Identifying and Treating NULL Values in the Oracle Database – Performance Case Study

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

***Abstract*—Relational scheme has been created to serve data in a precise format by accepting the structure, conditions, and constraints. It consists of the entities and relationships between them. Each entity is formed by the attributes defined by the data type and various column constraints ensuring integrity and consistency. In this environment, however, undefined values can be present. This paper aims at identifying NULL values by optimizing the storage capacity and data retrieval performance in Oracle Database. Additionally, it provides new unique NULL pointer layer for the undefined value reference. Besides, it summarizes existing approaches by focusing on representation, 3-valued logic, and indexing.**

I. INTRODUCTION

A database is a set of data files, which are block oriented and provide the layer for storing the data. A database is a physical data repository and its optimization is associated with the storage capacity and demands, as well as internal data organization, which impacts the overall performance of accessing and retrieving data tuples. Over the decades, various discussions and research strategies were identified, delimited by the physical infrastructure, data discs, interfaces, and access rate. These factors influence the technical means in a hardware operation manner. Alongside, data storage architectures were evaluated, either defined by the physical data structures, or the size of the blocks. An important aspect, in addition to physical infrastructure and performance, is also security, persistency, availability, durability of data, and resistance to crashes. The database itself, however, does not have sufficient power and an additional layer managing the data must be present [1].

An instance is mostly defined by the software – processes of the instance operating the database and memory. These processes are responsible for managing the instance, and database managing, as well as ensuring data transfer, logging, durability, memory optimization, and sanitizing [2].

The critical part during the data processing and evaluation relates to the tuple identification, particular block memory loading, and evaluation based on the conditions of the statement, followed by the result set composition [2] [3]. From the opposite side, by inserting new tuples, instance processes must identify a free block in the memory to hold a new tuple, followed by transaction management, logging, and finally, transferring the changed blocks from the memory to the database. Conversely, the approach is similar, even if there is an attempt to update the existing data. In that case, the free block must be identified not to hold new data tuple, but the existing block in the first phase. Then, the update operation can be executed, if the tuple is not locked. During the update operation, however, the storage demands can be extended, consequencing in the impossibility to store new data versions in an original position. In that case, data must be migrated to another available block, to which the data pointer will be created and stored in the original data block. Data migration is a significant performance problem while retrieving the data, whereas the pre-loaded block does not store the required data, instead, just the locator to another repository is present, which results in the necessity to load multiple blocks, instead of only one.

To optimize the performance and access to the data, indexes are present, limiting the necessity to scan the data files and related blocks sequentially [4]. Thus, the index is a specific locator, its structure is defined by the attributes, function results, or expressions forming the key. Typically, B+tree indexes are present in the relational platform, whereas they maintain efficiency with the growth of the tuple version number. Thus, the traversing across the index is done based on the index key, up to the leaf layer, which holds the addresses of the data in the database storage layer. The physical address in the Oracle Database is defined by the 10-byte value, delimited by the data file identifier, block, and position of the tuple inside the block. Thus, the ROWID access is the fastest access to the data, based on the pre-condition, that the addresses are accurate and the data migrations are not present, negatively influencing the I/O operation number [4].

The database index is associated with the traverse path to locate particular key index values on the leaf. It is based on value comparison to identify, which element covers the defined range. The limitation is just the NULL value, which does not hold any value and thus, cannot be mathematically compared and evaluated, resulting in refusing NULL value indexing. Therefore, an additional index layer is proposed in this paper, to ensure the proper performance and index usage, even for the undefined values, or general values, which could contain NULLs. It is based on a specific index extension stored physically in the database, pointed from the root node [5].

Besides, the NULL value representation is discussed, delimited by the 3-valued logic and reflection in the temporal database systems, whereas they are a bit specific. Namely, a NULL value represents a completely undefined value, not applicable value or value, which is not obtained, stated,

relevant for the processing, or is delayed, caused by various reasons, like non-reliable network, improper measurement, precision ranges, etc. However, for the temporal systems, how to refer to the unlimited validity? Although the NULL value is used, representing an undefined value, it is clear, that such an event has not occurred yet and will be present in the future (if ever). Thus, in the temporal systems definition of the duration frame, the NULL value reflects only a partially undefined value, from the timeline reference, only future timepoints can be associated [5], [6].

To serve the complexity of the NULL value management in relational systems, the proposed paper is structured as follows: Section 2 deals with the meaning and representation of the NULL values by forming the 3-valued logic. Section 3 deals with the sorting techniques by extending order by clause of the Select statement to handle undefined values. Section 4 provides an evaluation study of the default value management using a clause or trigger, followed by handling conversion errors (section 5). Section 6 deals with the indexing and undefined value reference in the temporal systems using B+tree index enhancements. Finally, section 7 provides a *NULL pointer layer* extension to serve undefined values to be part of the indexing.

