関係者外秘

PJ関係者限り

サンプルプロジェクト

SQLコーディング規約

第１．０版

2024年9月30日

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| No. | 版数 | 変更日 | 区分 | 変更箇所 | | 変更内容 | 担当者 |
| 頁 | 項番 |
| 1 | 1.0版 | 2018/8/24 | 新規 | - | - | (新規作成) | TIS |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

[1． 本規約について 1](#_Toc522525821)

[2． 可読性観点の記述 2](#_Toc522525822)

[2.1. 大文字、小文字の記述ルール　★ 2](#_Toc522525823)

[2.1.1. Nablarchアプリケーションフレームワークで拡張された機能を使用する場合の例外 2](#_Toc522525824)

[2.2. SQL文のフォーマット　★ 3](#_Toc522525825)

[2.2.1. SQLファイルの構成 4](#_Toc522525826)

[2.3. テーブル名、カラム名の別名について　★ 6](#_Toc522525827)

[2.4. WHERE句のカラム名の記述方法　★ 6](#_Toc522525828)

[2.5. 複合索引列を条件式で使用する場合の条件式の記述順　★ 7](#_Toc522525829)

[2.6. 表(TABLE)結合の記述位置　★ 7](#_Toc522525830)

[3． 性能観点の規約 8](#_Toc522525831)

[3.1. SQL関数、演算子を使用する際の注意点 8](#_Toc522525832)

[3.2. SELECT句のカラムリスト 9](#_Toc522525833)

[3.3. テーブルの全レコードを削除する方法 10](#_Toc522525834)

[3.4. フラグの判定方法　★ 10](#_Toc522525835)

[3.5. not in や否定形(!=、<>)は使用しない　★ 10](#_Toc522525836)

[3.6. IS NULLの使用　★ 10](#_Toc522525837)

[3.7. 部分一致、後方一致検索をできるだけ使用しない 11](#_Toc522525838)

[3.8. 暗黙の型変換　★ 12](#_Toc522525839)

[3.9. BETWEENの使用 12](#_Toc522525840)

[3.10. UNION ALLの使用　★ 12](#_Toc522525841)

[3.11. 同じ列に対する不等号条件をORで結んでいる場合、UNIONの使用を検討する 13](#_Toc522525842)

[3.12. テーブルを結合する際の注意事項 13](#_Toc522525843)

[3.13. ヒント句(インデックスヒント、結合ヒント)は使用しない　★ 13](#_Toc522525844)

[4． 保守、運用性観点の規約 14](#_Toc522525845)

[4.1. ロジックの分散を防ぐ 14](#_Toc522525846)

[4.2. INSERT文におけるカラム指定　★ 15](#_Toc522525847)

[4.3. レコード追加時の重複チェック　★ 15](#_Toc522525848)

[4.4. LIKE検索時のエスケープ処理　★ 15](#_Toc522525849)

[4.5. テーブルアクセス順序の厳守　★ 15](#_Toc522525850)

[4.6. TRUNCATEの注意事項 15](#_Toc522525851)

[4.7. 条件値や更新値に指定する値が固定の場合 16](#_Toc522525852)

[4.8. $if構文の注意点 ★ 17](#_Toc522525853)

[5． 注意事項 18](#_Toc522525854)

# 本規約について

本ドキュメントは、アプリケーションプログラマがSQL文をコーディングするに際に準拠しなければならない標準ルールについて記述をしている。

本ドキュメントに記載したルールのうち、必ず守るべきものについては、タイトルの文末に「★」マークをつけている。

「★」マークがついていないルールは、SQLをコーディングする際に知っておくべき心得や注意事項である。内容をよく読み、ルールの適用可否を慎重に判断すること。

※本ドキュメントは、下記の前提をもとに標準ルールを定義している。

* SQL文をJavaコードとは分離し外部ファイルに記述することを前提としている。
* SQLを記述する際の規約と注意点を記載する。索引の定義の仕方など設計に関する内容は記述しない。

