Матеріалізовані представлення – це надлишково збережені в базі даних результати виконання запитів. Вони можуть бути використані для часткової або повної відповіді на запити, які будуть з'являтися в подальшому замість повторного виконання запиту з нуля. Існує велика кількість опублікованих робіт, присвячених обслуговуванню, особливо інкрементному оновленню, матеріалізованих уявлень і переписуванню запитів для їх використання. Деякі з них підтримують матеріалізовані уявлення на основі рекурсивного запиту на мові datalog. Хоча більшість datalog запитів можуть бути перетворені в SQL запити і навпаки, це не відноситься до рекурсивних запитів. Рекурсивні запити на мові datalog намагаються знайти всі можливі транзитивні замикання. Рекурсивні запити в SQL (Common Table Expression – CTE (узагальнений табличний вираз – УТВ) повертають прямі посилання, але не транзитивні замикання. У даній статті запропоновано ефективні методи інкрементного оновлення матеріалізованих уявлень на основі CTE, а також алгоритм генерації вихідних кодів на мові програмування Сі для будь-яких вхідних рекурсивних SQL запитів. Синтезовані вихідні коди реалізують запропоновані нами алгоритми інкрементного оновлення відповідно до набору вставлених/видалених/оновлених записів в базових таблицях. В даній статті основна увага приділяється рекурсивним запитам, результатами виконання яких є спрямовані деревовидні структури даних. Розглянуто два випадки вузла дерева. У першому випадку дочірній вузол має тільки один батьківський вузол, а в другому випадку дочірній вузол може мати багато батьківських вузлів. Ці два випадки представляють два типи зв'язків між сутностями в реальному світі: один-до-багатьох і багато-до-багатьох відповідно. Для зв'язку один-до-багатьох дані зв'язку супроводжуються записами, що описують дочірній елемент з використанням деяких полів. Ці поля задаються порожніми при видаленні конкретного зв'язку. Для зв'язку багато-до-багатьох, зберігаються в окремій таблиці, а конкретні зв'язки видаляються шляхом видалення описуючих записів з цієї таблиці. Розгляд забезпечення посилальної цілісності може допомогти зменшити простір пошуку і, отже, підвищити продуктивність. Проте, набором вузлів або ребер дерева можна управляти. Всі ці комбінації призводять до різних алгоритмів. Для підтвердження ефективності запропонованих в даній роботі методів наводяться й обговорюються результати експерименту

Ключові слова: матеріалізоване уявлення; рекурсивний SQL запит; CTE (УТВ); інкрементне оновлення; генерація вихідного коду

UDC 004.65
DOI: 10.15587/1729-4061.2019.180226

A SOLUTION FOR SYNCHRONOUS INCREMENTAL MAINTENANCE OF MATERIALIZED VIEWS BASED ON SQL RECURSIVE QUERY

Nguyen Tran Quoc Vinh
PhD*
E-mail: ntquocvinh@ued.udn.vn
Dang Thanh Hao
Student*
E-mail: dngthnhhao@gmail.com
Pham Duong Thu Hang
PhD Student*
E-mail: ntquocvinh@ued.udn.vn
PW Chandana Prasad
PhD, Associate Professor**
Email: CWithana@studygroup.com
Abeer Alsadoon
PhD, Associate Professor**
Email: AAlsadoon@studygroup.com
Nguyen Viet Anh
PhD
Department of Data Science and Application
Institute of Information Technology
Vietnam Academy of Science and Technology
Hoang Quoc Viet, 108, Cau Giay Dist., Hanoi City, Vietnam, 100000
E-mail: anhnv@ioit.ac.vn
*Faculty of Information Technology
The University of Da Nang — University of Science and Education
Ton Duc Thang, 459, Lien Chieu Dist.,
Da Nang city, Vietnam, 550000
**School of Computing and Mathematics, Sydney Campus
Charles Sturt University
Level 1, 63 Oxford str., Darlinghurst NSW 2010, Australia

Received date 16.07.2019
Accepted date 02.10.2019
Published date 31.10.2019

1. Introduction

There are many types of SQL queries for calculating upon data and returning execution result as a table. Some of them support select, project, joins and where predicates (SPJ queries). Some others may support group by predicate with aggregate functions. There is also a special type of query called recursive query. In SQL, it goes with Common Table Expression (CTE).

There is the kind of view as virtual by default because the query that the view is based on will be executed from the scratch each time when the view is accessed. The query

execution result is not stored anywhere. This kind of view exists and is supported parallelly by many database management systems.

From 1998, Oracle started supporting another kind of view called materialized view (MV) although its idea was raised from the 1980s. Essentially, MV is a table that stores the execution result of a query. When a future query that can partially or fully use the result stored in MV appears, that table content can be used to answer the query instead of calculation from the scratch. This may help to support real-time applications.

It is known that query execution requires system resource, mainly of CPU time, main memory and disk load. The join and group by operations of a query often require much of CPU time and memory load. Disk load often leads to bottleneck because of reading a large volume of needed data for the query execution through the low speed disk interface.

If the MV has small enough number of records that are based on a complex query, it means that including joins and group by operations will use a large volume of data to answer a query [1–9], it could improve the query execution time and required system resources in many times. The above effectiveness may be respectively multiplied with the appearance frequency of the queries that use MV. The queries are often being rewritten to be answered using MVs if possible.

However, when data in the base tables that related to MV query is changed, the stored query execution result in MV becomes inaccurate, i. e. to be inconsistent with MV definition. It must be updated to be actual to the latest data in the base tables, then it could reflect the MV definition.

There are two types of MV update according to actualization time. The first type is synchronous (immediate) that does the update of MV as a part of transaction which changes data in the base tables. The second one is asynchronous (deferred, lazy) that does the update outside of that transaction, so that the update will be done periodically with some schedule or when users call the update procedure or when the MV is accessed. Some ideas from asynchronous incremental update algorithms are useful to synchronous ones.

There are incremental (differential) update [10–15] and complete (fully refresh) update according to the ways of maintenance process. Fully refresh update clears the MV table content, then re-executes the MV query and fills the result into the MV table. Incremental update does the changes forward the MV table according to the changes in the base tables, and vise versa.

Certainly, MV brings not only benefit but also maintenance process that requires additional system resources. This might significantly impact the system performance, sometimes the maintenance cost and negative cost of change may go over the benefit. The query DBMS optimization engine must work harder with existed MVs. This additional cost comes from analysis, choosing MV for query answering and rewriting the query using the chosen MV. So that selection of views [16–19] to be materialized is a very important task. The views may be chosen to be materialized periodically and/or schedulable based on the organization working schedules [17].

All published papers are devoted to the MV with SPJ queries, queries with aggregations, recursive queries in datalog that product transitive closures. This research focuses mainly on the synchronous incremental update of MV based on recursive query (recursive MV) in SQL (CTE) whose execution results are directed tree-structured data. The algorithms undertaking incremental update of recursive MV for different combinations of the parameters are proposed:

1) set of base table records being manipulated (describing tree nodes or tree edges);

2) types of the relationship between tree nodes (one child can have one parent or many parents);

3) referential integrity (enforcing or not).

A generator that synthesizes source codes in C language for implementing proposed algorithms that supports PostgreSQL is built. Experiments on large sample database and sample recursive query in PostgreSQL to verify the rightness of the trigger function source code generating algorithm, the incremental update algorithms and the effectiveness of the proposed method are performed. Experimental results and discussion of the results are also provided.

2. Literature review and problem statement

The work [20] summarized all the aspects of MV using almost of the works which were published till 2019, reviewed them, recommended related algorithms and future directions in research and application of MV. We remind the representative papers related to synchronous incremental maintenance of MV algorithms and implementing those ones, which help additionally proving the possibility of proposed methods and understanding the provided discussions.

Obviously, there is nothing interesting in completely new updates of MV although sometimes it's effective. On the contrary, a large number of publications are devoted to incremental update of MVs [10–15, 21–25].