For the performance computational study, the *Oracle Cloud* environment located in the *Frankfurt* data region was used. Provisioned Autonomous Transaction Database version was *Oracle Database 21c Enterprise Edition Release 21.0.0.0.0 – Production Version 21.2.0.0.0*. The storage capacity was *20 GB* for internal data management. Backups were part of the *Object storage* outside the database itself. The used data set consisted of 5 000 000 of the sensor-based data provided by the air transport systems. It was spatio-temporal database oriented using group granularity. The precision of the date value processing was one second.

In this paper, we focus on the Autonomous Transaction Database provisioned in Oracle Cloud. There are several reasons for selecting it with no further references and comparisons to other database systems. Firstly, it provides advanced ML (machine learning) and AI (artificial intelligence) techniques to provide sel-driving, tuning and optimizing database, which does not require additional specific administration intervence. Moreover, it allows fast, reliable and robust worldwide access, supervised by the huge scalability. Thirdly, Oracle Database is the most powerful and provides various enhancements and improvements, so the provided solution can be compared to the latest trends and the best offers. Fourtly, Oracle Database is a the best player for the commercial database storage with high data flow, in which the data streaming can be anhanced by the reliability, undefined values and NULL reference treatment. And finally, this contribution is part of the Erasmus+ project EverGreen dealing with data analytics, in which the Oracle Corporation is the associative partner.

II. NULL VALUE REPRESENTATION AND MEANING

NULL notation represents undefined value, which can, however, arise from various causes, like inapplicable value, not stated, out of precision, value delivered late or undelivered, at all, etc. But the NULL value itself has only one representation and internal meaning is hidden. Thus, without any additional rules and records source and origin cannot be identified. Therefore, undefined values are in some cases modeled by the specific value part of the domain. This results in additional storage costs, as well as additional demands ensuring proper representation. Namely, the obtained value must always be processed and evaluated. The original value does not have to express the domain membership, but rather a specific symbol expressing the cause of the undefined value. The approach defined in this way has four main disadvantages. Firstly, the domain must be able to provide specific values allocated for the undefined values clearly distinguishable from the correct data. Secondly, by changing the precision and ranges over time, originally reserved values do not need to be proper later on. Thirdly, the types of undefined values can originate from various sources, and categories can be added and altered dynamically. Last but not least, there are the additional costs of storage and indexing. Users must be aware of the internal representation, otherwise, improper data can be obtained by the querying.

Thus, original NULL values are more feasible and representation effective, however, some performance drawbacks should be stated. The main limitation is the impossibility to compare the NULL value mathematically by limiting the opportunity to sort the values and place them in an ordered list. Whereas they cannot be compared, nor sorted, it is impossible to set a unique constraint for the attribute ensuring, that only one NULL value would be present. The only solution is to set column constraint type UNIQUE and NOT NULL column definition constraint. It would, however, require a specific representation for holding the undefined value.

The following code snippet shows, that the unique constraint definition does not ensure using only one NULL value, because the checking is done using B+tree index with an unique flag, however, NULL values are not indexed, so excluded from the consideration and checking process:

```
create table null_man_tab(id integer unique);
insert into null_man_tab values(null);
-- 1 row inserted.
insert into null_man_tab values(null);
-- 1 row inserted.
```

By treating NULL values, they cannot be mathematically compared using equality and non-equality sign, resulting in forming 3-valued logic. Next subsection introduces 3-valued logic by summarizing AND, OR and NOT operation results.

A. 3-valued logic

The NULL values management extends the conditional processing of the TRUE (T) and FALSE (F) values by the NULL (N) as the result, forming 3-valued logic. Fig. 1 shows the matrix for the evaluation, defined by the logical sum, logical product, and negation [5].

OR	T	F	N
T	T	T	T
F	T	F	N
N	T	N	N

AND	T	F	N
T	T	F	N
F	F	F	F
N	N	F	N

NOT	
T	F
F	T
N	N

Fig. 1. 3-valued logic

The performance study related to the condition evaluation was based on holding 10% of undefined values (500 000 records). The rest values were uniquely identifying the flight and sequence number. The study evaluated the following conditions by focusing on processing time and costs. Note, that there was no explicitly specified index to be used, forcing the system to perform sequential data block scanning. The identifier itself required 7808 blocks and 61 MB (the size of the block was set to the default option – 8KB. Performance study and impacts of the various block size to the loading can be found in [6]).

The following code block shows the query to get the size of the stored data objects. It is stored in the bytes, by transforming the value into MB precisions in the output:

```
select blocks, bytes/1024/1024 as MB
  from user_segments
    where segment_name = 'AIR_MONITORING';
```

The results of the query evaluation can be found in Fig. 2 by considering various conditions dealing with the undefined values, represented by the NULL notation. ECTRL_ID is the flight identifier, sequence number (SN) references the timepoint, during which the state of the airplane has been provided.

