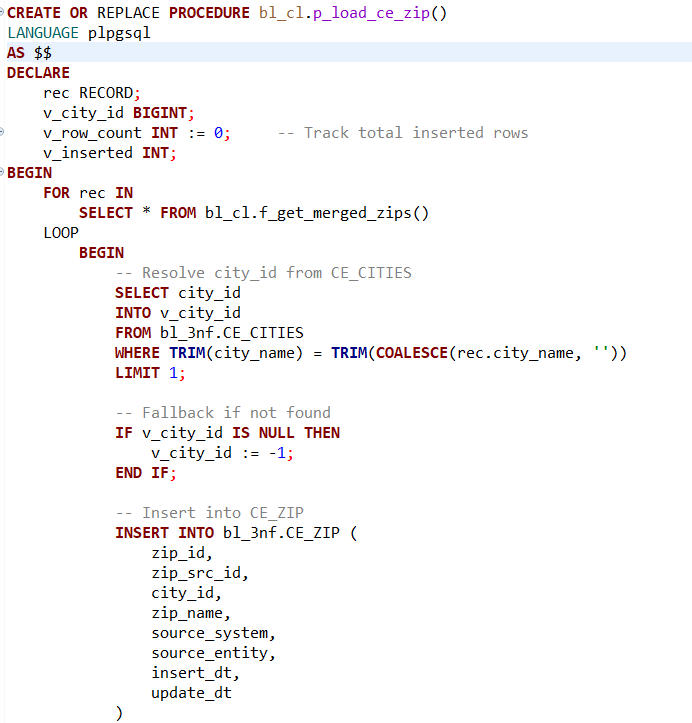
The procedure p\_load\_ce\_cities() loads the transformed data into the bl\_3nf.CE\_CITIES table. It processes each row individually to handle a special requirement: resolving the correct state\_id from the CE\_STATES table. This mapping is essential for establishing a foreign key relationship between 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.

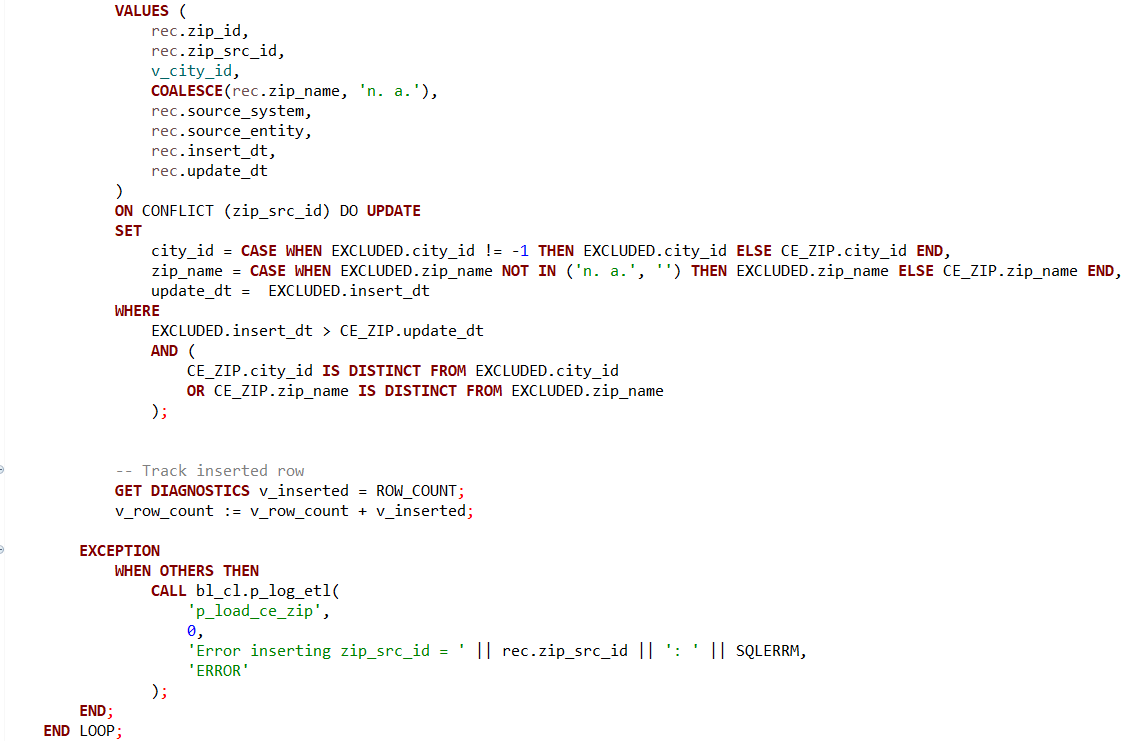
Each insertion is wrapped in a BEGIN ... EXCEPTION ... END block to gracefully handle any potential issues on a per-row basis. If an error occurs while inserting a particular city, it is logged using p\_log\_etl, capturing the failed city\_src\_id and the error message. This ensures that the overall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

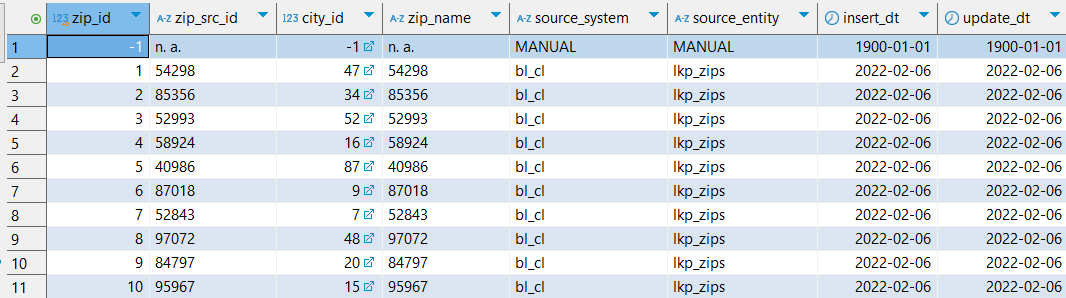
The function f\_get\_merged\_zips() performs the initial data consolidation. It extracts ZIP code records from the raw source table bl\_cl.lkp\_zips and groups them by their zip\_src\_id. For each group, it selects the maximum zip\_id as the primary identifier and uses conditional aggregation (COALESCE(MAX(NULLIF(...)))) 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.

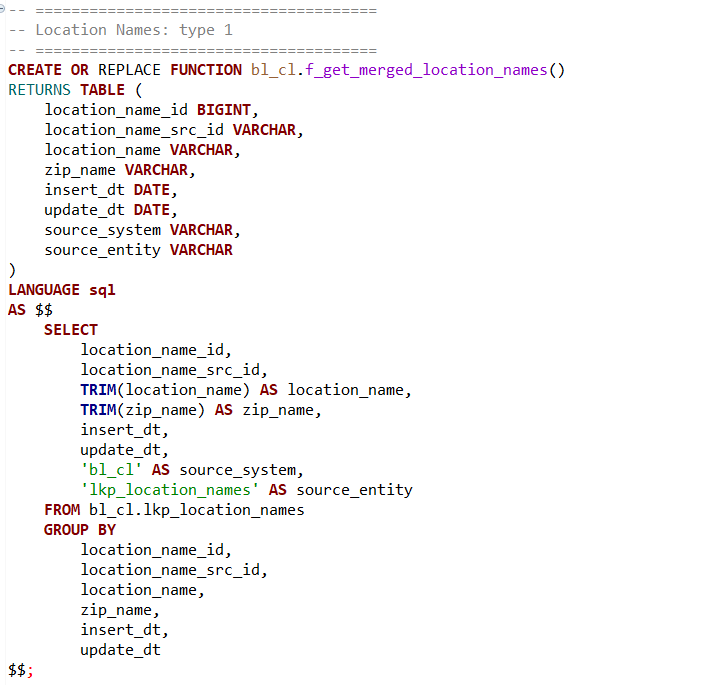
Since the staging layer is implemented with **Slowly Changing Dimension Type 1 (SCD Type 1)** semantics, it always reflects the most recent and accurate values for each source record. This simplifies the downstream loading process, as no history tracking or versioning logic is required—data can simply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

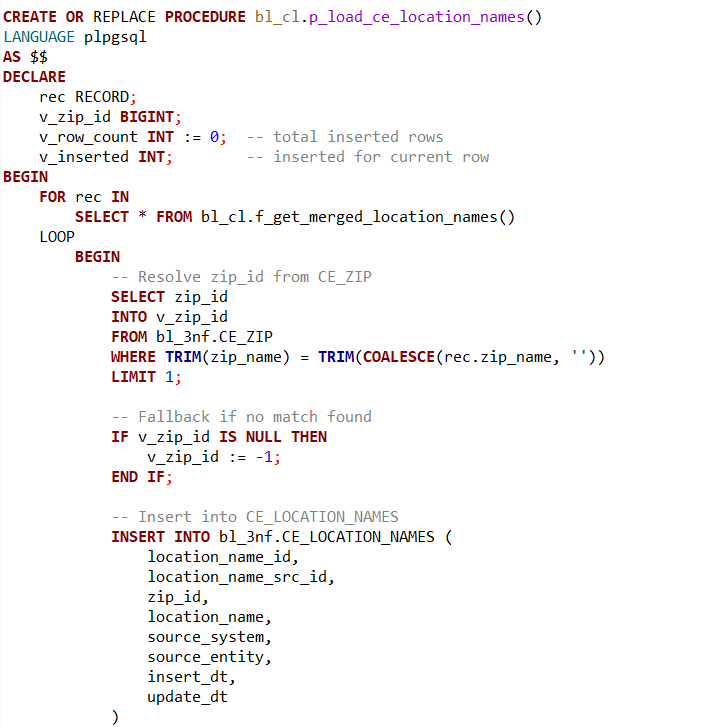
The procedure p\_load\_ce\_zip() iterates over the cleaned ZIP records returned by the function and handles the logic necessary to load them into the CE\_ZIP dimension table. A critical part of this process is resolving the city\_id from the CE\_CITIES table based on the city\_name. This lookup ensures 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.