Data manipulation events in the base table let MV change are insert, delete and update actions. Insert operation increases a set of new records that will be inserted into the base table. Delete operation decreases a set of old records that will be removed from the base table. Update events are often divided into two sequences: delete old records from the base table and then insert new records [12–14]. Therefore, the incremental update algorithm must separately perform "delete event" procedure and "insert event" procedure. Sometimes, this division is not necessary [15]. It means that update events are manipulated in only one operation instead of two steps like above.

For SPJ views, the work [15] suggested adding at least one key of each base table into the MV, so that we can do insert, update and delete directly in MV based on the set of key values corresponding to changed records in the base table.

The work [23] approached existing incremental update methods that are proposed in the works [21] and suggested an algorithm. This algorithm bases on the concept of version store for older versions of the base tables and transaction ID in data warehouse environment. The work [26] analyzed MVs in data warehouse environment by collecting 25 papers that were published until 2010. These papers not only dedicated incremental update of MV in data warehouse environment but also in general. They showed techniques, issues addressed, changes handled, types of queries, the advantages and disadvantages of each proposed solution via tabular manner. The work [25] developed MV that are stored and incrementally updated by asynchronous way in the distributed databases based on distributed log-structured merge-tree, which provides high data write performance.

The work [24] proposed the first solution for incremental maintenance of positive nested relational calculus on bags for collection processing engine, which supports also XML data. The work [27] introduced a framework called ViewDF that focuses on the problem of incrementally propagating changes to MV according to appending a set of new data from streams. It is effective when the base tables, MV and data stream are partitioned by time such that each part can be accessed directly, and the incremental update process requires only a small number of parts.

Most of incremental update algorithms of MV using queries with aggregate functions [12, 15] (sum, count, avg, min, max...) are based on the ones of SPJ MV. The approach has its own reason. Indeed, although sometimes the aggregations can move up and down in the query tree for optimization on some classes of queries, the order of intrinsic evaluation of relational algebra expressions that contain group by operations and aggregate functions is as follows: SPJ (joins, where predicates, select) and then aggregation (group by, aggregate functions, having). It seems reasonable to build algorithms of MV incremental update with aggregations based on the ones of SPJ MV. So that, we can calculate the set of deleted (and/or inserted) records for the SPJ part, and then group by combine calculate aggregate functions. Each record in the result of this step corresponds to one record of MV.

The work [15] suggested some improvements in incremental update process of MV with aggregate functions. When referential integrity is enforced, the inserted records into parent base tables do not affect MV, so that, they can be ignored. Once an attribute only participates in the group by predicate, not in parameters of aggregate functions, its value changes in the base table can transfer directly into the corresponding attribute in MV. Otherwise, once all attributes from a key of the base table participate in group by predicate, each their value set presents a reference between a record in the base table and records in MV, so that, we can delete the corresponding records from MV directly as soon as a record is deleted from the base table. The authors also show that it is necessary to transfer the query before materialization because of usefulness of MV in future toward some cases of selection expression.

All the incremental update of recursive MV algorithms are dedicated to the type of recursive queries that they can calculate transitive closures. The paper [28] has comprehensive reviews about relative issues and recommended algorithms for recursive MVs published to 2017. It focused on DRed and EPF algorithms introduced by many works before. It also indicated that EPF algorithm is more effective than DRed algorithm. Those algorithms cannot be applied to MV based on SQL CTE. In this paper, we focus on incremental update of recursive MVs, concretely, MVs here are based on SQL recursive queries using CTE.

Although a large number of works are devoted to algorithms doing incremental update of MV, the works [15, 22, 23] showed those algorithms are implemented rarely. The work [22] presents how to manually write trigger functions and how triggers are executed when data in the base table are being changed/changed. It did not provide any incremental update algorithm as others. The work [15] shows the method to synthesize the source code of triggers and triggers functions in C programming language implementing algorithms that undertake synchronous incremental update of MVs in PostgreSQL. It provided source code generating algorithm for each MV incremental update algorithms (for events: insert, update, delete; for types of queries: SPJ, with aggregations; for cases of optimization) and the generator as well. The authors compared source codes of many trigger functions for each combination (group) of <type of query, event, update algorithm> and separated them into two types: fixed codes and variable codes. Fixed codes are the same for a group of triggers functions. Variable codes depend on the base tables, attributes, expression and data types. Their experiments show that the generated codes satisfy all the requirements, fully coincide with the triggers written manually.

Thus, there is an absolutely large number of published works relative to incremental maintenance of MV based on SPJ queries, queries with aggregations, queries with outer joins and based on recursive queries as well. In case of MV based on SPJ queries, queries with aggregations and queries with outer joins, existing solutions are effective, cover almost cases of queries and optimization. They may implement synchronous incremental update of MV based on SQL queries and datalog language automatically. The absence of good abstraction within the implementation of incremental update system may be the latest remain problem. On the other hand, in case of MV based on recursive queries, the published solutions are devoted to datalog recursive queries, which tend to transitive closures calculation and cannot apply for SQL recursive queries with CTE.

This work addresses the algorithms for incremental update of MV devoted to SQL recursive queries with CTE containing inner join SPJ queries. Furthermore, we create a generator tool that does automatic synthesis of source codes in C programming language which implements the proposed incremental update algorithms in PostgreSQL.

3. The aim and objectives of the study

The study aims to understand underlying approaches for incremental update of MV implementation. More specifically, it focuses on solving issues related to incremental update of MV which base on SQL recursive queries.

To achieve this aim, the following objectives are accomplished to:

– carefully and thoroughly review related works regarding to incremental update of MV problem, especially for the case of MV based on SQL recursive query and then formally formulate the problem;

– build the algorithms for incremental maintenance of MV with SQL recursive query basing on relational algebra;

– build the generator to synthesize source codes of trigger functions that undertake incremental update of MV with SQL recursive queries implementing the proposed algorithms;

– provide experiments to prove the accuracy of the built generator, discussions on algorithms and testing results.

4. The proposed method

4. 1. SPJ query

Each x^{th} SPJ query Q^x which x^{th} MV is based on consists of:

$$Q^x\left(S^x,T^x,J^x,W^x\right), \quad (1)$$

where:

– $S^x = \{S_1^x, S_2^x, .., S_p^x\}$ – set of fields that are selected and presented in SELECT predicates;

– $T^x = \{T_1^x, T_2^x, .., T_n^x\}$ – set of the base tables that participate in FROM predicates. FROM predicate F^x is the combination of T^x and J^x: $F^x = T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x$;

– J^x join conditions between the base tables in T^x;

– W^x – WHERE predicates, the conditions on each record in joining result of F^x. In case of implicit joins, J^x is empty and it is contained in W^x. Otherwise, it is not empty. Let $C^x = J^x \wedge W^x$. Suppose that J^x and W^x are converted into conjunctive canonical form.

4. 2. Incremental update of MV based on SPJ query

Inner joins have distributive property, so that:

$$\begin{aligned} &newF^x = \\ &= T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} \left(T_i^x \cup dnewT_i^x\right) .. \bowtie_{J_{n-1}^x} T_n^x = \\ &= T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x \cup \\ &\cup T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} T_n^x \end{aligned} \quad (2)$$

and

$$\begin{aligned} &oldF^x = \\ &= T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} \left(T_i^x \setminus doldT_i^x\right) .. \bowtie_{J_{n-1}^x} T_n^x = \\ &= T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x \setminus T_1^x \bowtie \\ &\bowtie_{J_1^i} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} doldT_i^x .. \bowtie_{J_{n-1}^x} T_n^x. \end{aligned} \quad (3)$$

It is known that a record of the j^{th} base table T_j^i can take participance in the result of Q^i if and only if its cartesian product with records of other tables in T^i satisfies J^i and W^i.