ECTRL_ID = 10 526	ROWCOUNT provided	COSTS	Result set SIZE
SN = 1	1	2 140	1 092 B
SN != 1	4 499 999	2 140	40 MB
not(SN=1)	4 499 999	2 140	40 MB
SN is null	500 000	2 132	4 084 KB
SN = 1 or SN is null	500 001	2 145	4 084 KB
SN <> 1 or SN is null	4 999 999	2 145	3 619 KB
no condition	5 000 000		

Fig. 2. Condition performance evaluation

As evident from the results, the number of rows part of the result set has a strong impact on the result set size and the direct proportion can be identified. However, from the processing costs point of view, only the evaluation strategy falls into consideration. In all cases, sequential block scanning operation was performed, so each data block was moved to the memory for the evaluation, which brings significant demands on the I/O operations, system sources, and activity of the background processes.

Besides, dynamic statistics sampling (level=2) was used. Dynamic sampling is an optimization strategy to improve the ability of the optimizer to make a good execution plan. It indicates, that the data statistics the optimizer attempts to use are not sufficient and should be extended. Thus, it does not substitute existing statistics, rather it extends them. During the Select statement evaluation by the database optimizer, statistics are considered. If they are not sufficient to produce a relevant and performance-effective execution plan, dynamic sampling is activated. Note, that dynamic sampling statistics are not so complex and exact, it is rather the fast estimation produced very quickly. Oracle database uses 12 levels of dynamic sampling, level 2 disables it, while level 1 defines at least one non-partitioned table that lacks the statistics. The default option is 2, applied in this case, as well means, one table referenced in the statement does not have the proper statistics, like a histogram of the values present for the attributes. A sample size (blocks) in this case is 64. More about dynamic sampling can be found in [2], [8], [9].

During the evaluation, no active transactions were present, all expired tuple locks were removed [10]. Thus, no sort strategy and DB block get were necessary. Consistent gets value takes 341 027 expressing the metric of the number of logical RAM buffer I/O reads to get data from a data block. SQL*Net message to the client and SQL*Net message from the client define waiting events, whereas the client process can be busy to accept the message delivery immediately.

Fig. 3 shows the executed Select statement statistics.

Statistics	
5	recursive calls
0	db block gets
341 027	consistent gets
0	physical reads
0	redo size
115 611 644	bytes sent via SQL*Net to client
3 666 715	bytes received via SQL*Net from client
333 335	SQL*Net roundtrips to/from client
0	sorts (memory)
0	sorts (disk)
5000000 rows	processed

Fig. 3. Select statement statistics

III. SORTING DATA

As the NULL values are not comparable and sortable, ordering the data in the result set requires additional clauses or specific functions to convert undefined values and place them in the sorted set. In this section, several approaches are evaluated to identify the performance impacts and costs. The environment and used data strategies are the same as already described in the previous section.

By default, NULL values are considered the maximal value in the range, so by ordering the data in an ascending manner, such values are placed at the end. It can be optionally enhanced by using NULLS FIRST and NULLS LAST clauses. Thus, based on the defined preconditions, the first and second evaluated solutions provide the same result set content. However, as evident from the results, although the total processing costs are almost the same, explicit clause definition brings additional demands for sorting, which takes 1% of the CPU for the SORT ORDER BY operation execution. The obtained results are shown in Fig. 4. Among the ORDER BY clause extensions, also user-controlled sorting operations have been used to evaluate performance impacts.

	processing time
no ORDER BY clause	00:01:00.85
order by value	00:01:03.89
order by value NULLS FIRST	00:01:04.91
order by value NULLS LAST	00:01:04.34
order by case when value is null then -1 else null end	00:01:05:12
order by nvl(value, -1)	00:01:06:78
order by transform_null(value)	00:01:08:24

Fig. 4. Processing time demands of Order by clause

Based on the results, it can be concluded, that the user definition brings additional demands, due to the necessity to compile the code. Moreover, if it is done in a PL/SQL function, processing time demands are significantly higher. Note, that even using NULLS LAST clause, which provides the same results as the ordinary ascending definition, requires additional demands.