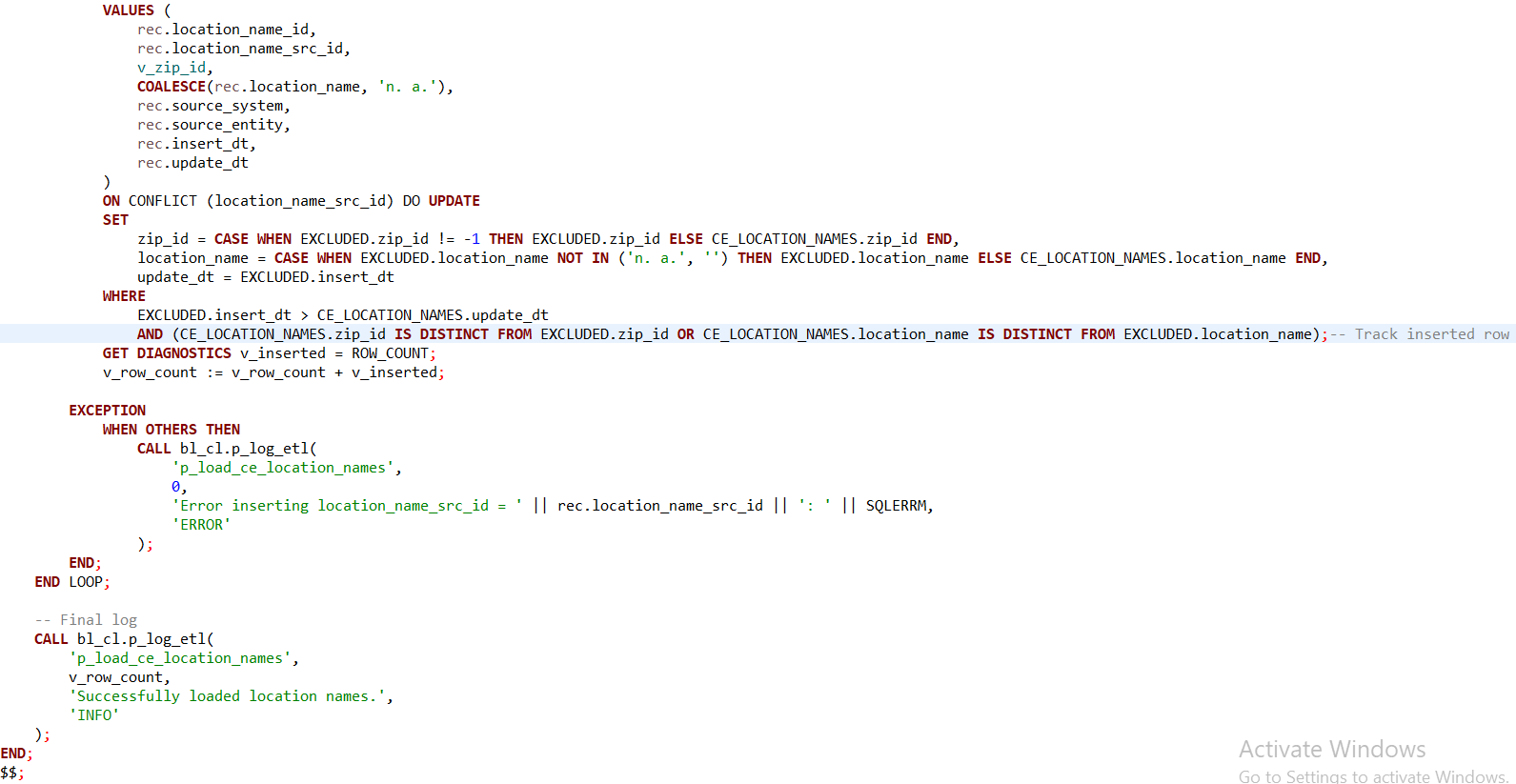
The procedure inserts the ZIP records one by one using a FOR loop and tracks the number of successful insertions. It employs a BEGIN...EXCEPTION...END block within the loop to ensure that errors with individual records do not halt the entire load process. Any insertion failures are logged through p\_log\_etl,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









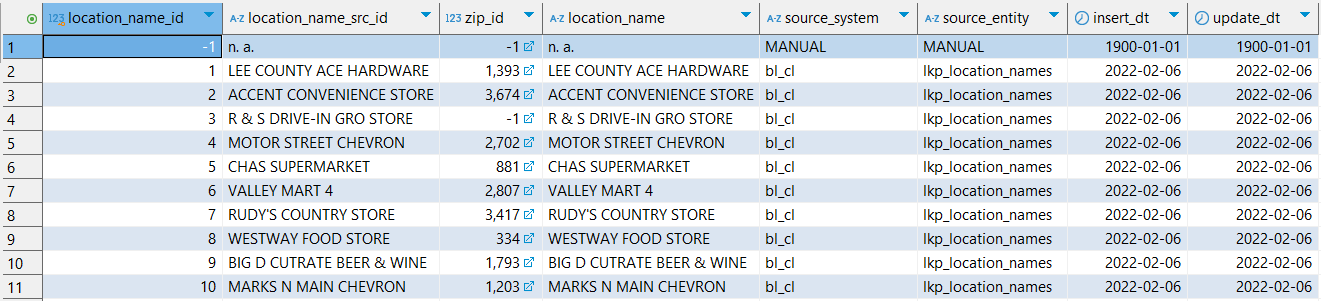
This portion of the ETL framework is responsible for transforming and loading **location name** reference data into the third normal form (3NF) model. The logic ensures that location names are clean, unique, and relationally connected to ZIP codes, creating a well-structured geographic hierarchy within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

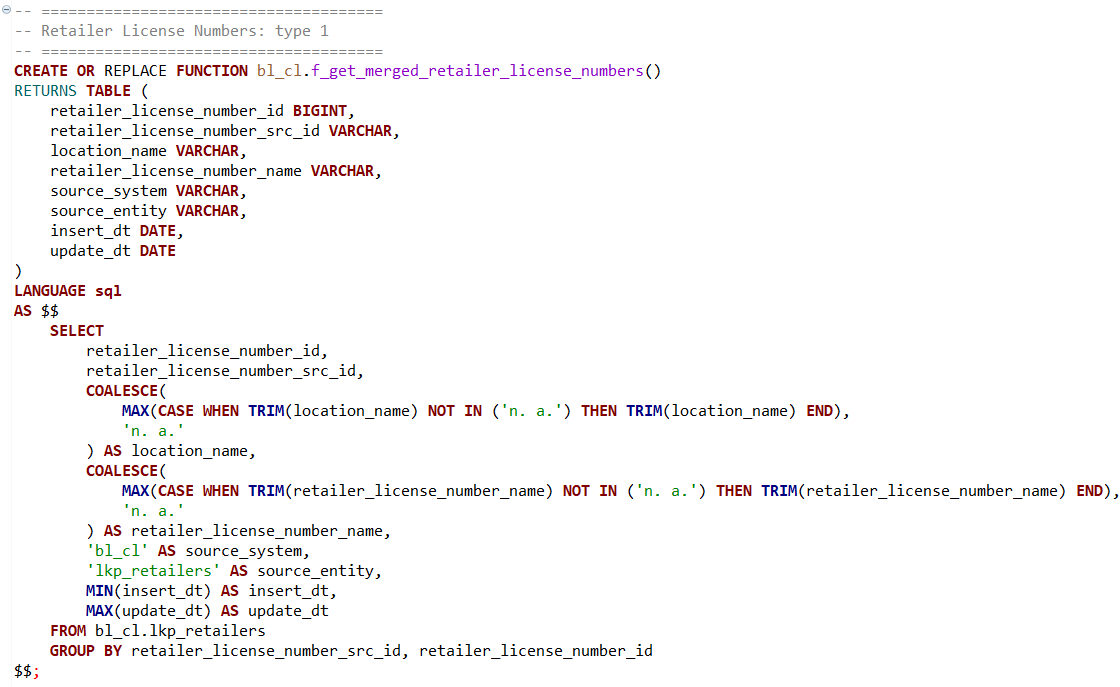
The process begins with the SQL function f\_get\_merged\_location\_names(), which retrieves raw location data from bl\_cl.lkp\_location\_names. The function performs basic data cleansing—trimming whitespace from location\_name and zip\_name—and assigns metadata such as the source system (bl\_cl) and 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.

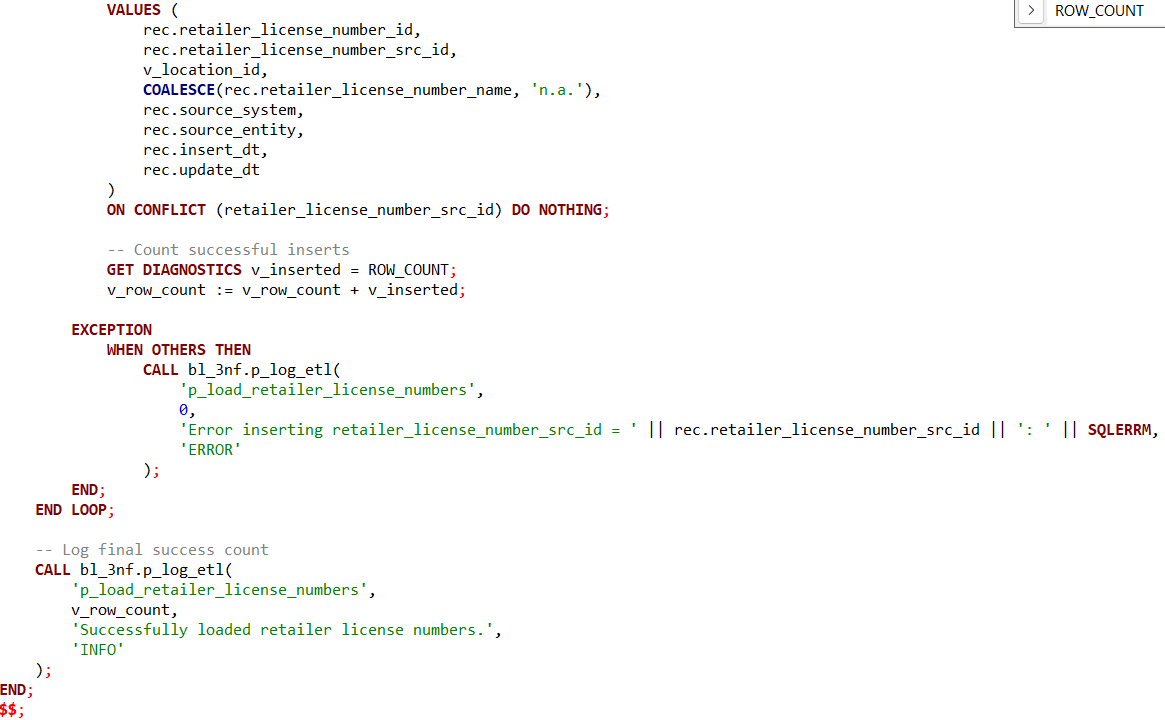
The procedure also includes robust error handling. Any errors that occur during the insertion of a specific record are caught and logged via the p\_log\_etl procedure. This ensures that faulty records do not interrupt the overall ETL process and that all errors are traceable by source identifier (location\_name\_src\_id)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

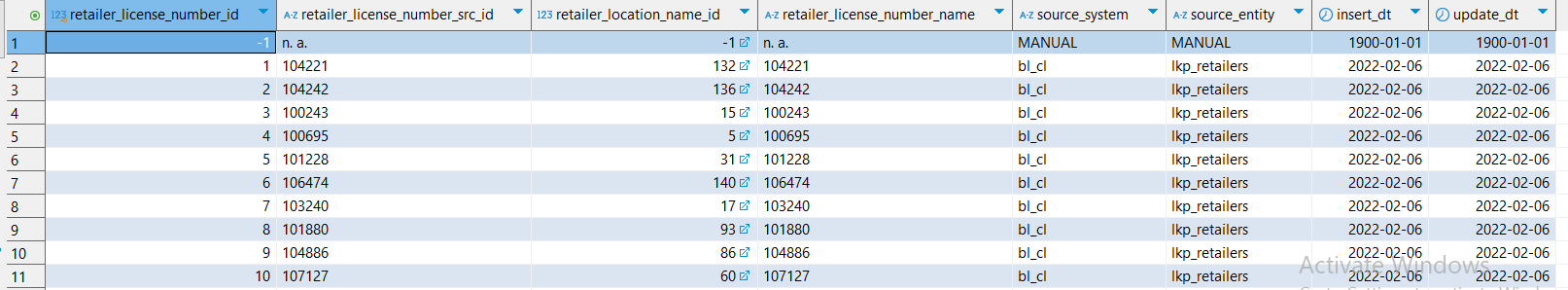
The process begins with the SQL function f\_get\_merged\_retailer\_license\_numbers(), which queries the source bl\_cl.lkp\_retailers lookup table. The function applies transformation logic to clean and consolidate data. It uses MAX(CASE WHEN ...) to extract the most meaningful non-null and non-placeholder ('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.

