QC Utility - User's Guide v24.0

Author: Philippe Debois (European Commission)

Table of content

1. Introduction

1.1. Background

Adhering to best practices in database design is essential for ensuring readability, maintainability, adaptability, and integrity. Database architecture is a specialized discipline that requires expertise and knowledge that may not be universally possessed. Even with a dedicated database designer on the team, their availability can be limited. Without a rigorous quality review process, poorly designed database objects may emerge, particularly when unqualified developers are involved.

CASE tools (e.g., Oracle Data Model, ERWIN) can significantly aid in database engineering by providing built-in functionalities to check naming conventions and other design rules prior to object creation. However, even with such tools, controls can sometimes be bypassed. For instance, production tables may be inadvertently retained during a release, complicating the distinction between backup or temporary tables and operational ones when naming conventions are lax. Consequently, implementing quality checks directly within the database is crucial for identifying anomalies that might otherwise go unnoticed.

The QC Utility facilitates quality checks on any database schema against your naming standards and best practices. It features a predefined set of rules that can be activated or customized, along with default naming conventions. These naming patterns utilize standard regular expressions, enhanced with specific keywords. Continuous quality monitoring of database objects, including PL/SQL code identifiers, is achievable by scheduling a job that runs the tool regularly, reporting only the anomalies discovered or resolved since the last run. The utility supports integration with SonarQube and Bamboo, and it can automatically rectify many detected issues by renaming incorrectly named database objects, dropping unnecessary items (e.g., unused sequences), and creating missing elements (e.g., foreign key indexes, audit columns). Names are generated according to configurable patterns for each object type, and before renaming or dropping objects, the tool conducts an impact analysis to identify any references in the database (note that references in the Java layer cannot be checked).

1.2. Key Features

The tool offers the following features:

- Support for multiple applications and schemas
- Ability to define which database objects are in scope or out of scope of the quality checks
- Ability to define and check naming conventions applicable to all database objects, including identifiers of PL/SQL code (the tool comes with predefined naming conventions that can be customized)
- Ability to check database design best practices (in addition to naming conventions)
- Customizable and extendable dictionary of keywords
- Automatic extraction of table aliases
- Manual launch or automated run for continuous checking via the database scheduler
- Automatic detection of changes made via DDL since last run

- Incremental execution reports (show newly detected anomalies and those resolved since last run)
- Execution reports sent by email
- Impact analysis for objects needing to be renamed or dropped
- Automatic fixing of some anomalies (renaming, creation of missing objects, etc.)
- Ability to report (online or via email) on database changes made between two dates or runs
- Possible integration with SonarQube via its generic import JSON format
- Possible integration with Bamboo via JUnit XML format

1.3. Quick Start

1. **Install the Application**
   Install the QC tool using the DBM tool (see "installation" section below)
2. **Set Up the Tool**
   Register the applications and schema(s) to be quality checked using the setup command of the DBM tool (see "setup" section below).
3. **Configure Naming Patterns**
   Review and adapt the default patterns provided in the QC_PATTERNS table according to your specific naming conventions.
   For each object type:
   i. Define the include pattern, i.e., objects that are in the scope of the quality checks.
   ii. Define the exclude pattern, i.e., objects that are out of scope of the quality checks.
   iii. Define the check pattern, i.e., how objects should be named according to the standards.
   iv. Define the anti-pattern, i.e., the pattern that object names should not match.
   v. Define the fix pattern, i.e., how the object name is generated when fixing anomalies.
   Patterns are defined using a combination of regular expressions (see "Regular expressions") and keywords like `{app alias}` or `{table alias}` (see "Keywords").
4. **Extract Table Aliases**
   Execute the command:
   ```
   exec qc_utility_krn.extract_table_aliases_from_pk();
   ```
   Table aliases are extracted from the names of primary or unique keys by default. You can pass another extract pattern as a parameter if necessary (see "Extraction of aliases").
5. **Review Table Aliases**
   Check the extracted table aliases in the QC_DICTIONARY_ENTRIES table and define any missing ones.
6. **Run Quality Checks**
   Execute the command:
   ```
   exec qc_utility_krn.check_all()
   ```
7. **Review Results**
   Check the results sent by email. If necessary, return to step 3 to refine your settings and rerun the quality checks.
8. **Analyse Impact**
   Perform impact analysis (see "Impact analysis").
9. **Fix Anomalies**

Address any detected anomalies (see "Fixing anomalies").

2. Installation

2.1. Installation type

The tool can be installed in its own schema (**central installation**) or in the schema of the application to be quality checked (**private installation**). When the tool is used to quality check multiple applications and/or schemas, a central installation avoids the burden of installing and subsequently maintaining it multiple times. Note that if your application schemas are spread across several databases, the tool must still be installed in each of them.

2.1.1. Central installation

If the tool is installed in a separate dedicated schema, it must have the necessary privileges to access the schema(s) where application database objects are stored. Write access is necessary to allow the tool to fix anomalies (e.g., renaming objects, creating missing ones, etc.). The required privileges are:

- SELECT ANY TABLE
- SELECT ANY DICTIONARY
- CREATE ANY <object_type>
- DROP ANY <object_type>

2.1.2. Private installation

When the tool is installed in the same schema as the application to be quality checked, the database objects of the QC tool must be excluded from the scan using the “exclude pattern” (this is the default behavior).

2.2. Installation procedure

The QC utility is part of the EC PL/SQL toolkit and utilises the MAIL and LOG utilities, which are also available in the toolkit.

The steps to install these tools are the following:

- Set `DBM_USERNAME`, `DBM_PASSWORD`, and `DBM_DATABASE` environment variables corresponding to the schema in which you want to install the tool (central or application specific).
- Execute the following shell command while located in the home directory of the toolkit:
  `dbm-cli install log, mail, qc`.