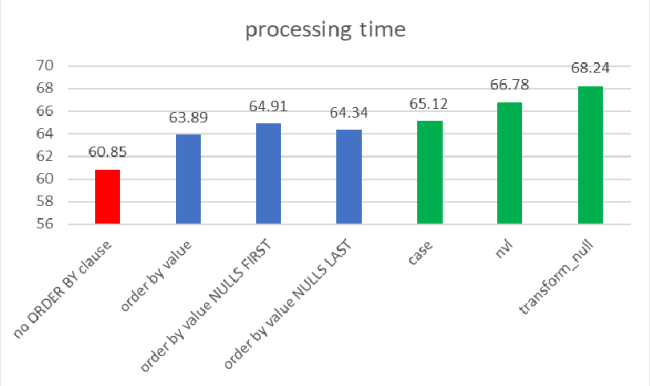

Fig. 5. Performance of the Order by clause

Fig. 5 shows the results in a graphical form. The reference solution does not sort the data and provides them as a heap. It requires 60.85 seconds. Ordinary ORDER by clause requires additional 3.04 seconds, which reflects 5% increase. Forcing the system to put the undefined values first requires additional 1.02 seconds. An interesting solution is provided by using the explicit NULLS LAST clause. Although the results are the same as an ordinal solution, the additional processing time is 0.45 seconds, just for the evaluation and order selection. Explicit management does not provide sufficient power. Three functions were evaluated – CASE, NVL, and explicit user function. The case provided the best results, internally optimized for SQL usage, while NVL is primarily used for the PL/SQL and requires a shift between SQL and PL/SQL environment processing. Finally, the own function (transform_null) requires parsed form loading in the memory for the shared function content. It requires 68.24 seconds, which represents a 7.39 second increase (12.14%). Compared to ordinary sorting, additional demands are 4.35 seconds, expressing 7.15%.

IV. Managing NULLs as Default Values

This section aims at evaluating NULL values and replacing them with the default value. The default value definition depends on the application and attribute domains. In the past, it was necessary to distinguish between value, which was not specified, and value, which was explicitly marked as undefined. Oracle database used to offer default clause for the attribute. The following code block shows the default value definition using attribute reference.

```
create table air_monitoring
  (ectrl_id integer,
   sn integer default -1);
insert into air_monitoring(ectrl_id)
 values(1);
insert into air_monitoring(ectrl_id, sn)
 values(2, null);
select * from air_monitoring;
```

The result of the above Select statement is depicted in fig. 6.

ECTRL_ID	*SN*
1	-1
2	(null)

Fig. 6. Select statement result

Prior to Oracle Database version 12c released in 2012 – explicit NULL value for a column bypassed the default value. This version introduced the DEFAULT ON NULL clause. In this section, the performance of the default value clause extension and explicit management using a trigger is evaluated, referring to the processing time. The results for 50 000 tuples are in Fig. 7. A significant performance difference can be identified. Inside the trigger, IF condition and default value assignment is present, if the condition is evaluated as TRUE meaning, the NULL value is identified. The evaluation has been associated with the Insert operation.

Insert	processing time
default on null	+00 00:00:03.031000
trigger	+00 00:00:04.751000

Fig. 7. Default on null performance – Insert statement

The trigger definition brings an additional increase of more than half (56.75%) compared to the DEFAULT ON NULL clause. It is due to the necessity of passing between the SQL language and the PL/SQL functionality defined in the trigger body.

When dealing with the Update statements, the difference reflects 16.89% (Fig. 8).

Update	processing time
default on null	+00 00:00:03.673000
trigger	+00 00:00:04.294000

Fig. 8. Default on null performance – Update statement

Results in the form of a chart are presented in Fig. 9. The difference for the Update operation is lowered,

whereas the size of the tuple after the change operation cannot be extended, so no data migration can be present.

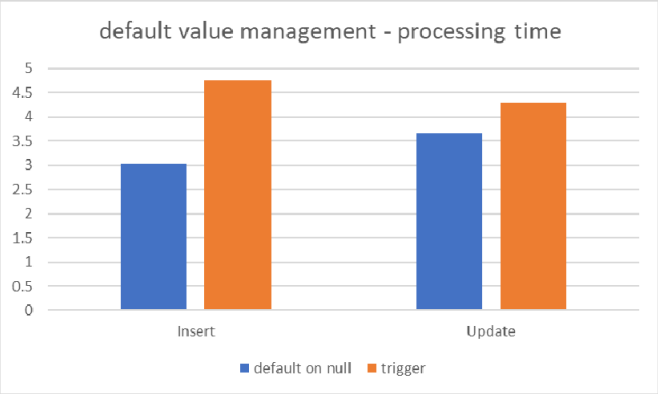

Fig. 9. Handling default value – performance results

V. Treating Data Conversions

External bulk loading is commonly associated with the character strings, which must be converted to a particular data type. The possibility of conversion needs to be verified and if any problem occurs, the NULL value should be used, instead of the original value, by recording it in the log. This section deals with three architectural solutions. The first solution is based on using an exception handler coded inside the block content body. Generally, no checking is done and conversion attempts to be performed. Any issue is covered by the exception handler, using the row granularity. The principle is shown in the following code snippet. The block content takes the Insert operation. If it fails, it is encapsulated by the exception handler inserting NULL.

```
begin
 insert into air_monitoring
   values(to_date('15.13.2000', 'DD.MM.YYYY'));
exception when others
  then insert into air_monitoring values(null);
end;
/
```