Now, suppose the current state (instance) of the database is with the set of the base tables T^x. The execution result of $Q^x\left(S^x, T^x, J^x, W^x\right)$ is:

$$M^x = \left(S^x, T^x, J^x, W^x\right). \quad (4)$$

The eq. (4) can be presented in the form of a relational algebra expression as follows:

$$M^x = \pi_{\left(S^x\right)} \sigma_{\left(W^x\right)} \left(T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x\right).$$

If there is a set of records $dnewT_i^x$ is inserted into T_i^x, suppose

$$newT_i^x = T_i^x \cup dnewT_i^x,$$

$$dnewT^x = \{T_1^x, T_2^x, ..dnewT_i^x, ..T_n^x\},$$

$$\begin{aligned} &newT^x = \{T_1^x, T_2^x, ..newT_i^x, ..T_n^x\} = \\ &= \{T_1^x, T_2^x, .. \left(T_i^x \cup dnewT_i^x\right), ..T_n^x\}. \end{aligned}$$

The database now has new instance and inferring from eq. (2)–(4), new execution result of Q^x is then:

$$\begin{aligned} &newM^x = \left(S^x, newT^x, J^x, W^x\right) = \\ &= \left(S^x, T^x, J^x, W^x\right) \cup \left(S^x, dnewT^x, J^x, W^x\right); \end{aligned} \quad (5)$$

$newM^x$ is in the form of a relational algebra expression as follows:

$$\begin{aligned} &newM^x = \\ &= \pi_{\left(S^x\right)} \sigma_{\left(W^x\right)} \left(T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x\right) \cup \\ &\cup \pi_{\left(S^x\right)} \sigma_{\left(W^x\right)} \left(T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} T_n^x\right). \end{aligned}$$

$dnewM^x = \left(S^x, dnewT^x, J^x, W^x\right)$ is the set of records that must be inserted into MV M^x according to insertion of $dnewT_i^x$ into T_i^x.

If there is a set of records $doldT_i^x$ is deleted from T_i^x, suppose

$$oldT_i^x = T_i^x \setminus doldT_i^x,$$

$$doldT^x = \{T_1^x, T_2^x, ..doldT_i^x, ..T_n^x\},$$

$$\begin{aligned} &oldT^x = \{T_1^x, T_2^x, ..oldT_i^x, ..T_n^x\} = \\ &= \{T_1^x, T_2^x, .. \left(T_i^x \setminus doldT_i^x\right), ..T_n^x\}. \end{aligned}$$

The database now has a new instance and new execution result of Q^x as in eq. (6) below:

$$\begin{aligned} &oldM^x = \left(S^x, oldT^x, J^x, W^x\right) = \\ &= \left(S^x, T^x, J^x, W^x\right) \setminus \left(S^x, doldT^x, J^x, W^x\right); \end{aligned} \quad (6)$$

$oldM^x$ is in the form of a relational algebra expression as follows:

$$\begin{aligned} &oldM^x = \pi_{\left(S^x\right)} \sigma_{\left(W^x\right)} \\ &\left(T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x\right) \setminus \pi_{\left(S^x\right)} \sigma_{\left(W^x\right)} \\ &\left(T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} doldT_i^x .. \bowtie_{J_{n-1}^x} T_n^x\right). \end{aligned}$$

$doldM^x = \left(S^x, doldT^x, J^x, W^x\right)$ is the set of records that must be deleted from MV M^x according to deleting operation of $dnewT_i^x$ from T_i^x.

If there is a set of records $doldT_i^x$ of T_i^x is updated to $dnewT_i^x$, it is equivalent with delete a set of records $doldT_i^x$ from T_i^x and then insert a new one $dnewT_i^x$ into T_i^x. The database now has new instance and new execution result of Q^x is:

$$\begin{aligned} &updM^x = \left(S^x, updT^x, J^x, W^x\right) = \\ &= \left(\left(S^x, T^x, J^x, W^x\right) \setminus \left(S^x, doldT^x, J^x, W^x\right)\right) \cup \\ &\cup \left(S^x, dnewT^x, J^x, W^x\right). \end{aligned} \quad (7)$$

It means that the update operation is converted into a delete operation followed by an insert operation. It is enough clear for MV with SPJ query and can be applied to MV with recursive query when it is converted to an iterative program.

4. 3. SQL recursive query

There may be many types of recursive queries producing different sorts of results. We focus on the most important type of recursive queries that produce the results with hierarchical or tree-structured data, which are in tabular form and can be presented by the directed tree. Suppose that there may be many trees and the trees that time can share common edges. So that, the result table does not contain duplicates. For the case of one – many relationships, i. e. one child can have only one parent, the key of the result table is

the key that identifies a unique tree node. For the many – many relationships, i.e. one child can have more than one parent, the set of attributes help link two records of the result table, i. e. describe edges of trees will be key of the result table, for example, the set {id, parentId}. Each record (child) in the result table has at least one "directed" link to (at least one) other records (parents), except the root record. The graph can be cyclic or acyclic. We choose the case of acyclic graphs first to solve the problem of incremental update of recursive MV. The general SQL recursive query may have a form as Fig. 1.

```
1:  R:  WITH RECURSIVE R AS
2:      (
3:          nrt_query
4:          UNION
5:          rt_query
6:      )
7:      SELECT * FROM R
```

Fig. 1. Original general recursive query

SQL recursive queries have two parts: non-recursive term nrt_query (anchor) and recursive term rt_query. The terms nrt_query and rt_query are SPJ queries. They can have select, project and join predicates. The rt_query has join predicates including exactly one time inner-joining with R. This constraint is given by SQL standard. Our paper considers rt_query has join predicates that contains only inner join, not outer join operations. The UNION operation discarding duplicate records is used excluding "UNION ALL" because of the chosen type of recursive queries.

R in Fig. 1 is evaluated as follows:

1) nrt_query is calculated and returns the result as a table; this table is the intermediate result of recursive query *R* too;

2) rt_query is calculated, certainly, it calls join operation with the intermediate result of recursive query *R*; empty intermediate table and fill it with the current result; do UNION operation removing any duplicates with the previous result; the process repeats until current rt_query returns empty table.

From eq. (1), (4), nrt_query and nrt_query have the form of $Q^n\left(S^n,T^n,J^n,W^n\right)$ and $Q^r\left(S^r,T^r,J^r,W^r\right)$ with the execution results $M^n=\left(S^n,T^n,J^n,W^n\right)$ and $M^r=\left(S^r,T^r,J^r,W^r\right)$ respectively. Since rt_query contains inner join recursive query, so that $T^r=\left\{T_1^r,..,T_i^x,..T_n^r,\ R\right\}$. J^r, certainly, contains joining condition between *R* and remaining in the T^r base tables. This join operation creates an edge between children created by $\left\{T_1^r,..,T_i^x,..T_n^r\right\}$ and parents created by *R*. Suppose T_i^x is a member of both T^n and T^r, $T^n=\left\{T_1^n,..,T_i^x,..T_m^n\right\}$.

The observations show that it is iterative process but not recursive essentially, recursive terminology is chosen by the SQL standards committee. It is an important confirmation and the algorithms proposed within this paper are based on it.

So that, *R* in Fig. 1 which has execution result *v* is now can be converted to the program using loop structure as in Fig. 2, in which the indexing variable *k* just has a role presenting the number of iteration. Certainly, T^r and J^r in the equation in step 6 Fig. 2 contain join operation with *v*.

The "programs" illustrated in Fig. 1, 2 are equivalent, but it seems to be unable to build an incremental update algorithm for the one in Fig. 1 because it is in the form of recursive calculation. All the algorithms developed within this paper are based on the program in Fig. 2.

```
1:  v = M^n = (S^n, T^n, J^n, W^n)
2:  k = 0
3:  M_k = v
4:  WHILE (M_k is not empty)
5:  {
6:      M_{k+1} = M^r = (S^r, T^r, J^r, W^r)
7:      v = v ∪ M_{k+1}
8:      k = k + 1
9:  }
10: Return v
```

Fig. 2. Transferred general recursive query

4. 4. Tree evaluation with recursive query

For example, we have the base table in Fig. 4, *a* with schema people(id, parentId, fullName, city). It is necessary to build a genealogy tree of persons who were born in the city "*c*1" and recursive query in Fig. 3 can support this. It returns the result in Fig. 4, *b*. The recursive query is executed and builds trees as an iterative process shown in Fig. 5.