2.3. Installed objects

The following QC objects are created in the database schema:

Object Type	Object Name	Description
PACKAGE	QC_UTILITY_KRN	Main package of the QC utility (KRN = Kernel)
PACKAGE	QC_UTILITY_VAR	Global variables of the QC utility
PACKAGE	QC_UTILITY_MSG	API for managing versions of QC_RUN_MSGS records (internal use)
PACKAGE	QC_UTILITY_STAT	API for managing versions of QC_RUN_STATS records (internal use)
PACKAGE	QC_UTILITY_ORA_04068	Parser for detecting packages containing global variables/cursors (internal use)
TABLE	QC_CHECKS	List of quality checks
TABLE	QC_APPS	Applications to check
TABLE	QC_DICTIONARY_ENTRIES	Entries of the QC utility dictionaries
TABLE	QC_PATTERNS	Various patterns (check, in/out scope, fix, anti-pattern, etc.)
TABLE	QC_RUNS	Runs (invocations) of the QC utility
TABLE	QC_RUN_MSGS	Messages raised during each run
TABLE	QC_RUN_STATS	Statistics collected during each run
TABLE	QC_RUN_LOGS	Information logged during each run
TABLE	QC_SCHEMA_VERSION	Schema version and status
VIEW	QC_CURRENT_SCHEMA_VERSION	Current schema version and status
SEQUENCE	QC_RUN_SEQ	Sequence for QC_RUNS.RUN_ID
SEQUENCE	QC_STAT_SEQ	Sequence for QC_RUN_STATS.STAT_IVID
SEQUENCE	QC_MSG_SEQ	Sequence for QC_RUN_MSGS.MSG_IVID
JOB	QC_UTILITY_JOB	Job to run the QC utility every hour

Objects of MAIL and LOG utilities are not described in this document.

2.4. Uninstallation

To uninstall the QC tool, execute the following command: `dbm-cli uninstall qc`.

To uninstall the QC tool and the tools it depends on: `dbm-cli uninstall qc, mail, log`.

3. Setup

3.1. Configuration types

The tool supports several configurations:

- The tool can quality check one or several applications (e.g., in a microservices architecture, each microservice can be considered a separate application).
- Each application can have one or more schemas (commonly, application data, application logic, and access layers are organized in separate schemas).
- Schemas are usually dedicated to a single application; however, the tool supports the rare case of one schema being used for multiple applications if the ownership of database objects can be determined based on naming conventions.

Quality check rules (e.g., naming conventions) can be configured as shared across all applications or defined privately for each application. In a **shared configuration**, patterns and dictionary entries are attached to the “ALL” dummy application; in a **private configuration**, they are linked to the specific application. Note that a mix of shared and private rules is not supported, but shared rules can be converted into private rules and vice versa.

Anomalies are reported per application and per schema. If email parameters are shared, a single email is sent for all applications; if private, one email is sent per application.

3.2. Setup wizard

Once the tool has been installed, you must use the setup wizard to register your applications and schemas and decide on the type of configuration (shared or private). To setup the QC tool, execute the follwing shell command: `dbm-cli setup qc` .

Available **options** are the following:

1. Reset setup to factory settings (to start from scratch).
2. Register one or more applications and their schemas.
3. Register additional schemas for one or more applications.
4. Unregister one or more applications (and their schemas).
5. Unregister one or more schemas of an application.
6. Check registered applications and their schemas.
7. Check system privileges required to operate.
8. Convert a shared configuration into multiple private configurations (one per application).
9. Convert the private configuration of an application into a shared configuration.
10. Copy a configuration from one application to another.

Remarks:

- To terminate a repetitive step or exit the setup, enter a dot (“.”) + RETURN.
- Names of schemas are validated online (must exist and be accessible).
- When registering the first application, you will be asked whether your configuration will be shared by all applications (yes/no). Enter “yes” for a shared configuration or “no” for private configurations.
- A **warning** is displayed for critical operations (e.g., reset or convert):

  ```
  WARNING: This is an irreversible operation: are you sure (yes/no)?
  Enter “yes” to continue or “no” to abort.
  ```

  “y” and “n” are valid answers equivalent to “yes” and “no.”

3.3. Parameters

The following parameters will be requested (once for all applications in a shared setup, once per application in a private setup):

Variable	Description	Mandatory?
TAB_TS	Tablespace for table data	N
IDX_TS	Tablespace for indexes	N
LOB_TS	Tablespace for large objects	N
EMAIL_RECIPIENTS	Recipients of anomalies report	N
EMAIL_CC	Carbon Copy of anomalies report	N
EMAIL_BCC	Blind Carbon Copy of anomalies report	N
AUDIT_INSERTED_BY	Audit column name (who created?)	N

Variable	Description	Mandatory?
AUDIT_INSERTED_ON	Audit column name (when created?)	N
AUDIT_UPDATED_BY	Audit column name (who updated?)	N
AUDIT_UPDATED_ON	Audit column name (when updated?)	N

Tablespaces are used in QC013, and audit columns are used in QC016. These parameters can be modified later in the `QC_DICTIONARY_ENTRIES` table.

Parameters (tablespaces, email addresses, and audit column names) are requested for each application (in the case of private configurations) or once for all applications (in the case of a shared configuration).

By default, patterns and dictionary entries are those recommended by the DBCC and cannot be changed interactively via the setup (it would be too tedious). To customize them, it is advised to create a script that calls the provided APIs (see section 7.6). When managed as code, the configuration can be placed under version control and easily reapplied when needed.

4. Operations

4.1. Naming Conventions

The tool can check the naming conventions of various database objects, including primary objects like tables, views, and stored procedures. It also evaluates secondary objects, which cannot exist independently of a primary object, such as table columns, constraints, indexes, and more. Additionally, identifiers within PL/SQL code can be assessed.

Naming conventions are defined using extended regular expressions that can incorporate predefined keywords. These keywords are replaced with their corresponding values before the regular expression is evaluated. Some keywords apply universally to any object type, while others are specific to certain object types.

4.2. Keywords

The following predefined keywords are replaced with their corresponding single values:

Keyword	Description
{app alias}	Application alias (configured during setup) - applicable to any object type
{object}	Name of the object being checked
{table}	Table name
{table entity}	Name of the entity corresponding to a table
{table alias}	Table alias - applicable to tables and their secondary object types (columns, constraints, indexes, etc.)
{parent table}	Name of the table referenced by a foreign key (FK) - applicable to FK and their indexes
{parent entity}	Name of the entity corresponding to a parent table
{parent alias}	Alias of the parent table
{parent cons}	Name of the primary key (PK) or unique key (UK) referenced by a FK - applicable to FK and their indexes
{tab column}	Name of the table column - applicable only to not null constraints
{tab col alias}	Alias of the table column - applicable only to not null constraints, used for long column names that result in a constraint name length ≥ 30
{ind column}	Name of the index column - applicable only to indexes made up of a single column
{cons column}	Name of the constraint column - applicable only to constraints based on a single column
{cons role}	Role qualifying a FK constraint, derived from column names (when two FKs exist between the same set of tables) - applicable to FK constraints only
{ind cons}	Index constraint: name of the constraint whose columns match those of the index - applicable to indexes only
{triggering event}	Triggering event (e.g., insert/update/delete) - applicable only to triggers

Keyword	Description
{trigger type}	Trigger type (e.g., before/after row/statement) - applicable only to triggers
{argument in out}	In out argument
{seq no}	Sequence number used to generate unique names when fixing anomalies - applicable to all object types

Note: The "name" qualifier is omitted from the keyword name (e.g., `{table name}` becomes `{table}`).