The second evaluated solution places the checking directly to the conversion operation by introducing the DEFAULT NULL ON CONVERSION ERROR clause. In this case, no explicit exception handler is used, because it is implicitly associated with the conversion operation itself. The following code block shows an example of th usage in PL/SQL data block.

```
insert into air_monitoring
 values(to_date('15.13.2000'
                default null on conversion error,
               'DD.MM.YYYY'));
```

The last solution uses the *validate_conversion* function. Thus, before the operation itself, it is verified that the conversion can be successfully performed. If not, the original value is replaced by NULL. Thanks to this, no exception can occur, and therefore there is no need to process the exception, define a handler, etc. The snippet of the code in the procedural language reference is stated in the following code block (for the readability and simplicity, we omit the declaration plasem as well as the whole block encapsulation):

```
select validate_conversion('15.13.2000' as date,
                           'DD.MM.YYYY')
  into result from dual;
if result=1 then
  insert into air_monitoring
    values(to_date('15.13.2000', 'DD.MM.YYYY'));
else
  insert into air_monitoring values(null);
end if;
```

Fig. 10 shows the results. The best solution was provided by using a conversion error clause, which required 5.45 seconds. The validate_conversion function takes the input, format, and output data type and checks, whether the conversion can be done, by getting value 1 (if possible) or 2 (if removed). The total demands are 8.61 seconds, which is also delimited and caused by the shift between the SQL and PL/SQL languages. Thus, it requires an additional 57.98% of the processing time. The worst solution uses PL/SQL code, as well. But the exception handler is defined, covering any unsuccessful attempts for the conversion. 50 000 values are incorrectly specified, resulting in getting a NULL value. The total processing time demands of the explicit exception management were 11.42 seconds. Compared to the validate_conversion usage, it requires an additional 32.66%. By taking the conversion error clause as a reference, additional processing time demands reflect more than 109%.

	processing time
exception handler	+00 00:00:11.422000
conversion error clause	+00 00:00:05.453000
validate_conversion	+00 00:00:08.610000

Fig. 10. Performance – treating conversion errors

A graphical representation of the conversion technique management is shown in Fig. 11.

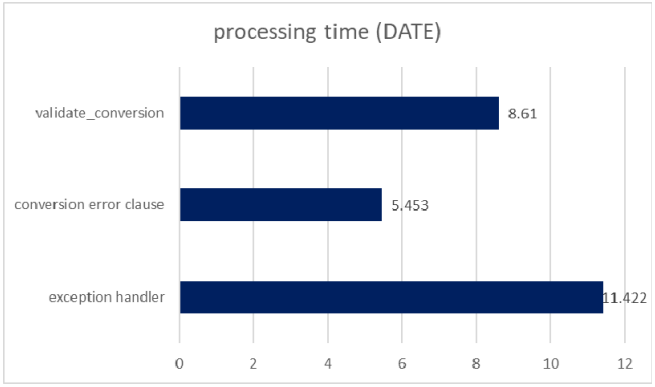

Fig. 11. Graphical representation – treating conversion errors - DATE

For the conversion to the numerical format, the obtained results are analogous, however, for the validate conversion,

date mapping format does not need to be treated, so the total processing costs are lowered to the value 6.86 seconds, compared to the DATE value processing, which requires 8.61 seconds. Concluding, format mapping for the specified environment required 1.75 seconds. Results are graphically presented in Fig. 12.

