Data Management in Hierarchical Database - Branches Notification

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

Karol Matiaško
University of Zilina
Zilina, Slovakia
Karol.Matiasko@fri.uniza.sk

Abstract—**Current information system approaches require a database to manage data in a complex manner, to monitor the evolution and manage changes during the whole time spectrum. Several data architectures have been proposed with an emphasis on the data granularity. This paper deals with the temporal data changes management in a hierarchical database, where the whole access path branch must be notified if any change occurs. Thanks to that, individual changes are very effective to be identified and located. Our proposed solution originates from the self-relationship table and extends the principles with the opportunity to manage heterogeneous data streams with an opportunity to change the structure and references dynamically. In the first model, references are managed and notified using ROWIDs, the final solution uses object pointers based on the property of child object records covering the parents. Proposed solutions are implemented in intelligent transport system environment highlighting dynamics of electricity and hybrid vehicles and temporally changing environment.**

I. Introduction

Information systems and almost every application need to store data in the database. Systems are data-oriented with emphasis on changes monitoring and evolution management. In the past, data were located directly in the application, which is currently completely insufficient not only from the abstraction point of view but mostly due to the effectivity and robustness. Later, data were separated into the files. Thus, at least three layers could be identified – data storage (file system), data manipulation and evaluation formed by application codes. The last layer is presentation. Nowadays, data are mostly located in the database, thus the first layer has been modified without no significant changes in the other layers, only data access provider has been changed. Current database systems have many streams, from semantical models, file systems up to relational principles and big data. These systems, however, can be distinguished and divided based on transaction management. Most often used database system is still based on relational principles and algebra. Managed data are covered by the transactions, which means, that each change must be approved by the transaction manager before becoming visible to other applications, users and sessions. A transaction is a process of transferring one consistent and valid database state to the new one by passing all requirements and constraints. It is formed by four requirement properties – *ACID* – *atomicity* (either the entire transaction is successfully executed or it is completely canceled), *consistency* (after the transaction execution, all constraints must be passed), *isolation* (transaction data are not provided to other transactions/sessions before commit operation) and *durability* (confirmed data are persistent. No data can be lost, even after database crash) [1], [10].

Many times, data are formed into the hierarchy (car consists of individual components, the train consists of locomotive and wagons, individual wagons are formed by coupes, seats, etc.). If any component property is changed, the whole structure up to the hierarchy root must be notified (e.g. if the locomotive for the train is upgraded, it means, that the train can go faster, the train can consist of more wagons, etc.). Thus, it is necessary to propose a robust solution for dealing with database hierarchy, composition, and aggregation. This paper is transaction oriented and provides techniques for query definition and data management in the hierarchical database approach. The motivation is based on intelligent transport systems and electrical vehicles management. Individual charging machines are connected to the electricity grid. In this sense, the whole infrastructure is interconnected. If any subelement is corrupted, the whole architecture is influenced. Particular electricity branch must be notified and react automatically to adjust its properties, accessibility and capacity .

The aim of this research is to cover the complexity of the temporal database environment for dealing with hierarchy. Current relational databases can deal with such hierarchical data, but there is a lack of notification of all upper layers if the change occurs. Before proposing our solution, data change was only reflected as the update of a particular part of the object, but the whole structure image remained the same – the object validity itself was not changed. It was just only reflected on the individual component. It is, however, very important to store data change pointer in the object header, if the object composition is used. Thus, change on any level of the hierarchy must update the whole object definition, not just the sub-element (component) itself. Thanks to that, identification of the change with emphasis on data monitoring can be done on only one layer - the root. Moreover, internal tree architecture does not influence our proposed solution, thus the structure can evolve over time with no data management change necessity.

Section 2 deals with the conventional a temporal extension, summarizes existing principles and approaches reflecting the granularity. Section 3 deals with the hierarchy definition and current limitations. Section 4 offers a definition of our own proposed solution for dealing with hierarchical data. Afterward, properties, performance characteristics, and results analytics are proposed.