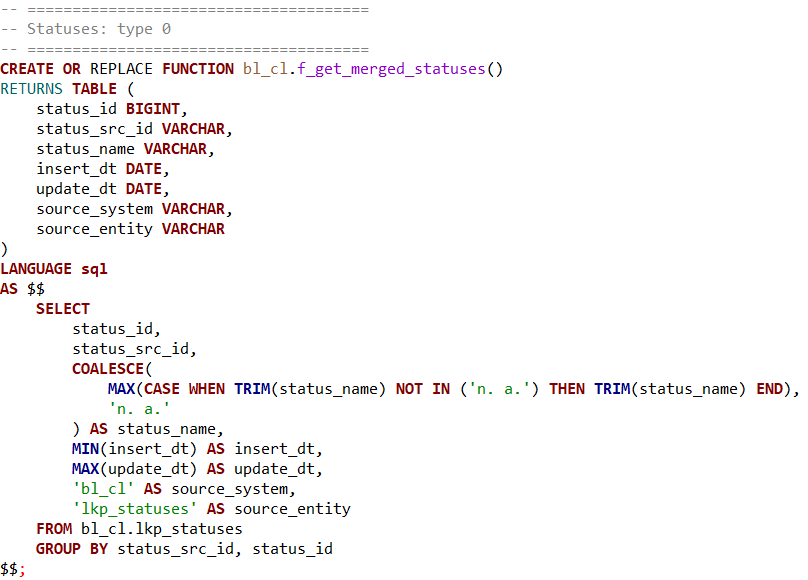
The associated procedure, p\_load\_retailer\_license\_numbers(), iterates over the cleaned dataset and attempts to resolve a foreign key relationship with CE\_LOCATION\_NAMES. It looks up the location\_name\_id by matching the trimmed location name from the current record with the entries in the location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

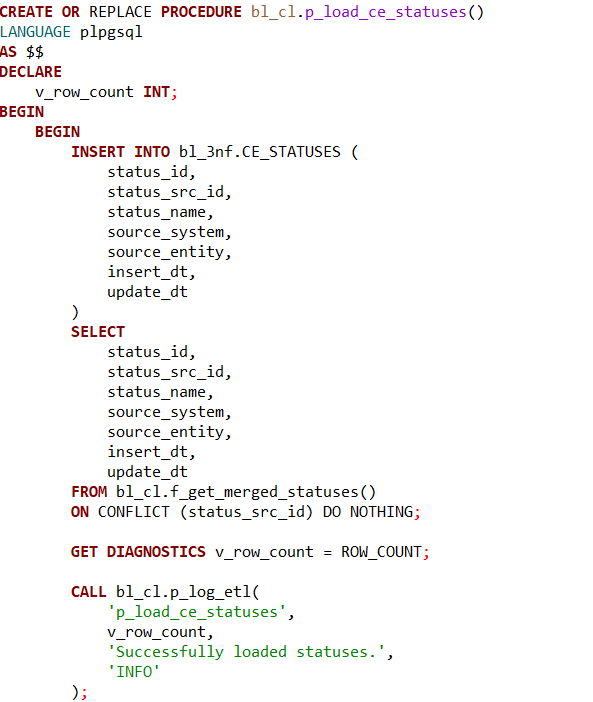
The core of the loop is an INSERT statement that loads each retailer license record into the CE\_RETAILER\_LICENSE\_NUMBERS table. The fields include both identifiers (id, src\_id), descriptive name, location foreign key, and audit metadata. The use of ON CONFLICT (retailer\_license\_number\_src\_id)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

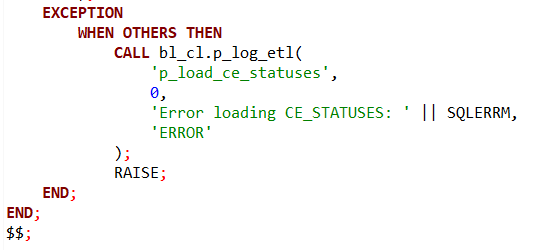
For each row processed, the procedure tracks the number of successful insertions using GET DIAGNOSTICS. This value is aggregated in v\_row\_count to produce a final load count. Any exceptions during individual insert operations are caught and logged via the p\_log\_etl procedure, capturing the source 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.









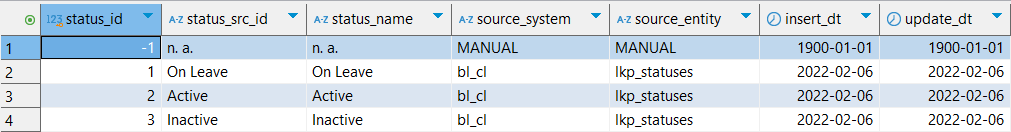
The ETL routine for **Statuses** in the bl\_3nf schema facilitates the normalization and consolidation of status reference data into the central entity table CE\_STATUSES. This process supports consistent status tracking across the data model and enhances downstream analytics, particularly in areas like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

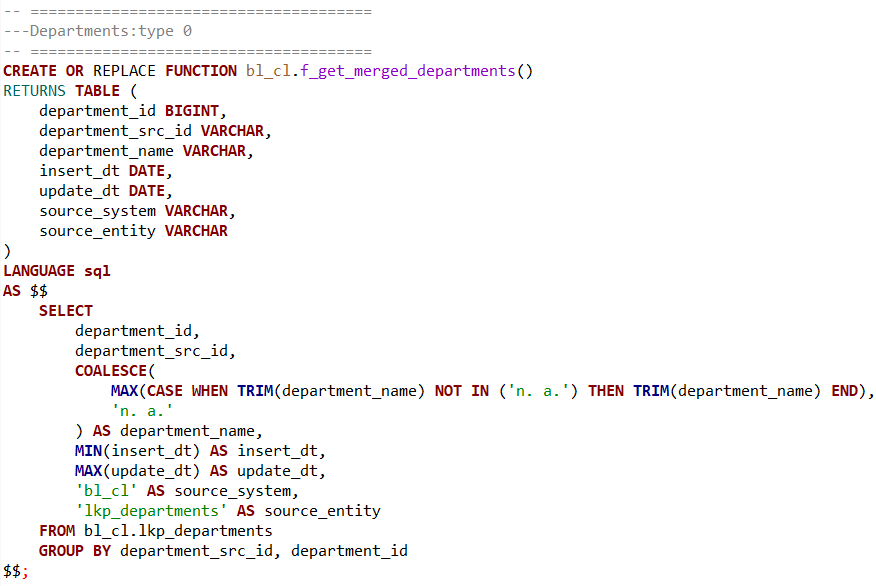
The function f\_get\_merged\_statuses() is responsible for extracting and preparing the raw status data from the bl\_cl.lkp\_statuses source table. Within this function, each record is grouped by its source and internal identifiers—status\_src\_id and status\_id. This grouping enables deduplication and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

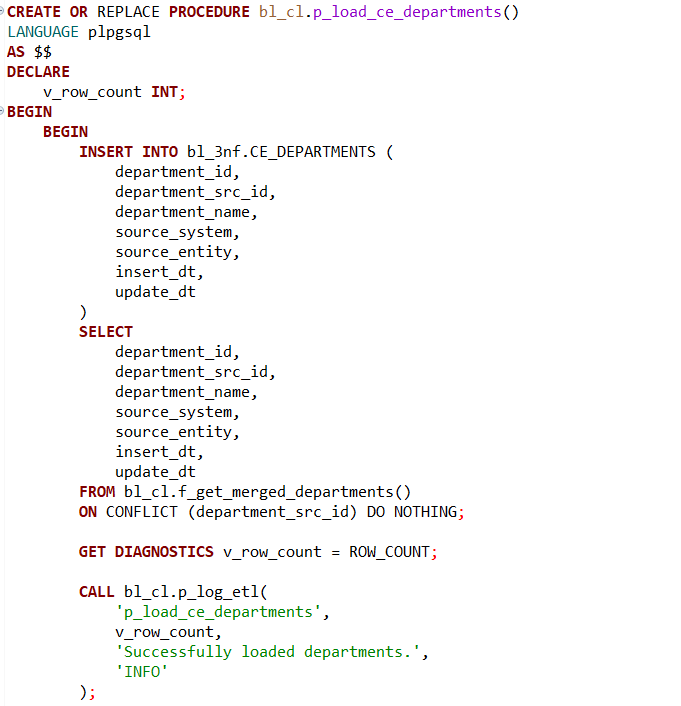
For the transformation step, the function applies a MAX(CASE WHEN ...) pattern to ensure that placeholder values such as 'n.a.' are ignored when selecting the most accurate status\_name. If no valid name is found, it defaults to 'n. a.', preserving data completeness. Timestamps are also normalized: MIN(insert\_dt) 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.

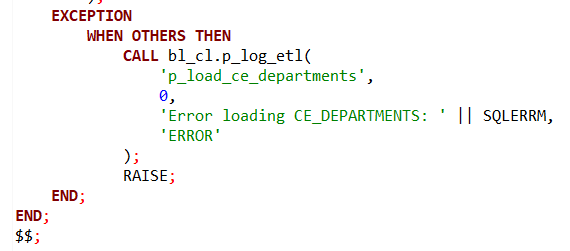
The corresponding procedure p\_load\_ce\_statuses() orchestrates the loading of this curated dataset into the target CE\_STATUSES table. It begins by invoking the function and attempting to insert each record. The use of ON CONFLICT (status\_src\_id) DO NOTHING enforces uniqueness based on the source 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.

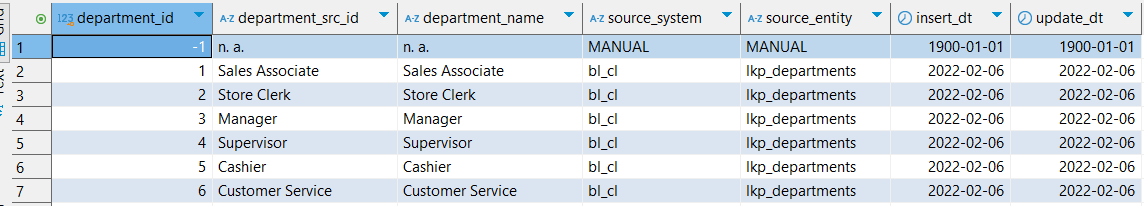
The process starts with the function f\_get\_merged\_departments(), which performs the extraction and transformation of data from the raw source table bl\_cl.lkp\_departments. This function handles potential inconsistencies in naming by applying a MAX(CASE WHEN ...) expression to select the most meaningful, 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.