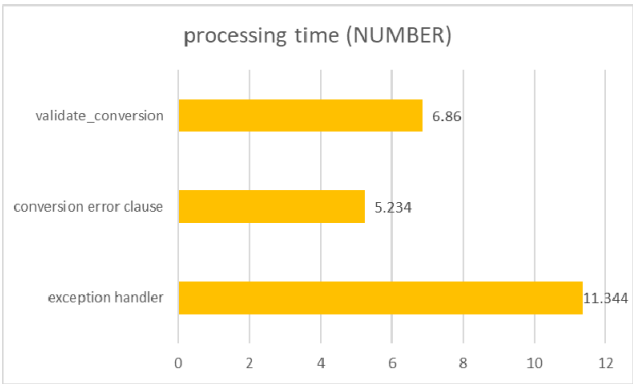

Fig. 12. Graphical representation – treating conversion errors - NUMBER

VI. INDEXING

Index as a structure of enhancing data path gives the optimized way to access the data blocks, compared to the sequential scanning necessity. By default, the B+tree index structure is created and defined implicitly for each primary key or unique constraint. The B+tree index is characterized by walking from the top (root node) to the leaf node-set, which contains the pointers to the data. Thus, at least one header block and one leaf block always exist. Each block, which is not marked as a leaf, has several descendants, forming the balanced tree. Moreover, the keys in the leaf layer are interconnected, so the data on the leaf layer are automatically sorted based on the key values. Besides the implicit indexes, the user can specify any index explicitly. However, the limitation of the B+tree is covered by the traversing, which is done by the mathematical comparisons to locate relevant tuple references [11] [12]. Whereas NULL values cannot be mathematically compared and evaluated, tuples, which have NULL value as an index key, are excluded, resulting in the necessity to scan the whole data block set sequentially, if the result set can potentially contain such rows. The limitation of the NULL value management, processing, and index coverage is shown in the following execution plans. Even though there are no undefined values in the table, the database optimizer cannot guarantee this, because it only refers to statistics that do not change with each data update, but only in defined time frames, or on request. Thus, in the case of defining a query that may contain a NULL value, a sequential search of the entire structure is used. Fig. 13 shows the execution plan of the ordinary solution.

```
Execution Plan
----------------------------------------------------------
Plan hash value: 352122309

-------------------------------------------------------------------------------------------------
| Id  | Operation          | Name             | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                  |  5389K|    66M|       | 28023   (2)| 00:00:02 |
|   1 |  HASH UNIQUE       |                  |  5389K|    66M|   103M| 28023   (2)| 00:00:02 |
|   2 |   TABLE ACCESS FULL| AIR_MONITORING6  |  5389K|    66M|       |  2147   (4)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Fig. 13. Execution plan (1)

By limiting undefined values by the additional condition, significant performance improvement can be identified (Fig. 14 representing the execution plan). However, note, that many times, undefined, non-reliable, or delayed data need to be identified and reflected. That would cause additional processing demands, whereas they are commonly represented by the NULL notation.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation          | Name             | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                  |  4665K|    57M|       | 24553   (2)| 00:00:01 |
|   1 |  HASH UNIQUE       |                  |  4665K|    57M|    89M| 24553   (2)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| AIR_MONITORING6  |  4665K|    57M|       |  2155   (4)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Fig. 14. Execution plan (2)

Fig. 15 shows the results dealing with the undefined values.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation          | Name             | Rows  | Bytes |TempSpc| Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |                  |   723K|    17M|       |  7656   (4)| 00:00:01 |
|   1 |  HASH UNIQUE       |                  |   723K|    17M|    25M|  7656   (4)| 00:00:01 |
|*  2 |   TABLE ACCESS FULL| AIR_MONITORING6  |   723K|    17M|       |  2270   (9)| 00:00:01 |
-------------------------------------------------------------------------------------------------
```

Fig. 15. Execution plan (3)

In the next section, the proposed solution extending the B+tree index by the NULL handler layer is discussed.

VII. IDENTIFYING UNDEFINED VALUES INSIDE THE INDEX

As evident from the results stated in Section VI., undefined values cannot be efficiently referenced by the index structures forcing the system to scan block-by-block sequentially. That results in various performance limitations, whereas data fragmentation can be present. Moreover, in dynamic systems, even empty blocks can be present. This section prepares a strategy for undefined value identification. The first approach relates to extending database table statistics by introducing a *NULL value counter* for each attribute, which can hold it, as expressed as *SOL_DD_COUNT*. This solution stores the number of NULL values in the data dictionary. However, unlike standard statistics, this value is updated directly when the transaction is approved. This ensures that the stored data is valid and accurate at any moment. Thus, even if the query does not limit undefined values and it is clear from the extended statistics that undefined values are not present, the optimizer can choose to use the index. Naturally, with an emphasis on constructing a consistent data image over time. And here we come to the limitation of the proposed solution. In an environment supporting huge parallelism, it would be necessary to store these extended statistics for each data image with the possibility of tracking the evolution over time based on transaction identification. It would require moving these extended statistics to a separate repository and associating them with the transaction-oriented log. So, if the transaction log expires, particular statistics would be vacuumed and original sequential data block scanning would be necessary to be performed.

The above solution is depicted by the following code block expressed by a trigger in a logical scheme:

```
Create or replace trigger update_data_dictionary
  before transaction approval
begin
   -- recalculate statistics for the used objects
   --              in the transaction;
```

```
   null_man.update_null_in_DD(changed_table_set);
   -- optionally
 for i in changed_table_set
  loop
   dbms_stats.gather_table_stats
    (tabname => 'i.table_name', cascade => true);
  end loop;
end;
/
```

The second solution relates to the B+tree index extension. In this case, the root element of the B+tree is extended to hold the *NULL pointer layer* used as the list of references to the rows, which hold undefined value for the particular data attribute. In [7], various architectures and data structures have been evaluated. Namely, pure B+tree, bitmap, and function-based indexes are evaluated. Besides, partitioning using global and local indexes are performance studied, pointing to the processing time, costs, and storage demands. In this paper, the *NULL pointer layer* is stored primarily in the instance memory and loaded automatically during the instance startup, respectively on demand. Moreover, this layer is shared among all indexes associated with the table, compared to the original solution described in [13]. Fig. 16 shows the architecture of the proposed solution by focusing on the undefined value management. Data input is represented by the various data stream, typically defined by the sensor based network. Then, each data tuple is extracted to identify, whether NULL values are present. If so, processing is navigated to *the NULL pointer layer* to record the ROWID reference. Otherwise, particular tuple is directly indexed. ROWID reference is shown red. The general data flow is marked black.