The following predefined keywords match any value defined in the specified dictionary:

Keyword	Description
{any <object type>}	Matches any object of the given type (e.g., table, constraint)
{any <pk|uk|fk> column}	Matches any column of any constraint of the given type (in the context of a given table)
{any <dict name>}	Matches any value defined in the named custom dictionary (e.g., "table alias", "plural")

Note: To avoid performance issues, refrain from using two `{any \<...\>}` keywords in the same pattern.

The tool includes the predefined dictionaries above, but additional ones can be defined. For example, large applications are often decomposed into modules, and abbreviations can be used in table names. These modules should be defined in the “module” dictionary and referenced in the matching patterns using the following keyword:

Keyword	Description
{any module}	Matches any value defined in the “module” dictionary

4.2.1. Special Syntax

Any keyword name can be prefixed and/or suffixed with an underscore (e.g., `{_keyword}`, `{keyword_}`, or `{_keyword_}`) to indicate that an underscore should be generated before and/or after the value when its value is not null.

For example, the following check pattern for table names will work whether the application has an alias or not: `{app_alias_}([A-Z]|[0-9]|'_')+`.

4.3. Patterns

The following patterns are defined for each object type:

- **Check pattern:** Specifies how objects should be named according to the established standards.
- **Anti-pattern:** Specifies patterns that objects should not match.
- **Include pattern:** Specifies which objects are within the scope of the quality checks.
- **Exclude pattern:** Specifies which objects are outside the scope of the quality checks.
- **Fix pattern:** Specifies how to generate object names when correcting naming anomalies.

As a rule, the patterns associated with an object type are always applied to the names of the objects. For example, the pattern for the TABLE object type is applied to table names. Some patterns may also apply to other attributes, as indicated by the name of the object type itself. For instance, the patterns associated with the TABLE ALIAS object type are applied to table aliases rather than table names.

4.4. Dictionaries

The tool includes several predefined dictionaries used for various purposes. Each dictionary consists of key-value pairs. The predefined dictionaries are as follows:

- **PARAMETER:** Holds various parameters for the tool (e.g., parameters for email notifications).
- **TABLE ALIAS:** Created when extracting table aliases (see Extraction of Aliases).
- **TABLE ENTITY:** Created when extracting entities from tables (see Extraction of Entities).
- **PLURAL WORD:** A list of non-trivial plural words, i.e., those not ending with an "s" (e.g., information, staff, data).
- **TRIGGER TYPE:** A list of trigger types with their abbreviations (e.g., BEFORE/AFTER EACH ROW trigger is abbreviated as BR/AR).
- **TRIGGERING EVENT:** A list of triggering events with their abbreviations (e.g., INSERT/UPDATE/DELETE trigger is abbreviated as I/U/D).
- **AUDIT COLUMN:** A list of audit columns that must exist in every table (e.g., date and author of creation and last update).
- **TABLE COLUMN ALIAS:** Used to replace long column names with shorter ones to avoid exceeding 30 characters when generating not null constraint names; key format is `<table>.<column>` and the corresponding value is `<tab col alias>` (not prefixed with the table name). This dictionary is completely project-dependent and is therefore empty by default.

Custom dictionaries can be added as necessary. For example, if your table names include a second prefix that represents the short name of a module (assuming the application comprises multiple modules), a custom dictionary could be created to hold the list of modules. This custom dictionary can then be referenced in the naming pattern using the `{any module}` keyword.

4.5. Extraction of Aliases

Table aliases can be extracted from existing database objects, provided that certain naming conventions have been followed. The tool includes a default rule to extract table aliases based on the following conventions:

- **Primary key:** `^(<table_alias>)_PK$`
- **Unique key:** `^(<table_alias>)_UK[1-9]*$`

Table aliases are extracted using the `\1` meta-character, which references the first matching subexpression (the one enclosed in parentheses).

If constraint names are prefixed with an application alias, the following conventions can also be applied:

- **Primary key:** `^(<app_alias>)_(<table_alias>)_PK$`
- **Unique key:** `^(<app_alias>)_(<table_alias>)_UK[1-9]*$`

In this scenario, the pattern for extracting table aliases would be `\2` to reference the second subexpression.

4.6. Extraction of Entities

Entity names can be extracted from existing table names, assuming that specific naming conventions have been adhered to. The tool provides a default rule for extracting entity names based on the following convention:

- **Table name:** `^<app_alias>(.+)$`

Entity names are extracted using the `\1` meta-character, which references the first matching subexpression (the one enclosed in parentheses). By default, entity names are derived from table names by removing the application alias prefix.

4.7. Incremental Reporting

Execution reports are incremental, meaning they can highlight the differences between two consecutive runs. Each report consists of three distinct parts:

1. **New Anomalies:** Anomalies detected since the last run.
2. **Resolved Anomalies:** Anomalies that were present in the previous run but have been resolved.
3. **Backlog:** A record of other anomalies that were detected previously and remain unresolved.

4.8. Email Notifications

Execution reports can be sent via email to a designated list of recipients, with the option to include CC (Carbon Copy) or BCC (Blind Carbon Copy) recipients. The recipient types (TO, CC, BCC) can be specified in the **PARAMETER** dictionary. Additionally, the subject line and sender information of the emails can be customized. Execution reports are only sent when there are differences detected since the last run.

4.9. Message Types

The tool generates three types of messages: error, warning, and information messages.

- **Error Messages:** Raised for rules that are universally applicable (e.g., table names must adhere to naming standards).
- **Warning Messages:** Raised for rules that allow exceptions (e.g., each table must be linked to at least one other table; an exception could be a calendar table that holds a single record for each calendar day).
- **Information Messages:** These provide additional context or insights related to the execution process.

This structure improves readability and conveys the information clearly.

4.10. Fixing Anomalies

4.10.1. Fixing Database Object Anomalies

Some anomalies reported by the tool can be fixed automatically (refer to the individual descriptions in the **List of Quality Checks**). The following actions can be taken on database objects:

- **Rename:** Change the name to comply with naming conventions.
- **Drop:** Remove unnecessary objects, such as unused sequences.
- **Create:** Generate missing objects, like foreign keys or indexes.
- **Enable:** Activate disabled objects.
- **Compile:** Compile invalid objects.

