Referring Null Values in Partitioned Tables

Martina Hrínová Durneková
University of Žilina
Žilina, Slovakia
Martina.Durnekova@fri.uniza.sk

Michal Kvet
University of Žilina
Žilina, Slovakia
Michal.Kvet@fri.uniza.sk

***Abstract*—Nowadays, working with data is very important in the decision-making process. A large amount of data is generated daily and needs to be stored efficiently. The database system makes it possible to divide tables into smaller, more manageable chunks, so-called partitions. Partitions are created based on the partition key column. If we insert records that do not contain any anomalies, there is no problem with inserting records into the partitioned table. The problem can arise if we are trying to insert records that contain null values of the partition key column. Therefore, the aim of this paper is to show how records with null values of the partition key columns behave when inserted into a partitioned table and to create methodology for managing partitions with records with null values. We focus on three types of partitioning: *Range*, *List* and *Hash* partitioning. As part of the experiments, we will show how it is possible to modify the definition of the created tables and partitions so, that the records with null value of the partition key column can also be inserted into the table.**

I. Introduction

Nowadays, a huge amount of data is generated every day. The data needs to be efficiently stored, processed and the results interpreted for further decisions.

As the amount of data stored in the database tables increases, the efficiency of the executed operations decreases, which can lead to an increase in the total cost of the executing queries. During creating queries, we try to reduce the total cost of queries execution as much as possible. There are several ways we can achieve this. Either by creating indexes, replications, or dividing the table into smaller parts, so-called partitions.

However, the obtained data might not acquire the required structure. In many cases, we come across that the data was incorrect, even some records were incomplete, or some data is empty, has no value. We say that such data acquire a null value [6].

Null values are discussed topic in general. In database tables, they can occur only in those columns whose integrity constraint is not set to *NOT NULL* or *PRIMARY KEY*. How to work with such columns to be sure that we ensure the correct interpretation of the records? In database systems, it is not possible to compare null values through the = assign, but through the condition *IS NULL* or *IS NOT NULL* [2]. In case we want to treat null values in some way and work with some value defined by instead of null, there are functions that can transform null values to a default value or value defined by us. These function include: *NVL(), Coalesce(), Decode()* [2].

These methods can be used in the case when the records are already stored in the database tables. But how do we insert records that contain null values into the table? Inserting records into ordinary table might not be a problem if the integrity constraints are respected. But in the case of partitioned tables, whose partitions are created based on a column that can be defined as null, it can already be a problem.

Therefor, the main aim of this paper is to create a methodology for managing partitions with records with null values and to show how null values behave during inserting records into partitioned table, how it is necessary to modify the definitions of partitioned tables so that it is possible to insert records with null values of the partition key columns.

The Oracle database environment will be used for the experiments in this paper. Oracle is a powerful and versatile database system that provides a wide range of features and benefits. It delivers high performance and scalability solutions, ensures high availability and reliability, and also offers a large set of data backup and recovery tools. Oracle Database additionally supports multiple partitioning methods including range, hash, and list partitioning. However, handling null values requires a special approach in each database system. MySQL and PostgreSQL database systems are not different in this regard. While both systems support partitioning, they each have several limitations. The MySQL database system does not allow creating indexes on columns that contain null values, which can potentially impact query performance [5]. Additionally, the use of foreign keys on partitioned tables is not supported, which can reduce data integrity. PostgreSQL and the Oracle database system has fewer restrictions on partitioned tables in comparison to the previously mentioned database system.

This paper contains several chapters. At the beginning, the database tables will be described in more detail. The theoretical part will then continue in the next chapter, which will be focused on partitioning. In this chapter we will discuss in more detail the different types of partitioning provided by the database system. The experimental part will be found in chapter four. We will show a methodology for modifying the definition of database tables for various types of partitioning. At the end, individual experiments will be summarized.

