Confidential

Only for PJ employees

Sample Project

SQL Coding Conventions

Version No. 1.0

September 30, 2024

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| No. | Version No. | Revision date | Classification | Change locations | | Revision details | Contact person |
| Page | Item No. |
| 1 | Version 1.0 | 9/30/2024 | New | - | - | (New creation) | TIS |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

[1． Information on rules 1](#_Toc35793955)

[2． Description of readability viewpoint 2](#_Toc35793956)

[2.1. Description rules for upper and lowercase letters (!) 2](#_Toc35793957)

[2.1.1. Exceptions when using extended functions in Nablarch application framework 2](#_Toc35793958)

[2.2. Format of SQL statement(!) 4](#_Toc35793959)

[2.2.1. Structure of SQL file 5](#_Toc35793960)

[2.3. Aliases for table names and column names (!) 7](#_Toc35793961)

[2.4. Naming method of column name in WHERE clause (!) 7](#_Toc35793962)

[2.5. Description order of conditional expressions when complex index columns are used in conditional expressions (!) 9](#_Toc35793963)

[2.6. Table (TABLE) join description position 9](#_Toc35793964)

[3． Performance perspective conventions 10](#_Toc35793965)

[3.1. Precautions when using SQL functions and operators 10](#_Toc35793966)

[3.2. Column list of SELECT clause 11](#_Toc35793967)

[3.3. How to delete all records in a table 12](#_Toc35793968)

[3.4. Flag determination method (!) 12](#_Toc35793969)

[3.5. Do not use "not in" or negative forms (!=, <>)(!) 12](#_Toc35793970)

[3.6. Use of IS NULL (!) 12](#_Toc35793971)

[3.7. Do not use partial or backward match searches as much as possible 13](#_Toc35793972)

[3.8. Implicit type conversion (!) 14](#_Toc35793973)

[3.9. Use of BETWEEN 14](#_Toc35793974)

[3.10. Use of UNION ALL (!) 14](#_Toc35793975)

[3.11. Consider using UNION if the inequality condition for the same column is connected by OR 15](#_Toc35793976)

[3.12. Notes on joining tables 15](#_Toc35793977)

[3.13. Do not use hint phrases (index hints, join hints)　(!) 15](#_Toc35793978)

[4． Rules for maintenance and operability 17](#_Toc35793979)

[4.1. Prevent the distribution of logic 17](#_Toc35793980)

[4.2. Column specification in INSERT statement (!) 18](#_Toc35793981)

[4.3. Check for duplicates when adding records (!) 18](#_Toc35793982)

[4.4. Escape processing during LIKE search (!) 18](#_Toc35793983)

[4.5. Strict adherence to table access order (!) 18](#_Toc35793984)

[4.6. Notes on TRUNCATE 18](#_Toc35793985)

[4.7. When value specified for the condition value or update value is fixed 19](#_Toc35793986)

[4.8. Notes on $if syntax (!) 20](#_Toc35793987)

[5． Note 22](#_Toc35793988)

# Information on rules

This document describes the standard rules that application programmers must follow when coding SQL statements.

The rules which are marked with "(!)" at the end of the title in this document must be followed without fail.

Rules that are not marked with "(!)" are tips and precautions that you should know when coding in SQL. Read the contents carefully and prudently determine whether the rules apply.

\* The standard rules are defined in this document based on the following assumptions.

* It is assumed that SQL statements are separated from Java code and described in an external file.
* The rules and notes to write SQL statements are described. The design contents such as how to define the index are not described.

\* Examples of SQL statements described in the text may not follow [SQL statement format](#_SQL文のフォーマット) such as line breaks, etc., so that the text is easier to read.

# Description of readability viewpoint

## Description rules for upper and lowercase letters (!)

Reserved words are all uppercase, column names and table names are all lowercase.

* Good example

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

### Exceptions when using extended functions in Nablarch application framework

When using the following extended functions in the Nablarch application framework, there are cases where lowercase letters must be used as a specification of the Nablarch application framework.

* Function for the easy implementation of LIKE search
* Function to assemble SQL statement with variable conditions

An example is given below.

- if is lowercase.

- Match the property name to the field name of the corresponding object.

SELECT

c1

FROM

user\_mst

WHERE

-"if" is lowercase. Match property name (userName) to field name

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

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

ｓｓｓｓｓｓｓｓｓｓ

## Format of SQL statement(!)

The SQL statement is described in an external file from the security perspective (in a file separate from the Java source code).

The rules for writing SQL statements in external files are given below.

* Write from SQL\_ID to "=" in a single line.
* Write from the beginning of the line. Use an indent of 4 spaces for reserved words.

\* Reserved words to be written from the beginning of a line are only those which are written at the beginning of each unit when the SQL statement is divided into meaningful units (see the table below).

