Матеріалізовані представлення можуть значно підвищити швидкість виконання запитів, використовуючи всі або частину збережених попередньо обчислених результатів запитів. При інкрементному обслуговуванні матеріалізовані представлення оновлюються відповідно до змін у відповідних базових таблицях. Часто це більш ефективно, ніж повне оновлення, що замінює таблиці матеріалізованих уявлень новим результатом виконання запиту. Асинхронне обслуговування, що приводить матеріалізовані представлення до фактичного стану, не є складовою частиною транзакції, що вносить зміни в базові таблиці. Більшість опублікованих робіт присвячено синхронному інкрементному оновленню представлень, алгоритми якого вимагають доступу до стану базових таблиць перед оновленням і не можуть застосовуватися безпосередньо до асинхронним оновлень, які виконуються в стані після оновлення. Кілька робіт присвячено асинхронному обслуговуванню представлень, або обмежують зміни тільки в одній з базових таблиць, або передбачають наявність лише однієї базової таблиці, що недоцільно, або невірно, або надає занадто високий рівень і складні алгоритми інкрементного оновлення, або може бути реалізовано, тільки якщо система управління базами даних підтримує управління версіями даних на рівні таблиць і рядків. У даній роботі запропоноване рішення для асинхронного інкрементного оновлення представлень, яке може бути реалізовано з будь-якими системами управління базами даних. Ми збираємо зміни в базових таблицях, отримуємо доступ до стану перед оновленням базових таблиць, використовуючи процес ущільнення, і застосовуємо алгоритми інкрементного обслуговування перед оновленням для асинхронного обслуговування до стану базових таблиць після оновлення, враховуючи особливості асинхронного обслуговування. Це може бути застосовано для запитів SPJ з внутрішніми з'єднаннями, запитів з внутрішніми з'єднаннями і агрегатами. Створено прототип та наведено експерименти з автоматичної генерації вихідних кодів на мові Сі для збору змін в базових таблицях і виконання асинхронного інкрементного оновлення матеріалізованих представлень в PostgreSQL

Ключові слова: матеріалізоване представлення, стан перед оновленням, асинхронне інкрементне обслуговування, синтез вихідного коду, PostgreSQL

UDC 004.65
DOI: 10.15587/1729-4061.2020.193715

A NEW SOLUTION FOR ASYNCHRONOUS INCREMENTAL MAINTENANCE OF MATERIALIZED VIEWS

Nguyen Tran Quoc Vinh
PhD*
E-mail: ntquocvinh@ued.udn.vn
Tran Trong Nhan
Master of Science in Computer Science*
E-mail: trongnhan.tran93@gmail.com
Le Van Khanh
Master of Computer Science*
E-mail: khanhlv@lekhanhtech.com
Tran Dang Hung
PhD, Associate Professor
Faculty of Information Technology
Hanoi National University of Education
Xuan Thuy str., 136, Cau Giay District, Hanoi, Vietnam, 110000
Email: hungtd@hnue.edu.vn
Abeer Alsadoon
PhD, Associate Professor**
Email: AAlsadoon@studygroup.com
PW Chandana Prasad
PhD, Associate Professor**
Email: CWithana@studygroup.com
Pham Duong Thu Hang
PhD student*
E-mail: ntquocvinh@ued.udn.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, Oxford str., 63, Darlinghurst NSW 2010, Australia

Received date 22.12.2019
Accepted date 24.01.2020
Published date 28.02.2020

1. Introduction

Materialized views (MV) are special tables where the query execution results are stored, which can be used to answer other queries that appear later. MV is critical and raises many research questions, such as MV maintenance, including incremental update and maintaining strategies, using MV to answer queries and optimization in different fields, in traditional ones such as data warehouse, data streaming, web and semantic web, distributed system and concern future

directions such as IoT, business intelligence and analysis, emergent web application [1].