The example shows that the execution result of one recursive query may construct more than one tree. The nrt_query creates roots of trees and decides the number of trees. The

```
1:  RQ:          WITH RECURSIVE ctefamilyinc1 AS
2:               (
3:  nrt_query:       SELECT id, parentId, fullname, city FROM people WHERE
                     city='c1'
4:                   UNION
5:  rt_query:        SELECT people.id, ctefamilyinc1.id, fullname, city FROM
                     ctefamilyinc1 join people on
                     people.parentId=ctefamilyinc1.id
6:               )
7:               SELECT * FROM ctefamilyinc1
```

Fig. 3. Example of recursive query

id	idparent	fullname	city
1	null	name1	*c*1
2	null	name2	*c*1
3	1	name3	*c*3
4	1	name4	*c*3
5	2	name5	*c*4
6	2	name6	*c*4
7	2	name7	*c*1
8	4	name8	*c*2
9	4	name9	*c*2
10	9	name10	*c*2
11	9	name11	*c*3
12	Null	Name12	*c*2
13	Null	Name13	*c*3

a

Id_mv	Idparent	id	fullname	city
1	Null	1	name1	*c*1
2	Null	2	name2	*c*1
3	2	7	name7	*c*1
4	1	4	name4	*c*3
5	1	3	name3	*c*3
6	2	6	name6	*c*4
7	2	5	name5	*c*4
8	4	9	name9	*c*2
9	4	8	name8	*c*2
10	9	11	name11	*c*3
11	9	10	name10	*c*2

b

Fig. 4. Tree evaluation by recursive query: *a* – base table people, *b* – CTE execution result

rt_query develops tree branches step by step through its iterations.

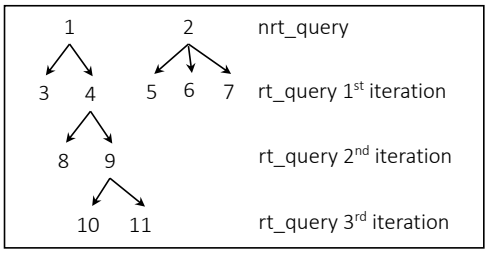

Fig. 5. Tree evaluation by recursive query

4. 5. Problem Formulation

Problem Definition. Given a SQL recursive query R that produces execution result v as tree-structured data. There is a table MV that contains and equal to execution result v. The problem is how to do synchronous incremental maintenance of MV according to data changes in the base tables that participate in R upon data manipulation events on them.

We focus on the type of recursive queries having execution result in the table which is tree-structured data, only the data create structure of the tree is interested; properties of tree nodes can be skipped. Tree structure is determined by join operation between one base table (or join expression) and the CTE. There are two considered cases of relationship between entities in real world that may create edges between tree nodes: one – many and many – many. Without loss of generality when suppose that T^n has only one or two base tables that determine the parent – child relationship, and T^r has only two or three tables (i. e. one or two base tables and the CTE itself depending on the type of relationship between entities creating the parent – child relationship). Certainly, a tree node can have many child nodes. On the opposite side, we consider two separate cases, they are:

1) a child node may have only one parent node;

2) a child node may have many parent nodes.

For one – many relationships, there are key values of record describing parent node in the record describing child. Those fields are set as null to delete the relationship. For many – many relationships, there is a record in a separate table that contains key values of both records describing children and parents. That record describing a concrete relationship is deleted to remove a relationship, i.e. a tree edge linking a child node and a parent node. Deletion of records – nodes and deletion of records – edges must be considered differently.

The data in the base tables are always in one of the two cases: either forced to ensure the referential integrity or not forced to ensure the referential integrity. Enforcing referential integrity also sets options (no action, cascade, set null) on update and delete events. The insurance of referential integrity on attributes creating tree structure is the most important issue needed to be considered. Two cases above may also lead to different algorithms for incremental maintenance of recursive MV.

Finally, there should be a solution to implement all the being developed algorithms for incremental update automatically for every SQL recursive MV. Source code synthesis may be an appreciate approach that can help to solve the problem.

4. 6. Algorithms for incremental update of recursive MV

4. 6. 1. Without enforcing referential integrity

The case that each tree child node has only one parent node is firstly considered. We try to build an algorithm for incremental update of MV for data manipulation events (insert, update, delete) on each base table.

Once a SQL recursive query is converted to an iterative program, each iteration consists of rt_query, which is an SPJ query. The eq. (5), (6) could be applied to calculate the set of records that will be inserted into/deleted from the MV table within each iteration and then for the whole iterative program. Combination of this with the eq. (7) helps to calculate the set record that will be deleted from MV and the set record that will be inserted into MV.

So that, updating a set of records $doldT_i^x$ of T_i^x to $dnewT_i^x$ can be considered equivalent to delete a set of records $doldT_i^x$ from T_i^x and then insert a new one $dnewT_i^x$ into T_i^x. The algorithm in Fig. 8 will be called and then the algorithm in Fig. 7 will be executed.

4. 6. 1. 1. Insert

Generally, the record inserted into the base table may satisfy nrt_query and/or rt_query. If it satisfies nrt_query, it creates a new root of a tree. This new root can link to already existed nodes – records in the base table to create a new tree. If it satisfies rt_query, it creates a tree node that can become child or parents of already existed nodes – records in the base table. All nodes that can be its parent have already existed in MV. All its children nodes – records are in the base table.

In the case of non-enforcing referential integrity, parentId values may be not found in id column but the base tables can contain other "sub-trees" that don't exist in MV. When a new record is inserted, it can link the tree contained already in MV with some of those sub-trees and subtree created within new inserted data. The edges with solid lines in Fig. 6 demonstrate edges that have been described already in the base tables, and the dashed lines (8→20 and 6→26) show edges described by new inserted data.

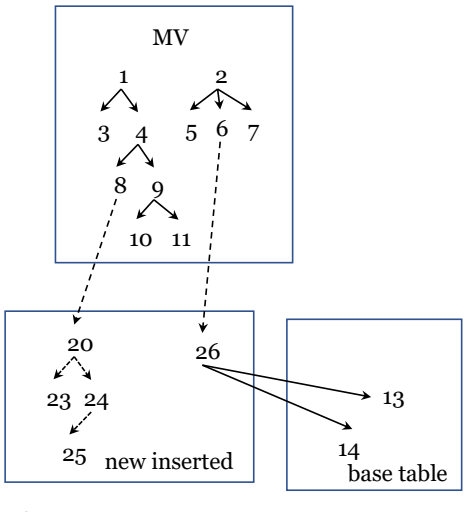

Fig. 6. Insert in case of without referential integrity

The algorithm in Fig. 7 calculates the record set that will be added into MV according to the new inserted record set $dnewv$ being inserted $dnewT_i^n$ to the base table T_i^n. $dnewv$ creates the nodes and edges that will be added into the tree. It creates a new being inserted set of roots, does union with new being inserted set of nodes that are children of already existed one in the tree nodes and then finds all nodes that are their children.

Input: M, $dnewT_i^x$, T^n, T^r
Output: *dnewv*

1: $dnewM^n = \left(S^n, dnewT^n, J^n, W^n\right) = \left(S^n, \left\{T_1^n, .., dnewT_i^x, ..T_n^n\right\}, J^n, W^n\right)$
2: $k = 0$
$dnewM^r = \left(S^r, dnewT^r, J^r, W^r\right) = \left(S^r, \left\{T_1^r, .., dnewT_i^x, ..T_n^r, M\right\}, J^r, W^r\right)$
3: $dnewv = M_k = dnewM^n \cup dnewM^r$
4: WHILE (M_k is not empty)
5: Begin
6: $M_{k+1} = M^r = \left(S^r, newT^r, J^r, W^r\right)$
$= \left(S^r, \left\{T_1^r, T_2^r, ..\left(T_i^x \cup dnewT_i^x\right), ..T_n^r, M_k\right\}, J^r, W^r\right)$
7 $dnewv = dnewv \cup M_{k+1}$
8: $k = k + 1$
9: End
10: Return *dnewv*

Fig. 7. Incremental update for insert event without enforcing referential integrity

Because of ignoring referential integrity, the searching space in step 6 includes $T_i^x \cup dnewT_i^x$ to search not only within the set of new being inserted records $dnewT_i^x$, but also the old instance of the database containing T_i^x.