※本文中の記述しているSQL文の例は、文章の見易さのために改行など[SQL文のフォーマット](#_SQL文のフォーマット)に従わない表記をすることがある。

# 可読性観点の記述

-- ifは、小文字とする。

-- プロパティ名は、対応するオブジェクトのフィールド名に合わせる。

SELECT

c1

FROM

user\_mst

WHERE

--"if"は小文字。プロパティ名(userName)はフィールド名に合わせる

$if(userName) {user\_name LIKE :userName%}

AND $if(userKanaName) {user\_kana\_name LIKE :userKanaName%}

SELECT col1, col2 FROM table WHERE col1 = ? ORDER BY col2

## 大文字、小文字の記述ルール　★

予約語はすべて大文字、カラム名、テーブル名はすべて小文字で記述する。

* 良い例

### Nablarchアプリケーションフレームワークで拡張された機能を使用する場合の例外

Nablarchアプリケーションフレームワークで拡張された下記機能を使用する場合は、Nablarchアプリケーションフレームワークの仕様として、小文字を使用しなければならないケースがある。

* LIKE検索を簡易的に実装出来る機能
* 条件が可変のSQL文を組み立てる機能

以下がその一例である。

## SQL文のフォーマット　★

SQL文は、セキュリティの観点から外部ファイルに記述(Javaのソースコードとは別のファイルに記述)する。

以下は、SQL文を外部ファイルに記述する際のルールである。

* SQL\_ID～「=」までは、単独行に記述する。
* 行頭から記述する。ただし、予約語以外は予約語に対して4スペース字下げする。

※行頭から記述する予約語は、SQL文を意味のある単位(下記表を参照)で区切った場合に、それぞれの単位の先頭に記述する予約語のみとする。

それ以外の予約語は、4スペース字下げを行ない記述すること。

≪意味のある単位の例≫

|  |  |  |
| --- | --- | --- |
| No | 単位 | 先頭に記述される予約語の例 |
| 1 | 取得項目リスト | SELECT |
| 2 | 対象テーブル | FROM  UPDATE  INSERT INTO  DELETE FROM  INNER JOIN  LEFT OUTER JOIN |
| 3 | 条件 | WHERE  ON |
| 4 | グループ化 | GROUP BY |
| 5 | ソート | ORDER BY |
| 6 | 集合演算子 | UNION(ALL)  MINUS |

* 基本的に1カラム、1テーブル、1条件ごとに改行する。
* カンマ(,)は文末に記述する。
* コメントは、必要がある場合のみ行末に記述する。
* 副問い合わせは、予約語に対して4スペース字下げして記述する。その他は上記に従う。

### SQLファイルの構成

SQLファイルの構成は以下のとおりとする。

* SQLファイルは、対応するJavaクラス単位に作成する。
  + SQLファイルは、Javaクラスのクラス名と同じパッケージ、同じファイル名とする。(拡張子は、「sql」)

|  |  |  |
| --- | --- | --- |
| データベース  アクセス機能 | 配置場所 | 例 |
| JDBCラッパー | データベースアクセスクラスと同じパッケージ | クラスが「nablarch.sample.ss11AC.W11AC01Action」のとき、SQLファイルは「nablarch/sample/ss11AC/W11AC01Action.sql」 |
| ユニバーサルDAO | 検索結果をマッピングするBeanと同じパッケージ | クラスが「nablarch/sample/entity/User」のとき、SQLファイルは「nablarch/sample/entity/User.sql」 |

* SQL\_IDは、SQLファイル単位で一意とし、意味のある名前(そのSQL文の意図がわかる名前)をつけること。
  + 良い例

・SELECT\_USERS

・INSERT\_USER

・UPDATE\_USER\_ADDRESS

・UPDATE\_USER\_ATTRIBUTES

* + 悪い例

意味のないIDを付与してはならない。

ID\_01

SQL\_ID001

* SQLファイルのエンコーディングは、Javaファイルのエンコーディングと合わせること。
* SQL文とSQL文の間には空行を1行入れること。

以下に例を示す

user\_id,

use\_case\_id,

insert\_user\_id,

insert\_date,

updated\_user\_id,

updated\_date

)

VALUES

(

:userId,

:useCaseId,

:insertUserId,

:insertDate,

:updatedUserId,

:updatedDate

)

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

-- 副問い合わせの例

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

SELECT\_LATEST\_USER =

SELECT

users.kanji\_name,

users.kana\_name,

users.mail\_address,

users.extension\_number\_building,

users.extension\_number\_personal

FROM

users

INNER JOIN

system\_account

ON

users.user\_id = system\_account.user\_id

WHERE

system\_account.effective\_date\_to = (SELECT

MAX(effective\_date\_to)