Before executing the tool, it is advisable to review the proposed fixes by examining the following columns in **qc_run_msgs**:

- **fix_op:** Indicates the fixing operation (e.g., rename, drop, create, enable, compile).
- **fix_type:** Specifies the type of object being addressed.
- **fix_name:** Represents the name of the object to be created or the new name of the object to be renamed.

You can modify the proposed name and/or operation. To ensure that your changes are preserved in subsequent runs of the quality check, set the **fix_locked** column to **Y**.

The following columns will be updated as a result of the fixing operation:

- **fix_ddl:** The exact Data Definition Language (DDL) statement used to fix the anomaly.
- **fix_status:** Displays **SUCCESS** or **FAILURE** depending on the outcome of the operation.
- **fix_msg:** Contains the error message generated during the operation (in case of FAILURE).
- **fix_time:** Records the date and time when the anomaly was fixed.

It is recommended to check the status of the operation. The most common reasons for failure include:

- **Name length exceeds 30 characters:** Shorten the name.
- **Object is currently locked:** Try again later.

Anomalies can be fixed by calling the **qc_utility.krn.fix_anomalies** procedure. If you prefer to target specific anomalies (which is often advisable rather than fixing all anomalies at once), you can specify the following parameters (wildcards `%` and `_` are allowed):

- **qc_code:** Quality check code.
- **object_type:** Type of object.
- **object_name:** Name of the object.
- **fix_op:** Fixing operation (e.g., rename, drop, create, enable, compile).
- **msg_type:** Type of message (e.g., error, warning).

Be aware that fixing an anomaly may lead to the generation of new anomalies during subsequent quality checks. For example, creating a missing foreign key may result in a "foreign key without index" anomaly. Once an anomaly has been successfully fixed, it will be reported as resolved in the next run of the quality checks.

4.10.2. Fixing PL/SQL Identifiers Anomalies

Support for checking and fixing the names of PL/SQL identifiers was added in version 1.2. The code analysis utilizes the **PLSCOPE** feature of Oracle, which, when enabled, collects information about PL/SQL identifiers (e.g., where they are declared and referenced) during compilation. This information, accessible via the **user_identifiers** system view, is subsequently used by the QC utility to detect and fix naming non-compliances.

It is advisable to enable **PLSCOPE** while developing PL/SQL code. You can enable or disable it for your session by invoking:

- **qc_utility_krn.enable_plscope** to enable PLSCOPE.
- **qc_utility_krn.disable_plscope** to disable PLSCOPE.

The procedure for fixing PL/SQL identifier anomalies is similar to that for database objects (described in the previous section). However, there are two key reasons why the results of fix operations cannot be logged in the **qc_run_msgs** table:

1. **Scope of Renaming:** Entries in **qc_run_msgs** correspond to declarations of poorly named identifiers, whereas the renaming operation applies not only to declarations but also to references. Some database objects not listed in **qc_run_msgs** may contain references to poorly named variables that could also be changed and recompiled.
2. **Optimized Recompilation:** The recompilation process is optimized to change and recompile each database object only once, allowing multiple naming anomalies to be fixed in a single operation.

Before fixing anomalies, please review the **fix_name** proposed in **qc_run_msgs**. The proposed name may be incorrect if the usage has not been accurately determined. For example, if the **CONSTANT** keyword has been omitted, a variable named "k_xxx" may be reported as poorly named because it has been considered a normal variable instead of a constant. In such cases, simply adding the missing **CONSTANT** keyword is preferable to renaming the variable.

Anomalies can be fixed by calling the **qc_utility.krn.fix_plsql_anomalies** procedure. Changed and recompiled database objects are reported via the **log_utility**, whose output is by default sent to **dbms_output**. If you wish to persist this information in the **log_output** table, first set a context via **log_utility.set_context** (the context is a unique identifier for retrieving your logs) and specify a log mask containing "I" (indicating information). To

receive details on which identifier references have been changed, in which database object, and at which line and column, activate the debugging mode by including "D" (indicating debug) in the log mask.

Note that **PLSCOPE** is automatically disabled before the fix operation and re-enabled afterward. This prevents any recompilation during the fix process from modifying **user_identifiers**, which should remain unchanged until the operation is complete (to avoid the risk of a moving target).

The order of recompilation for database objects considers dependencies (i.e., references). Database objects that do not depend on any others are changed and recompiled first.

Any error detected during the recompilation of code after the substitution of identifiers will halt the fixing operation, potentially leaving your schema in an inconsistent state. If your code is not under version control, it is recommended to back it up before starting the fix operation, allowing you to restore the backup in case of failure.

4.11. Impact Analysis

Before fixing anomalies, the tool can perform an impact analysis of the objects that need to be renamed or dropped. To accomplish this, it searches the **user_dependencies** and **user_source** Oracle data dictionary views. The following columns in **qc_run_msgs** are updated during this process:

- **dep_ref_cnt:** Number of references found in **user_dependencies**.
- **usr_ref_cnt:** Number of references found in **user_source**.

4.12. Database Changes

The tool can detect, track, and report on database changes, both online and via email (refer to QC000 for more details).

4.12.1. Online Report

To obtain a report of changes made to database objects between two specific dates, you can execute the following SQL query:

```
SELECT *
  FROM TABLE(qc_utility_krn.get_db_changes(
               TO_DATE('16/08/2019 00:00:00','DD/MM/YYYY HH24:MI:SS'),
               TO_DATE('27/08/2019 23:59:59','DD/MM/YYYY HH24:MI:SS')
           ));
```

The **get_db_changes** pipelined function accepts the following four optional parameters:

- **Start date/time**
- **End date/time**
- **Start run ID**
- **End run ID**

You can specify either a date/time range or a run ID range. If no start parameter is provided, the function considers database changes from the beginning of time. If no end parameter is specified, it will include database changes up to the current moment.

4.12.2. Email Report

A report can also be sent via email using the following invocation:

```
EXEC qc_utility_krn.send_db_changes(
        TO_DATE('16/08/2019 00:00:00','DD/MM/YYYY HH24:MI:SS'),
        TO_DATE('27/08/2019 23:59:59','DD/MM/YYYY HH24:MI:SS'));
```

The **send_db_changes** procedure accepts the same four parameters as the **get_db_changes** function. The recipients (TO, CC, and BCC) are specified in the **PARAMETER** dictionary.