It is necessary to make MV actual to the base tables after data changes on them. Incremental update of MV calculates the part of records needed to be removed or inserted into MV according to the changed records in the base tables. The updating process can be done in a synchronous or asynchronous manner. Synchronous (eager, immediate) update is performed within the transaction which changes data in the base tables. In opposite, asynchronous (deferred, lazy) is done outside of those transactions, by user requests, on-demand when a query that uses the MV appears, periodically or by some schedule. Synchronous maintenance of MV can be performed before (more often) or after the base tables are updated, i. e. on the pre-update or post-update states of the base tables and often after changes in each base table. On the opposite, asynchronous one is done on the post-update state of base tables and often after data changes in more than one table.

Asynchronous incremental maintenance of MV is important. There are a number of works dedicated to the asynchronous incremental update of views, which either restrict changes in only one of base tables or assume the query uses only one base table, which is impractical, either is wrong, either provides too high level and complicated incremental update algorithms or can be implemented only if the database management system supports data versioning at the table and row levels. Most of database management systems don't support data versioning at the table and row levels and access to full system transaction log is not always available. Therefore, it is necessary to develop a new technique to incrementally maintain MV in asynchronous manner that can be applied and implemented to any database management system.

2. Literature review and problem statement

Most of the works are devoted to the synchronous incremental update [2–7], a little number of them are dedicated to the asynchronous incremental update of MV [8–10]. The works [5, 11] build and provide experiments on triggers in programming languages PL/pgSQL and C, implementing the same incremental update algorithms in a synchronous manner on the same MV based on the same query with the same base tables with the clustered indices on primary keys and foreign keys. It is shown that triggers written in C are more effective by about 13 %. The work [5] suggests a sequence of optimizations to the incremental update algorithm relative to [11]. The work [2] proposes a solution for synchronous incremental of recursive materialized view based on SQL query. Although it may be more effective if the incremental update code is embedded into a database management system, solution using triggers has its advantages because it is almost independent of database management system versions and has higher mobility.

It is clear that data in base tables are different at the two states that lead to the necessity of different incremental update algorithms, at least the calculations of the sets of records need to be removed from and/or inserted to the MV according to the data changes made in the base tables. However, most of the published researches didn't care about those different states, which leads to state bug, so that the incremental update algorithms proposed by those works may be wrong [9]. The work [9] develops a new algorithm especially for asynchronous maintenance with different update strategies for helping to reduce the time the MV tables and base tables are locked, which may affect other applications that use the tables negatively. The solution proposed by this paper seems to be very complicated and very difficult to implement. It also doesn't show how to access the pre-update state of the base table to calculate the delta stream for MV.

After the work [8], almost asynchronous incremental maintenance is studied for warehousing and distributed environment [12–16] employing a versioning mechanism of the database management systems. The work [12] delivers that in the very large scale distributed shared-nothing data storage systems, remote view tables are better for index, equijoin, and selection MV, while local ones are more useful for MV with aggregations so that those two types of MV implementation can provide a good tradeoff between system throughput and minimizing view staleness. The work [13] shows the effectiveness of combining eager and lazy MV maintenance policies for social networking applications on a database management system using a distributed memory cache, but there is a tradeoff between accuracy and freshness of the MV. The work [14] tends to find the cost metric which helps to balance the accuracy, the latency, and the source utilization effectively for updating policies of MV with joins on continuous queries over data stream and linked data.

The work [16] utilizes the versions store and deferred maintaining mechanism of the work [8] and approach proposed by the work [17] to build an incremental update algorithm using version store in a warehousing environment. The work [8] exploits the ability of Microsoft SQL Server (from ver. 2005) that supports data versioning at the table level down to record level to access the pre-update state of base tables. That paper also provides condense operators to "compress" the record sets dT_i^x removing intermediate operations on a base table T_i^x record as an optimization, which can help to dramatically reduce the size of the sets of changed record sets in base tables to be processed in deferred maintaining mode. It improves performance. The experiments provided by the paper show that asynchronous maintenance is more effective than synchronous one even in on-demand mode, i. e. the asynchronous update is performed when there is a query that uses the MV appears. The work [15] states the same. In fact, if the data changes in base tables are accumulated too much, the incremental update process may take too much time and system resources, the query owner has to wait longer and there may be also a negative effect on other applications executing parallelly.