II. Database table

A database table is the basic unit of data storage in a database. Data in a database table is stored using rows and columns. Each row has to be identified by unique value [1].

Each database table has a precisely defined structure, therefore the data that needs to be stored must also meet these requirements [4], [7]. Fig. 1 shows the simplified syntax for creating a database table.

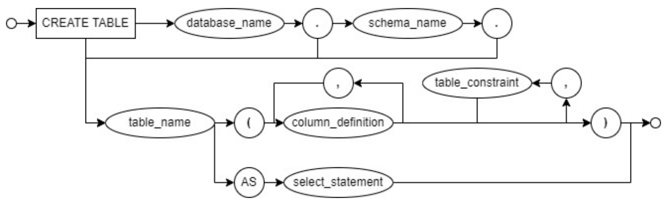

Fig. 1. Create table syntax

When creating a table, it is necessary to define the table name and columns, or it is also possible to create a table as a result of the Select statement. In that case, the created table will have the same structure as defined in the Select statement. Database and schema name is optional [7].

In the case of defining columns, it is necessary to define the column name and data type (Varchar, Date, Number, etc.). furthermore, it is possible to define rules for table columns, so-called integrity constraints. These integrity constraints include for example NOT NULL. It ensures whether the column value has to be filled or not [7], [10].

The database table can also contain virtual columns. Virtual columns differ in that their value is derived based on a defined expression. The expression represents, for example, constants, the value of other column of the same table, user-defined functions, or SQL functions [7].

The Oracle database system allows us to create different types of database tables, for example, an ordinary table that is created by default, an index-organized table, a partitioned table, an external table, etc [7]. In this paper, we are focusing on partitioned tables, which are described in more detail in the following section.

III. Partitioning

As the amount of data stored in database tables increases, data processing operations might become slower and less efficient. Therefore, the idea arose to divide the table into smaller, more manageable chunks so-called partitions [8], [12], thanks to which the easier maintenance of the database will be ensured and at the same time the performance and efficiency of the query execution will increase [9], [13].

Partitioning is a technique that allows database objects (tables and indexes) to be subdivided into smaller chunks called partitions. Each partition is defined by its name and might also have its own storage characteristics defined. A database object, which consists of several partitions, can be managed more easily, while these partitions can be managed separately or collectively [8]. Partitioning helps reduce the total cost of executing operations [12].

It is important to note, that accessing a partitioned table does not differ in any way from accessing a classic non-partitioned table. It means that a partitioned table is still only one table that can be accessed using DML operations in the same way as a non-partitioned table.

Partitioning key divides the database object into individual partitions. It can include one or set of columns. Using it, it is possible to determine in which partition each record will be included [3].

The graphical representation of a non-partitioned table and a partitioned table is shown in Fig. 2.

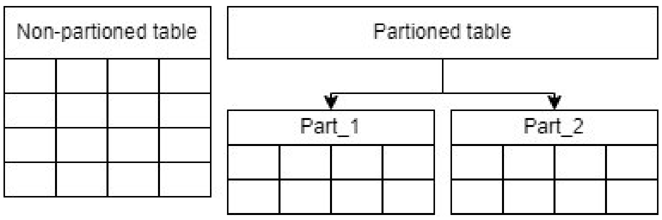

Fig. 2. Graphical representation of the non-partitioned table and partitioned table

There are many requirements for data processing. Oracle provides comprehensive partitioning solution, the so-called partitioning strategies. There are two ways to divide tables. Single (one-level) partitioning or composite (two level) partitioning [8]. Single partitioning is displayed in Fig 2. It means that table will be divided only into partitions. The second option (composite partitioning) ensures that each partition is divided to the subpartitions [11]. Fig. 3 shows graphical representation of the subpartitioned table.