5. Appendices

5.1. Predefined Patterns

Matching patterns are defined for the following categories:

- **Native Oracle Object Types**

- **Object Types Qualified by Their Properties** (e.g., index type, constraint type, object comment, object alias, etc.)
- **Non-Oracle Object Types** (any non-database object type for which patterns must be defined, e.g., Quality Check)

Below are the default check patterns provided:

OBJECT_TYPE	CHECK_PATTERN
CONSTRAINT: CHECK	^{app alias_}{table alias}_([A-Z]
CONSTRAINT: FOREIGN KEY	^{app alias_}{table alias}*{parent alias}*([A-Z]
CONSTRAINT: NOT NULL	^{app alias_}{table alias_}{tab column}_NN$
CONSTRAINT: PRIMARY KEY	^{app alias_}{table alias}_PK$
CONSTRAINT: UNIQUE KEY	^{app alias_}{table alias}_UK[0-9]*$
FUNCTION	^{app alias_}([A-Z]
INDEX: FOREIGN KEY	^{app alias_}{ind cons}(_I)?$
INDEX: NON UNIQUE	^{app alias_}([A-Z]
INDEX: PRIMARY KEY	^{app alias_}{ind cons}(_I)?$
INDEX: UNIQUE KEY	^{app alias_}{ind cons}(_I)?$
MATERIALIZED VIEW	^{app alias_}([A-Z]
PACKAGE	^{app alias_}([A-Z]
PACKAGE BODY	^{app alias_}([A-Z]
PROCEDURE	^{app alias_}([A-Z]
QUALITY CHECK	^QC[0-9]{3}$
SEQUENCE	^{app alias_}{any table alias}_SEQ$
TABLE	^{app alias_}([A-Z]
TABLE ALIAS	[1]([A-Z]
TRIGGER	^{app alias_}{table alias}_{triggering event}{trigger_type}_TRG$
VIEW	^{app alias_}([A-Z]

You can add any missing native database object types, provided they appear in the `USER_OBJECTS` data dictionary view. Additionally, you can adapt the provided patterns to suit your needs and naming conventions.

5.1.1. Default Patterns and Observations

Regarding the default patterns, note the following:

- All object names (both primary and secondary) are prefixed with the application alias to ensure uniqueness within a database schema.
- Table aliases consist of a minimum of 2 and a maximum of 4 characters, with an ideal length of 3 characters.
- Table names are plural, meaning they end with an "S" or another defined plural word from the relevant dictionary.
- Trigger names include abbreviations for their type and event, as defined in their respective dictionaries.
- Quality check codes are formatted as `QC001` to `QC999`.
- Comments can be bilingual by choice, though they may also be monolingual.
- Constraint names vary depending on their type (e.g., primary key, unique key, foreign key, check constraints).
- Index names vary based on their type (e.g., pk, uk, fk, non-unique).

5.1.2. Default Fix Patterns

The following fix patterns are provided by default to generate names according to these naming conventions:

OBJECT_TYPE	FIX_PATTERN
CONSTRAINT: CHECK	{app alias_}{table alias}_CK{seq nr}
CONSTRAINT: FOREIGN KEY	{app alias_}{child alias}_{parent alias}{_cons role}_FK
CONSTRAINT: NOT NULL	{app alias_}{table alias_}{tab column}_NN

OBJECT_TYPE	FIX_PATTERN
CONSTRAINT: PRIMARY KEY	{app alias_}{table alias}_PK
CONSTRAINT: UNIQUE KEY	{app alias_}{table alias}_UK{seq nr}
INDEX: FOREIGN KEY	{ind cons}_I
INDEX: NON UNIQUE	{app alias_}{table alias}_I{seq nr}
INDEX: PRIMARY KEY	{ind cons}_I
INDEX: UNIQUE KEY	{ind cons}_I
TRIGGER	{app alias_}{table alias}_{triggering event}{trigger_type}_TRG

5.1.3. Additional Observations

- Not all object types can have their names fixed automatically.
- A sequence number (`{seq nr}`) is used for certain object types to ensure unique names.
- The constraint role (`{cons role}`) helps to generate unique foreign keys when multiple foreign keys exist between the same pair of tables.
- Index constraints (`{ind cons}`) are used to derive index names from their corresponding constraint names.
- Trigger names depend on their type and event.

5.1.4. PL/SQL Identifier Patterns

Here are the patterns for PL/SQL identifiers, aligned with the naming conventions recommended by the Database Engineering Centre of Excellence (DBCC):

OBJECT_TYPE	CHECK_PATTERN	FIX_PATTERN
IDENTIFIER: GLOBAL CONSTANT OBJECT	^gko_.*$	gko_{name}
IDENTIFIER: GLOBAL CONSTANT RECORD	^gkr_.*$	gkr_{name}
IDENTIFIER: GLOBAL CONSTANT SCALAR	^gk_.*$	gk_{name}
IDENTIFIER: GLOBAL CONSTANT TABLE	^gk[ta]_.*$	gkt_{name}
IDENTIFIER: GLOBAL CURSOR	^gv?c_.*$	gc_{name}
IDENTIFIER: GLOBAL EXCEPTION	^gv?e_.*$	ge_{name}
IDENTIFIER: GLOBAL OBJECT	^gv?o_.*$	go_{name}
IDENTIFIER: GLOBAL RECORD	^gv?r_.*$	gr_{name}
IDENTIFIER: GLOBAL RECORD TYPE	^gr_.*_type$	gr_{name}_type
IDENTIFIER: GLOBAL SCALAR	^gv?_.*$	g_{name}
IDENTIFIER: GLOBAL TABLE	^gv?[ta]_.*$	gt_{name}
IDENTIFIER: GLOBAL TABLE TYPE	^g[ta]_.*_type$	gt_{name}_type
IDENTIFIER: IN OBJECT PARAMETER	^pi?o_.*$	po_{name}
IDENTIFIER: IN OUT OBJECT PARAMETER	^pioo_.*$	pioo_{name}
IDENTIFIER: IN OUT RECORD PARAMETER	^pior_.*$	pior_{name}
IDENTIFIER: IN OUT SCALAR PARAMETER	^pio_.*$	pio_{name}
IDENTIFIER: IN OUT TABLE PARAMETER	^pio[ta]_.*$	piot_{name}
IDENTIFIER: IN RECORD PARAMETER	^pi?r_.*$	pr_{name}
IDENTIFIER: IN SCALAR PARAMETER	^pi?_.*$	p_{name}
IDENTIFIER: IN TABLE PARAMETER	^pi?[ta]_.*$	pt_{name}
IDENTIFIER: LOCAL CONSTANT OBJECT	^l?ko_.*$	ko_{name}
IDENTIFIER: LOCAL CONSTANT RECORD	^l?kr_.*$	kr_{name}
IDENTIFIER: LOCAL CONSTANT SCALAR	^l?k_.*$	k_{name}