II. A CONVENTIONAL AND TEMPORAL APPROACH

Conventional paradigm is based on storing only current valid data, thus if the transaction is committed, the original version is removed and replaced by a newer one. Commonly, historical images are accessible only in the time limited manner using transaction log files. The system consists of a defined number of transaction log files with a defined size, which are cyclically rewritten [7], [8]. The primary purpose of them is to ensure transaction management - *consistency*, *isolation,* and *durability* with availability to restore the database after the crash. They are also inevitable for creating snapshots of the object inside the local transactions – other transactions must access original object values, if the transaction modifying them has not been approved, yet. Thus, if the transaction modifies particular data, which are not, however, approved, yet, other sessions and transactions must access original data, which are produced from the logs. Unfortunately, it is not always possible. As already mentioned, log files are managed in a cyclical manner. It means, that if the log does not consist of the active transaction data, it can be replaced with newer data. In that case, original data versions necessary for the other transaction can be unavailable. Let have two transactions (*T1* and *T2*), the first (T1) updates the object (*O*). Particular versions for such object (*O*) are stored in the log file (original (*UNDO* version) and new (*REDO* version)). Before the transaction ends, a new transaction is produced (*T2*), which gets the original data before the transaction start. However, if the original transaction (*T1*) is confirmed, changes take place and become permanent, so the transaction log is freed and can be rewritten. At that point, however, transaction *T2* may require original data (as they were at the beginning of the transaction). Unfortunately, such data cannot be provided. As a consequence, exception *ORA-01555 Snapshot Too Old* is raised. Database system tries to avoid such situation by using parameter *Undo_retention*, which influences time interval, during which historical images must be accessible without the possibility to rewrite them inside the logs. With the rising of its value, the accessibility time interval is also extended. On the other hand, the size of log files is significantly rising and transaction operations can be delayed, which have a significant impact on the system reliability [4], [5], [12]. Fig. 1 shows the principle of log management. Log files are cyclically rewritten if they do not store relevant data for existing (active) transactions. Historical data of the confirmed transactions are removed from the log files, therefore there is no possibility to lose data. Data themselves cannot be modified before storing *undo* (before image) and *redo* (after image) data in the log [7]. These log segments are operated by the *Log Writer* background processes. Another approach is based on *archive log mode* configuration – individual log files are copied and backed up in the file system before rewritten. With the cooperation and support of backups, historical images can be created and evolution can be monitored, although the process is complicated with high resource and time demands, whereas transaction logs do not store only data themselves, but also other support values, like *SCN* (*system change number - database state order number*), operations, images, etc.

Fig. 2 shows the processing demands to create a consistent image of the data valid in the past, delimited by the time point or time interval. First of all, the closest full backup from the left (historical) site of the temporality is loaded. Then, incremental backups are applied (in Fig. 2, it is complexly expressed by backup management), if available, merged by the archive log files. The active log files are used at the end. All these data groups point relevant data to the result set (gray strong arrows of the Fig. 2).

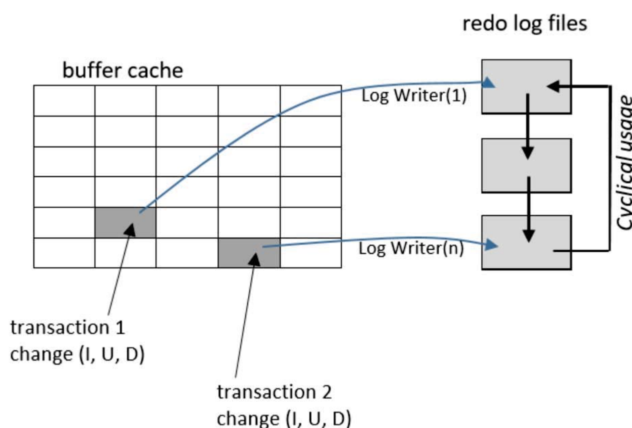

Fig. 1. Log file management

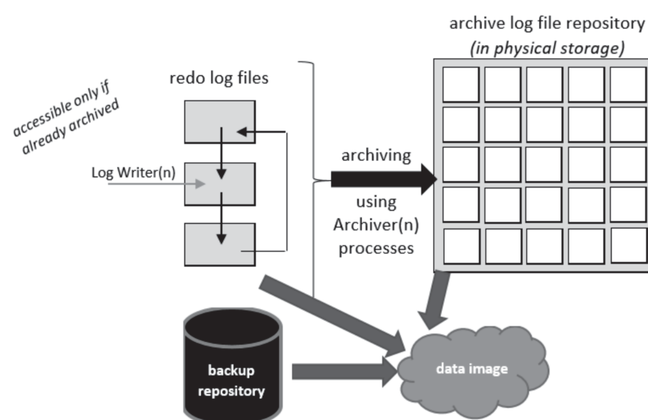

Fig. 2. Getting data image delimited by historical time frame

Temporal paradigm extends the existing conventional principles by storing all data images and changes inside the main database structure. Thanks to that, individual changes are a direct part of the solution. It, moreover, allows you to manipulate with future valid states and planning, thus the architecture is full time oriented.

Temporal evolution has been created soon after the first releases of the relational database systems in 60ties of the 20th century. Complexity and usability were, however, covered just in 90ties of the 20th century by proposing object-oriented temporal architecture [7], [8], [14]. Fig. 3 shows the logical scheme. Identifier of the object (*primary key*) is extended by the time interval forming a uni-temporal model with validity aspect. Optionally, it can store also transaction validity – time, during a particular object was stored in the database and assumed to be correct. It can be modeled by the time interval or by just one attribute expressing the insertion date of the row (tuple). The first model of the Fig. 3 is conventional with no temporal support. The second model is uni-temporal, *BD* expresses begin timepoint of the referenced tuple validity, *ED* ends the validity. Several approaches for date interval modeling

have been proposed, the description can be found in [2], [3], [7], [8]. The third model is bi-temporal managing two-time spectra – validity BD_{val}, ED_{val} and transaction aspect modeled by BD_{trans}.