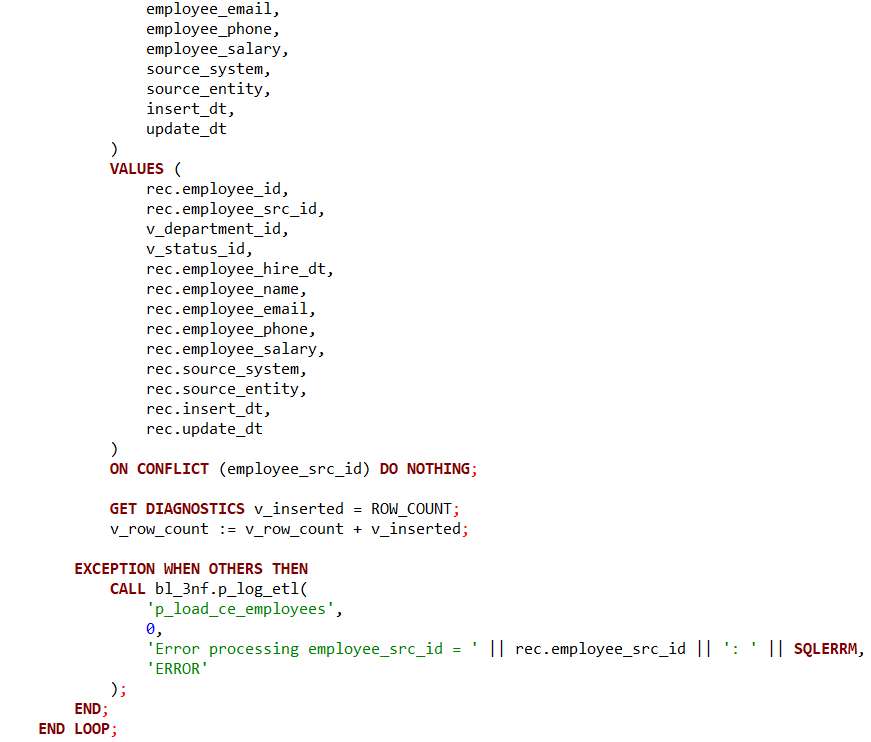
The procedure uses GET DIAGNOSTICS to determine how many rows were successfully inserted and logs this information through the p\_log\_etl procedure. This logging mechanism enhances operational monitoring and traceability by capturing both success metrics and error conditions.

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.

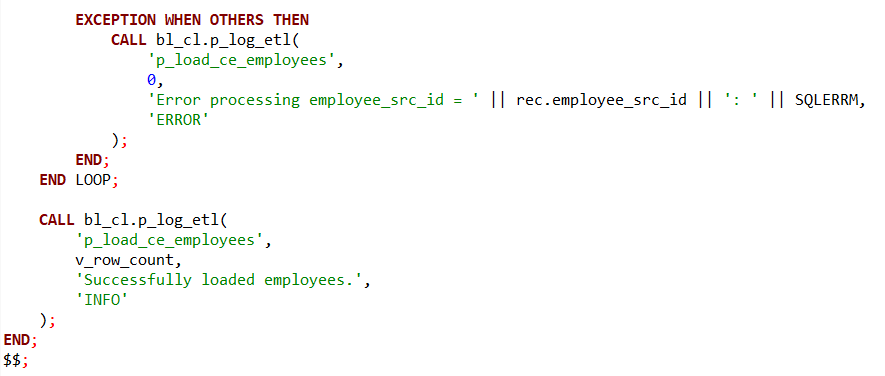












This segment of the ETL pipeline is responsible for loading and maintaining employee data using a **Slowly Changing Dimension Type 1 (SCD Type 1)** approach. In contrast to Type 2, where historical versions are preserved, Type 1 focuses solely on keeping the most **current version** of each record. Any changes to an employee’s attributes **overwrite** the previous values, ensuring that the target table always reflects the latest state.

The ETL process begins with the function f\_get\_merged\_employees(), which consolidates employee data from the source table bl\_cl.lkp\_employees. This table may contain multiple entries per employee, often with slight variations in values due to updates or inconsistencies. The function groups these records by employee\_id and employee\_src\_id, which collectively identify a unique employee across systems.

To ensure high data quality, the function filters out placeholder values such as 'n. a.' and trims excess whitespace. For each textual attribute—like department name, status, name, email, and phone—it uses a MAX(CASE WHEN ...) construct to pick the most complete and non-placeholder version of the value. Numeric and date fields like employee\_salary, employee\_hire\_dt, insert\_dt, and update\_dt are aggregated using MAX() or MIN() as appropriate. These operations ensure that the function outputs **a single, cleaned, and deduplicated record** per employee.

With the merged dataset prepared, the procedure p\_load\_ce\_employees() takes over to insert this data into the target table bl\_3nf.CE\_EMPLOYEES. For each record, it first attempts to resolve surrogate keys for both the employee’s **department** and **status** by looking up bl\_3nf.CE\_DEPARTMENTS and bl\_3nf.CE\_STATUSES. If no match is found, it assigns a default value of -1 to indicate a missing or unmatched reference.

The heart of the SCD Type 1 logic lies in the INSERT INTO ... ON CONFLICT clause. This pattern allows the procedure to **insert new employees** or **update existing ones** based on a conflict in the employee\_src\_id, which serves as the natural key. If an employee already exists, the procedure will selectively update fields only if new values are valid and newer. For example:

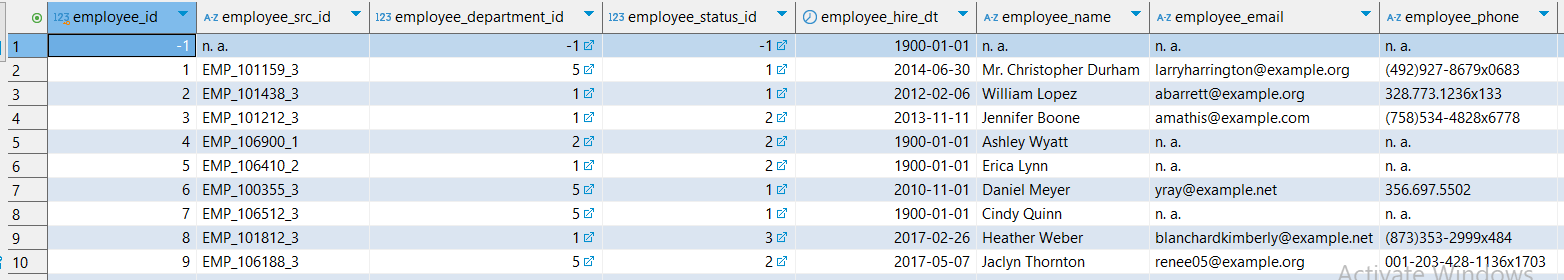
* If the new department or status ID is not -1, it will replace the old one.
* If the new hire date isn’t '1900-12-31', a known placeholder, it’s accepted.
* Text fields are updated only if they are not empty or 'n. a.'.
* The salary is updated only if it’s present and positive.
* The update\_dt is set to the current insert\_dt to track freshness.

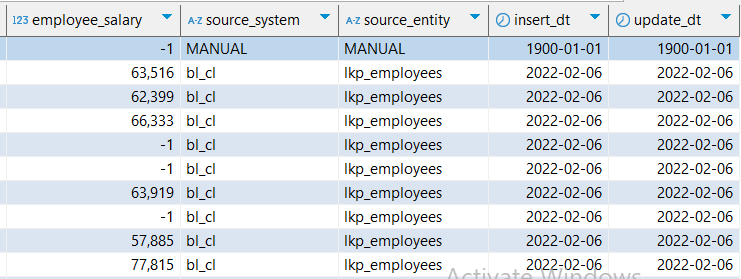
To avoid unnecessary overwrites, the procedure checks if the new data is actually **different** (IS DISTINCT FROM) from what already exists. Only if a meaningful change is detected and the insert\_dt is **more recent** than the existing update\_dt will the update be applied.

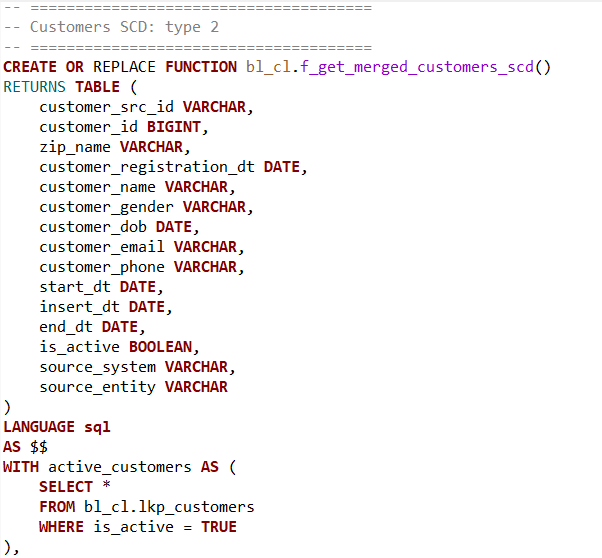
Throughout the process, the procedure maintains a counter v\_row\_count to track how many records were inserted or updated. Errors are gracefully caught and logged using a dedicated logging function, p\_log\_etl(), which records the error message along with the affected employee\_src\_id for easier debugging.

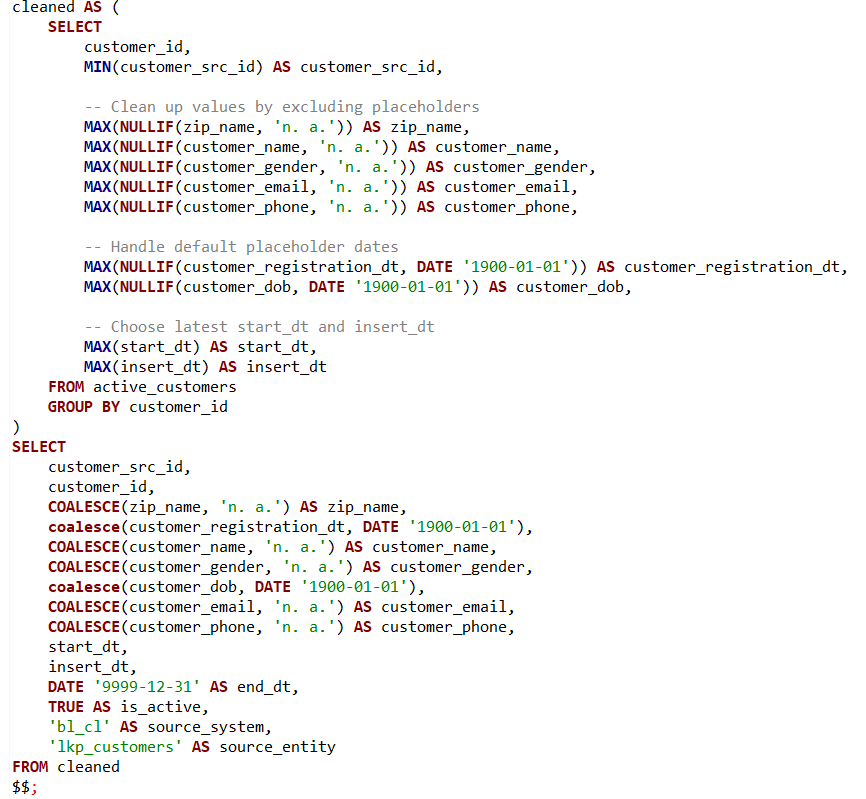
At the end of execution, a summary message is logged to confirm how many records were successfully processed, ensuring visibility into each ETL run’s impact.