For other reserved words, write with an indent of 4 spaces.

<<Example of a meaningful unit≫

|  |  |  |
| --- | --- | --- |
| No | Unit | Example of reserved words written from the beginning |
| 1 | Acquisition item list | SELECT |
| 2 | Target table | FROM  UPDATE  INSERT INTO  DELETE FROM  INNER JOIN  LEFT OUTER JOIN |
| 3 | Conditions | WHERE  ON |
| 4 | Grouping | GROUP BY |
| 5 | Sort | ORDER BY |
| 6 | Set operator | UNION(ALL)  MINUS |

* Basically, use a new line for each column, table, and condition.
* Use comma (,) at the end of the sentence.
* Write comments at the end of the line only when necessary.
* Write subquery using an indent of 4 spaces with respect to the reserved word. Others are as described above.

### Structure of SQL file

The structure of the SQL file is as follows.

* SQL files are created for each corresponding Java class.
  + The same package and file names as the class name of the Java class are used for the SQL file. (Extension is "sql")

|  |  |  |
| --- | --- | --- |
| Database  Access function | Location | Example: |
| JDBC wrapper | Same package as database access class | When class is "nablarch.sample.ss11AC.W11AC01Action", SQL is "nablarch/sample/ss11AC/W11AC01Action.sql" |
| Universal DAO | Same package as Bean that maps search results | When class is "nablarch/sample/entity/User", SQL is "nablarch/sample/entity/User.sql" |

* SQL\_ID must be unique for each SQL statement and have a meaningful name (a name that indicates the intent of the SQL statement).
  + Good examples

・SELECT\_USERS

・INSERT\_USER

・UPDATE\_USER\_ADDRESS

・UPDATE\_USER\_ATTRIBUTES

* + Bad example

Do not assign meaningless IDs.

ID\_01

SQL\_ID001

* The encoding of the SQL file must match the encoding of the Java file.
* Insert one blank line between SQL statements.

An example is shown below.

user\_id,

use\_case\_id,

insert\_user\_id,

insert\_date,

updated\_user\_id,

updated\_date

)

VALUES

(

:userId,

:useCaseId,

:insertUserId,

:insertDate,

:updatedUserId,

:updatedDate

)

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

-- Example of subquery

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

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

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

-- Example of SELECT statement

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

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

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

-- Example of INSERT statement

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

INSERT\_SYSTEM\_ACCOUNT\_AUTHORITY =

INSERT INTO

system\_account\_authority

(

login\_id,

## Aliases for table names and column names (!)

Add table aliases when necessary to join. When using aliases, you will not know which table column you are fetching, so add an alias to every column.

A column alias is assigned only to the column when a duplicate column name exists in the SQL.

It should be possible to deduce the meaning of the original table or column name from the alias. Do not use the key word "AS" when describing an alias.

* Good example

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

* Bad example

- Do not add an alias whose original table name is not known

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

-- Do not use AS

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

## Naming method of column name in WHERE clause (!)

For a conditional expression that compares a column name with an external variable, describe the column name on the left side. However, when BETWEEN is used, it may be described on the right side.

* Good example

SELECT c3 FROM t1 WHERE c1 = 'ABC'

* Good example (when using BETWEEN)

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

* Bad example

Violation of convention because external variables are written on the left side

SELECT c3 FROM t1 WHERE 'ABC' = c1

## Description order of conditional expressions when complex index columns are used in conditional expressions (!)

When using a complex index column in a conditional expression, describe in the order of definition of the index.

Example: When the following complex index is used as a conditional expression

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

* Good example

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

* Bad example

The convention is violated because the index is not specified in the conditional expression according to the defined order (it has been reversed)

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

## Table (TABLE) join description position

The join condition is described in the FROM clause. This is to make sure that the description is the same for outer join and inner join, and to clarify the intention of the join condition or the extraction condition.

Note that the JOIN clause must be described in an abbreviated form (LEFT JOIN or RIGHT JOIN). (Use of complete systems such as LEFT OUTER JOIN is prohibited.)

* Good example

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

* Bad example

Violation of convention because join condition is written in WHERE clause

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

# Performance perspective conventions

## Precautions when using SQL functions and operators

Use SQL functions and operators with caution, as they may degrade the performance. In particular, note that the index may not be not used in the following cases and cause performance degradation.

* The index will not be used when a function is set for an index column
* The index will not be used when an operator is set for an index column

Example: An index is defined for c1

* Good example

SELECT c1, c2 FROM t1 WHERE c1 = 1

SELECT c1, c2 FROM t2 WHERE c1 = 50

* Bad example

The index is not used if a function is set for the index column, which causes performance degradation.