FROM

system\_account

WHERE

login\_id = :loginId)

ORDER BY

users.user\_id

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

-- SELECT文の例

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

SELECT\_USERS =

SELECT

sa.login\_id,

usr.kanji\_name,

usr.kana\_name,

usr.mail\_address,

usr.extension\_number\_building,

usr.extension\_number\_personal,

ugrp.ugroup\_id,

ugrp.ugroup\_name

FROM

users usr

INNER JOIN

system\_account sa

ON

usr.user\_id = sa.user\_id

INNER JOIN

ugroup\_system\_account usa

ON

usr.user\_id = usa.user\_id

INNER JOIN

ugroup ugrp

ON

ugrp.ugroup\_id = usa.ugroup\_id

WHERE

$if (loginId) {sa.login\_id = :loginId}

AND $if (kanjiName) {usr.kanji\_name LIKE :kanjiName}

AND $if (kanaName) {usr.kana\_name LIKE :kanaName}

AND $if (ugroupId) {ugrp.ugroup\_id = :ugroupId}

ORDER BY

sa.login\_id

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

-- INSERT文の例

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

INSERT\_SYSTEM\_ACCOUNT\_AUTHORITY =

INSERT INTO

system\_account\_authority

(

login\_id,

## テーブル名、カラム名の別名について　★

--元のテーブル名がわからない別名はつけない

SELECT t1.login\_id, t2.kanji\_name FROM system\_account t1 INNER JOIN users t2 ON t1.user\_id = t2.user\_id

--ASは使用しない

SELECT sa.login\_id, usr.kanji\_name FROM system\_account AS sa, INNER JOIN users AS usr ON sa.user\_id = usr.user\_id

SELECT sa.login\_id, usr.kanji\_name FROM system\_account sa INNER JOIN users usr ON sa.user\_id = usr.user\_id

テーブルの別名は、結合するために必要な場合に付ける。別名を付けた場合は、どのテーブルのカラムを取得しているのかわからなくなるため、全てのカラムに対して別名を付加する。

カラムの別名は、そのSQL内に重複するカラム名が存在する場合に、そのカラムにのみ付ける。

別名は、元のテーブル名やカラム名の意味を推測できるものとする。また、別名を記述する際、キーワード”AS”は使用しない。

* 良い例

SELECT c1 FROM t1 WHERE '20101126' BETWEEN effective\_date\_from AND effective\_date\_to

SELECT c3 FROM t1 WHERE c1 = 'ABC'

SELECT c3 FROM t1 WHERE 'ABC' = c1

* 悪い例

## WHERE句のカラム名の記述方法　★

カラム名と外部変数の比較を行う条件式は、左辺にカラム名を記述すること。 ただし、BETWEENを使用する場合は、右辺に記述しても良い。

* 良い例
* 良い例(BETWEENを使用する場合)
* 悪い例

外部変数が左辺に記述してあるため規約違反

## 複合索引列を条件式で使用する場合の条件式の記述順　★

SELECT c3 FROM t1 WHERE c2 = 'ABC' AND c1 = 1

SELECT c3 FROM t1 WHERE c1 = 1 AND c2 = 'ABC'

条件式に複合索引列を使用する場合は、索引の定義順に記述する。

SELECT t1.c1, t2.c2 FROM t1, t2 WHERE t1.c1 = t2.c1

SELECT t1.c1, t2.c2 FROM t1 INNER JOIN t2 ON t1.c1 = t2.c1

例:以下の複合索引を条件式として使用する場合

CREATE INDEX t1\_index\_1 ON t1(c1, c2)

* 良い例
* 悪い例

索引が定義された順番どおりに、条件式に指定されていない(逆になっている)ため規約違反

## 表(TABLE)結合の記述位置　★

結合条件はFROM句に記述する。これは、外部結合と内部結合で記述がそろえられることと、結合条件か抽出条件かの意図を明確にするためである。

なお、JOIN句は省略形(LEFT JOINやRIGHT JOIN)で記述すること。（LEFT OUTER JOIN等の完全系の使用は禁止する。）

* 良い例
* 悪い例

結合条件がWHERE句に書かれているため規約違反

# 性能観点の規約

SELECT c1, c2 FROM t2 WHERE c1 / 5 = 10

SELECT c1, c2 FROM t1 WHERE TO\_CHAR(c1) = '001'