According to optimizations suggested by the work [5], the positive effects are gained from condense operators that may be not so high as obtained in the work [8]. The reason is that changes in base tables on attributes participating in GROUP BY clause but not in WHERE clause and are not parameters of an aggregate function can be applied directly to the MV without any relational algebra expression calculation. Anyway, the solution can be applied only to the database management systems which support data versioning to the table and record levels with the ability to access full system transaction log as needed.

PostgreSQL is the world's advanced open source relational database management system. It supports materialized view with the asynchronous full update of MV, which re-executes the underlying query and replaces the previous result in the MV table. As almost open source database management systems, PostgreSQL doesn't support data

versioning on the table and record levels. So, we can't apply the solution proposed in the work [8] to implement support of the asynchronous incremental update of MV.

Although spending on system resource among the system operational time with the transactions is one of the advantages of synchronous incremental maintenance of MV, it always exploits system resource so can affect the system negatively. It can lengthen the transaction execution time too much especially when multiple MV are affected and needed to be updated, which is not acceptable in many applications. In some cases, for example in data warehouse and distributed systems, the base tables are not always available for synchronous maintenance. On the opposite, asynchronous maintenance may allow MV to be not actual to data in the base tables in some period between the last update time to the next update time. Thus, asynchronous maintenance is very important and needs to be studied and implemented.

3. The aims and objectives of the study

This research aims to find a solution for asynchronous incremental maintenance of MV which can be applied to any database management system. The following objectives are established and archived to reach the aims:

– carefully study the related published works regarding the incremental update of MV at all and asynchronous incremental update in particular;

– formally show the state bug error and propose a technique to access the pre-update state of base tables exploiting the condensing process which avoids the state bug error;

– build the updating expressions accessing the pre-update state of base tables and the mechanism of asynchronous incremental maintenance which can be implemented with every database management system;

– build a prototype for implementing the proposed solution with PostgreSQL.

4. Proposed method

4 .1. SQL query

It is necessary to formulate the SQL queries that are used to create MV. Each x^{th} SPJ query Q^x which x^{th} MV is based on consists of:

$$Q^x\left(S^x,T^x,J^x,W^x\right), \tag{1}$$

where:

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

– $T^x=\left\{T_1^x,T_2^x,..,T_n^x\right\}$ – set of 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 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 a conjunctive canonical form.

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

$$Q^x\left(S^x,T^x,J^x,W^x,G^x\right), \tag{2}$$

where S^x now can contain aggregate functions and G^x is the set of groups by attributes.

4. 2. Incremental update of MV and state bug

The work [9] first shows examples about state bug when the pre-update incremental maintenance algorithms are applied to the post-update state of the database in the asynchronous incremental update of MV. In this paper, this problem is shown in a formal form.

The standard execution order for SPJ query is join operations, where predicates and then selection. The standard execution order for queries with aggregations is join operation, where predicates, group-by operations and aggregate functions and then selections. Incremental update of those both MV types is based on the distributive property of inner join operation of relational algebra. It is known that a record of a i^{th} base table T_i^x can take participance in the result of Q^x if and only if its cartesian product with records of other tables in T^x satisfies J^x and W^x. So that we will give proof that the pre-update algorithms cannot be applied to the post-update state directly only for the case of SPJ MV defined in (1). Suppose there is a record set $dnewT_i^x$ inserted into the base table T_i^x, then,

$$\begin{aligned}&newF^x=\\&=T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} \left(T_i^x \cup dnewT_i^x\right) .. \bowtie_{J_{n-1}^x} T_n^x=\\&=T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x \cup\\&\cup T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} T_n^x\end{aligned} \tag{3}$$

and there is a record set $doldT_i^x$ deleted from the base table T_i^x, then,

$$\begin{aligned}&oldF^x=\\&=T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} \left(T_i^x \setminus doldT_i^x\right) .. \bowtie_{J_{n-1}^x} T_n^x\\&=T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} T_i^x .. \bowtie_{J_{n-1}^x} T_n^x \setminus\\&\setminus T_1^x \bowtie_{J_1^x} T_2^x \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} doldT_i^x .. \bowtie_{J_{n-1}^x} T_n^x.\end{aligned} \tag{4}$$