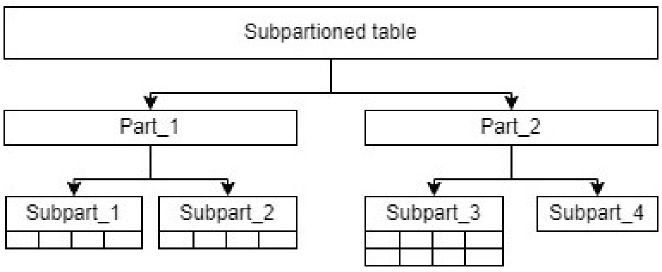

Fig. 3. Graphical representation of the subpartioned table

So, there are many ways to divide the table into partitions and subpartitions. It is also possible to create partitions directly in the definition or dynamically when inserting records into the table. In addition, partitioning has many extensions, increases flexibility, and it is also possible to increase performance by creating global and local indexes.

There are three basic methods of data distribution into partitions provided by Oracle partitioning – *Range* partitioning, *List* partitioning and *Hash* partitioning.

A. *Range partitioning*

Range partitioning ensures the data distribution into partitions based on the range of values of the partitioning key column/columns. The syntax of this type of partitioning is as follows:

```
CREATE TABLE [schema.]<table_name>(
  <table_definition>
)
PARTITION BY RANGE (<column> [, <column>, ...])
 [INTERVAL (<constant> | <expression>)]
 ...
(
    PARTITION [<partition_name>]
         VALUES LESS THAN (<value>[, <value>]...),
    [TABLESPACE <tablespace_name>]
  ...
)/
```

Range partitioning represents a continuous distribution without gaps [3]. Let's imagine that we want to divide the data into partitions according to years. When defining a partition, we always define the upper boundary of the partition. The lower boundary is automatically set to the value of the upper boundary of the previous partition. The boundaries have an increasing character. It means that the first partition covers all records with a lower value than defined. It is also possible to define the last partition with the highest partition boundary [8]. This partition boundary can be defined by *MAXVALUE* value [11], [12], [13]. Each boundary represents an open interval.

B. *List partitioning*

List partitioning is a type of partitioning in which the set of values in the range is not defined, but the set takes on the discrete values of the partition key column [11]. It means that each partition has precisely defined set of values. In the case, that some inserted records do not acquire any of these values, and therefore do not belong to any partition, it is possible to define a *DEFAULT* partition type, that caches all other records [8], [13].

The syntax of list partitioning is as follows:

```
CREATE TABLE [schema.]<table_name>(
  <table_definition>
)
PARTITION BY LIST (<column>) [AUTOMATIC]
...
(
   PARTITION [<partition_name>]
     VALUES (<value>[, <value>]...),
    [TABLESPACE <tablespace_name>]
  ...
)/
```

As can be seen in the syntax of list partitioning, using the key word *AUTOMATIC* is possible to create partitions automatically according to inserting records.

C. *Hash partitioning*

The difference between HASH partitioning and the above types of partitioning is that HASH partitioning has its own internal hash algorithm built in. This algorithm divides records into partitions based on a partition key column [3], [11]. The syntax of this type of partitioning is as follows:

```
CREATE TABLE [schema.]<table_name>(
  <table_definition>
)
PARTITION BY HASH (<column> [, <column>, ...])
 ...
 [PARTITIONS <num>] [STORE IN <tablespace_name>
                             [, <tablespace_name>, ...]]
(
   PARTITION [<partition_name>]
    [TABLESPACE <tablespace_name>]
  ...
)/
```

Hash partitioning is used when the range or list key is not obvious. This type of partitioning provides roughly equi-balanced sizes of the partitions. Therefore, it is difficult to expect association between partitions and data. With hash partitioning, it is possible to define the number of partitions to be created. However, this number must be a power of 2, so this attribute also greatly affects the partition size balance [8], [13].

In the following section, experiments are described in more detail. Using examples, we demonstrate individual types of partitioning in connection with null values.

IV. Experiments

The aim of the performed experiments was to find out how records with null values are stored in database tables divided into partitions through range, list, and hash partitioning. For each type of partitioning, we created separate database tables in which we inserted different types of records, while these records also contained null values of the columns according to which the tables were divided into partitions.