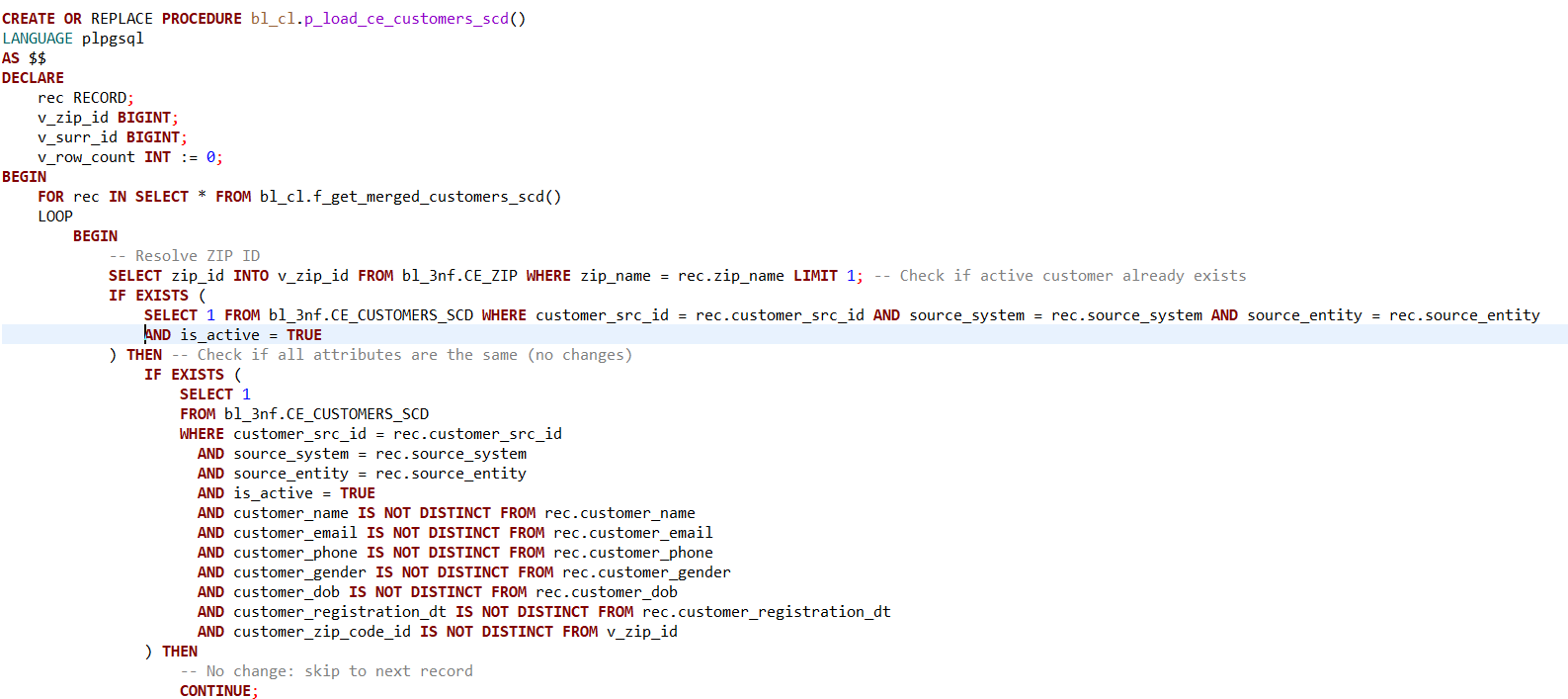
In summary, this SCD Type 1 logic prioritizes **data freshness and accuracy**, ensuring that the most current and valid employee information is always available in the data warehouse. By avoiding history tracking and overwriting outdated attributes, this approach is particularly suited for volatile, non-historic attributes where only the latest state matters—such as contact details, employment status, or salary figures.



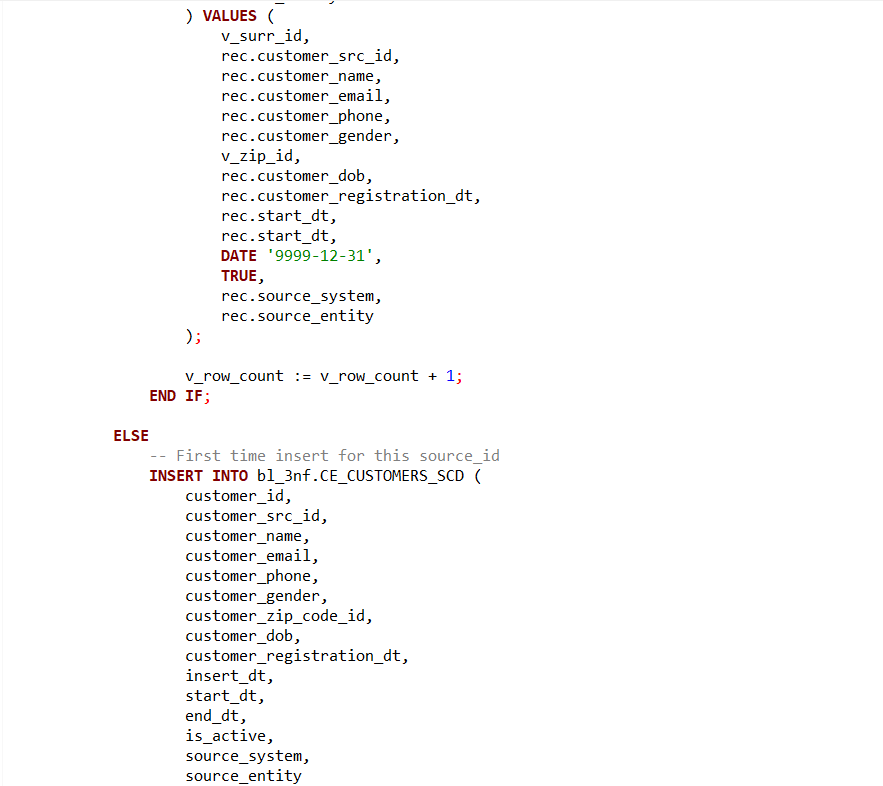


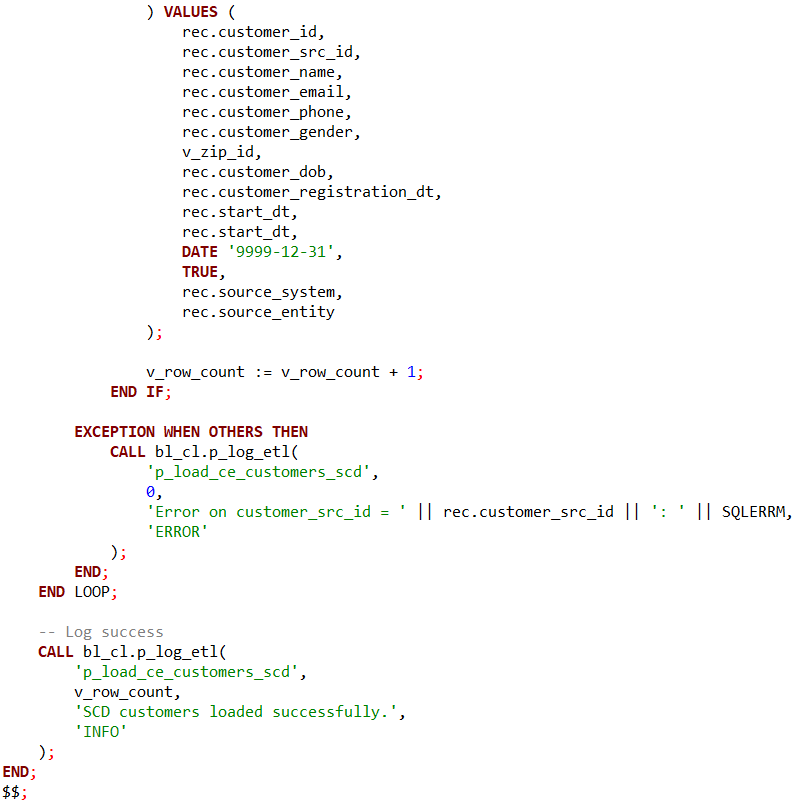












This part of the ETL pipeline is designed to manage customer data using Slowly Changing Dimension Type 2 (SCD Type 2) logic, which preserves historical changes to customer attributes over time. The source data is stored in the bl\_cl.lkp\_customers table, where each row represents a version of a customer’s information, including key attributes such as name, contact details, gender, and registration date. Records marked as active (is\_active = TRUE) are considered the latest version from the source system.

To prepare the data for insertion into the target SCD table, the function f\_get\_merged\_customers\_scd() performs several data cleaning and consolidation steps. It begins by selecting only the active customer records from the source. Within this filtered set, it groups records by customer\_id and derives a single cleaned record per customer. Placeholder values such as 'n.a.' or invalid default dates like '1900-01-01' are replaced with NULL, and the function selects the most meaningful available values using aggregate functions like MAX() for string and date fields. If valid data cannot be found, the final selection uses COALESCE to safely default back to placeholders like 'n.a.' or the original fallback date.

Each cleaned record is then enriched with standard metadata to fit the SCD Type 2 model. The function sets start\_dt and insert\_dt to reflect the latest known values, and assigns a distant future end\_dt of '9999-12-31' to mark the record as currently active. It also includes metadata fields for source\_system and source\_entity to support traceability across ETL operations.