4. 6. 1. 2. Delete

MV contains all possibly linked records that satisfy nrt_query and rt_query. The (being) deleted set of records $doldT_j^x$ from the base table T_j^x that satisfy nrt_query and rt_query can create nodes and branches of the tree only within existing trees in MV.

As mentioned above, because rt_query contains inner join recursive query, so that $T^r = \left\{T_1^r, .., T_i^x, ..T_n^r, R\right\}$. J^r, certainly, contains joining condition between R and remaining in the T^r basetables; once a record would like to satisfy $\left(S^r, \left\{T_1^r, .., T_i^x, ..T_n^r, R\right\}, J^r, W^r\right)$, it firstly has to satisfy $rt\_temp = \left(S^r, \left\{T_1^r, .., T_i^x, ..T_n^r\right\}, J^r, W^r\right)$ including only fields and conditions that requires only the base tables $\left\{T_1^r, .., T_i^x, ..T_n^r\right\}$. M contains all records that are the result of joining between *rt_temp* and execution result of each iteration of rt_query. Infer, M contains all records that satisfy $\left(S^r, \left\{T_1^r, .., doldT_i^x, ..T_n^r, M\right\}, J^r, W^r\right)$.

A node visually can participate in a tree many times with many roles. It may be a root node because it satisfies nrt_query and/or a descendant of other roots because it satisfies rt_query. If a node is being deleted, all its descendants must be deleted. Once a record M that creates a node of a tree is deleted, we must delete all the records that create its children and the process goes on recursively. But if a relationship is being deleted, some inferred relationships generated by rt_query will be deleted. At that time, a node can change its state of child node and becomes root if it satisfies nrt_query. For example, node with (id=k, city='$c1$') is a root that has a set of descendants {(id=k+1, city='$c2$'), (id=k+2, city='$c2$'), (id=k+3, city='$c1$'), (id=k+4, city='$c2$')}. Node (id=k+3, city='$c1$') has child (id=k+4, city='$c2$'). Recursive query builds the trees of people that has rooted born in city '$c1$'. If we delete parent-child relationship between (id=k, city='$c1$') and (id=k+1, city='$c2$'), the set of nodes {(id=k+1, city='$c2$'), (id=k+2, city='$c2$')} has to be deleted recursively from MV. But the node (id=k+3, city='$c1$') now becomes root and {(id=k+3, city='$c1$'), (id=k+4, city='$c2$')} are still retained in MV. Anyway, if the node (id=k+3, city='$c1$') itself is in the set of being deleted nodes $doldT_j^x$, then all mentioned nodes will be removed.

The algorithm in Fig. 8 calculates the record set *doldv* that is necessary to be deleted from MV according to the deletion of the record set $doldT_j^x$ from the base table T_j^x in case of that T_j^x describes tree nodes. It is the set of nodes that are necessary to be deleted from the tree. The condition M.id NOT IN dRootId is used at each iteration to skip removing the nodes that become root after deleting some relationships.

It isn't difficult to see that after step 5 in Fig. 8, *doldv* can contain many records creating the trees. Each of those trees can be a subtree of each other's. Although all duplicates will be filtered in the iterative process, omitting records that can create trees within *doldv* after this step can significantly improve the performance.

Input: $doldT_i^x, T^n, M$
Output: *doldv*

1: $doldM^n = \left(S^n, doldT^n, J^n, W^n\right) = \left(\{id\}, \left\{T_1^n, .., doldT_i^x, ..T_n^n\right\}, J^n, W^n\right)$
2: $M^n = \left(S^n, T^n, J^n, W^n\right) = \left(\{id\}, \left\{T_1^n, .., T_i^x, ..T_n^n\right\}, J^n, W^n\right)$
3: $dRootId = \left(\{M.id\}, \left\{M, M^n\right\}, \left\{M.id = M^n.Id\right\}, \{\}\right)$
$\setminus \left(\{M.id\}, \left\{M, doldM^n\right\}, \left\{M.id = doldM^n.Id\right\}, \{\}\right)$
4: $doldM^r = \left(\{M.id\}, oldT^r, J^r, W^r\right)$
$= \left(\{M.id\}, \left\{T_1^r, .., doldT_i^x, ..T_n^r, M\right\}, J^r, W^r \cup \left\{M.id\ NOT\ IN\ dRootId\right\}\right)$
5: $doldv = \left(\{M.id\}, \left\{M, doldM^n\right\}, \left\{M.id = doldM^n.Id\right\}, \left\{M.id\ NOT\ IN\ dRootId\right\}\right) \cup$
$\cup\ doldM^r$
6: $k = 0$
7: $M_k = doldv$
8: WHILE (M_k is not empty)
9: Begin
10: $M_{k+1} = M^r = \left(\{M.id\}, \left\{M, M_k\right\}, \left\{M.parentId = M_k.id\right\}, \left\{M.id\ NOT\ IN\ dRootId\right\}\right)$
11: $doldv = doldv \cup M_{k+1}$
12: $k = k + 1$
13: End
14: Return *doldv*

Fig. 8. Incremental update for delete event without enforcing referential integrity

4. 6. 2. Enforcing referential integrity

When enforcing referential integrity is set to enabled, any foreign key field must either agree with the primary key that is referenced by the foreign key or be null. It focuses on only the reference between id and parentId columns that creates links between records in MV, i. e. the branches of the trees. parentId values are always found in id column.

4. 6. 2. 1. Insert

Because of enforcing referential integrity, every record-node existed in the base tables already has a parent record-node, a new being inserted item can be a parent of other items added with or after it. So that, if the set of records $dnewT_i^x$ is inserted into the base table T_i^x, a child

of any new tree node created by any record from $dnewT_i^x$ can be only produced by some record from $dnewT_i^x$, but not from the base table T_i^x. The algorithm in Fig. 9 returns the set of records that will be inserted into MV according to insertion of the set of records $dnewT_i^x$ into the base table T_i^x.

Input:	$dnewT_i^x, M, T^n \setminus \{T_i^x\}, T^r \setminus \{T_i^x\}$
Output:	$dnewv$
1:	$dnewM^n = (S^n, dnewT^n, J^n, W^n)$
2:	$k = 0$ $dnewM^r = (S^r, \{T_1^r, ..., dnewT_i^x, ..T_n^r, M\}, J^r, W^r)$
3:	$dnewv = M_k = dnewM^n \cup dnewM^r$
4:	WHILE (M_k is not empty)
5:	Begin
6:	$M_{k+1} = M^r = (S^r, \{T_1^r, ..., dnewT_i^x, ..T_n^r, R\}, J^r, W^r)$
7:	$dnewv = dnewv \cup M_{k+1}$
8:	$k = k + 1$
9:	End
10:	Return $dnewv$

Fig. 9. Incremental update for insert event with enforcing referential integrity

The decreasing of searching space from $T_i^x \cup dnewT_i^x$ (step 6 in Fig. 7) down to $dnewT_i^x$ (step 6, Fig. 9) which is often much smaller should improve the performance significantly.

4. 6. 2. 2. Delete

There are three options (no action, cascade, set null) on update and delete events. Suppose null option is never set, so that the user must do update setting foreign key values to null before update or delete the related referenced record. Suppose that the enforcing referential integrity option set to update/delete cascade, so that deletion of a node will infer deletion of all the descendant nodes. All the being removed nodes will be in being deleted set $doldT_i^x$.

Anyway, all satisfying records – nodes that can infer from being deleted nodes have already existed in MV. The algorithm in this case is the same as when enforcing referential integrity is disabled.

4. 6. 2. 3. Update

It seems that in the case of enforcing referential integrity for delete and update events having cascade option, we cannot divide updating a set of records $doldT_i^x$ of T_i^x to $dnewT_i^x$ into the equivalent sequence of operations: 1) delete a set of records $doldT_i^x$ from T_i^x and then 2) insert a new one $dnewT_i^x$ into T_i^x and apply two algorithms for incremental update mentioned in Fig. 8, 9; but it is not true. For example, suppose that there is already the record (0, null, name0, *c*2) in the base table. We do update {(0, null, name0, *c*2), (1, null, name1, *c*1)} into {(0, null, name0, *c*1), (1, 0, name1, *c*1)}. It is equivalent to delete $doldT_i^x = \{(0, \text{null}, \text{name0}, c2), (1, \text{null}, \text{name1}, c1)\}$ and then insert $dnewT_i^x = \{(0, \text{null}, \text{name0}, c1), (1, 0, \text{name1}, c1)\}$. It seems that deleting (1, null, name1, *c*1) may call deleting the records with id 3, 4, 8, 9, 10 and 11, but it may be only true when we do remove using delete command separately. So, the algorithm for incremental update of MV is the same, i. e. apply removing the algorithm in Fig. 8 for a set of records $doldT_i^x$ from T_i^x following by the algorithm in Fig. 9 for the insertion of a new one $dnewT_i^x$ into T_i^x.