SELECT c1, c2 FROM t1 WHERE c1 = 1

SELECT c1, c2 FROM t2 WHERE c1 = 50

## SQL関数、演算子を使用する際の注意点

SQL関数や演算子を使用する場合は、性能劣化の原因となることがあるため注意して使用すること。特に、下記の場合には索引が使用されずに性能劣化の原因となるため注意すること。

* 索引列に関数を設定すると索引が使用されない
* 索引列に演算子を設定すると索引が使用されない

例:c1に索引が定義されている場合

* 良い例
* 悪い例

索引列に関数を設定すると索引が使用されない為、性能劣化の原因となる。

索引列に演算子を設定すると索引が使用されない為、性能劣化の原因となる。

SELECT \* FROM t1

SELECT c1, c2 FROM t1

## SELECT句のカラムリスト

全てのカラムを取得しないのであれば、『SELECT \*』は使用しない。『SELECT \*』を使用すると、性能劣化の原因となるケースがある。

ただし、テーブルの全カラムのデータを取得する場合やUniversalDao使用時に検索結果をEntityにマッピングしたい場合は、この限りではない。

* 良い例
* 悪い例

『SELECT \*』を使用することで、パフォーマンス改善のために作成したカバリングインデックスが有効活用出来なくなる。

## テーブルの全レコードを削除する方法

SELECT c2 FROM t1 WHERE c1 <> ‘001’

SELECT c2 FROM t1 WHERE c1 NOT IN ('001', '002')

SELECT c2 FROM t1 WHERE c1 = ‘001’

テーブルの全てのレコードを削除する場合は、DELETE文ではなくTRUNCATE文を使用すること。ただし、TRUNCATEには運用上の注意事項([TRUNCATEの注意事項](#_TRUNCATEの注意事項)を参照)があるので、使用前によく検討すること。

## フラグの判定方法　★

削除フラグなど0か1を判定する場合、WHERE句の条件は、<>0, <>1ではなく、=1、=0とする。<>0, <>1では索引が使用されないため性能劣化の原因となる。

* 詳細は[not in や否定形(!=、<>)は使用しない](#_not_in_や否定形(!=、<>)は使用しない)を参照すること。

## not in や否定形(!=、<>)は使用しない　★

『not in』や否定形は索引が使用されないので、極力使用しないこと。

仕様上否定形を使用する必要がある場合には、索引が定義されている項目で十分にレコードが絞れるようにすること。

* 良い例
* 悪い例

## IS NULLの使用　★

IS NULLの使用はできるだけ避ける。NULLは索引に存在しないので、IS NULLで検索すると索引が使用されないため。

仕様上IS NULLを使用する必要がある場合には、索引が定義されている項目で十分にレコードが絞れるようにすること。

SELECT c1 FROM t1 WHERE c2 LIKE :cond%

SELECT c1 FROM t1 WHERE c2 LIKE :%cond%

SELECT c1 FROM t1 WHERE c2 LIKE :%cond

## 部分一致、後方一致検索をできるだけ使用しない

部分一致、後方一致は索引を使用しないため、極力使用しないこと。

* 悪い例
* 良い例

## 暗黙の型変換　★

SELECT c1 FROM t1 WHERE c2 = 5000

SELECT c1 FROM t1 WHERE c2 = '5000'

パフォーマンスの劣化を防ぐため、文字列型(CHARやVARCHARなど)のカラムで、暗黙の型変換が発生しないようにすること。また対象が索引列の場合、関数を持つことになり、索引が使用されない。

例:c2がVARCHAR型の場合

* 良い例
* 悪い例

c2が文字列型のため、数値型への暗黙の型変換が起こり、パフォーマンスが劣化する。

また、c2が索引列であっても、型変換で関数が使用されるため、索引が使用されない

## BETWEENの使用

可能な限りBETWEENを使用すること。BETWEENは評価が1回で済むので、andを2回書くよりもパフォーマンスが良い。

## UNION ALLの使用　★

UNIONではなく、UNION ALLを使用すること。UNIONを使用すると、重複を除去するソート処理が行われるため性能劣化の原因となる。

## 同じ列に対する不等号条件をORで結んでいる場合、UNIONの使用を検討する

SELECT c1 FROM t1 WHERE c2 > 60 OR c2 <= 50

SELECT c1 FROM t1 WHERE c2 > 60

UNION

SELECT c1 FROM t1 WHERE c2 <= 50

同じ列に対する不等号条件(等号条件ではない)をORで結んでいる場合、その列の索引は使用されないため、UNIONの使用を検討する(UNIONで結ばれている各SELECT文で索引が使用される)。ただし、性能劣化の原因にもなりうる(参照)ので、使用前によく検討すること。