OBJECT_TYPE	CHECK_PATTERN	FIX_PATTERN
IDENTIFIER: LOCAL CONSTANT TABLE	^l?k[ta]_.*$	kt_{name}
IDENTIFIER: LOCAL CURSOR	^l?v?c_.*$	c_{name}
IDENTIFIER: LOCAL EXCEPTION	^l?v?e_.*$	e_{name}
IDENTIFIER: LOCAL OBJECT	^l?v?o_.*$	o_{name}
IDENTIFIER: LOCAL RECORD	^l?v?r_.*$	r_{name}
IDENTIFIER: LOCAL RECORD TYPE	^l?r_.*_type$	r_{name}_type
IDENTIFIER: LOCAL SCALAR	^l?v?_.*$	l_{name}
IDENTIFIER: LOCAL TABLE	^l?v?[ta]_.*$	t_{name}
IDENTIFIER: LOCAL TABLE TYPE	^l?[ta]_.*_type$	t_{name}_type
IDENTIFIER: OUT OBJECT PARAMETER	^poo_.*$	poo_{name}
IDENTIFIER: OUT RECORD PARAMETER	^por_.*$	por_{name}
IDENTIFIER: OUT SCALAR PARAMETER	^po_.*$	po_{name}
IDENTIFIER: OUT TABLE PARAMETER	^po[ta]_.*$	pot_{name}

Where `{name}` is the identifier name with any prefix or suffix removed.

5.2. SYSPER patterns

Here is an example of patterns for the SYSPER project.

5.2.1. Check Patterns for SYSPER

OBJECT_TYPE	CHECK_PATTERN
CONSTRAINT: PRIMARY KEY	^{table alias}_PK$
CONSTRAINT: UNIQUE KEY	^{table alias}_UK[0-9]*$
CONSTRAINT: FOREIGN KEY	^{table alias}*{parent alias}*([A-Z]
CONSTRAINT: CHECK	^{table alias}_([A-Z]
FUNCTION	^{app alias_}([A-Z]
INDEX: PRIMARY KEY	^{ind cons}(_I)?$
INDEX: UNIQUE KEY	^{ind cons}(_I)?$
INDEX: FOREIGN KEY	^{ind cons}(_I)?$
INDEX: UNIQUE	^{table alias}_(UK[0-9]*
INDEX: NON UNIQUE	^([A-Z]
MATERIALIZED VIEW	^{app alias_}([A-Z]
MVIEW COMMENT	^ENG:./ *FRA*:.$
PACKAGE	^{app alias_}([A-Z]
PACKAGE BODY	^{app alias_}([A-Z]
PROCEDURE	^{app alias_}([A-Z]
QUALITY CHECK	^QC[0-9]{3}$
SEQUENCE	^{app alias_}{any table alias}_SEQ$
TABLE	^{app alias_}([A-Z]
TABLE ALIAS	[2]([A-Z]
TABLE COLUMN COMMENT	^ENG:./ *FRA*:.$

OBJECT_TYPE	CHECK_PATTERN
TABLE COMMENT	^ENG:./ *FRA*:.$
TRIGGER	{table alias}
VIEW	^{app alias}(_

5.2.2. Fix Patterns for SYSPER

OBJECT_TYPE	FIX_PATTERN
CONSTRAINT: PRIMARY KEY	{table alias}_PK
CONSTRAINT: UNIQUE KEY	{table alias}_UK{seq nr}
CONSTRAINT: FOREIGN KEY	{child alias}_{parent alias}{_cons role}_FK
CONSTRAINT: CHECK	{table alias}_CK{seq nr}
INDEX: PRIMARY KEY	{ind cons}_I
INDEX: UNIQUE KEY	{ind cons}_I
INDEX: FOREIGN KEY	{ind cons}_I
INDEX: UNIQUE	{table alias}_UK{seq nr}_I
INDEX: NON UNIQUE	{table alias}_I{seq nr}
TRIGGER	{table alias}_{triggering event}{trigger_type}_TRG

5.3. Regular Expressions

5.3.1. Supported Meta-characters

Syntax	Description
.	Matches any character
*	Matches one or more occurrences of the preceding subexpression
?	Matches zero or one occurrence of the preceding subexpression
*	Matches zero or more occurrences of the preceding subexpression
{m}	Matches exactly m occurrences of the preceding subexpression
{m,}	Matches at least m occurrences of the preceding subexpression
{m,n}	Matches at least m, but not more than n occurrences of the preceding subexpression
[……]	Matches any character in list ……
[^ ……]	Matches any character not in list ……
(……)	Treat expression …… as a unit. The subexpression can be a string of literals or a complex expression containing operators.
\n	Matches the nth preceding subexpression, where n is an integer from 1 to 9.
\	Treat the subsequent meta-character in the expression as a literal.
^	Match the subsequent expression only when it occurs at the beginning of a line.
$	Match the preceding expression only when it occurs at the end of a line.
[:class:]	Match any character belonging to the specified character class. Can be used inside any list expression.
[.element.]	Specifies a collating sequence to use in the regular expression. The element you use must be a defined collating sequence, in the current locale.
[=character=]	Match characters having the same base character as the character you specify.

5.4. Predefined Dictionary Entries

5.4.1. Dictionary Entries

<table>
<tr><th>Name</th><th>Key</th><th>V</th></tr>
<tr><td>PARAMETER</td><td>APP ALIAS</td><td><&&app_alias></td></tr>
<tr><td>PARAMETER</td><td>TAB TS</td><td><&&tab_ts></td></tr>
<tr><td>PARAMETER</td><td>IDX TS</td><td><&&idx_ts></td></tr>
<tr><td>PARAMETER</td><td>EMAIL SENDER</td><td>AUTOMATED D
CHECK automa
notifications@n</td></tr>
<tr><td>PARAMETER</td><td>EMAIL RECIPIENTS</td><td><&&email_recip</td></tr>
<tr><td>PARAMETER</td><td>EMAIL CC</td><td><&&email_cc></td></tr>
<tr><td>PARAMETER</td><td>EMAIL BCC</td><td><&&email_bcc></td></tr>
<tr><td>PARAMETER</td><td>EMAIL SERVER</td><td>localhost</td></tr>
<tr><td>TABLE ALIAS</td><td colspan="2">
<table>
<tr><td>PLURAL WORD</td><td>data</td><td>data</td></tr>
<tr><td>PLURAL WORD</td><td>information</td><td>information</td></tr>
<tr><td>AUDIT COLUMN</td><td>INSERTED BY</td><td><&&inserted_by></td></tr>
<tr><td>AUDIT COLUMN</td><td>INSERTED ON</td><td><&&inserted_on></td></tr>
<tr><td>AUDIT COLUMN</td><td>UPDATED BY</td><td><&&updated_by></td></tr>
<tr><td>AUDIT COLUMN</td><td>UPDATED ON</td><td><&&updated_on></td></tr>
<tr><td>RESERVED WORD</td><td><keywords from v$reserved_words></td><td></td></tr>
<tr><td>OBJECT OWNER</td><td></td><td></td></tr>
</table>
Values prefixed with <code>&&</code> are prompted during installation. Reserved words are those that cannot be used as identifiers (from the <code>v$reserved_words</code> view).
</td></tr>
</table>