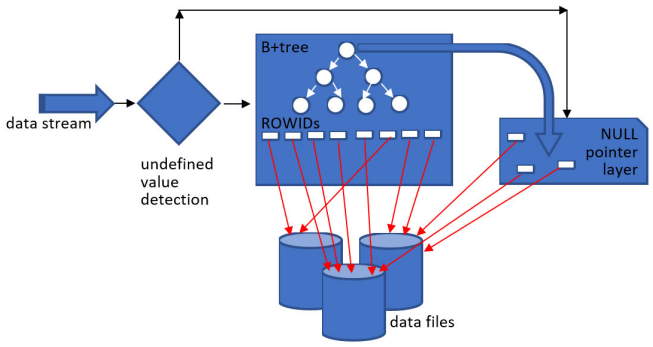

Fig. 16. Proposed architecture

Fig. 17 shows the results comparing BI-index introduced in [14] and the proposed memory structure. It is used in various optimization strategies [15] [16] and decision making support systems [17].

The limitation of the proposed solution is the pre-loading necessity. However, this operation can be done directly during the instance opening process, which would delay the timepoint of getting the database accessible or on demand before the first attempt to use that layer. In that case, however, loading would be strongly delayed, because the whole *NULL pointer layer* must be formed. The third relevant solution relates to the pre-indexer, which compresses the structure and loads in during the weak workload to make it already accessible when trying to reference the *NULL pointer layer*. Based on the results shown in Fig. 17, by comparing physical structure and memory structure, total improvements range from 16.2% up to 18.7%.

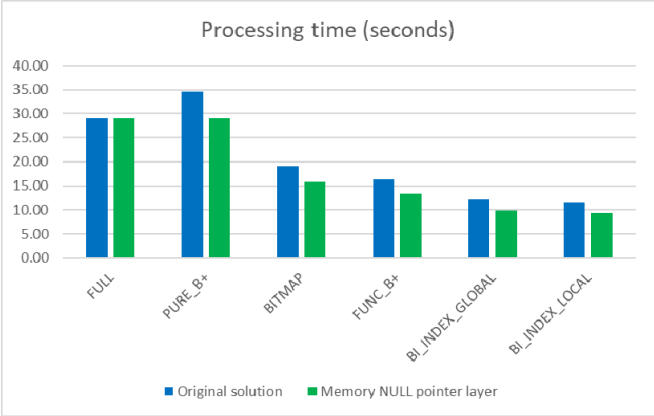

Fig. 17. Index performance

To conclude, memory loading operations provide sufficient power and improve the performance of the ad-hoc dynamic queries. On the other hand, memory loading must be present to ensure the structure accessible. Similarly, before shutting down the instance ordinary, particular memory *NULL pointer layer* must be binary exported to the database. But if the database is not closed correctly, the *NULL pointer layer* is automatically marked as untrusted and cannot be used later. In that case, the *NULL pointer layer* is composed from the scratch by scanning the table block set and identifying undefined values. During this period, the original indexes are accessible, but if a NULL value were referenced, a sequential search would be necessary. On the other hand, even this search itself can be used to construct a *NULL pointer layer*.

VIII. Methodology

This paper deals with undefined value management, which is critical for temporal management, as well as the sensor-based environment. Communication infrastructure can also cause delays and unreliable data, which should be considered.

This paper is practically oriented by providing the performance evaluation study, starting with Where the condition of the Select statement (data retrieval). Whereas undefined values cannot be mathematically compared, nor evaluated, significant additional processing time demands, and costs can be identified. Besides, several additional clauses have been considered. Namely, the Order by clause can be extended by the NULLS FIRST or NULLS LAST clause, which, based on the defined environmental conditions, brings 1.02 or 0.45 seconds, reflecting 26.22% or 11.59%, respectively. Own explicit sortage method does not bring a relevant solution, while the additional processing time demands rise to 2.89 seconds for the NVL function call or 4.35 seconds for the complete own function body definition. It is caused by the necessity to shift between SQL and PL/SQL environments.

Then, replacing the undefined value with NULL notation is evaluated. Default on NULL is optimized and reduces processing time demands, compared to the trigger, up to 36.20%.

Data can be originated from various sources and need to be commonly converted to the destination data type. This activity requires additional checking to ensure the conversion can be properly done. To highlight the performance demands, three solutions were evaluated. The best solution is provided by the conversion error clause extension. The most demanding solution is just exception management, while the additional universal handler must be defined. Similarly, checking conversion is not fair, because it is not directly associated with the operation, but it must precede the conversion itself.