The main disadvantage of the previously described model is object granularity if the data updates are not synchronized. Let have the object consisting of five temporal attributes. If the individual update statement does not change all of them, original values would be copied and stored in the system multiple times increasing storage demands (Fig. 4). Remind also static and conventional attributes, the evolution of which cannot be monitored. Gray color in the Fig. 4 expresses not-changed values, which are copied to the new images. *NULL* values undefined reference pointers do not provide sufficient power in this case [2], [6].

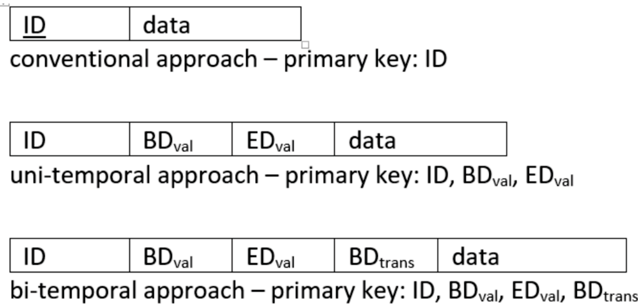

Fig. 3. Object level temporal model

ID	Time attributes (validity, resp. transaction definition)	data				
		A1	A2	A3	A4	A5
1	Val_1	$A1_{val1}$	$A2_{val1}$	$A3_{val1}$	$A4_{val1}$	$A5_{val1}$
1	Val_2	$A1_{val1}$	$A2_{val1}$	**$A3_{val2}$**	$A4_{val1}$	$A5_{val1}$
1	Val_3	$A1_{val1}$	**$A2_{val2}$**	$A3_{val2}$	$A4_{val1}$	$A5_{val1}$
1	Val_4	$A1_{val1}$	$A2_{val2}$	$A3_{val2}$	**$A4_{val2}$**	$A5_{val1}$
1	Val_5	$A1_{val1}$	$A2_{val2}$	$A3_{val2}$	$A4_{val2}$	**$A5_{val2}$**
1	Val_6	$A1_{val1}$	**$A2_{val3}$**	$A3_{val2}$	$A4_{val2}$	$A5_{val2}$
1	Val_7	$A1_{val1}$	$A2_{val3}$	$A3_{val2}$	$A4_{val2}$	**$A5_{val3}$**

original row values

Fig. 4. Object level temporal data effectivity

To solve the limitation, therefore, the attribute-oriented approach was defined in 2013 consisting of three layers (Fig. 5) [10], [11]. The core of the system is managed by the temporal manager and table storing all changes referencing the table of origin, row and changed the attribute. Thanks to architecture, no duplicate values are present. Moreover, such a solution can store also data, which do not change their values at all (e.g. code lists) or monitoring is not necessary, even forbidden (e.g. based on GDPR).

The temporal table consists of these attributes [11]:

- *ID_change* – got using sequence and trigger – primary key of the table.
- *ID_previous_change* – references the last change of an object identified by *ID*. This attribute can also have a *NULL* value that means, the data have not been updated yet, so the data were inserted for the first time in the past and are still actual.
- *ID_tab* – references the table, record of which has been processed by DML statement *(Insert, Delete, Update, Restore)*.
- *ID_orig* - carries the information about the identifier of the row that has been changed.
- *ID_column* – holds the information about the changed attribute (each temporal attribute has defined value for the referencing).
- *Data_type* – defines the data type of the changed attribute:
- C = char / varchar, N = numeric values (real, integer, …), D = date, T = timestamp, …
- This model can be also extended by the definition of other data types like binary objects.
- *ID_row* – references to the old value of an attribute (if the DML statement was *Update*). Only update statement of temporal column sets not *NULL* value.
- *Operation* – determines the provided operation:
- I = insert, D = delete, U = update, R = restore
- The principles and usage of proposed operations are defined in the part of this paper.
- *BD* – the begin date of the new state validity of an object.

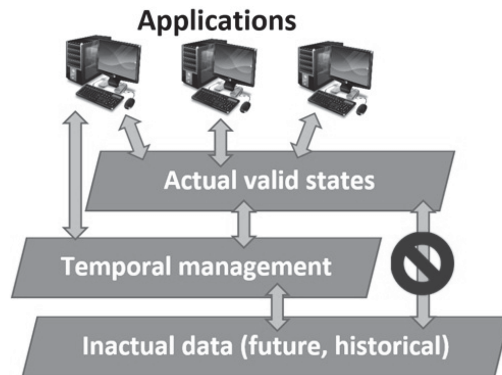

Fig. 5. The architecture of the attribute-oriented temporal system [11]

Interlayer between the object and attribute-oriented approach is just the group granularity, which allows creating a synchronization group internally processed as one attribute. Thanks to that, only one row is inserted into the temporal layer, if the group is updated, regardless of the number of attributes inside it. In general, in the beginning, synchronization group is created from each attribute separately, afterward, based on the data update time, synchronization groups can be detected and merged either automatically using background processes (*group detector* and *synchronizer*) or manually [9]. The solution from the data model view is shown in the Fig. 6. It is protected by the ISA hierarchy (a group can be composed either from individual attributes or by using existing groups). Vice versa, if the group is to be dropped, it is split into individual attributes or into groups based on the hierarchy evolution.