5.5. List of Quality Checks

5.5.1. QC000: Inventory of Objects and Statistics (Internal Use Only)

This internal QC makes an inventory of all database objects and computes some statistics per object type. Include and exclude patterns are considered to determine which objects must be included in or excluded from this inventory. This inventory is a versioned subset of the data dictionary views used to report on database changes made between two dates or two runs. "Versioned" means that all changes are tracked like in a source control system.

The information stored locally includes:

- Object name and object type.

Name	Key	V

- For object types with source code (package, package body, procedure, function, view, materialized view, trigger, check constraint, type, type body): a checksum computed on all source code lines (changes can be detected but not precisely identified).
- For primary key (PK), unique key (UK), foreign key (FK) constraints and indexes: their list of columns.
- For table/view/mview columns: their type, precision/scale, and optionality.

The tool can detect the creation and suppression of any database object, as well as any change to the properties or source code listed above. Statistics per object type can be used to compute the error rate of other quality checks.

5.5.2. QC001: Each Table Must Have an Alias

Table aliases are crucial as they are used in various naming conventions and SQL statements. Therefore, table aliases are mandatory. Aliases cannot easily be derived from table names but can be extracted from constraint names when naming conventions are followed. This tool permits the extraction of table aliases from the names of primary and/or unique keys based on the given extract pattern. As no exceptions are allowed, an error is raised when this rule is violated.

5.5.3. QC002: Table Aliases Must Be Unique

Since table aliases are used to define constraint names and these must be unique within a schema, table aliases must also be unique. As no exceptions are allowed, an error is raised when this rule is violated.

5.5.4. QC003: Not Used

This quality check is no longer used.

5.5.5. QC004: Each Table Must Have a Primary Key

According to relational theory, each row of a table must be unique. Therefore, a primary key must be defined in each table to ensure the uniqueness of its records. Primary or unique keys are necessary to create foreign keys and also support the creation of unique indexes. Using technical keys (e.g., based on sequences) instead of business keys allows changes to business keys without needing to update foreign keys in child tables. As no exceptions are allowed, an error is raised when this rule is violated.

5.5.6. QC005: Each Table Should Have at Least One Foreign Key to Another Table

This rule reinforces data model quality. Tables without any foreign keys to other tables are exceptional and therefore suspicious. This rule allows for the detection of temporary tables created for backup reasons, which should be excluded from quality checks. Exceptions are allowed (typically for configuration/parameter tables), so a simple warning is raised when this rule is violated.

5.5.7. QC006: Not Used

This quality check is no longer used.

5.6. List of Quality Checks

Here is the list of provided quality checks:

5.6.1. QC007: Each table/column/mview must have a comment

It is a best practice to put a comment on all database objects where it can be defined. Therefore, each table, column, view, and materialized view must have a comment. As no exception is allowed, an error is raised when this rule is violated.

5.6.2. QC008: Object names must match standard pattern

All database objects (tables, views, materialized views, constraints, triggers, indexes, sequences, procedures, functions, packages, package bodies, etc.) must have a name (and an alias when applicable) that follows naming conventions. Having clear naming conventions makes the data model easier to understand. As no exception is allowed, an error is raised when this rule is violated. This check is NOT

Name	Key	V
	performed on tables and their secondary object types that do not have an alias defined. Anomalies reported by this tool can be fixed automatically, i.e., objects can be renamed according to the standard (using the so-called fix pattern).	

5.6.3. QC009: Potentially missing foreign keys

A column that has exactly the same name as the primary column of another table might indicate a missing foreign key (e.g., XXX_ID in table A and XXX_ID in table B where XXX is the alias of table A or B). This check is not performed on composite primary keys (i.e., keys made up of more than one column). A warning message is raised when a candidate is found. Anomalies reported by this quality check can be fixed automatically, i.e., missing foreign keys can be created (using the FK fix pattern).

5.6.4. QC010: Foreign key columns must be indexed

For performance reasons, each foreign key must have a corresponding index. Otherwise, some operations like joins and deletion of master records might lead to unacceptable performance. More precisely, there must exist an index such that the foreign key columns are a prefix of the index columns (e.g., if FK has A and B columns, an index on A, B, C is acceptable). As no exception is allowed, an error is raised when this rule is violated. This check is not performed on tables that do not have an alias defined. Anomalies reported by this quality check can be fixed automatically, i.e., missing indexes can be created (using the index fix pattern).

5.6.5. QC011: Disabled database objects

All database objects like constraints and triggers should be enabled. A warning message is raised when this is not the case. Anomalies reported by this quality check can be fixed automatically by enabling the disabled objects.

5.6.6. QC012: Invalid database objects

All database objects like procedures, functions, packages, package bodies, and views should be valid. A warning message is raised when this is not the case. Anomalies reported by this quality check can be fixed automatically by recompiling invalid objects.

5.6.7. QC013: Tables and indexes must be stored in their respective tablespace

Tables and indexes must be stored in the tablespace identified respectively by TAB TS and IDX TS parameters. An error message is raised when this is not the case. Anomalies cannot be fixed automatically for the time being as objects must be recreated.

5.6.8. QC014: Package body/spec should not contain global variables

Packages and package bodies that contain global variables or cursors cannot be hot deployed (i.e., without restarting the application server). Modifying such a package would make existing sessions fall into the following error: ORA-04068: existing state of packages has been discarded. It is therefore recommended to isolate global variables and cursors into specific packages. These specific packages must be excluded from the check by specifying an exclude pattern on the QC014 object type. A warning message is raised when a faulty package is found. Anomalies detected by this quality check cannot be fixed automatically.