Finally, the indexing techniques are discussed. The focus is on the B+trees, which are used as a default option in relational databases, while they maintain efficiency with the data source expansion. The proposed solution serves relevant solution because it is stored in the memory and completely shared for all indexes of the table.

The source of the undefined values and references can be placed in the memory, requiring synchronization across the instance and database during the maintenance windows or shutdown request. If the synchronization is not done, after reloading, the structure is marked as invalid and the system will launch additional background processes to compose a new structure by scanning the whole associated table block set sequentially.

IX. Conclusions & Future Research

NULL value management is an inseparable part of relational data processing. To ensure proper management and performance, individual options must be taken into account to choose the best solution based on the workload and strategy. In this paper, NULL value representations and meanings are stated. Then, the techniques for sorting data consisting of NULL values are present, followed by the default value management replacing original NULL values. Data conversion techniques are also taken into consideration during the definition and performance evaluation. This option is critical during the bulk data loading and moving data.

A significant emphasis of this paper is related to indexing by proposing own memory *NULL pointer layer*, which is a general repository referenced by all indexes using a root node pointer. It discusses the techniques, synchronization operations, as well as rebuilding after the failure.

In the future, we will focus on the development of other methodological procedures and techniques, with an orientation towards distributed environments, synchronization, and a multi-tenant cloud environment. Besides, the analytical environment will be considered, which uses data aggregations and pre-calculated results.

Acknowledgment

It was partially supported by the Erasmus+ project: Project number: 022-1-SK01-KA220-HED-000089149, Project title: Including EVERyone in GREEN Data Analysis (EVERGREEN).

References

[1] R. Greenwald, R. Stackowiak, and J. Stern, *Oracle Essentials: Oracle Database 12c*, O'Reilly Media, 2013.

[2] D. Kuhn and T. Kyte, *Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity.* Apress, 2021.

[3] D. Kuhn and T. Kyte, *Oracle Database Transactions and Locking Revealed: Building High Performance Through Concurrency*, Apress, 2020.

[4] J. Lewis, *Cost-Based Oracle Fundamentals*, Apress, 2005.

[5] S.Y.W. Su, S.J. Hyun and H.M. Chen, "Temporal association algebra: a mathematical foundation for processing object-oriented temporal databases", IEEE Transactions on Knowledge and Data Engineering, vol. 4, issue 3, 1998.

[6] M. Kvet and K. Matiaško, "Analysis of current trends in relational database indexing", 2020 International Conference on Smart Systems and Technologies (SST), Croatia, 2020.

[7] M. Kvet, J. Papán, "The Complexity of the Data Retrieval Process Using the Proposed Index Extension", IEEE Access, vol. 10, 2022.

[8] T. Cunningham, "Sharing and Generating Privacy-Preserving Spatio-Temporal Data Using Real-World Knowledge", 23rd IEEE International Conference on Mobile Data Management, Cyprus, 2022.

[9] X.Yao, J. Li, Y. Tao and S. Ji, "Relational Database Query Optimization Strategy Based on Industrial Internet Situation Awareness System", 7th International Conference on Computer and Communication Systems (ICCCS), China, 2022.

[10] M. Kvet, "Autonomous Temporal Transaction Database", 30th Conference of Open Innovations Association FRUCT, 2021.

[11] Z. Liu, Z. Zheng, Y. Hou and B. Ji, "Towards Optimal Tradeoff Between Data Freshness and Update Cost in Information-update Systems", 2022 International Conference on Computer Communications and Networks (ICCCN), USA, 2022.

[12] W. Wang, Y. Jin, B. Cao, "An Efficient and Privacy-Preserving Range Query over Encrypted Cloud Data", 2022 19th Annual International Conference on Privacy, Security & Trust (PST), Canada, 2022

[13] M. Kvet, "Temporal bi-index", unpublished

[14] M. Kvet, "Relation between the Temporal Database Environment and Disc Block Size", IEEE 16th International Scientific Conference on Informatics (Informatics), Slovakia, 2022.

[15] A. Dudáš, J. Škrinárová, and E. Vesel, "Optimization design for parallel coloring of a set of graphs in the High-Performance Computing, " Proceedings of 2019 IEEE 15th International Scientific Conference on Informatics. pp 93-99. ISBN 978-1-7281-3178-8.

[16] W. Steingartner, J. Eged, D. Radakovic, V. Novitzka, "Some innovations of teaching the course on Data structures and algorithms, " In 15th International Scientific Conference on Informatics, 2019.

[17] J. Janáček and M. Kvet, "Shrinking fence search strategy for p-location problems", 2020 IEEE 20th International Symposium on Computational Intelligence and Informatics (CINTI), Hungary, 2020