4. 6. 3. One child has many parents

We consider the case of a tree node can have only one parent node, i. e. one – many relationships between entities. Normally, a tree node has many parents, i. e. the type of relationships between entities is many – many. Then, each MV record describes an edge of the tree. The set {id, parentID} now is the key of MV table. The case is much more sophisticated and requires other algorithms for incremental maintenance for delete event.

The default understanding is removing records that describe tree nodes. In fact, we can remove edges too. In case of one – many relationships between entities, an edge can be deleted by updating parentId to NULL. In case of many – many relationships, an edge can be removed by deleting the record that describes that edge. If we delete a set of records – edges, we must detect the set of direct children within that set of edges first.

Because a node can have more than one parent, we can only remove it accordingly data changes in the base tables when it is not a child of any other ones that records of which will have to be retained. It can solve by using bag algebra, but that time it is necessary to implement some operators of this calculus, such as IN, MINUS/EXCEPT. PL/pgSQL does not support bag algebra for those operators. We have another approach using counting idea. At each iteration of procedure to calculate a set of nodes that will be removed from MV, we separate it into two sets:

1. the set of nodes that have the number of being removed parents equal to the current number of its parents;

2. the set of nodes that have the current number of parents that is greater than the number of parents being deleted. So that, we have to modify the algorithm in Fig. 8 as in Fig. 10 for the case of that:

1) one child can have more than one parent;

2) nrt_query calculates nodes that will be removed;

3) base T_j^x describes node properties.

doldv contains object ID M.oid that serves for the operation of removing records from MV. Without oid, the key {id, parentID} is used, but it would be disadvantageous when the number of being removed records is large because in this case, we have to remove each record instead of a set of records once using IN operator. If T_j^x describes child – parent relationships and/or nrt_query returns the set of being removed edges, another algorithm with modifications accordingly will be applied.

It is necessary to define the query with group by and aggregation such as $Q^x(S^x, T^x, J^x, W^x, G^x)$ to count parents of each one from children nodes. S^x now can contain aggregate functions. G^x is the set of group by fields.

Let consider the case when rt_query calculates and returns tree edges. If nrt_query determines edges, i. e. the "root edges", the algorithm in Fig. 10 must be modified, especially for steps 1–7 and step 10, as shown in Fig. 11, "Root edge" is the edge that can start a tree, i. e. the node has id equivalent to parentId of this edge that will become a root node of a tree. This time we cannot remove an edge that can become a new "root edge" but not belong to the set of "root edges" that are initially removed.

The differences between algorithms shown in Fig. 10, 11 consist in the last one has an additional step to find child nodes of edges that were returned by nrt_query. It is inferred from the difference between two nrt_query.

Input: $doldT_i^x, T^n, M$

Output: $doldv$

1: $doldM^n = \left(S^n, doldT^n, J^n, W^n\right) = \left(\{id\}, \left\{T_1^n, .., doldT_i^x, ..T_n^n\right\}, J^n, W^n\right)$

2: $M^n = \left(S^n, T^n, J^n, W^n\right) = \left(\{id\}, T^n, J^n, W^n\right)$

3: $dRootId = \left(\{M.id\}, \{M, M^n\}, \{M.id = M^n.id\}, \{\}\right)$
$\setminus\left(\{M.id\}, \{M, doldM^n\}, \{M.id = doldM^n.Id\}, \{\}\right)$

4: $k = 0$

5: $doldId_k = doldM^n$

6: $doldArcs_k = \varnothing$

7: $doldv = \varnothing$

8: WHILE ($doldId_k$ is not empty)

9: Begin

10: $doldArcs_{k+1} = M^r = \left(\{M.oid, M.id, M.parentId\}, \{M, M_k\}, \{M.parentId = M_k.id\}, \{M.id \ NOT \ IN \ dRootId\}\right)$

11: $doldParentsCnt = \left(\left\{id, count\left(parentId\right) as \ cntOldParents\right\}, \{doldArcs_{k+1}\}, \{\}, \{id\}\right)$

12: $dParentsCnt = \left(\left\{id, count\left(parentId\right) as \ cntParents\right\}, \{M\}, \{\}, \{id\}\right)$

13: $doldId_{k+1} = \left(\{doldParentsCnt.id\}, \{doldParentsCnt, dParentsCnt\}, \{doldParentsCnt.id = dParentsCnt.id\}, \{cntOldParents = cntParents\}\right)$

14: $doldv = doldv \cup doldArcs_{k+1}$

15: $k = k + 1$

16: End

17: Return $doldv$

Fig. 10. Incremental update without enforcing referential integrity for delete event

Input: $doldT_i^x, T^n, M$

Output: $doldv$

1: $doldM^n = \left(S^n, doldT^n, J^n, W^n\right) = \left(\{id, parentId\}, \left\{T_1^n, .., doldT_i^x, ..T_n^n\right\}, J^n, W^n\right)$

2: $M^n = \left(S^n, T^n, J^n, W^n\right) = \left(\{id, parentId\}, T^n, J^n, W^n\right)$

3: $dRootOid = \left(\{M.oid\}, \{M, M^n\}, \{M.id = M^n.Id \wedge M.parentId = M^n.parentId\}, \{\}\right)$
$\setminus\left(\{M.oid\}, \{M, doldM^n\}, \{M.id = doldM^n.Id \wedge M.parentId = doldM^n.parentId\}, \{\}\right)$

4: $k = 0$

5: $doldParentsCnt = \left(\left\{id, count\left(parentId\right) as \ cntOldParents\right\}, \{doldM^n\}, \{\}, \{id\}\right)$

6: $dParentsCnt = \left(\left\{id, count\left(parentId\right) as \ cntParents\right\}, \{M\}, \{\}, \{id\}\right)$

7: $doldId_k = \left(\{doldParentsCnt.id\}, \{doldParentsCnt, dParentsCnt\}, \{doldParentsCnt.id = dParentsCnt.id\}, \{cntOldParents = cntParents\}\right)$

8: $doldArcs_k = \left(\{M.oid, M.id, M.parentId\}, \{M, doldM^n\}, \{M.id = doldM^n.Id \wedge M.parentId = doldM^n.parentId\}, \{\}\right)$

9: $doldv = doldArcs_k$

10: WHILE ($doldId_k$ is not empty)

11: Begin

12: $doldArcs_{k+1} = M^r = \left(\{M.oid, M.id, M.parentId\}, \{M, M_k\}, \{M.parentId = M_k.id\}, \{M.oid \ NOT \ IN \ dRootOid\}\right)$

13: $doldParentsCnt = \left(\left\{id, count\left(parentId\right) as \ cntOldParents\right\}, \{doldArcs_{k+1}\}, \{\}, \{id\}\right)$

14: $dParentsCnt = \left(\left\{id, count\left(parentId\right) as \ cntParents\right\}, \{M\}, \{\}, \{id\}\right)$

15: $doldId_{k+1} = \left(\{doldParentsCnt.id\}, \{doldParentsCnt, dParentsCnt\}, \{doldParentsCnt.id = dParentsCnt.id\}, \{cntOldParents = cntParents\}\right)$

16: $doldv = doldv \cup doldArcs_{k+1}$

17: $k = k + 1$

18: End

19: Return $doldv$

Fig. 11. The case when nrt_query returns tree edges

4. 7. Generating source codes of triggers

Firstly, it is necessary to determine the type of relationship between entities in real world that the tree will illustrate. Once again, the join between the base tables and the CTE in rt_query decides relationship and the tree structure. If id and parentId come from one base table and only id is key of that table, then the relationship type is one – many. If id and parentId are two foreign keys of the base table referenced to two keys of two separate ones, then the relationship type is many – many. Secondly, we must analyze and confirm whether nrt_query returns tree nodes or tree edges. After that, the source code generating procedure can start.