For these experiments, the Oracle Database environment was used, specifically version Oracle 19c.

A. *Range partitioning*

First, we focused on *RANGE* partitioning. In this section, we looked at two options for creating database tables divided into partitions. The first option was to define the partitions directly. It means that we defined as many partitions as we needed in the table definition. The second option was to create partitions dynamically. It means that in the table definition we defined only the initial partition and during inserting records, partitions were created as needed.

The following code shows the table definition with name *tab_partition_null*. This table contains three columns (identification number, record creation date, text) and partition definition. The table contains four partitions that are created for years based on the created date column. The first partition covers all records that were created up to 2021, the second partition covers only records from 2021, the third partition stored only records from 2022, and the last partition stores all records that were created from 2023 onwards.

```
CREATE TABLE tab_partition_null(
  record_id             NUMBER NOT NULL PRIMARY KEY,
  creation_date         DATE,
  text                  VARCHAR2(30))
PARTITION BY RANGE (creation_date)(
 PARTITION part_2020
   VALUES LESS THAN (TO_DATE('01.01.2021', 'DD.MM.YYYY')),
 PARTITION part_2021
   VALUES LESS THAN (TO_DATE('01.01.2022', 'DD.MM.YYYY')),
 PARTITION part_2022
   VALUES LESS THAN (TO_DATE('01.01.2023', 'DD.MM.YYYY')),
 PARTITION part_20XX
   VALUES LESS THAN (MAXVALUE))
/
```

After creating the database table with partitions, we inserted some records into the table. Two of these records contained a null value in the *creation_date* column. Inserted records are displayed in the following code.

```
INSERT INTO tab_partition_null
  VALUES(1, TO_DATE('21.12.2022', 'DD.MM.YYYY'), 'Text1');

INSERT INTO tab_partition_null
 VALUES(2, TO_DATE ('21.12.2021', 'DD.MM.YYYY'), 'Text2');

INSERT INTO tab_partition_null
  VALUES(3, NULL, 'Text3');
```

```
INSERT INTO tab_partition_null
  VALUES(4, TO_DATE ('21.10.2023', 'DD.MM.YYYY'), 'Text4');

INSERT INTO tab_partition_null
  VALUES(5, NULL, 'Text5');
```

All records were successfully inserted into the table *tab_partition_null*, even the records with null value of the *creation_date* column.

After running the following SELECT statement, which returns the number of records in each partition, we found that in the last partition (PART_20XX) are stored three records.

```
SELECT 'PART_2020', COUNT(*)
   FROM TAB_PARTITION_NULL PARTITION (PART_2020)
 UNION ALL
SELECT 'PART_2021', COUNT(*)
   FROM TAB_PARTITION_NULL PARTITION (PART_2021)
 UNION ALL
SELECT 'PART_2022', COUNT(*)
   FROM TAB_PARTITION_NULL PARTITION (PART_2022)
 UNION ALL
SELECT 'PART_20XX', COUNT(*)
   FROM TAB_PARTITION_NULL PARTITION (PART_20XX);
```

The result of previous SELECT statement is displayed in Table I.

TABLE I. NUMBER OF RECORDS IN THE *TAB_PARTITION_NULL* TABLE

PART_	COUNT(*)
PART_2020	0
PART_2021	1
PART_2022	1
PART_20XX	3

When we run SELECT statement for the last partition (PART_20XX) we see those records with null values in *creation_date* column is stored in this partition, because this partition covers all records that do not belong to the previous partitions.