例:c2が索引列の場合

* 良い例

各SELECT文の条件項目で索引が使用される。

* 悪い例

条件項目で索引が使用されず性能劣化の原因となる。

## テーブルを結合する際の注意事項

パフォーマンスの劣化を防ぐため、以下の点に注意する。

* 結合表数は最小限に抑える。
* 結合条件に使用する列に対し索引をつける。また、結合列が複数存在する場合は複合索引を定義する。

## ヒント句(インデックスヒント、結合ヒント)は使用しない　★

ヒント句を使用するとアクセスプランが固定化されてしまう。これにより、統計情報が最新化された場合でも、固定化されたアクセスプランでSQLが実行され、性能改善目的のヒント句が逆に性能劣化の原因となることがある。

このため、ヒント句の使用には細心の注意を払う必要があり、その場凌ぎの性能改善目的で使用すべきではない。よってヒント句の使用は禁止する。

# 保守、運用性観点の規約

SELECT c1, LENGTH(c1) FROM t1 WHERE c2 = ?

SELECT c1 FROM t1 WHERE c2 = ?

## ロジックの分散を防ぐ

ロジックがSQL文、プログラムコード双方に分散すると保守性が落ちる。これを防ぐため、SQLで実装する必要がないもの(プログラムコードで実現できるもの)はSQLではなく、プログラムコードで実装する。

SELECT c1 FROM t1 WHERE c2 = ?

* 文字数をカウントする場合
  + 良い例

SQL文では、対象のカラムのデータを取得し文字数のカウントはJavaで行う。

SELECT c1 \* 1.05 FROM t1 WHERE c2 = ?

* + 悪い例

文字数のカウントをSQL文で実現している。

* 計算を行う場合
  + 良い例

SQL文では、対象のカラムのデータを取得し計算はJavaで行う。

* + 悪い例

計算をSQL文で実現している。

## INSERT文におけるカラム指定　★

INSERT文を使用する場合、必ず明示的にカラム指定を行うこと。

テーブルに対してカラムの追加を行う際、ALTER文を使用すると、追加したカラムは最終カラムとして追加される。このため、CREATE TABLE文で作成した場合とカラム順序が異なるケースが想定される。いずれの場合でも正しくINSERT文が発行されるよう、必ず明示的にカラム指定を行うようにする。

## レコード追加時の重複チェック　★

レコード追加時のレコード重複チェックは、SELECT文を発行して重複レコードの存在をチェックしてからINSERT文を発行するのではなく、INSERT文のみを発行して、重複エラーを検出する。

## LIKE検索時のエスケープ処理　★

LIKE条件に設定する文字列のエスケープ処理を確実に行うため、LIKE検索は、Nablarchアプリケーションフレームワーク提供機能を使用すること。

## テーブルアクセス順序の厳守　★

デッドロックを回避するため、システム毎に定められたテーブルアクセス順序を厳守すること。

## TRUNCATEの注意事項

TRUNCATEは発行後にロールバックを行うことが出来ないので、バッチ処理にて使用する際は十分な検討を行うこと。また、障害時のリトライで再度TRUNCATEが発行されて問題がある場合は、TRUNCATEのみを別プロセスとして構築する必要がある。

※SQL Serverでは、TRUNCATE文もロールバック可能である。SQL Serverを使用する場合は、その特性をふまえて検討を行うこと。

## 条件値や更新値に指定する値が固定の場合

コード値や区分値等の固定値をSQLの条件や更新値(SET句やVALUE句)に使用する場合には、SQL文にリテラルで記述を行うこと。

なぜなら、バインド変数にしてしまうと、そのSQLの用途が複数あるように見えパフォーマンスチューニングやメンテナンスの妨げになるからである。

また、不要な値をバインド変数化することにより、ネットワーク経由で送信される値が増加し、パフォーマンスに影響を与えるためである。

* + 良い例

リテラルで設定されており、SQLの用途が明確になっている。（SQLの実行の度に値が変わる部分と、常に変わらない部分が明確でありメンテナンス性も高い）

-- ユーザIDに紐付くレコードを取得する。(論理削除されていないレコードが対象)