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

$$M^x=\left(S^x,T^x,J^x,W^x\right). \tag{5}$$

The eq. (5) can be presented in the form of a relational algebra expression as follows:

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

If there is a set of records $dnewT_i^x$ inserted into T_i^x, which move the table from the pre-update state to the new post-update state, suppose $postT_i^x=preT_i^x\cup dnewT_i^x$, $dnewT^x=\left\{preT_1^x,preT_2^x,..dnewT_i^x,..preT_n^x\right\}$,

$$postT^x = \{preT_1^x, preT_2^x, ..postT_i^x, ..preT_n^x\} =$$
$$= \{preT_1^x, preT_2^x, ..(preT_i^x \cup dnewT_i^x), ..preT_n^x\}.$$

The database now has a new instance and inferring from eq. (3)–(5), a new execution result of Q^x is then:

$$postM^x = (S^x, postT^x, J^x, W^x) =$$
$$= (S^x, preT^x, J^x, W^x) \cup (S^x, dnewT^x, J^x, W^x); \quad (6)$$

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

$$postM^x = \pi_{(S^x)}\sigma_{(W^x)} \begin{pmatrix} preT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup preT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \end{pmatrix}.$$

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

Suppose an MV query has two base tables T_1^x and T_2^x changed by insert operations between two sates, pre-update and post-update, of the database. The record set that must be inserted to MV with SPJ query according to data changes in base tables is:

$$dnewpreM^x = \pi_{(S^x)}\sigma_{(W^x)} \begin{pmatrix} dnewT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup preT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup dnewT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \end{pmatrix}. \quad (7)$$

If the pre-update expression (7) is applied directly to the post-update state of the database, we have:

$$dnewpostM^x = \pi_{(S^x)}\sigma_{(W^x)} \begin{pmatrix} dnewT_1^x \bowtie_{J_1^x} (preT_2^x \cup dnewT_2^x) \bowtie \\ \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup (preT_1^x \cup dnewT_1^x) \bowtie_{J_1^x} dnewT_2^x \bowtie \\ \bowtie_{J_2^x} .. \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup dnewT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \end{pmatrix}. \quad (8)$$

The record set that must be inserted to MV with SPJ query according to data changes in base tables in every case must be identified. But the result of expression (8) is wrong and now they are different in expressions (7) and (8):

$$dnewpostM^x \setminus dnewpreM^x =$$
$$= \pi_{(S^x)}\sigma_{(W^x)} \begin{pmatrix} dnewT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \end{pmatrix}. \quad (9)$$

In general, if all n base tables have changed with insertions, it is not difficult to prove that:

$$dnewpostM^x \setminus dnewpreM^x = \pi_{(S^x)}\sigma_{(W^x)} \begin{pmatrix} dnewT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup dnewT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cdots \\ \cup dnewT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} dnewT_n^x \cup \\ \cdots \\ \cup preT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} dnewT_n^x \cup \\ \cup preT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} dnewT_n^x \cup \\ \cdots \\ \cup dnewT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} dnewT_i^x .. \bowtie_{J_{n-1}^x} dnewT_n^x \end{pmatrix}. \quad (10)$$

$dnewpreM^x$ and $dnewpostM^x$ must be identical, but expressions (9) and (10) show the opposite. It means that the expression (8) is wrong, we can't directly apply pre-update MV incremental maintenance algorithms to the post-update state of the database. It is analogical for delete operations.

4. 3. Proposed technique

Suppose we have a base table T_i^x with the pre-update state $oldT_i^x$. There is a record set dT_i^x which moves T_i^x from the pre-update state $preT_i^x$ to the post-update state $postT_i^x$. Certainly, records in dT_i^x are with action information (inserted, deleted) and are ordered ascendingly by the time of the operations.