The second option of creating partitions is using an interval. These partitions are created automatically. At the beginning, the first partition is determined. When inserting a record, the value of the record is taken and inserted into the appropriate partition. If this partition does not exist, it will be created dynamically. The following code shows definition of the database table together with definition of the partitions according to *creation_date* column using the interval function.

```
 CREATE TABLE tab_interval_partition_null(
   record_id              NUMBER NOT NULL PRIMARY KEY,
   creation_date          DATE,
   text                   VARCHAR2(30))
PARTITION BY RANGE (creation_date)
  INTERVAL (NUMTOYMINTERVAL(1, 'YEAR'))(
 PARTITION part_1
   VALUES LESS THAN (TO_DATE('01.01.2021', 'DD.MM.YYYY')))
/
```

After creating the table, we inserted the same records into it as in the previous example. Records with the existing value of the *creation_date* column were successfully inserted into the table. The appropriate partitions were created. The initial partition is created always, but the system automatically created additional partitions for records with other year of *creation_date* column.

The system failed to insert records with null value of the *creation_date* column and gave the following error (ORA-14300).

```
SQL Error: ORA-14300: partitioning key maps to a partition
outside maximum permitted number of partitions
```

The system could not assign the null value of the *creation_date* column to any partition, nor could it create a new partition, because null key values for interval partitioning are not supported by the system.

The solution to this problem with null values for the partition key column could be a virtual table column that will either take the value of the existing *creation_date* column or transform the null value to a value we have predefined. So, we extended the *tab_interval_partition_null* table with virtual column named *date_def*. The virtual column is automatically calculated based on whether or not the *creation_date* column is filled. If the *creation_date* column has null value, the predefined date 01.01.2999 will be set. It was also necessary to change the *creation_date* column to *date_def* column in the PARTITION BY RANGE section. The modified table definition is shown in the following example.

```
 CREATE TABLE tab_interval_partition_null(
   record_id      NUMBER NOT NULL PRIMARY KEY,
   creation_date  DATE,
   text           VARCHAR2(30),
   date_def       DATE GENERATED ALWAYS AS
         (COALESCE(creation_date,
            TO_DATE('01.01.2999','DD.MM.YYYY'))) VIRTUAL)
PARTITION BY RANGE (date_def)
   INTERVAL( NUMTOYMINTERVAL (1, 'YEAR'))(
 PARTITION part_1
   VALUES LESS THAN (TO_DATE('01.01.2021', 'DD.MM.YYYY')))
/
```

After re-creating the *tab_interval_partition_null* database table and inserting given records we received information that all records have been successfully inserted to the table.

The names, interval values and number of rows in partitions are accessed via the following statement.

```
SELECT partition_name, high_value, num_rows
 FROM   user_tab_partitions
 WHERE table_name = 'TAB_INTERVAL_PARTITION_NULL'
 ORDER BY partition_name;
```

The result of the SELECT statement is displayed in the Table II.

TABLE II. NAMES AND INTERVAL VALUES OF THE PARTITIONS

PARTITION NAME	INTERVAL VALUE	REC. NUMBER
PART_1	TO_DATE(' 2021-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS)	0
SYS_P34691	TO_DATE(' 2023-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	1
SYS_P34692	TO_DATE(' 2022-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	1
SYS_P34693	TO_DATE(' 3000-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	2
SYS_P34694	TO_DATE(' 2024-01-01 00:00:00', 'SYYYY-MM-DD HH24:MI:SS')	1

As can be seen in the Table II, initial partition defined by us was created and then there were four partitions created according to needed. Partitions which are created by system start with prefix SYS. We can see that in the partition

SYS_P34693 are two records. That records contain null value in *creation_date* column.

B. List partitioning

LIST partitioning is similar as a *RANGE* partitioning. Different is that list partitioning defines list of values which the partitions can acquire. In this section, we will demonstrate an example of the *tab_list_partition_null* database table partitioned using *LIST* partitioning. The effort is to find out how to store the records that contain null values of the column that defines the partitions.