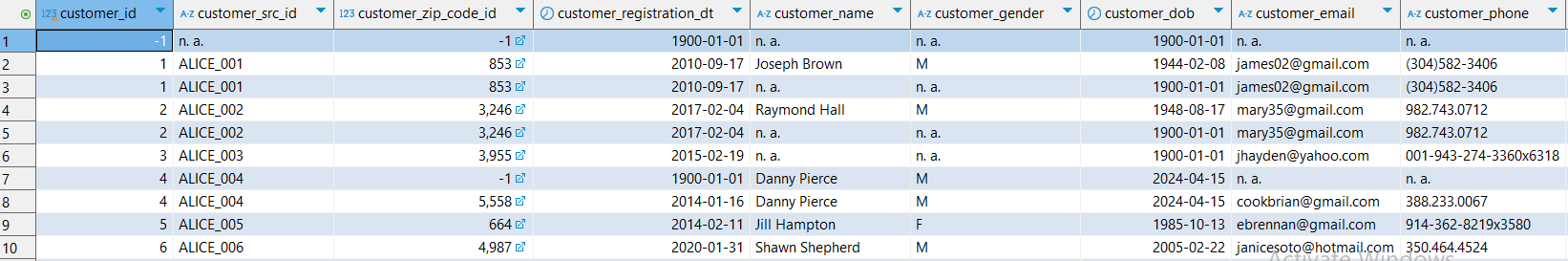
The corresponding procedure, p\_load\_ce\_customers\_scd(), is responsible for loading this processed data into the target table bl\_3nf.CE\_CUSTOMERS\_SCD. This procedure implements the logic required to maintain historical integrity of the SCD table. For each incoming record, it first attempts to resolve the customer’s ZIP code ID by looking up zip\_name in the bl\_3nf.CE\_ZIP table. It then checks if there is already an active version of this customer in the target table (based on customer\_src\_id, source\_system, and source\_entity).

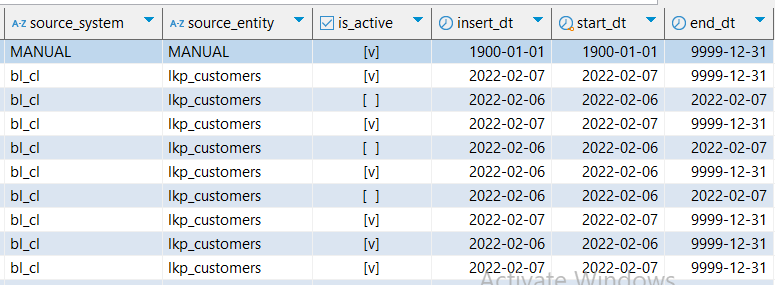
If an active record is found, the procedure performs a field-by-field comparison between the incoming record and the currently active one. If all significant attributes are unchanged (using IS NOT DISTINCT FROM to allow for NULL-safe comparison), the procedure skips the record—preserving the existing active version as still current. However, if any attribute has changed, it "expires" the existing active record by setting its is\_active flag to FALSE and adjusting its end\_dt to match the start\_dt of the new version. A new row is then inserted with the updated values, keeping the same surrogate customer\_id, and marking it as the active record with an open-ended end\_dt.

If no active record exists for the customer (i.e., it’s a first-time insert), the procedure simply inserts the record as a new active row, using the source-provided customer\_id as the surrogate key.

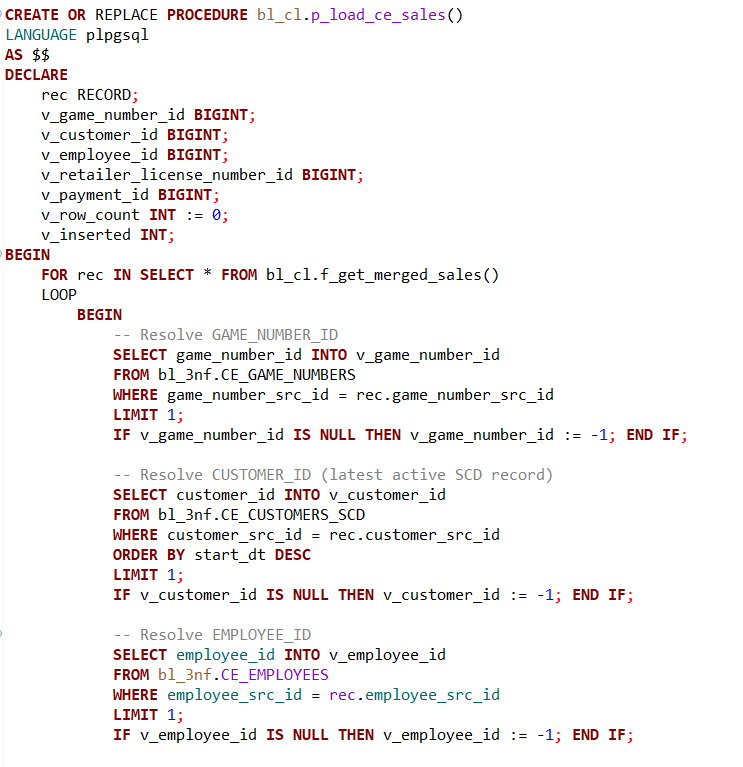
To ensure operational transparency, the procedure captures any errors encountered during processing and logs them using a centralized p\_log\_etl() function. For each failed record, it logs the customer\_src\_id along with the error message and error status. Upon successful completion, it logs the number of records inserted and confirms the load with an informational message.

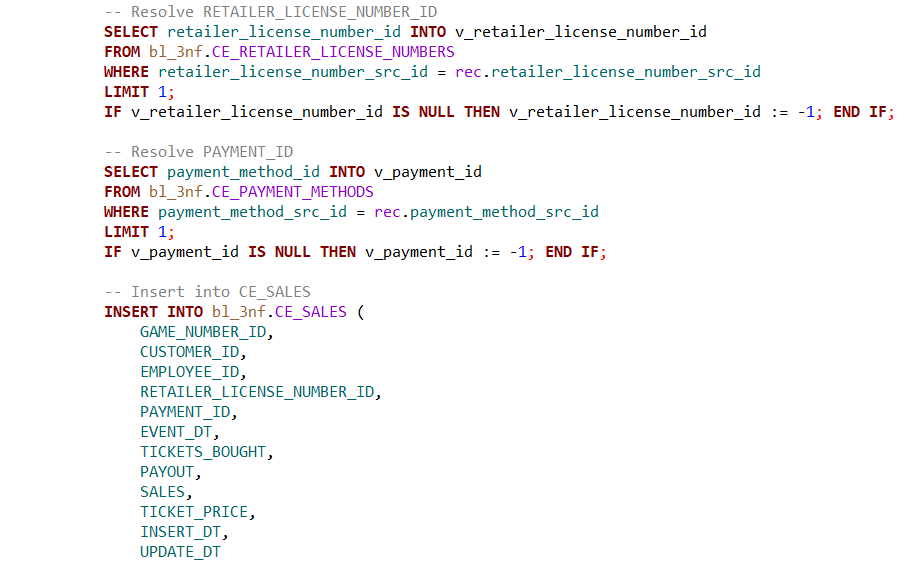
Overall, this ETL component ensures that customer data in the warehouse reflects not only the latest information but also preserves a detailed historical trail. By applying SCD Type 2 techniques, it supports full auditability and time-based analysis while gracefully handling data quality issues, deduplication, and source variations.

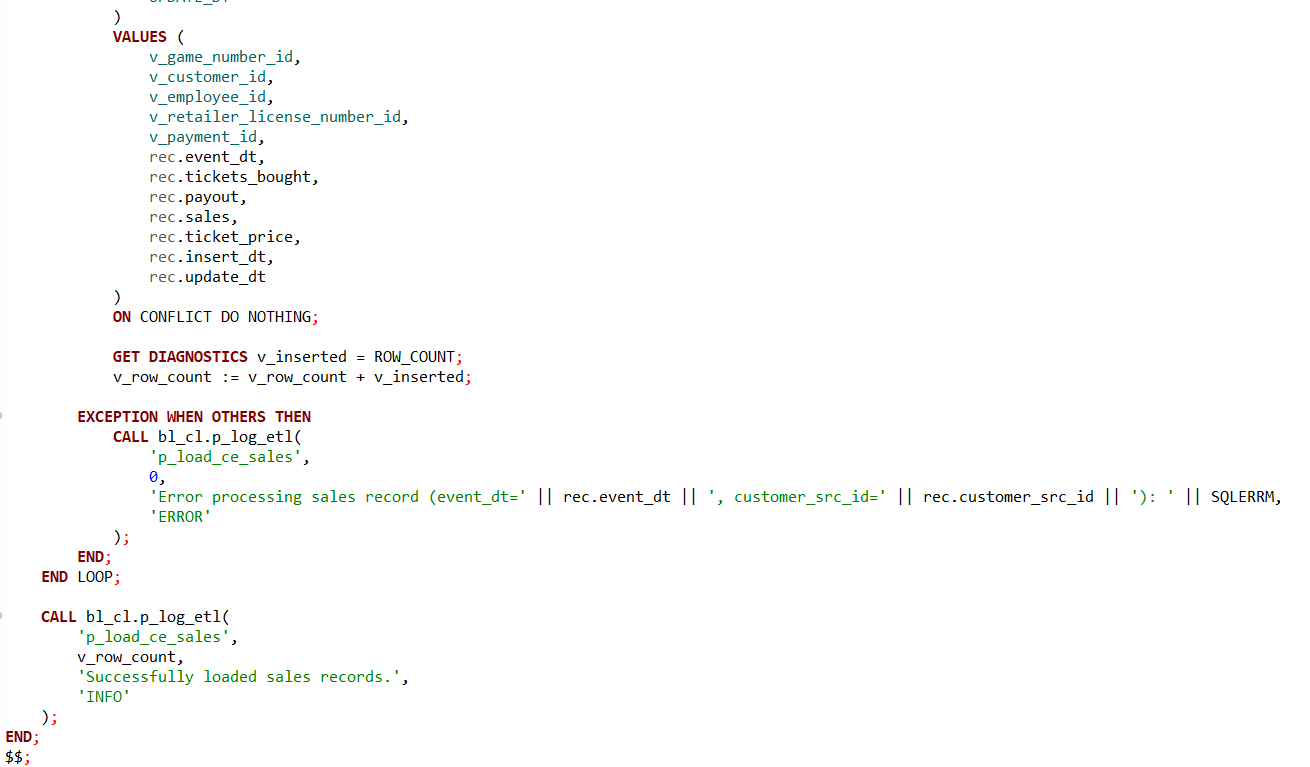












The procedure p\_load\_ce\_sales() is designed to load transactional sales data into the CE\_SALES table within the bl\_3nf schema. Unlike dimension tables, which store relatively stable reference data, the sales fact table captures **event-level transactions** and serves as a critical foundation for analytics related to revenue, customer behavior, game performance, and more.

The procedure begins by iterating over records returned from the function f\_get\_merged\_sales(). This function is assumed to return clean, consolidated sales data, with deduplicated and formatted fields ready for integration into the 3NF structure. For each sales record, several foreign key references must be resolved to maintain referential integrity in the target fact table.