SELECT user\_id, user\_name FROM user\_info WHERE user\_id = :userId AND sakujo\_sgn = '0'

-- ユーザIDに紐付くレコードを論理削除する。

UPDATE user\_info SET sakujo\_sgn = '1' WHERE user\_id = :userId AND sakujo\_sgn = '0'

* + 悪い例

固定の値がバインド変数となっているため、SQLを実行しているJavaコードを見ないとSQL文の用途がわからないためメンテナンス性が低い。

-- ユーザIDに紐付くレコードを取得する。(論理削除されていないレコードが対象)

SELECT user\_id, user\_name FROM user\_info WHERE user\_id = :userId AND sakujo\_sgn = :sakujoSgn

-- ユーザIDに紐付くレコードを論理削除する。

UPDATE user\_info SET sakujo\_sgn = :updateSakujoSgn WHERE user\_id = :userId AND sakujo\_sgn = :sakujoSgn

## $if構文の注意点 ★

$if構文は、検索画面等で任意の検索条件がある場合に使用する機能なので、SQLを複数処理で共通化する目的で使用しないこと。

なぜなら、$ifを使用してむやみに共通化されたSQL文は、アクセスパスが想定できず索引作成やパフォーマンスチューニングの障害となるからである。

例えば、A処理では「user\_id」を使用して、B処理では「user\_name」を使用して検索処理を行う場合には、A処理とB処理用の2種類のSQLを作成する必要がある。

* + 良い例

処理毎にSQL文を用意しているので、アクセスパスがわかりやすくパフォーマンスチューニングやメンテナンスがしやすい。

-- A処理用

SELECT user\_id, user\_name FROM user\_info WHERE user\_id = :userId

-- B処理用

SELECT user\_id, user\_name FROM user\_info WHERE user\_name = :userName

* + 悪い例

異なる条件で実行されるSQL文を、$if構文を使用して共通化している。

SELECT

user\_id,

user\_name

FROM

user\_info

WHERE

**$if(userId) {user\_id = :userId} -- A処理で使用する条件**

**AND $if(userName) {user\_name = :userName} -- B処理で使用する条件**

# 注意事項

'APPLE' NOT IN ('LEMON', 'ORANGE', NULL)

これは次の式と等価となる。

NOT ('APPLE' = 'LEMON' OR 'APPLE' = 'ORANGE' OR 'APPLE' = NULL)

NULLとの論理演算の結果はUNDEFINEDとなるので

上記の各演算子を展開すると、NOT (FALSE OR FALSE OR UNDEFINED)

となる。

"FALSE OR UNDEFINED"はUNDEFINEDとなるので

さらにNOT UNDEFINEDと展開される。

UNDEFINEDに対する論理演算はFALSEとなるので

最終的にはNOT FALSEとなる。

よって、この例は、"'APPLE' NOT IN ('LEMON', 'ORANGE', NULL) = TRUE"となってしまう。

* NULLになる可能性があるカラムに対する論理演算は十分注意すること。

下記例のような事象があるため、十分注意すること。

例:

* 件数取得時(COUNT関数)の列指定について。

COUNT関数の引数には、「\*」または「列名」を指定すること。これ以外の値（例えば、「1」や「'X'」等のリテラル値）の指定は禁止する。

なお、「\*」と「列名」の使い分けは、下記のルールに従うこと。

* + 条件に紐付くレコード数を取得したい場合には、「\*」を指定する。
  + 条件に紐付くレコードの特定カラムの「NULL」以外のレコード数を取得したい場合は、そのカラムの「列名」を指定する。
  + 条件に紐付くレコードの特定カラムの種類数を取得したい場合は、そのカラムの「DISTINCT 列名」を指定すること。
* 集合演算を使用する場合。

UNION、MINUSなどの集合演算は、パフォーマンスが劣化する可能性があるため、使用する場合は十分な性能設計、検証を行うこと。

* 再帰SQLの使用

再帰SQLの使用は、再帰のレベルや対象の行数により、パフォーマンス劣化やリソース不足の原因となりうるため、使用前に十分な性能設計、検証を行うこと。