The *tab_list_partition_null* database table is extended by another *state_code* attribute. Based on this attribute, four partitions will be created. The first partition contains only records that acquire the value SK or CZ of the *state_code* column. The second partition covers records with BE and FR *state_code* column, records with DE and AT *state_code* column are stored in the third partition and the last partition contains records with HU and PL value of the *state_code* column. The definition of the *tab_list_partition_null* is displayed below.

```
CREATE TABLE tab_list_partition_null(
  record_id      NUMBER NOT NULL PRIMARY KEY,
  creation_date  DATE,
  state_code     VARCHAR(2),
  text           VARCHAR2(30))
PARTITION BY LIST (state_code)(
  PARTITION part_1 VALUES ('SK', 'CZ'),
  PARTITION part_2 VALUES ('BE', 'FR'),
  PARTITION part_3 VALUES ('DE', 'AT'),
  PARTITION part_4 VALUES ('HU', 'PL'))
/
```

After creating the table, we tried to insert the following records. The records also contained null values of the *state_code* column, but also values that do not fit into any partition.

```
INSERT INTO tab_list_partition_null VALUES
  (1, TO_DATE ('21.12.2022', 'DD.MM.YYYY'), 'SK', 'Text1');

INSERT INTO tab_list_partition_null VALUES
  (2, TO_DATE ('21.12.2021', 'DD.MM.YYYY'), NULL, 'Text2');

INSERT INTO tab_list_partition_null VALUES
  (3, NULL, 'HU', 'Text3');

INSERT INTO tab_list_partition_null VALUES
  (4, TO_DATE ('21.10.2023', 'DD.MM.YYYY'), 'AT', 'Text4');

INSERT INTO tab_list_partition_null VALUES
  (5, NULL, NULL, 'Text5');

INSERT INTO tab_list_partition_null VALUES
  (6, NULL, 'GB', 'Text6');
```

Records with a filled value of the *state_code* column were successfully inserted into the table, but records with a null value and a different value of this column could not be inserted into the database table. The system threw the following error (ORA-14400).

```
SQL Error: ORA-14400: inserted partition key does not map to
any partition
```

Therefore, we had to extend the definition of the *tab_list_partition_null* table with additional types of partitions. The system allows us to use the default clause. Subsequently, all other records that do not meet the conditions of the previous partitions will be inserted to the default partition.

Records with null value of the partition key column will also be stored in this partition. However, if we would like to separate records with null values from records whose value is filled but does not belong to any defined partition, the system allows creating a partition only for records with null values through the null clause.

The following example shows extending the *tab_list_partition_null* database table definition to include two more partitions. To the definition, we have added a partition that will store only records with null values in a given column, and another partition will store records with column values that do not fit into any other partition.

```
CREATE TABLE tab_list_partition_null(
  record_id      NUMBER NOT NULL PRIMARY KEY,
  creation_date  DATE,
  state_code     VARCHAR(2),
  text VARCHAR2(30))
PARTITION BY LIST (state_code)(
  PARTITION part_1 VALUES ('SK', 'CZ'),
  PARTITION part_2 VALUES ('BE', 'FR'),
  PARTITION part_3 VALUES ('DE', 'AT'),
  PARTITION part_4 VALUES ('HU', 'PL'),
  PARTITION part_null VALUES (NULL),
  PARTITION part_default VALUES (DEFAULT))
/
```

Subsequently, we again tried to insert the above records into the *tab_list_partition_null* table. All records were successfully inserted. Using the following SELECT statement, we checked how many records are in individual partitions.

```
SELECT 'PART_1', COUNT(*)
   FROM TAB_LIST_PARTITION_NULL PARTITION (PART_1)
 UNION ALL
SELECT 'PART_2', COUNT(*)
   FROM TAB_LIST_PARTITION_NULL PARTITION (PART_2)
 UNION ALL
SELECT 'PART_3', COUNT(*)
   FROM TAB_LIST_PARTITION_NULL PARTITION (PART_3)
 UNION ALL
SELECT 'PART_4', COUNT(*)
   FROM TAB_LIST_PARTITION_NULL PARTITION (PART_4)
UNION ALL
SELECT 'PART_NULL', COUNT(*)
   FROM TAB_LIST_PARTITION_NULL PARTITION (PART_NULL)
UNION ALL
SELECT 'PART_DEFAULT', COUNT(*)
   FROM TAB_LIST_PARTITION_NULL PARTITION (PART_DEFAULT);
```

The resulting table with the name of the partitions and the number of records in each partition is shown in Table III.