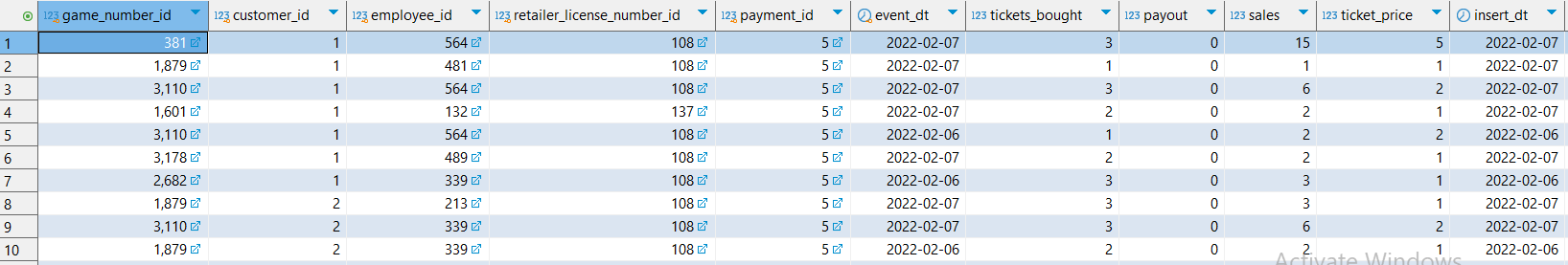
The procedure performs five critical lookups for each sales transaction:

1. **Game Number**: A reference to the CE\_GAME\_NUMBERS table using game\_number\_src\_id from the source. If no match is found, it defaults to -1, which signals a missing or unmatched reference during analysis.
2. **Customer**: Perhaps the most nuanced lookup, the procedure resolves the customer via the SCD-enabled CE\_CUSTOMERS\_SCD table. It selects the latest active version of the customer based on start\_dt, reflecting the business requirement that facts should relate to the **most current valid dimension state** at the time of data load.
3. **Employee**: A reference to the CE\_EMPLOYEES table using employee\_src\_id. Again, if no match exists, the placeholder -1 is used.
4. **Retailer License Number**: This identifies the retailer associated with the sale. The lookup is performed against CE\_RETAILER\_LICENSE\_NUMBERS.
5. **Payment Method**: This maps to a normalized list of payment methods stored in CE\_PAYMENT\_METHODS.

Once all reference keys are resolved (or defaulted), the procedure proceeds to insert the transaction into the CE\_SALES fact table. The key transactional attributes include: event\_dt( the date of the transaction),tickets\_bought, payout, sales, ticket\_price( key quantitative measures associated with the transaction), metadata such as insert\_dt and update\_dt for tracking when the fact was loaded or last refreshed.  
 Importantly, the INSERT statement uses the **ON CONFLICT DO NOTHING** clause. This suggests that **sales records are treated as immutable**—once loaded, they are not updated or overwritten. This is a common pattern in fact table design, where the primary key is typically a compound of business keys (or a surrogate key), and duplicates are avoided by skipping conflicting inserts. The decision to "do nothing" on conflict reflects a business rule that fact data is either fully trusted to be immutable or that upstream de-duplication guarantees uniqueness.

As with other procedures in this ETL pipeline, robust error handling is implemented using a BEGIN...EXCEPTION block within the loop. Any failures encountered during lookups or inserts trigger a call to the generic ETL logging procedure p\_log\_etl(), which records the issue along with identifying information (such as event\_dt and customer\_src\_id). This ensures full traceability of problematic records.

The procedure also tracks the number of successfully inserted rows using GET DIAGNOSTICS, which retrieves the row count after each insert. This count is aggregated in v\_row\_count and logged at the end of the procedure for auditing and monitoring purposes.



----Test that the procedure can be executed repeatedly with consistent results--------------------

--------------------------------------------------------------------------------------------------------------------------

---1.1.Run the procedure:

-----------------------------------------------------------------------------------------------

**CALL** bl\_cl.sp\_run\_batch\_etl\_by\_day(**DATE** '2022-02-06', **DATE** '2022-02-07');

----------------------------------------------------------------------------------------------------------------------------------------

---1.2.Query the logging table for affected rows:

-------------------------------------------------------------------------------------------------------------------------------------------------

**SELECT** \*

**FROM** bl\_3nf.etl\_logs

**WHERE** procedure\_name **IN** (

'sp\_upsert\_lkp\_game\_types',

'sp\_upsert\_lkp\_game\_categories',

'sp\_upsert\_lkp\_game\_numbers',

'sp\_upsert\_lkp\_payment\_methods',

'sp\_upsert\_lkp\_states',

'sp\_upsert\_lkp\_cities',

'sp\_upsert\_lkp\_zips',

'sp\_upsert\_lkp\_location\_names',

'sp\_upsert\_lkp\_retailers',

'sp\_upsert\_lkp\_statuses',

'sp\_upsert\_lkp\_departments',

'sp\_upsert\_lkp\_employees',

'sp\_upsert\_lkp\_customers',

'sp\_upsert\_lkp\_sales',

'p\_load\_ce\_game\_types',

'p\_load\_ce\_game\_categories',

'p\_load\_ce\_game\_numbers',

'p\_load\_ce\_payment\_methods',

'p\_load\_ce\_states',

'p\_load\_ce\_cities',

'p\_load\_ce\_zip',

'p\_load\_ce\_location\_names',

'p\_load\_retailer\_license\_numbers',

'p\_load\_ce\_statuses',

'p\_load\_ce\_departments',

'p\_load\_ce\_employees',

'p\_load\_ce\_customers\_scd',

'p\_load\_ce\_sales'

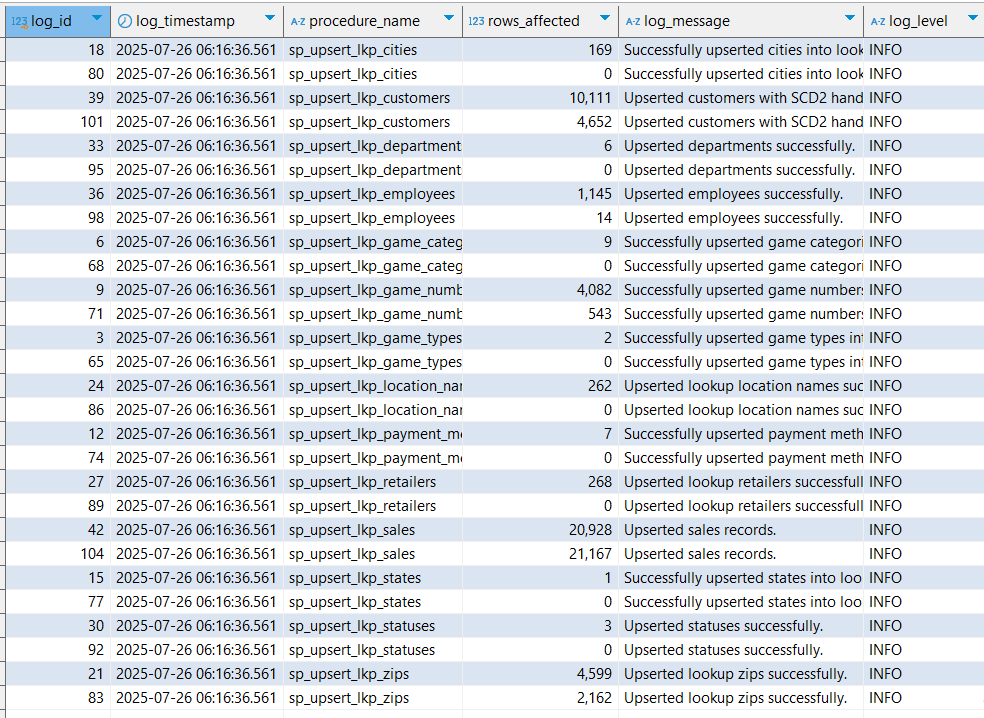
)

**order** **by** procedure\_name, log\_id ;

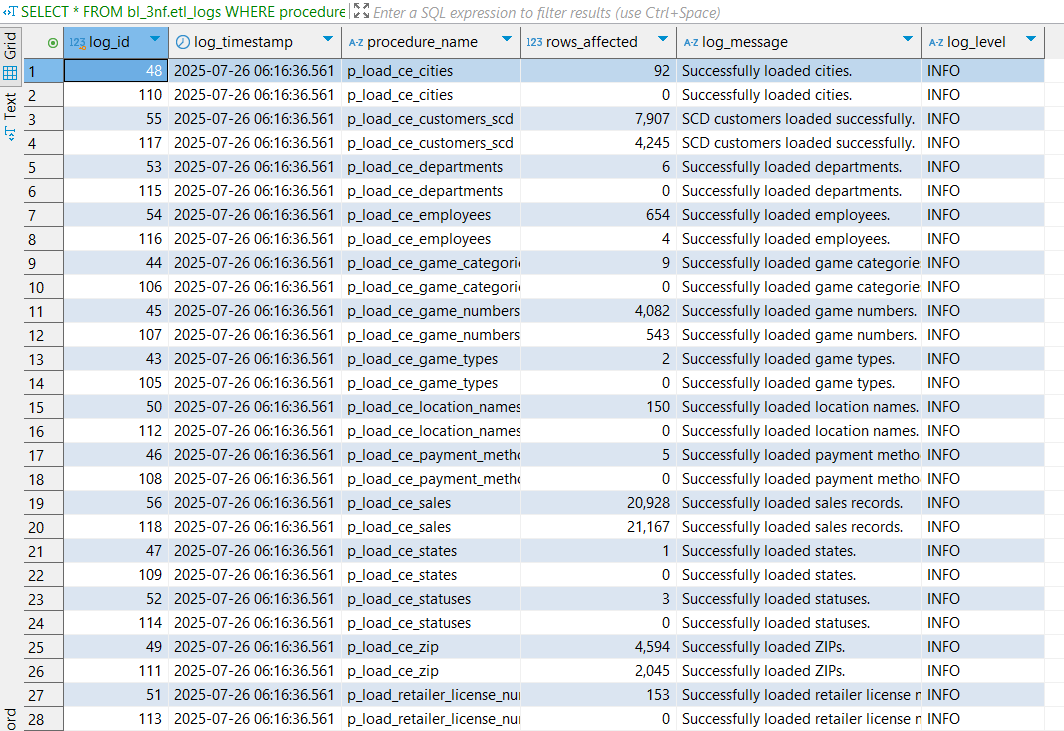
----------------------------------------------------------------------------------------------------------------------------------------

---1.3.Take a screenshot of the logging result

-----------------------------------------------------------------------------------------------------------------------------------------------



Results of the tables on the BL\_CL level.



Results of the tables on the 3nf level.