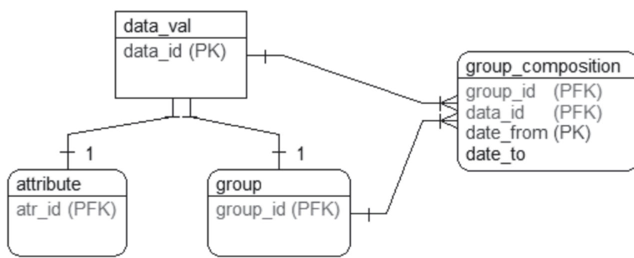

Fig. 6. Group management data model

III. HIERARCHY MODELING

Hierarchy modeling is just a problem to be faced in this paper. In existing temporal solutions, the emphasis is placed on just one table, over which the change is made. However, let have the object hierarchy (composition aggregation). It can be inevitable to notify all upper-level nodes about the update (if the property of the component is changed, the whole structure must be notified). Current temporal systems do not provide sufficient power to cover the hierarchical database models. In that case, it is necessary to message all layers manually sequentially. If the path length from the root up to the changed node is not the same for all attributes, the problem is even sharper forcing system to execute select and update statement dynamically in the cycle, which causes strict performance degradation. Moreover, there can be a problem with data row access using index [12], [15].

Database systems allow you to define a hierarchical query. In that case, self-relationship is used (the foreign key is referencing the same table primary key – Fig. 7). Based on technical reasons, many times, the temporal environment removes the composite primary key by the transaction and introduces a new virtual. The original primary key set is delimited by the unique index constraint.

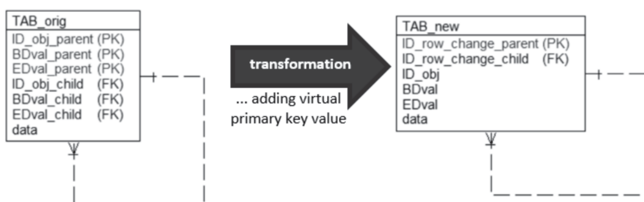

Fig. 7. Self-relationship transformation

Query getting current level data and parent requires to use the particular table twice in the *From* clause. Both tables must be delimited by the aliases.

```
SELECT *
 FROM TAB_new T1
    JOIN TAB_new T2
      ON (T1.id_row_change_parent
          = T2.id_row_change_child);
```

Query evaluation consists of scanning table using *Table Access Full* method – sequential scanning of all blocks under *High Water Mark* (*HWM*) sign (whereas the foreign key index is not defined), followed by *Index Range Scan* of the primary data. *High Water Mark* (*HWM*) sign points to the last block associated with the table. Individual blocks are linked. Fig. 8 shows the execution plan (*Autotrace*) of such a solution.

OPERATION	OBJECT_NAME	CARDINALITY	COST	LAST_CR_BUFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT			2		
NESTED LOOPS				6	66
NESTED LOOPS		2	2	5	58
TABLE ACCESS (FULL)	TAB	3	2	3	35
INDEX (RANGE SCAN)	TAB_IND	3	0	2	22
σ Access Predicates					
PK.PK=FK.FK					
TABLE ACCESS (BY INDEX ROWID)	TAB	1	0	1	6

Fig. 8. Execution plan of the self-relationship query

It is, however, valid only for a two-layer architecture of the hierarchy. If three layers are identified, the individual table must be listed three times requiring using nested loops to reach the results (Fig. 9):

```
SELECT *
 FROM TAB_new T1
    JOIN TAB_new T2
      ON (T1.id_row_change_parent
          = T2.id_row_change_child)
    JOIN JOIN TAB_new T3
      ON (T2.id_row_change_parent
          = T3.id_row_change_child);
```

In this case, the table is scanned three times and joined together using a nested loop operation. Fig. 9 shows the solution for three-level hierarchy based on self-relationship architecture.