In the framework of this research, the same technique as introduced in [15] is used for code generating. Since PostgreSQL currently supports trigger for each statement which can see the (being) changed data, the being generated triggers will be fired for each statement. The trigger headers are generated in PL/pgSQL language. The trigger functions are generated in C language. The procedure has main steps as shown in Fig. 12. A triggers and trigger functions source code generator is built as well.

The input query must have non-recursive and recursive terms that are SPJ queries with inner joins. The output source codes of trigger functions are synthesized in C language. The script describes trigger headers and does registration of triggers in PL/pgSQL. Certainly, nested query and temporary table concepts are employed here, especially, subqueries as virtual tables are used to implement the algorithms that were shown in Fig. 10, 11. The triggers are registered to be fired before events, otherwise, the algorithms must be modified accordingly concerning the instance of the base table which is being manipulated.

```
Input: SQL recursive query, base tables' metadata
Output: Trigger functions and triggers registration codes
1:  Foreach base_table doldT_i^x
2:  Begin
3:      Begin //trigger function for insert event
4:          If (enforced referential integrity is true)
5:              Generate code implements the algorithm in Fig. 9
6:          else
7:              Generate code implements the algorithm in Fig. 7
8:      End
9:      Begin //trigger function for delete event
10:         If (type of relationship is one – many)
11:             Generate code implements the algorithm in Fig. 8
12:         else
13:         Begin
14:             If (nrt_query returns a set of tree nodes)
15:                 Generate code implements the algorithm in Fig. 10
16:             Else
17:                 Generate code implements the algorithm in Fig. 11
18:         End
19:     End
20:     Begin //trigger function for update event
21:         Generate code for delete event
22:         Generate code for insert event
23:     End
24:     Generate script for compilation of trigger functions
25:     Generate trigger headers for all events
26: End
```

Fig. 12. Procedure for trigger source code generating

5. Experiments and performance evaluation

The experiments were provided. Suppose there is a table with 1,000,000 records describing tree nodes and tree edges as well and query as mentioned in section 4 for the case of that relationships type is one – many. For the case of many – many relationships, there is an additional table that describes the relationships with the number of records of 1,000,000 and the referential integrity is not enforced. Each insert/delete/update statement is issued upon 50,000 records. The being changed data is generated randomly and inserted into separate tables for insert/delete/update event accordingly, so that the process of data preparing almost does not impact on the evaluation. The being manipulated nodes/edges may be closer to the roots or the leaves of the tree. The system hardware and software configuration are Intel G4560 CPU, RAM 8GB DDR3, SSD WD Green 256GB drive, Win10 64bit, PostgreSQL v10 32bit.

The triggers and trigger functions source codes are synthesized using the built generator for any input recursive queries. Those source codes implement incremental update algorithms and are executed when data in the base tables are being inserted/updated/deleted. All synthesized codes by generator satisfy requirements and are equivalent to the codes written manually.

We tried to evaluate and compare the performance of the system while it executes the SQL commands like select, insert, delete, update when:

a) the MV are switched on;

b) the MV are switched off. Experiments are provided for different combinations of the cases:

1) the two cases of relationships between entities (One – Many and Many – Many);

2) the two cases of enforcing referential integrity; the two cases of objects to manipulate (tree-nodes and tree-edges). Execution time is milliseconds and shown in Table 1.

Table 1

Experimental results

<table>
<tr><td rowspan="3">Relationships type</td><td colspan="3" rowspan="2">One – Many</td><td colspan="4">Many – Many</td></tr>
<tr><td colspan="2">Manipulating edges</td><td colspan="2">Manipulating nodes</td></tr>
<tr><td>MV/ NoMV</td><td colspan="2">MV</td><td>Non-MV</td><td>MV</td><td>Non-MV</td><td>MV</td><td>Non-MV</td></tr>
<tr><td>Select</td><td colspan="2">603</td><td>5,682</td><td>715</td><td>3,825</td><td>715</td><td>3,825</td></tr>
<tr><td>Delete</td><td colspan="2">1,642</td><td>183</td><td>5,808</td><td>193</td><td>2,014</td><td>284</td></tr>
<tr><td>Update</td><td colspan="2">3,685</td><td>531</td><td>6,593</td><td>390</td><td>2,796</td><td>329</td></tr>
<tr><td rowspan="3">Insert</td><td colspan="2">Enforcing ref. int.?</td><td rowspan="3">797</td><td colspan="4">Not enforcing ref. int.</td></tr>
<tr><td>YES</td><td>NO</td><td rowspan="2">2,584</td><td rowspan="2">401</td><td rowspan="2">3,590</td><td rowspan="2">228</td></tr>
<tr><td>1,249</td><td>1,967</td></tr>
</table>

When the incremental maintenance of MV is switched on, certainly the triggers for each statement undertaking incremental update will be fired accordingly and it affects the performance of the data manipulating operations.

6. Discussion of the research results of the method

The experimental results confirm:

1) the correctness of the proposed incremental update of MV with recursive queries;

2) the correctness of the source code synthesizer;

3) the effectiveness of MV in terms of it can decrease the execution time of the queries significantly.

With the current experimental configuration, query execution time is different in 5.5–9.5 times. On the opposite

side, using of MV can increase data manipulating time with insert/delete/update events. For insertion of 50,000 records into the base table describing tree nodes and one – many relationships, the time is increased in 1.5–2.5 times with MV. The result shows that considering enforcing referential integrity minimizes the searching space, so that execution time is down from 2.5 times into 1.5 times. For the many – many relationships, adding 50,000 tree nodes causes augd mentation of execution time in 15.7 times versus 6.5 times when adding the same number of tree edges.

For the case of delete event, the speed is higher in 7.1–9 times for tree nodes, especially in 30.1 times for the deletion of tree edges. In the frame of this work, updating is considered as deletion next to insertion, the execution time is augmented in 7–8.5 and 16.9 times accordingly. It is proportional to augmentation of execution time of deletion plus insertion. Other data for testing may yield very different results in terms of execution time. The closer node/edge being removed to the root, the larger number of nodes/edges which are its descendants may be deleted consequently and the higher execution time certainly.

However, several challenges remain:

– It is mentioned that the result table of the recursive queries may present more than one tree separately, i. e. the MV may contain duplicates. The proposed incremental update algorithms using operations of relational algebra on set, which does not allow duplicates. So that, other algorithms must be developed for the case.

– Doing estimation and making choice between update strategies which are incremental update or full refresh can get an important role especially in case of MV based on recursive queries. If there is enforcing referential integrity, deletion of one record from the base table also can lead to deletion of a large number of records from themselves. Anyway, even in case of without enforcing referential integrity that deletion of one record does not lead to deletion of other records in the base tables, deletion of one record from the base tables may infer to deletion of a large number records from MV in case of recursive MV. For example, deletion of an edge from the graph can lead to deletion of the whole large branch from the tree. The calculation of the being deleted part with a large number of records of the base tables plus deleting process on a large number of records from MV to perform incremental maintenance may much overs the full refresh with a small number of records this time.

– The proposed incremental update of MV algorithms for the insertion event does not take place into account that case of records that are already in the MV table, which may occur when there is more than one tree and they share some tree edges. Although the duplicated records can be omitted during insertion into MV, the prodigal calculation is proportional to their volume. In the opposite side, the checking process will be repeated with the iterations.

– To simplify and improve the performance of the incremental maintenance of MV, the work [15] suggested a solution for MV based on SPJ query, so that at least one key of each base table is added into S^x and then we can manipulate records in MV directly according to the being changed records in the base table. For the case of the MV based on recursive queries, this solution can be applied but it may help improving performance only for the first from a large number of possible iterations of recursive query processing.