----------------------------------------------------------------------------------------------------------------------------------------

---1.4. Run the procedure again with the same input

--------------------------------------------------------------------------------------------------------------------------------------------------

**CALL** bl\_cl.sp\_run\_batch\_etl\_by\_day(**DATE** '2022-02-06', **DATE** '2022-02-07');

----------------------------------------------------------------------------------------------------------------------------------------

---1.5. Query the login table again

--------------------------------------------------------------------------------------------------------------------------------------------------

**SELECT** \*

**FROM** bl\_3nf.etl\_logs

**WHERE** procedure\_name **IN** (

'sp\_upsert\_lkp\_game\_types',

'sp\_upsert\_lkp\_game\_categories',

'sp\_upsert\_lkp\_game\_numbers',

'sp\_upsert\_lkp\_payment\_methods',

'sp\_upsert\_lkp\_states',

'sp\_upsert\_lkp\_cities',

'sp\_upsert\_lkp\_zips',

'sp\_upsert\_lkp\_location\_names',

'sp\_upsert\_lkp\_retailers',

'sp\_upsert\_lkp\_statuses',

'sp\_upsert\_lkp\_departments',

'sp\_upsert\_lkp\_employees',

'sp\_upsert\_lkp\_customers',

'sp\_upsert\_lkp\_sales',

'p\_load\_ce\_game\_types',

'p\_load\_ce\_game\_categories',

'p\_load\_ce\_game\_numbers',

'p\_load\_ce\_payment\_methods',

'p\_load\_ce\_states',

'p\_load\_ce\_cities',

'p\_load\_ce\_zip',

'p\_load\_ce\_location\_names',

'p\_load\_retailer\_license\_numbers',

'p\_load\_ce\_statuses',

'p\_load\_ce\_departments',

'p\_load\_ce\_employees',

'p\_load\_ce\_customers\_scd',

'p\_load\_ce\_sales'

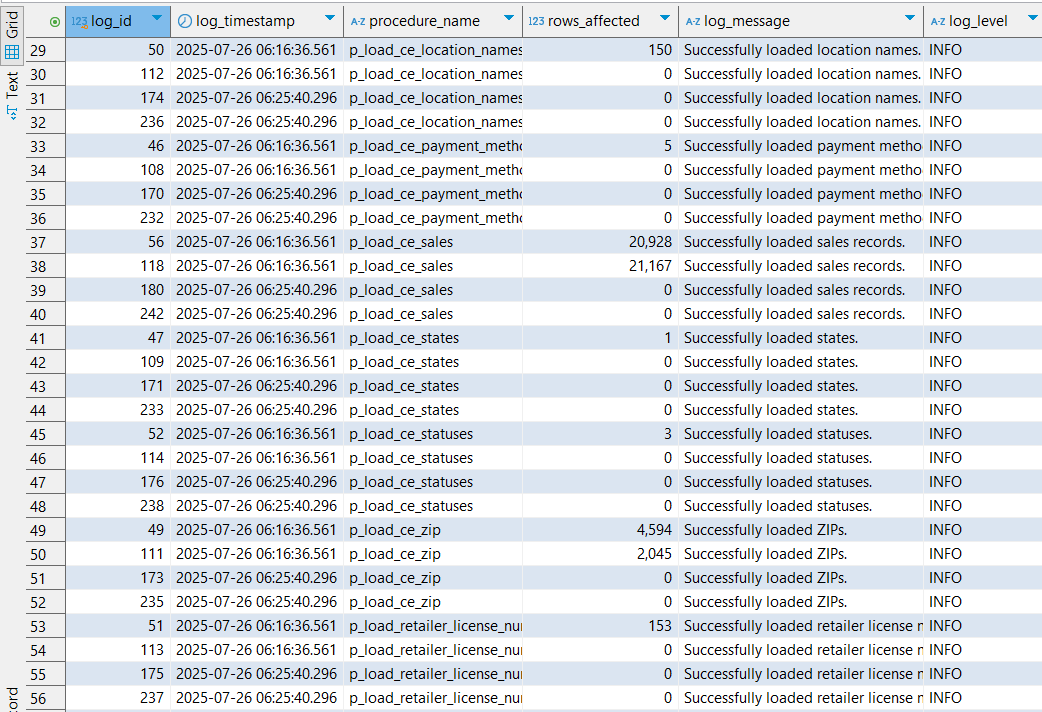
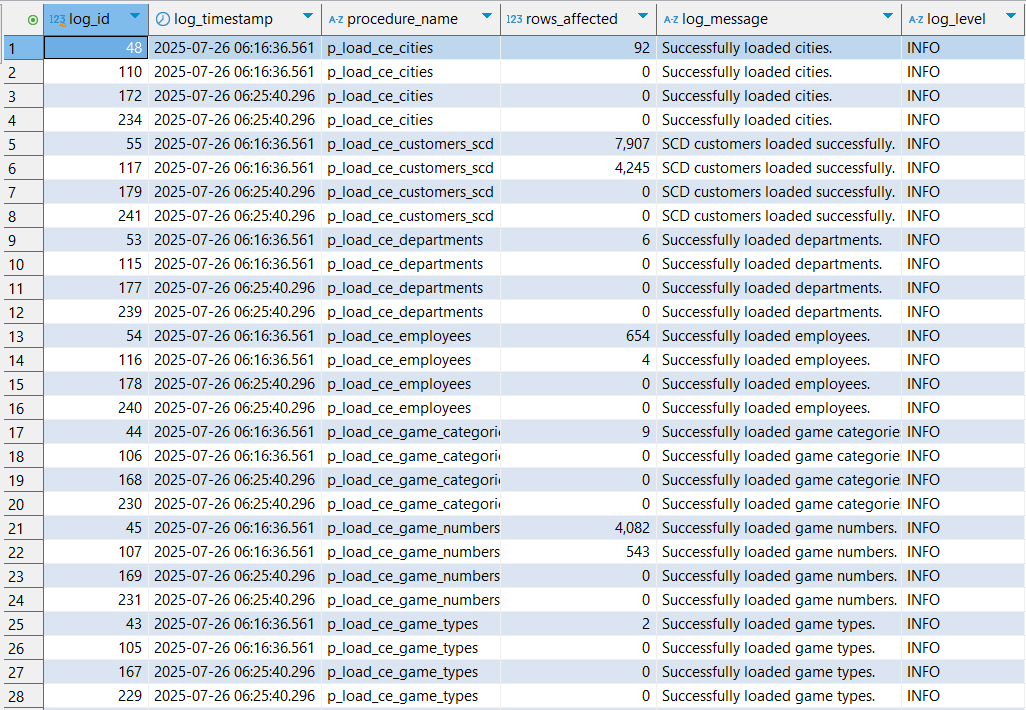
)

**order** **by** procedure\_name, log\_id ;

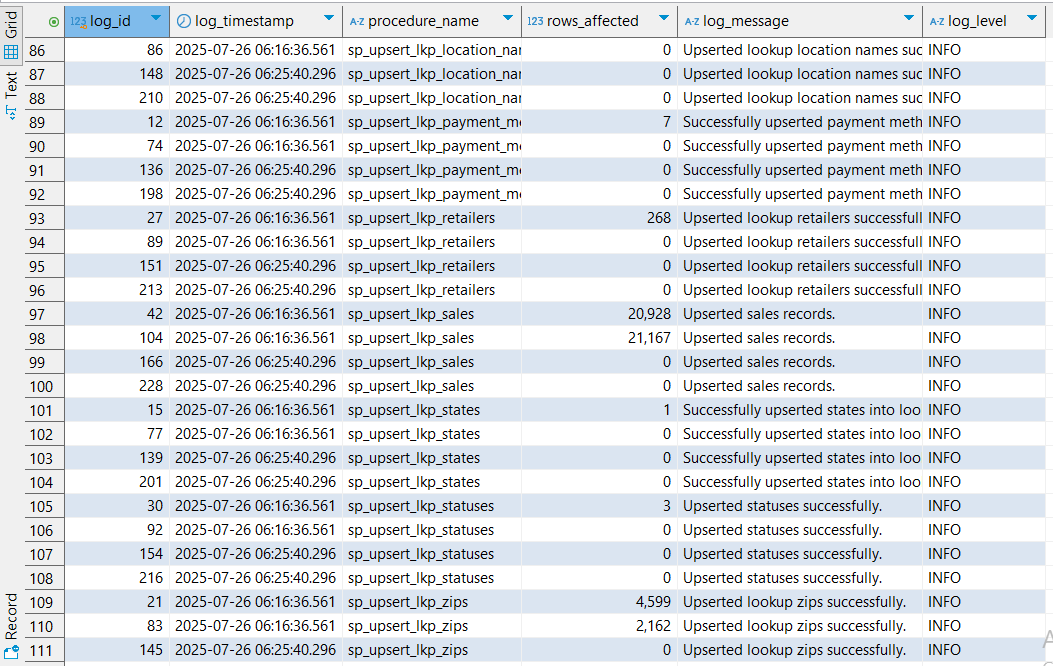
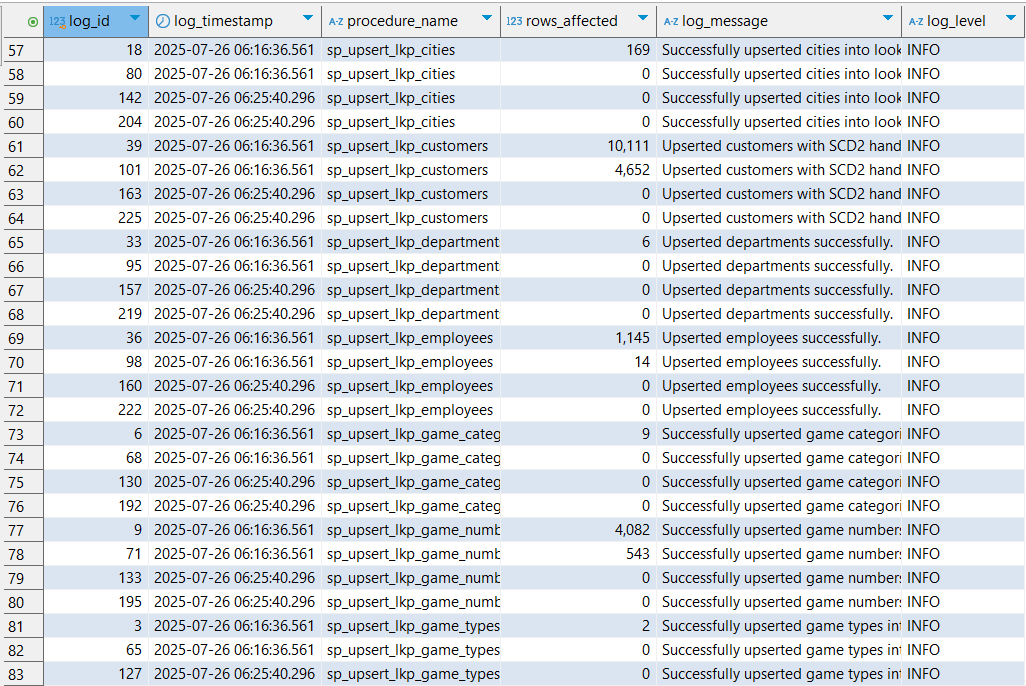
----------------------------------------------------------------------------------------------------------------------------------------

---1.6. Confirm number of rows affected is 0 (or explain why)

----------------------------------------------------------------------------------------------------------------------------------------



Results of the procedures on loading 3nf tables.



Results of the procedures on loading BL\_CL tables.

The number of rows is zero because running the process for the same period again will not update any rows based on our pipeline logic. According to this logic, an update only occurs if **at least one of the attributes changes** and the insertion date of the new candidate row is later than the update date of the existing row with the same source ID. Otherwise, the ETL pipeline’s behavior wouldn’t make sense, since we assume row frequency changes happen daily, not within the same day.

Therefore, running the procedure for the same dates again will have no effect—any candidate row will be rejected, and no rows will be updated or inserted.