OPERATION	OBJECT_NAME	CARDINALITY	COST	LAST_CR_BUFER_GETS	LAST_ELAPSED_TIME
SELECT STATEMENT			2		
NESTED LOOPS				6	108
NESTED LOOPS		1	2	5	100
TABLE ACCESS (FULL)	TAB	4	2	3	60
INDEX (RANGE SCAN)	TAB_IND	4	0	2	29
σ Access Predicates					
FK.FK=FKK.PK					
TABLE ACCESS (BY INDEX ROWID)	TAB	1	0	1	7

Fig. 9.Autotrace – three-level architecture

As a result, it is extremely inefficient, individual steps and evaluation is highly dependent on the architecture and forces the user to compose the command dynamically, which does not only slowdowns the whole system, but also complicates the possibility of defining access methods and indexes, whereas the structure is constantly changing. Moreover, if a new layer is added, all queries must be rewritten - the table must be referenced one more time.

To solve the problem, the Oracle database system has proposed *Connect by the prior* clause. In that case, the hierarchy can be listed in one query regardless of the length from the root to the particular node. The table is referenced only once. The user does not need to determine and reconstruct query if new intermediate step (new level) is added. Thus, from the user point of view, the depth of the hierarchy is not important. Internally, the query is divided and executed gradually for each level applying the conditions of all lower layers. It consists of *Connect by prior* filtering followed by *Index Unique Scan* for primary key and *Index Range Scan* for foreign keys. Fig. 10 shows the execution plan.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		2	4
CONNECT BY (WITH FILTERING)			
σ Access Predicates			
TAB.FK=PRIOR TAB.PK			
TABLE ACCESS (BY INDEX ROWID)	TAB	1	1
INDEX (UNIQUE SCAN)	SYS_C0016830	1	1
σ Access Predicates			
PK=1			
NESTED LOOPS		1	1
CONNECT BY PUMP			
TABLE ACCESS (BY INDEX ROWID)	TAB	1	0
INDEX (RANGE SCAN)	TAB_IND	1	0
σ Access Predicates			
connect$_by$_pump			

Fig. 10. Execution plan based on using Connect by level clause

Performance limitation is just the index created on the foreign key attributes in this case. If not available – not created or even unusable [12], solution significantly degrades, *Table Access Full* method must be used, instead (Fig. 11). Unusable index means, that particular references to the data have been changed, thus index cannot be used and is not maintained any more [12]. Data references are provided by the value *ROWID*, which points to the physical data in the database. *ROWID* consists of the identifier of the data file, block, and position of the block, where the particular object row resist.

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		3	3
CONNECT BY (NO FILTERING WITH START)			
σ Access Predicates			
TAB.FK=PRIOR TAB.PK			
σ Filter Predicates			
PK=1			
TABLE ACCESS (FULL)	TAB	3	2

Fig. 11. Performance limitation

IV. OWN HIERARCHY MANAGEMENT SOLUTION

Previously mentioned solutions described in chapter 3 are robust, only if the structure of individual layers is the same, whereas all the data are stored in one table. If not, models are not optimal suffering from all the negatives of the non-normalized data relation and the whole model – a potential loss of operations, the necessity to change multiple records, when changing only one value, etc. However, the main limitation is just the performance expressed by the time consumption of the processing, as well as size demands. It is necessary to evaluate the type of the layer and then, access to relevant attributes. So, one attribute definition type is added. Afterward, individual attributes are accessed, mostly by using additional query.

Reliability of the solution is strictly limited by the correctness of the object type, which must be secured and authorized by the trigger before each destructive *DML* operation, which also slowdowns execution plan [16], [17].

Our proposed solution originates from the standard non-hierarchical query. If the structure is steady, path using referential integrity without self-relationship can be used. The proposed solution is, moreover, resistant against the structure of individual layer modification. They can be changed anytime dynamically, however, the architecture and number of layers must remain original. Fig. 12 shows the solution.

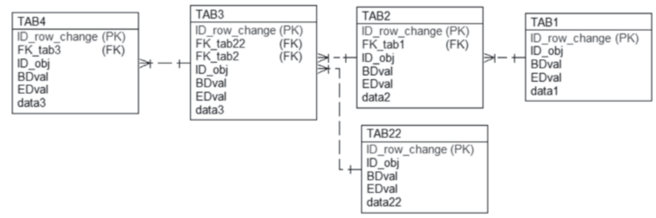

Fig. 12. The proposed solution data model

Regarding the necessity to add a new layer, indexes and all queries must be rebuilt, thus, if there is a chance to modify and cover architecture evolution, the solution is not usable.

Therefore, we propose another solution using the fact, that the best and quickest access to the data is the pointer. *ROWID* has the following structure (up to Oracle 8i – 8bytes; From Oracle 8i – 10 bytes):

- **The data object number** *(1 - 32 bits),*
- **Data file** in which the row resides (the first file is 1; file number is relative to tablespace) *(33 - 44 bits),*
- **Data block** in the data file in which the row resides *(45 - 64 bits),*
- **The position of the row** in the data block (the first row is 0) *(65 - 80 bits).*

ROWID is used in the leaf layer of the index structure. Particular data are located in the index and accessed by the *Rowid Scan*. In our case, however, index access is absent, whereas it would require B+tree traversing consuming time (Fig. 13). Our solution stores the *ROWID* locators directly in the table structure. Thanks to that, the specific index can be omitted and does not need to be defined, at all.