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

The index is not used if an operator is set for the index column, which causes performance degradation.

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

## Column list of SELECT clause

If you do not want to fetch all columns, do not use "SELECT \*". Using "SELECT \*" may cause performance degradation in some cases.

However, this is not applicable when fetching data of all columns of the table or if you want to map search results to Entity when using UniversalDao.

* Good example

SELECT c1, c2 FROM t1

* Bad example

By using "SELECT \*", the covering index created to improve performance cannot be used effectively.

SELECT \* FROM t1

## How to delete all records in a table

To delete all records in a table, use a TRUNCATE statement instead of a DELETE statement. Since TRUNCATE has operational precautions (see [TRUNCATE precautions](#_TRUNCATEの注意事項)), examine carefully before use.

## Flag determination method (!)

When determining 0 or 1 such as the deletion flag, the condition of the WHERE clause is = 1, = 0 instead of <> 0, <> 1. Index is not used with <> 0 and <> 1, which causes performance degradation.

\* For details refer to [Do not use "not in" or negative forms (!=, <>)](#_not_in_や否定形(!=、<>)は使用しない).

## Do not use "not in" or negative forms (!=, <>)(!)

Do not use "not in" or negative forms as much as possible because the index is not used.

If the specification requires the use of negative forms, ensure that the fields in which the index is defined are sufficiently narrowed down.

* Good example

SELECT c2 FROM t1 WHERE c1 = ‘001’

* Bad examples

SELECT c2 FROM t1 WHERE c1 <> ‘001’

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

## Use of IS NULL (!)

Avoid using IS NULL as much as possible. Because NULL does not exist in the index, searching with IS NULL does not use the index.

If the specification requires the use of IS NULL, ensure that the fields in which the index is defined are sufficiently narrowed down.

## Do not use partial or backward match searches as much as possible

Do not use partial or backward matches as much as possible because they do not use the index.

* Bad examples

SELECT c1 FROM t1 WHERE c2 LIKE :%cond%

SELECT c1 FROM t1 WHERE c2 LIKE :%cond

* Good example

SELECT c1 FROM t1 WHERE c2 LIKE :cond%

## Implicit type conversion (!)

To prevent performance degradation, make sure that implicit type conversions do not occur for string type columns (such as CHAR and VARCHAR). If the target is an index column, it will have a function, and the index will not be used.

Example: When c2 is of VARCHAR type

* Good example

SELECT c1 FROM t1 WHERE c2 = '5000'

* Bad example

Since c2 is a string type, an implicit type conversion to a numeric type occurs, which degrades the performance.

Even if c2 is an index column, the index is not used because the function is used for type conversion

SELECT c1 FROM t1 WHERE c2 = 5000

## Use of BETWEEN

Use BETWEEN whenever possible. BETWEEN requires only one evaluation, so it performs better than writing and twice.

## Use of UNION ALL (!)

Use UNION ALL, and not UNION. When UNION is used, a sort process is performed for eliminating duplication, which causes performance degradation.

## Consider using UNION if the inequality condition for the same column is connected by OR

If an inequality condition (not an equal sign condition) is connected by OR for the same column, the index for that column is not used, so consider using UNION. (Each SELECT statement linked by UNION uses an index.). However, it may cause performance degradation (refer), examine carefully before use.

Example: When c2 is an index column

* Good example

Index is used in the condition item of each SELECT statement.

SELECT c1 FROM t1 WHERE c2 > 60

UNION

SELECT c1 FROM t1 WHERE c2 <= 50

* Bad example

Index will not be used in the condition item, which causes performance degradation.

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

## Notes on joining tables

Note the following to prevent performance degradation:

* Keep the number of joined tables to a minimum.
* Index the columns used in the join condition. If there are multiple join columns, define a composite index

## Do not use hint phrases (index hints, join hints)　(!)

The access plan is fixed when you use the hint clause. As a result, even when the statistical information is updated, the SQL is executed with the fixed access plan, and the hint phrase used for the purpose of improving the performance may cause performance degradation.

For this reason, the use of hint phrases requires extreme caution and should not be used as a temporary measure for improving performance. Therefore, the use of hint phrases is prohibited.

# Rules for maintenance and operability

## Prevent the distribution of logic

If the logic is distributed across both SQL statements and program code, maintainability will decrease. To prevent this, those that do not need to be implemented in SQL (those that can be implemented in program code) are implemented in program code, not in SQL.

* If you want to count the number of characters
  + Good example

Data of the target column is fetched with the SQL statement and the number of characters is counted in Java

SELECT c1 FROM t1 WHERE c2 = ?

* + Bad example

Character count is being realized using SQL statement.

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

* When performing calculations
  + Good example

Data of the target column is fetched with the SQL statement and the calculation is performed in Java.