TABLE III. NUMBER OF RECORDS IN THE *TAB_LIST_PARTITION_NULL* TABLE

PART_	COUNT(*)
PART_1	1
PART_2	0
PART_3	1
PART_4	1
PART_NULL	2
PART_DEFAULT	1

List partitioning offers two options to solve the problem of null values. The first option is the use of the default partition type, but with this option it is important to realize that in addition, records with null value, records with values that do not belong to any other partition will also be stored here. The second option is to use a null partition type, so only records with null value of the partition key column will be stored in this partition.

C. Hash partitioning

Another way to divide the database table into partitions is to use partitioning by *HASH*. There is no need to specify explicitly partitions as in the RANGE or LIST partitioning. We only specify column according to which the partition will be created and number of partitions which will be created. If the number of partitions is not included, the system creates only one partition as default.

The following code displays the creation of the *tab_hash_partitioning_null* database table together with the *HASH* partitioning definition. We created partitions using the *creation_date* column and defined the number of partitions to the number four. So, the system created four partitions.

```
CREATE TABLE tab_hash_partition_null(
  record_id      NUMBER NOT NULL PRIMARY KEY,
  creation_date  DATE,
  text           VARCHAR2(30))
PARTITION BY HASH (creation_date)
 PARTITIONS 4
/
```

After creating the table, we inserted the same records into the database table as in the previous examples. All records have been successfully inserted into the table. Even those records that did not have a defined *creation_date* column.

Subsequently, we searched for records in individual partitions using the following SELECT statement. While we found that there are four records in the first partition and one record in the last partition. We obtained the names of the partitions through the *partition_name* attribute from the *user_tab_partitions* table.

```
SELECT 'PART_1', COUNT(*)
   FROM TAB_HASH_PARTITION_NULL PARTITION (SYS_P34704)
 UNION ALL
SELECT 'PART_2', COUNT(*)
   FROM TAB_HASH_PARTITION_NULL PARTITION (SYS_P34705)
 UNION ALL
SELECT 'PART_3', COUNT(*)
   FROM TAB_HASH_PARTITION_NULL PARTITION (SYS_P34706)
 UNION ALL
SELECT 'PART_4', COUNT(*)
   FROM TAB_HASH_PARTITION_NULL PARTITION (SYS_P34707);
```

The result of the above SELECT statement can be found in Table IV.

TABLE IV. NUMBER OF RECORDS IN THE *TAB_HASH_PARTITION_NULL* TABLE

PART_	COUNT(*)
PART_1	4
PART_2	0
PART_3	0
PART_4	1

After getting the records from the first partition using the following SELECT statement, we got the records that are stored in the partition.

```
SELECT *
   FROM TAB_HASH_PARTITION_NULL PARTITION (SYS_P34704);
```

The result of the previous SELECT statement, which selects only data from SYS_P34704 partition, can be found in Table V.

TABLE V. RECORDS IN THE SYS_P34704 PARTITION

Record ID	Creation date	Text
2	21.12.2021	Text2
3	NULL	Text3
4	21.10.2023	Text4
5	NULL	Text5

In this example, we can see that there is no problem with null values in HASH partitioning. The system inserts a record with null value of the given column into the partition obtained by calculating its hash function. In general, in the case of HASH partitioning, records with the null value of the partition key column can be stored in any partition.