It is necessary to access the pre-update state of base tables to do incremental maintenance of MV. In this paper, we exploit the idea of condensing process mainly to access the pre-update state of the base tables, so that the state bug error is avoided and the pre-update expressions can be used to do the asynchronous incremental update of MV at the post-update state of base tables. Certainly, we use the condensing process to "compress" the record set dT_i^x omitting intermediate changes of records which are deleted from/updated in/inserted into base tables and then separate it to $doldT_i^x$ and $dnewT_i^x$ to access the pre-update state of T_i^x from its post-update state.

4. 3. 1. Condense processing

Update operation is divided to delete operation following by insert operation. Each originating base table record can have one from 6 updating scenarios between the pre-update and post-update states of the database. Each originating record

may have many versions between the two states of a base table. For the base table T_i^x, dT_i^x contains the set of records that are deleted from and/or inserted to T_i^x, which moves T_i^x from the pre-update state $preT_i^x$ to the post-update state $postT_i^x$.

Scenario 1.

There is only one inserted record with one insertion operation, i. e. it is inserted and isn't changed between the two states of a base table. There is nothing to omit here.

Scenario 2.

There is only one deleted record by one deletion action. There is nothing to omit here.

Scenario 3.

First: Inserted.

Intermediate: Deleted.... Inserted (new record with the same key values).

Last: Deleted.

For this scenario, it is clear that all the operations do not have any influence on the 'final' post-update state of the base table although they can affect the intermediate states of the base table between the pre-update and post-update states. Since we are interested in the post-update state of T_i^x, so we can omit all the operations removing all the related records from dT_i^x.

Scenario 4.

First: Inserted.

Intermediate: Deleted...Inserted...Deleted...

Last: Inserted (new record with the same key values).

For this scenario, only the last operation influences the post-update state of T_i^x. So we remove all those records from dT_i^x relating to first and intermediate operations, keeping only the last one.

Scenario 5.

First: Deleted.

Intermediate: Inserted (new record with the same key values)...

Last: Deleted.

The first deleted record was already in the pre-state of the base table T_i^x, removing it may affect the post-update state of T_i^x. All the intermediately inserted records are deleted between the pre-update and post-updates of T_i^x, the operations are self-compensated each to other. We can remove all those records from dT_i^x keeping only the first one relating to the first deletion. In this case, the work [8] suggests keeping the last deletion instead of the first one.

Scenario 6.

First: Deleted.

Intermediate: Inserted.... Deleted...

Last: Inserted (new record with the same key values).

The first deleted record was already in pre-state of base table T_i^x, removing it may affect the post-update state of T_i^x. Once the last inserted record is not duplicate of the first deleted one, the two records with the same key values have different values for other attributes. In this case, we must keep the first deleted and the last inserted records, removing all records relating to intermediate operations.

4. 3. 2. Accessing the pre-update state of base tables

Before the condensing process, there may be many records in dT_i^x corresponding to a base table record with concrete key values, we can't re-order them within a scenario because it can lead to another scenario and yield another result. For example, within a scenario, we can't do all the delete operations and then insertions, because there may be nothing to delete since the records are not inserted and inserted records need to be removed respectively to a deletion still are kept in dT_i^x.

Without generality, after condensing process, there are three cases of record from dT_i^x must be considered:

– an originating record is inserted into T_i^x represented by t_j^i;

– an originating record is deleted from T_i^x represented by t_j^d;

– an originating record is deleted from T_i^x and then another one is inserted into T_i^x represented by t_j^{ud} and t_j^{ui} respectively.