SELECT c1 FROM t1 WHERE c2 = ?

* + Bad example

Calculation is realized by SQL statement.

SELECT c1 \* 1.05 FROM t1 WHERE c2 = ?

## Column specification in INSERT statement (!)

When using the INSERT statement, be sure to explicitly specify the column.

If you use the ALTER statement when adding columns to the table, the columns will be added as the last column. Therefore, it is assumed that the column order is different from the case created in the CREATE TABLE statement. In any case, be sure to explicitly specify the column so that the INSERT statement is issued correctly.

## Check for duplicates when adding records (!)

In the record duplication check when adding a record, instead of issuing a SELECT statement to check for duplicate records and then issuing an INSERT statement, only the INSERT statement is issued to detect duplicate errors.

## Escape processing during LIKE search (!)

To reliably perform the escape processing of the character string set in the LIKE condition, LIKE search should use the function provided by Nablarch application framework.

## Strict adherence to table access order (!)

Strictly adhere to the table access order stipulated for each system to avoid deadlocks.

## Notes on TRUNCATE

Since TRUNCATE cannot be rolled back after issuance, carefully consider before using in batch processing. If TRUNCATE is issued again in the case of retry due to failure and there is a problem, then it is necessary to build TRUNCATE as a separate process.

\* In SQL Server, the TRUNCATE statement can also be rolled back. When using SQL Server, examine taking into account the server characteristics.

## When value specified for the condition value or update value is fixed

If fixed values such as code values and segment values are used for SQL conditions and update values (SET clause and VALUE clause), describe them literally in the SQL statements.

This is because if you make it a binding variable, the SQL will appear to have multiple uses and interfere with performance tuning and maintenance.

Further, by converting unnecessary values into binding variables, the values transmitted over the network increases, which affects performance.

* + Good example

It is set as a literal, and the purpose of SQL is clear. (The part where the value changes each time the SQL is executed and the part that does not always change are clear and maintainability is high.)

-- Fetch records linked to user ID. (records that have not been logically deleted)

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

-- Logically delete the record linked to the user ID.

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

* + Bad example

Since the fixed value is a binding variable, the purpose of the SQL statement cannot be understood without looking at the Java code that executes the SQL, and the maintainability is low.

-- Fetch records linked to user ID. (records that have not been logically deleted)

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

-- Logically delete the record linked to the user ID.

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

## Notes on $if syntax (!)

The $if syntax is a function that is used when there is any search condition on the search screen, etc., so it should not be used for the purpose of sharing SQL between multiple processes.

This is because if an SQL statement is shared unnecessarily using $if, then access paths cannot be assumed, which hinders index creation and performance tuning.

For example, to perform a search process using “user\_id” in the process A and “user\_name” in the process B, it is necessary to create two types of SQL for the process A and the process B.

* + Good example

- Since SQL statements are prepared for each process, the access path is easy to understand, and performance tuning and maintenance are easy.

-- Process A

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

-- Process B

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

* + Bad example

SQL statements executed under different conditions are shared using $if syntax.

SELECT

user\_id,

user\_name

FROM

user\_info

WHERE

**$if(userId) {user\_id = :userId} -- Condition used in process A**

**AND $if(userName) {user\_name = :userName} -- Condition used in process B**

# Note

* Be careful when using logical operations on columns that can be NULL.

Be careful because there are events such as the following example.

Example:

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

This is equivalent to the following expression:

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

Since the result of the logical operation with NULL is UNDEFINED

Expanding each of the above operators results in NOT (FALSE OR FALSE OR UNDEFINED)

"FALSE OR UNDEFINED" becomes UNDEFINED, and

it is expanded to NOT UNDEFINED.

Since the logical operation on UNDEFINED is FALSE,

eventually it becomes NOT FALSE.

Therefore, this example becomes "'APPLE' NOT IN ('LEMON', 'ORANGE', NULL) = TRUE".

* Column specification when acquiring the number of records (COUNT function).

Specify "\*" or "column name" as an argument of the COUNT function. Specifying any other value (for example, a literal value such as "1" or "'X'") is prohibited.

Note that the use of "\*" and "column name" should follow the rules below.

* + To fetch the number of records linked to the condition, specify "\*".
  + If you want to fetch the number of records other than "NULL" of the specific column of the record linked to the condition, specify the "column name" of that column.
  + If you want to fetch the number of types of specific column of the record linked to the condition, specify "DISTINCT column name" of that column.
* When using set operators.

Set operators such as UNION and MINUS may degrade performance, so when using them, ensure sufficient performance design and verification.

* Using recursive SQL

The use of recursive SQL may cause performance degradation or resource shortage depending on the level of recursion and the number of target rows. Therefore, perform sufficient performance design and validation before use.