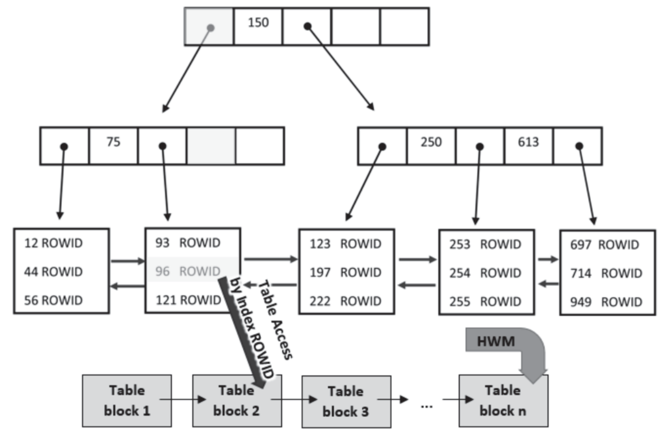

Fig.13. Table access using the index and Rowid Scan

In this case, the *Index Scan* can be omitted, whereas a particular data pointer is already stored in the database in the upper layer. Thus, to access defined row, only *Table Access By Index ROWID* method is used (Fig.14):

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		1	1
TABLE ACCESS (BY USER ROWID)	TAB	1	1

Fig.14.Execution plan of our proposed solution

In the first phase, we assumed, that index itself does not need to be defined, at all. Later, based on complex case studies, we came to the conclusion, that solution must have an index, due to the security and reliability aspect. First of all, *ROWID* is not a stable element. If there is an update of the row, it can happen, that new tuple does not fit the original data block and must be located in another. In that case, migrated row is created and originally stored *ROWID* is not valid, however, the system cannot determine it automatically (references are only one dimensional). If the index is not defined, particular migrated row pointer is not stored directly in the database and system would access inappropriate data block consequencing in scanning all the blocks under the *High Water Mark* (*HWM*) sequentially. The property of the B+tree index, as the default index type in the database, is just the effectivity and robustness of the data locators [12], [13], [15]. If the *ROWID* is changed, a particular pointer to another block is created automatically and reference stored in the original block (Fig. 15). Our proposed solution uses that fact, thus some index must be defined. However, it does not need to be created for the foreign key, in comparison with other mentioned techniques. Each table has a primary key pointing to the data, so the index is created automatically [7]. Moreover, it always uses *ROWID* locators, whereas the table generally consists also of non-key attributes.

The problem of the migrated row is shown in Fig. 15. The particular data block is accessed using the *ROWID* stored in the leaf layer of the index. Afterward, the referenced data block is loaded into memory buffer cache (arrow from the table block 3 to *buffer cache* structure, which reflects the repository for the data in the memory, Fig. 15), however, required the data are not there, thus, migrated row pointer is used and another block is loaded table block *n* must be loaded into the memory, Fig. 15). If the access path is wide consisting of several migrations, inappropriate blocks are loaded and the execution process becomes more complicated and time and resource demanding.

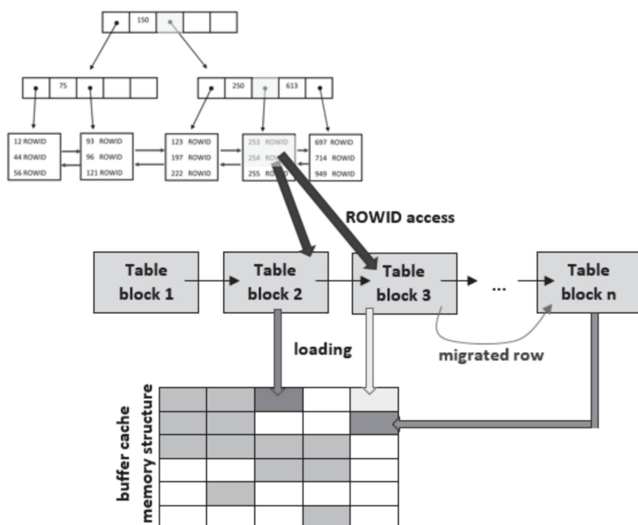

Fig. 15. Rowid access protection – security and reliability aspect

Problems can arise, if the *ROWID* values are changed in specific other situations, like using *flashback* technology (operation of getting a historical image of the table), moving data to other disks, tablespaces or files. These activities change all the *ROWID* values for the table. It is not, however, the problem in our solution, respectively it is easy to detect and evaluate such situation, whereas if the data positions are changed, the particular index is automatically listed as *UNUSABLE* [1], [12], [13]. By using triggers, the whole structure is notified and *ROWID* values are recalculated. Such a situation, however, occurs very rarely. During the recalculation process, *Table Access Full Method* (*TAF*) is used – all data blocks are sequentially scanned to locate to the row.