5.6.9. QC015: Potentially redundant indexes

An index that is a prefix of another index is potentially useless and consumes storage space for nothing. For example, index I1 on T(c1) is a prefix of index I2 on T(c1,c2) and should therefore normally be dropped, except when there are many values of c2 for a given value of c1 (in which case it might be more performant to keep it). A warning is raised for indexes that are considered redundant. Anomalies reported by this quality check can be fixed automatically, i.e., redundant indexes can be dropped or disabled.

5.6.10. QC016: Missing audit columns

Every table should have at least four audit columns indicating who created/updated each record and when. These columns should be populated by the application or set via database triggers (preferred). The names of these four audit columns are configured during installation and stored in the AUDIT COLUMN dictionary.

Name	Key	V

Additional audit columns can be specified, when necessary, by adding entries in this dictionary. Anomalies reported by this quality check can be automatically fixed by creating missing audit columns. The format of these columns must be specified via domain patterns.

5.6.11. QC017: Incorrect data type/length/optionality for domain-based columns

The QC tool allows defining domains (as standard data type, data length, and optionality) and to which columns they apply via naming patterns. For example, you might want to check that all columns holding a CODE (i.e., a string) or an ID (i.e., a number) have their name ending with “_CD” or “_ID”. A domain “XYZ” is defined by creating a pattern with the name “DOMAIN: XYZ.” Its include and exclude patterns define to which columns it applies while its check pattern defines its data type, data length, and optionality (e.g., VARCHAR2(30) NOT NULL). An error is raised when a column to which a domain pattern applies does not have the correct data type, data length, and/or optionality. Anomalies reported by this quality check can be fixed automatically by modifying the data type, data length, and/or optionality defined in the fix pattern.

5.6.12. QC018: Foreign key and referenced columns must have the same data type and length

Foreign key columns must have the same data type and length as the primary or unique key columns that they reference. Anomalies detected by this quality check can be fixed automatically by modifying the foreign key columns to ensure they have the same data type and length as the columns they reference.

5.6.13. QC019: PL/SQL identifiers must match standard naming patterns

Names of all PL/SQL identifiers must match standard naming patterns. All identifiers in PL/SQL source code (package, package body, procedure, function, trigger, type, and type body) must be named according to the naming conventions. No exception is allowed, so an error is raised when this rule is violated. Reported anomalies can be subsequently fixed automatically, i.e., identifier declarations and references can be renamed according to your configured standards. Be aware that the execution of this specific procedure is by default excluded from the Production environment. A warning message is raised during the installation steps of the QC Utility as a notification. If needed to force the usage of QC019 from Production, it is mandatory to update the QC_PATTERNS table related to Rule QC019 before launching the Quality Check Package. In that case, the action needed is to empty the column exclude_pattern from the qc_patterns table by filtering on the record with Object_Type = ‘QC019’.

5.6.14. QC020: Object names must not match anti-patterns

All database objects (tables, views, materialized views, constraints, triggers, indexes, sequences, procedures, functions, packages, package bodies, etc.) must have a name (and an alias when applicable) that follows naming conventions and do not match defined anti-patterns (if any). For example, you might want to check that column names are not prefixed with the alias of the table they belong to. As no exception is allowed, an error is raised when this rule is violated. This check is NOT performed on tables and their secondary object types that do not have an alias defined. Anomalies reported by this quality check cannot be fixed automatically.

5.6.15. QC021: Duplicate primary/unique key constraints

The set of columns of a primary or unique key constraint should not be equal or a superset of the set of columns of another primary or unique key constraint. Refer to “elementary key normal form” on Wikipedia. For example, if PK(A) is unique, then UK(A,B) is also unique, whatever column B is, but it is not elementary; similarly, if UK(A,B) is unique, UK(B,A) is also unique but completely redundant; it’s likely that UK(B,A) was created for performance reasons. Redundant constraints are created either by mistake, in which case they must be simply dropped, or for performance reasons, in which case they must be replaced with the same indexes created automatically by Oracle for primary/unique constraints. This explains why anomalies reported by this quality check cannot be fixed automatically.

5.6.16. QC022: Standalone procedures and functions are not allowed

Name	Key	V
	All database procedures and functions must be part of a database package. As no exception is allowed, an error is raised when this rule is violated. The automatic fix consists of deleting them.	

5.7. APIs

5.7.1. General Principle

- Use change scripts that call provided APIs to adapt patterns and parameters stored in the data dictionary.
- Avoid direct modifications to internal tables to facilitate easy reinstallation without losing configurations.

5.7.2. Changing Patterns

You can configure patterns for various object types using the following procedures from the `qc_utility_krn` package:

1. **insert_pattern()**: Inserts a new object type and its patterns.
2. **update_pattern()**: Updates patterns of an object type (supports wildcards).
3. **upsert_pattern()**: Updates patterns if the object type exists; creates it otherwise.
4. **delete_pattern()**: Deletes an object type (supports wildcards).

Parameters:

- `p_app_alias` : Mandatory; use ‘ALL’ for all applications with shared patterns.
- `p_object_type` : Mandatory; supports SQL wildcards (`*` , `_`).
- `p_check_pattern` : Optional.
- `p_anti_pattern` : Optional.
- `p_include_pattern` : Optional.
- `p_exclude_pattern` : Optional.
- `p_fix_pattern` : Optional.
- `p_msg_type` : Optional.

Special Values:

- `NULL` : Property remains unchanged (equivalent to not specifying the parameter).
- `'NULL'` (as a string): Resets the property to NULL.

Note: Optional parameters do not apply to `delete_pattern()` .

5.7.3. Changing Dictionary Entries (Parameters)

Dictionary entries can be managed using the following procedures from the `qc_utility_krn` package:

1. **insert_dictionary_entry()**: Inserts a new entry into the dictionary.
2. **update_dictionary_entry()**: Updates an existing entry (supports wildcards).
3. **upsert_dictionary_entry()**: Updates the entry if it exists; inserts it otherwise.
4. **delete_dictionary_entry()**: Deletes a dictionary entry (supports wildcards).

Parameters:

- `p_app_alias` : Mandatory; use ‘ALL’ for shared parameters.
- `p_dict_name` : Mandatory; supports SQL wildcards.
- `p_dict_key` : Mandatory; supports SQL wildcard.
- `p_dict_value` : Optional.
- `p_comments` : Optional.

Special Values:

- `NULL` : Property remains unchanged (equivalent to not specifying the parameter).
- `'NULL'` (as a string): Resets the property to NULL.

Note: Optional parameters do not apply to `delete_dictionary_entry()` .

Name	Key	V
	1. A-Z ↩ 2. A-Z ↩	