Note that those records t_j^d, t_j^i and the pair of t_j^{ud} plus t_j^{ui} are independent of each other. Because relational algebra doesn't care about the order of the records in a set, till now, we can re-order them randomly keeping the order of t_j^{ud} and t_j^{ui}. It is clear that we can't re-order t_j^{ud} and t_j^{ui}, which can lead to wrong post-update state of T_i^x. If we divide dT_i^x into the set of deleted records $doldT_i^x$ and the set of inserted records $dnewT_i^x$, i. e. we have $doldT_i^x = \{t_j^d \mid j = 1..k\}$, $dnewT_i^x = \{t_j^i \mid j = 1..l\}$, $t_j^{ud} \in doldT_i^x$, $t_j^{ui} \in dnewT_i^x$ and $doldT_i^x$ is following by $dnewT_i^x$, then the order of the records within each set is not important. This time,

$$postT_i^x = \left(preT_i^x \setminus doldT_i^x\right) \cup dnewT_i^x, \tag{11}$$

and

$$preT_i^x = \left(postT_i^x \setminus dnewT_i^x\right) \cup doldT_i^x. \tag{12}$$

At the post-update state of T_i^x, $postT_i^x$ is already in the database.

4. 3. 3. Update expressions

It is not difficult to refer from expression (7) and we adopt the update expression $dnewM^x$ from [8] ignoring the order of records in dT_i^x in case of all n base tables are updated. The expressions (14) and (13) show $doldM^x$ and $dnewM^x$ – the set of records to be deleted and then inserted into the MV table for SPJ query based MV according to the changes made moving base tables from the pre-update state to the post-update one.

$$dnewM^x = \pi_{(S^x)}\sigma_{(W^x)}\begin{pmatrix} dnewT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup postT_1^x \bowtie_{J_1^x} dnewT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ ... \\ \cup postT_1^x \bowtie_{J_1^x} postT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} postT_i^x .. \bowtie_{J_{n-1}^x} dnewT_n^x \end{pmatrix} \tag{13}$$

and

$$doldM^x = \pi_{(S^x)}\sigma_{(W^x)}\begin{pmatrix} doldT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ \cup preT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} preT_n^x \cup \\ ... \\ \cup preT_1^x \bowtie_{J_1^x} preT_2^x \bowtie_{J_2^x} .. \bowtie \\ \bowtie_{J_{i-1}^x} preT_i^x .. \bowtie_{J_{n-1}^x} doldT_n^x \end{pmatrix}. \tag{14}$$

The expressions for MV based on a query with aggregative functions are different but can be referred analogically.

4. 3. 4. Used asynchronous incremental update algorithms

The incremental update algorithms used for MV based on SPJ query and MV based on a query with aggregations are the same and adopted from the work [5]. They are not repeated here to reduce the length of the paper. Anyway, the algorithms for the asynchronous maintenance must have specifics:

– changed data is collected by triggers which are fired on each data manipulation event for each base table with the transaction id and the timestamp the insert/update/delete statement started;

– the sets of changed rows in base tables must be condensed;

– the expression (13) is applied to one set of inserted into base tables records instead of (7) and similarly for the set of deleted from base table records, the expression (14) is used. It is the main difference between synchronous and asynchronous incremental update of MV;

– applied for the cases when there may be more than one base table changed at a time;

– the pre-update and post-update states are marked for each base table during each asynchronous update for an MV.

4. 4. System model

The system for asynchronous incremental maintenance of MV is suggested to consist of three components (Fig. 1):

i) MV manager;

ii) asynchronous incremental update manager.

MV manager i) analyses the input MV query getting meta-data about base tables (key, detail information about attributes...) from the database, ii) generates triggers and asynchronous incremental update source in C and trigger registration code in SQL and iii) updates MV configuration to the database and create the MV table. The generated triggers on each of insert/update/delete events for each base table which will gather the changed records with transaction id, data manipulating statement timestamp and action information in base tables and will save into the dT_i^x tables. This process is synchronously done within the transaction which makes changes in the base tables. It is committed or rollbacked together with the main transaction.

The source code in C is synthesized similarly as in the works [2, 5], will be compiled and linked, then saved in the form of dynamic link library (.dll). The MV manager uses generated SQL script and compiled trigger functions code in .dll to register trigger. All the codes implementing expressions (13) and (14) are synthesized at this stage. The structure "...((SELECT all_columns FROM new_table EXCEPT SELECT all_collums FROM dnew_table) UNION SELECT all_collums FROM dold_table) AS table..." is used to implement expression (12) to access the pre-update state of a base table.