– The nrt_query and rt_query may be queries with aggregate functions. Anyway, they must produce summaries that can be joined with each other to build trees as the final result. Let them as SPJ queries, it is possible to create MVs for them and replace them in recursive queries by MVs accordingly. We must generate triggers undertaking incremental updates of those MVs too.

– Let consider the case that nrt_query and/or rt_query contains outer joins. Since we focus on the recursive query that produces tree-structure data, the join between CTE and the remaining part in rt_query must be inner join. Now, we can modify the algorithms proposed in this paper to use the algorithms for incremental update of MV with outer joins.

7. Conclusions

1. We suggested to transfer recursive queries into the iterative process and then proposed the algorithms for incremental maintenance of MV with SQL recursive query considering the combinations of cases:

1) enforcing referential integrity of data or not;

2) types of relationships between entities in the real world – one – many and many – many;

3) manipulating of records that describe tree nodes or tree edges.

2. We built a generator that synthesizes source codes of trigger functions in C language and script that does triggers registration in PL/pgSQL implementing the proposed algorithms. The input SQL recursive query is the one with inner join SPJ queries in both of non-recursive and recursive terms, but it can be extended to support outer join and aggregation.

3. We provided the experiments to prove the accuracy of the proposed algorithms, built the generator and did exhausted discussion on the execution result of the experimentally synthesized source codes and the proposed algorithms as well. The MV helps to improve the queries' execution speed in 5.5–9.5 times. On the opposite side, MV slows down data insertion in 1.5–15.7 times and 7.1–30.1 for deletion.

4. However, some challenges on incremental maintenance of MV with recursive queries exist and query rewriting problem to use those MVs to answer queries is opened for future work.

Acknowledgment

This research is supported by the Ministry of Education and Training (Vietnam) under the grant B2017.DNA.06_KYTH-01 "Building a system for supporting materialized views in open source database management systems".

References

1. Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., Urata, M. (2000). Answering complex SQL queries using automatic summary tables. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data - SIGMOD'00. doi: https://doi.org/10.1145/342009.335390

2. Goldstein, J., Larson, P.-Å. (2001). Optimizing queries using materialized views. Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data - SIGMOD'01. doi: https://doi.org/10.1145/375663.375706

3. Halevy, A. Y. (2001). Answering queries using views: A survey. The VLDB Journal, 10 (4), 270–294. doi: https://doi.org/10.1007/s007780100054
4. Park, C.-S., Kim, M. H., Lee, Y.-J. (2002). Finding an efficient rewriting of OLAP queries using materialized views in data warehouses. Decision Support Systems, 32 (4), 379–399. doi: https://doi.org/10.1016/s0167-9236(01)00123-3
5. Chirkova, R., Li, C., Li, J. (2005). Answering queries using materialized views with minimum size. The VLDB Journal, 15 (3), 191–210. doi: https://doi.org/10.1007/s00778-005-0162-8
6. Ileana, I., Cautis, B., Deutsch, A., Katsis, Y. (2014). Complete yet practical search for minimal query reformulations under constraints. Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data - SIGMOD'14. doi: https://doi.org/10.1145/2588555.2593683
7. Afrati, F., Chandrachud, M., Chirkova, R., Mitra, P. (2009). Approximate Rewriting of Queries Using Views. Lecture Notes in Computer Science, 164–178. doi: https://doi.org/10.1007/978-3-642-03973-7_13
8. Larson, P.-Å., Zhou, J. (2006). View matching for outer-join views. The VLDB Journal, 16 (1), 29–53. doi: https://doi.org/10.1007/s00778-006-0027-9
9. Cohen, S., Nutt, W., Sagiv, Y. (2006). Rewriting queries with arbitrary aggregation functions using views. ACM Transactions on Database Systems, 31 (2), 672–715. doi: https://doi.org/10.1145/1138394.1138400
10. Chen, S., Rundensteiner, E. A. (2005). GPIVOT: Efficient Incremental Maintenance of Complex ROLAP Views. 21st International Conference on Data Engineering (ICDE'05). doi: https://doi.org/10.1109/icde.2005.71
11. Lee, K. Y., Kim, M. H. (2005). Optimizing the incremental maintenance of multiple join views. Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP - DOLAP. doi: https://doi.org/10.1145/1097002.1097021
12. Gupta, H., Mumick, I. S. (2006). Incremental maintenance of aggregate and outerjoin expressions. Information Systems, 31 (6), 435–464. doi: https://doi.org/10.1016/j.is.2004.11.011
13. Larson, P.-Å. (2018). Maintenance of Materialized Views with Outer-Joins. Encyclopedia of Database Systems, 2165–2170. doi: https://doi.org/10.1007/978-1-4614-8265-9_841
14. Nica, A. (2012). Incremental maintenance of materialized views with outerjoins. Information Systems, 37 (5), 430–442. doi: https://doi.org/10.1016/j.is.2011.06.001
15. Quoc Vinh, N. T. (2016). Synchronous incremental update of materialized views for PostgreSQL. Programming and Computer Software, 42 (5), 307–315. doi: https://doi.org/10.1134/s0361768816050066
16. Gupta, H., Mumick, I. S. (2005). Selection of views to materialize in a data warehouse. IEEE Transactions on Knowledge and Data Engineering, 17 (1), 24–43. doi: https://doi.org/10.1109/tkde.2005.16
17. Kungurtsev, O. B., Vozovikov, Y. N., Vinh, N. T. Q. (2012). Determination Of The Parameters of Periodic On / Off Materialized View in the Information System. Eastern-European Journal of Enterprise Technologies, 4 (2 (58)), 42–45. Available at: http://journals.uran.ua/eejet/article/view/4217/3980
18. Novokhatska, K., Kungurtsev, O. (2016). Developing methodology of selection of materialized views in relational databases. Eastern-European Journal of Enterprise Technologies, 3 (2 (81)), 9–14. doi: https://doi.org/10.15587/1729-4061.2016.68737
19. Novokhatska, K., Kungurtsev, O. (2016). Application of Clustering Algorithm CLOPE to the Query Grouping Problem in the Field of Materialized View Maintenance. Journal of Computing and Information Technology, 24 (1), 79–89. doi: https://doi.org/10.20532/cit.2016.1002694
20. Sebaa, A., Tari, A. (2019). Materialized View Maintenance: Issues, Classification, and Open Challenges. International Journal of Cooperative Information Systems, 28 (01), 1930001. doi: https://doi.org/10.1142/s0218843019300018
21. Zhou, J., Larson, P.-A., Elmongui, H. G. (2007). Lazy maintenance of materialized views. Proceedings of the 33rd international conference on Very large data bases. Vienna, 231–242. Available at: http://www.vldb.org/conf/2007/papers/research/p231-zhou.pdf
22. Chak, D. Materialized views that work. Available at: https://www.pgcon.org/2008/schedule/attachments/64_BSDCan2008-MaterializedViews-paper.pdf
23. Almazyad, A., Siddiquim, M. K. (2010). Incremental View Maintenance: An Algorithmic Approach. International Journal of Electrical & Computer Sciences IJECS-IJENS, 10 (03), 16–21.
24. Koch, C., Lupei, D., Tannen, V. (2016). Incremental View Maintenance For Collection Programming. Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems - PODS'16. doi: https://doi.org/10.1145/2902251.2902286
25. Duan, H., Hu, H., Qian, W., Ma, H., Wang, X., Zhou, A. (2018). Incremental Materialized View Maintenance on Distributed Log-Structured Merge-Tree. Lecture Notes in Computer Science, 682–700. doi: https://doi.org/10.1007/978-3-319-91458-9_42
26. Jain, H., Gosain, A. (2012). A comprehensive study of view maintenance approaches in data warehousing evolution. ACM SIGSOFT Software Engineering Notes, 37 (5), 1. doi: https://doi.org/10.1145/2347696.2347705
27. Yang, Y., Golab, L., Tamer Ozsu, M. (2017). ViewDF: Declarative incremental view maintenance for streaming data. Information Systems, 71, 55–67. doi: https://doi.org/10.1016/j.is.2017.07.002
28. Dietrich, S. W. (2017). Maintenance of Recursive Views. Encyclopedia of Database Systems, 1–7. doi: https://doi.org/10.1007/978-1-4899-7993-3_842-2