Proposed solution with *ROWID* management provides a relevant solution and fills the gap of data management in the hierarchical architecture of the database approach. Based on the time processing of the query, it provides benefits in the performance sphere. On the other hand, again, it is not possible to effectively change the structure and depth of the hierarchy.

The last proposed solution is universal and independent. It can deal with any number of levels in the architecture and automatically reacts to the structural evolution and any change of the architecture. The principle is based on the object references inside the relational structure. Solution model is shown in the Fig. 16. Individual changes are stored in the temporal table with self-relationship. References to the updated values are not a direct part of the table itself (like in attribute oriented temporal system), but the universal reference to the object is used. For the solution, individual objects are defined as child records (descendants) of the core object. Whereas each child record can be used as a replacement of the parent, the data structure can evolve anytime. It is just necessary to create a new child object and register it in the temporal layer of the solution. All other activities are maintained automatically by background processes.

Fig. 16 shows the solution architecture. It consists of three components. Individual changes and hierarchy are stored in the self-relationship table. It can be referenced to the temporal ecosystem based on defined granularity (third component of the solution). Data themselves are not, however, stored directly in the hierarchy, but the references using objects are used. The reference type is *tObjBase* as the root of the object types, all others are under that as extensions. Thanks to that, any object type can be referenced (as the child of the *tObjBase* type). Objects and references are managed in the second component of the solution. The last part is created from the temporal registration, where acknowledges, notifications and storage principles for temporal data are defined and maintained.

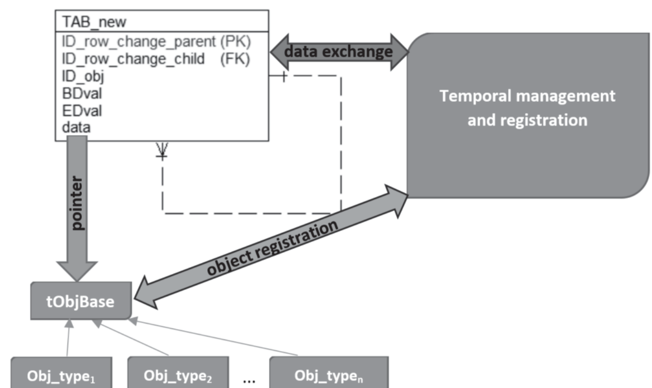

Fig. 16. Solution architecture using object pointers

V. Performance analysis

Experiment results were provided using Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - 64bit Production; PL/SQL Release 11.2.0.1.0 – Production. Parameters of the used computer are:

- Processor: Intel Xeon E5620; 2,4GHz (8 cores),
- Operation memory: 16GB,
- HDD: 500GB.

Performance is the whole solution in comparison with existing approaches is also covered in this paper. Let assume, that only 1 000 records are stored in the hierarchical database. In that case, if the number of levels is two (master and child record), notification and *Select* statement requires 6 values of the costs and lasts 187ms (existing solution described in section 3, performance shown in fig. 8).

If the system, however, consists of three levels, performance degrades – it requires 6 costs but elapsed 304ms (existing solution performance is shown in section 3, Fig. 9). A general solution is not usable, whereas the processing demands are rising very deeply.

Using *Connect by prior* clause solves the problem only partially, although it is level independent. Although the costs are only characterized by the value 5 (improvement using 16,7%), if the index is not defined, performance does not provide sufficient power. A more significant limitation is just the necessity to define the same structure for each level, which is not easy to ensure in a real environment. Moreover, this clause is defined for the DBS Oracle and other systems do not provide it. Therefore, we propose our own solution for dealing with hierarchy. It has been tested using DBS Oracle, but the solution is not dependent and can be implemented on any system type. Costs are lowered to the value 2 (reference Fig. 14), whereas direct access to the data using *ROWID* can be used.

To be critical, there are also limitations of our proposed solution. The most significant is based on de-duplication. There is no support for limiting the same component definition, to point the same data to several objects with the reflection of the change. If the component is linked to the several objects and is altered later, the particular change would be present in all the objects to which it belongs, which does not need to be correct in general. Therefore, in the future, we want to focus on the decomposition of the component after the change.

VI. Conclusions

Current relational systems do not provide complex management of hierarchy in the database. They highlight only architectures with the same data structure on each level. Temporal evolution requires storing the whole evolution during the object lifecycle. If the temporal aspect degree is added to the system, performance significantly degrades, whereas the architecture of the solution is not proper. Therefore, in this paper, we summarize technologies for dealing with hierarchical oriented data management in the temporal environment. We target the availability of storing heterogeneous data with the evolving structure inside the layer, as well as the whole architecture of the levels. Thanks to that, any change can be directly reflected and data structure and model can be changed dynamically with availability to get a historical image in the original structure.