The MV manager also creates the structure for the tables that contain MV configurations and information if needed. It also creates tables to store the changed data in the base table during the process of trigger definition.

The code that implements the asynchronous incremental update of MV is saved in the library and used by the update manager. When an MV update arrived, the update manager will mark the pre-update and post-update state for the changed base tables, mark the current point of auxiliary tables corresponding to those changed base tables for current MV, do the condensing process and then invoke the save code to undertake incremental maintenance of MV.

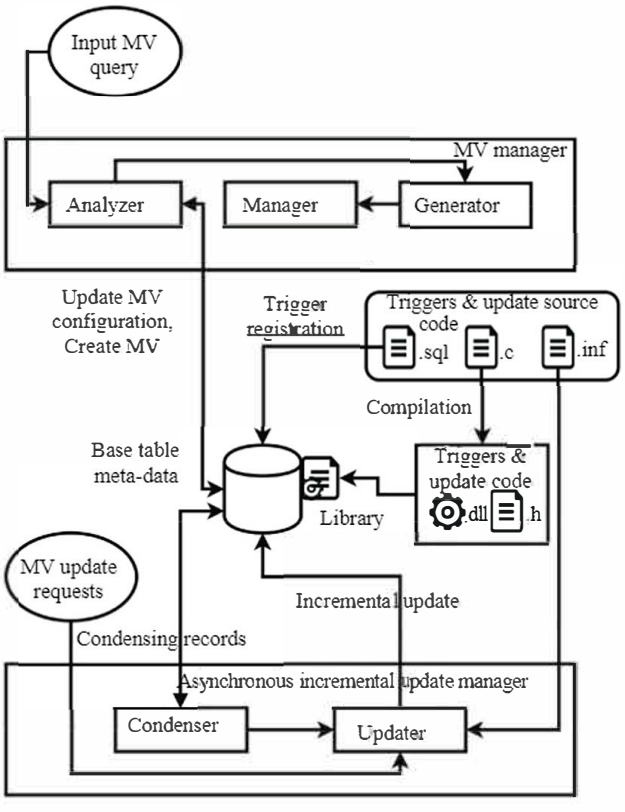

Fig. 1. System model

5. Experiments on changes gathering and asynchronous incremental update

PostgreSQL is chosen as a database management system to build a prototype with. It now supports trigger for statements allowing to see all the changed records in the body of the trigger. The trigger functions can be written in PL/pgSQL, C... There was the work showing that triggers written in C are more effective, so C is chosen as a programming language to generate trigger functions.

5. 1. Prototype

A high-level application programming interface is provided to simplify the process of source code synthesis. Different sub-modules have their own functions:

– connecting and communication director;
– definition of trigger function structures;
– trigger parameters checking;
– processing of numerical and string constants;
– data type conversions;
– short-hands for string processing;
– debug and log manager;
– SQL query executor;
– query execution result cache manager;
– query execution result data extractor;
– changed in base tables data extractor;
– query execution result validating;
– table structure creation;
– MV query analyzer;

– changed in base tables data condensing templates and rules;

– incremental update templates and rules.

The prototype is realized as two standalone applications. The first does the tasks of MV manager and the second – asynchronous incremental update manager.

5. 2. Experiments

There are base tables countries, customers, sales and costs have numbers of attributes of 10, 23, 8 and 6 with the numbers of records of 23, 55.500, 918.843 and 822.112 respectively. A MV with the name mv_total is created for the following query, which calculates the summary received from each customer with their location information: SELECT countries.country_id, countries.country_name, countries.country_region_id, countries.country_region, customers.cust_id, customers.cust_first_name, stomers.cust_last_name, SUM(sales.quantity_sold*costs.unit_price), COUNT(*) FROM countries JOIN customers ON countries.country_id=customers.country_id JOIN sales ON customers.cust_id=sales.cust_id JOIN costs sales.time_id=costs.time_id AND sales.promo_id=costs.promo_id AND sales.channel_id=costs.channel_id AND sales.prod_id=costs.prod_id GROUP BY countries.country_id, countries.country_name, countries.country_region_id, countries.country_region, customers.cust_id, customers.cust_first_name, customers.cust_last_name.