VII. CONCLUSION

The main aim of this paper was to create a methodology for managing partitions with records with null values and show how to store records in partitioned tables. If the records contain not null value in the partition key column, there is no problem with inserting them into the table. The problem arises if we try to insert a record with null value of the partition key column into the table.

In this paper, we decided to demonstrate examples of storing records with null values in partitioned database tables.

The introduction of this paper describes what null values are and why their storing and processing is important. Null values cause a problem in their storage and further processing. The following chapter focuses on partitions. In this chapter, partition meaning and types of partitions (range, list, hash, composite partitions) are described in more detailed. After so-called theoretical part, the chapter with our experiments follows. For each type of partition, the practical examples of how records with null values behave when they are inserted into database tables divided into partitions, are demonstrated.

To summarize, in our experiments we focused on three types of partitioning. In the first experiment, we showed how it is possible to store records with null values of the partition key column in a table that uses RANGE partitioning. We demonstrated a sample of two options for creating partitions. First, we created a database table in which we defined the partitions directly, and then we created a database table in which the partitions were defined dynamically. In both cases, we tried to solve the problem of storing records with null values of the partition key column. The second experiment was focused on LIST partitioning. In this experiment, we again looked for a suitable solution for storing records with null values of the partition key column. We have shown two ways in which these records can be stored. The last third experiment was focused on HASH partitioning and its way of handling records with null values of the partition key column.

ACKNOWLEDGMENT

It was supported by the Erasmus+ project: Project number: 2022-1-SK01-KA220-HED-000089149, Project title: Including EVERyone in GREEN Data Analysis (EVERGREEN) funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the Slovak Academic Association for International Cooperation (SAAIC). Neither the European Union nor SAAIC can be held responsible for them

Co-funded by the European Union

EverGreen DATA Analytics

This paper was also supported by the VEGA 1/0192/24 project - Developing and applying advanced techniques for efficient processing of large-scale data in the intelligent transport systems environment.

REFERENCES

[1] Fatima H., Wasnik k., “Comparison of SQL, NoSQL and NewSQL Databases for Internet of Things”, *IEEE Bombay Section Symposium*, 2016

[2] Kvet M., “Identifying and Treating NULL Values in the Oracle Database – Performance Case Study”, *Conference of Open Innovation Association, FRUCT*, 2023, pp.161-168

[3] Lim L., “Elastic Data Partitioning for Cloud-based SQL Processing Systems”, *International Conference on Big Data,* 2013, pp 8-16

[4] Matiaško K., Vajsová M and Kvet M., *Advanced database systems 2,* Žilina, Edis 2017

[5] MySQL website, Restrictions and Limitations on Partitioning, Web: https://dev.mysql.com/doc/refman/8.3/en/partitioning-limitations.html

[6] Neumann T., “Reasoning in the Presence of NULLs”, *International Conference on Data Engineering (ICDE),* 2018, pp 1682-1683

[7] Oracle, “Database Concepts”, 2021

[8] Oracle, “Oracle Partitioning”, 2019

[9] Oracle website, VLDB and Partitioning Guide, Web: https://docs.oracle.com/en/database/oracle/oracle-database/23/vldbg

[10] Spurthi B., Rutuja B., and Vaishnavi B., “Small-Scale Relational Database Management System”, *IEEE Pune Section International Conference, PuneCon,* 2022

[11] Šalgová V. and Matiaško K., “Reducing Data Access Time using Table Partitioning Techniques”, *International Conference on Emerging eLearning Technologies and Applications (ICETA)*, 2020, pp. 564-569

[12] Šalgová V. and Matiaško K., “The Effect of Partitioning and Indexing on Data Access Time”, *Conference of Open Innovation Association, FRUCT*, 2021, pp. 301-306

[13] Wisal K., Cheng Z, Bin L, Teerath K. and Ejaz A., “Robust Partitioning Scheme for Accelerating SQL Database”, *International Conference on Emergency Science and Information Technology, ICESIT,* 2021, pp. 369-376