This paper extends the existing approaches by proposing two solutions. The first solution is based on storing *ROWID* of the lower layer directly in the database as the data pointer. The specific index does not need to be defined, it aims only as the notifier if the data position is changed. In that case, the index is marked as *UNUSABLE* and all *ROWID* values are not relevant and must be recalculated. The second proposed solution is such a problem immune. It is characterized by the object-oriented data tuples, instead of standard relational theory. Thanks to that, pointers to the data are based on references to the objects. Access to the data in the *Select* queries is direct without the necessity of using any specific access method. If there is a necessity to change data, a function-based index using object reference is used. However, notice, that in temporal databases, most data changes are expressed by the *Insert* statements, corrections can be done only in bi-temporal architecture using physical data.

The proposed architecture is now implemented on the intelligent transport systems consisting of an expanding network of charging stations for electric vehicles and their management, where any component failure or change influences the whole system. The infrastructure and conditions evolve over the time dynamically.

Proposed solution is architecture and approach general and can be implemented in any field, like railway transport, individual networks or industry as well. Based on the performance analysis - not only experiments described in this paper, solution is robust and does not degrade performance with the data number increase.

In the future, we will extend the solution by the notification layer – if there is a change, the system will automatically notify ll the upper layers sequentially, up to the root of the hierarchy. Our preference will highlight the de-duplication techniques for modeling many-to-many relationship cardinality of the hierarchy, as well.

Acknowledgment

This publication is the result of the project implementation: *Centre of excellence for systems and services of intelligent transport II.,* ITMS 26220120050 supported by the Research & Development Operational Programme funded by the ERDF.

The work is also supported by the project VEGA 1/0089/19 - *Data analysis methods and decisions support tools for service systems supporting electric vehicles*.

"Podporujeme výskumné aktivity na Slovensku
Projekt je spolufinancovaný zo zdrojov EÚ

References

[1] K. Ahsan, P. Vijay. *Temporal Databases: Information Systems*, Booktango, 2014.

[2] A. Alelaiwi. “Evaluating distributed IoT databases for edge/cloud platforms using the analytic hierarchy process”, Journal of Parallel and Distributed Computing, pp. 41-46, 2019.

[3] R. Behling et al., “Derivation of long-term spatiotemporal lanslide

activity – a multisensor time species approach", 2016. In Remote Sensing of Environment, Vol. 136, pp. 88-104.

[4] C. J. Date, N. Lorentzos, H. Darwen. *Time and Relational Theory : Temporal Databases in the Relational Model and SQL*. Morgan Kaufmann, 2015.

[5] M. Doroudian, et al.: "Multilayered database intrusion detection system for detecting malicious behaviours in big data transaction" IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), 2016.

[6] M. Erlandsson et al., "Spatial and temporal variations of base cation release from chemical weathering a hisscope scale". 2016. In Chemical Geology, Vol. 441, pp. 1-13.

[7] T. Johnston. *Bi-temporal data – Theory and Practice.* Morgan Kaufmann, 2014.

[8] T. Johnston and R. Weis, *Managing Time in Relational Databases*, Morgan Kaufmann, 2010.

[9] M. Kvet, K. Matiaško, "Temporal Data Group Management", IEEE conference IDT 2017, 5.7. – 7.7.2017, pp. 218-226.

[10] M. Kvet, K. Matiaško, "Transaction Management in Temporal System", 2014. IEEE conference CISTI 2014, 18.6. – 21.6.2014, pp. 868-873.

[11] M. Kvet and K. Matiaško, "Uni-temporal modelling extension at the object vs. attribute level", IEEE conference UKSim, 20.11 – 22. 11.2014, , pp. 6-11, 2013.

[12] D. Kuhn, S. Alapati, B. Padfield, *Expert Oracle Indexing Access Paths*. Apress, 2016.

[13] S. Li, Z. Qin, H. Song. "A Temporal-Spatial Method for Group Detection, Locating and Tracking", In IEEE Access, volume 4, 2016.

[14] Y. Li et al., "Spatial and temporal distribution of novel species in China", 2016. In Chinese Journal of Ecology, Vol. 35, No. 7, pp. 1684-1690.

[15] A. Noury, M. Amini. "An access and inference control model for time series databases", Future Generation Computer Systems, Vol. 92, pp. 93 – 108, 2019.

[16] P. Rusnak, J. Rabcan, M. Kvassay and V. Levashenko. "Time-dependent reliability analysis based on structure function and logic differential calculus", Advances in Intelligent Systems and Computing, Volume 761, pp. 409-419, 2019.

[17] E. Zaitseva, V. Levashenko, M. Kvassay, P. Barach. "Healthcare system reliability analysis addressing uncertain and ambiguous data", Proceedings of the International Conference on Information and Digital Technologies, IDT 2017.