The built source code synthesizer is used to produce i) script to create all auxiliary tables, ii) codes of triggers on all data manipulation events for all 4 base tables and iii) codes for the incremental update of MV mv_total. All those source codes are identical to codes created manually and function well. The insert/update/delete events on base tables are now similar, almost without a difference, to the previous case when there was not any trigger included. It is because of that triggers are very 'light', they do only read the deleted/inserted records in base tables and write to auxiliary tables.

The experiments were provided on a personal computer with configuration CPU Intel Core i5 3317U, RAM DDR3 4GB, HDD SATA3 5400rpm, and PostgreSQL v11.6 64bit installed. The asynchronous maintenance policy in the experiment is on request. Table 1 shows the execution time measured in milliseconds. Two cases of the number of records were evaluated per updating action on each base table: i) one record is manipulated by one SQL statement and ii) 10 records are manipulated by 10 SQL statements. The time is measured for each record per statement. The time in Table 1 is cumulative in the case of synchronous maintenance with 10 records changed.

Table 1

Incremental update time in milliseconds

Table	Synchronous			Asynchronous			
	Insert	Delete	Update	Insert	Delete	Update	Combined
1 record/1 command/action/base table							
Sales	22	90	138	29	55	75	108
Customers	15	56	77	28	47	71	
10 records/10 commands/action/base table							
Sales	211	925	1174	40	66	89	117
Customers	139	612	757	28	54	77	

For asynchronous maintenance, we examine not only the general case when there is a mix of actions (insert or delete of update) but also the case when there is only one of the actions separately. The optimization mentioned for the case of base table customers is not applied to check the general case of that there are more than one base table changed between the two pre-update and post-update states.

5. 3. Discussion of experimental results

The closest to this solution is the one suggested in the work [8]. As mentioned above, its main distinguishing point to this research is the mechanism for accessing the pre-update states of base tables and gathering changed records. Its disadvantages are in system resources needed to read the right version of base tables and gathering changes from transaction log, the performance of which is strongly dependent on the concrete database management system. Our solution avoids those costs, but it has a disadvantage in system resources required to calculate the pre-update state of the base tables following expression (12). So, the performance of the two solutions may be competitive, but our solution can be implemented with any database management system, not only with the ones that support data versioning at the row and table levels.

The experimental results show that the commutative resource required for the synchronous incremental update is grown almost with the factor of the number of invokes, but fortunately, the cost is spread over time and is negligibly small for each call.

It may be because of the small number of manipulated records, the time of maintenance seems almost for code invoking for both synchronous and asynchronous cases. For asynchronous maintenance, the difference between updating 1 record and 10 records is different by 12 %. Opposite to synchronous one, the accumulative update in an asynchronous manner costs about 30–33 % of that when the updates are performed separately. The total asynchronous updating time is about 3 % (10 records/10 commands per action in each base table) – 27 % (1 record/1 command per action in each base table) in comparison to the synchronous incremental update. These good results are due to the effects of condensing process and smaller number of code invoking.

If we apply the optimization suggested in the work [5] for the cases like base table customers, the update of MV according to data changes in customers will be performed separately.

6. Conclusions

1. We formally showed the state bug when applying the expressions for the incremental update of MV at the pre-update state of base tables, which is used in synchronous maintenance to the post-update state of base tables, which is required in asynchronous one.

2. We proposed a solution for the asynchronous incremental update of MV with the new technique for accessing pre-state of base tables. So that the state bug is avoided and the expressions for calculation of changes to MV which is often used in the synchronous incremental update can be correctly applied to the asynchronous update. We exploited the idea of condensing operators and described them detailly to prove the correctness of the suggested accessing pre-state of base tables technique. The incremental update algorithm is adopted from other our published works applying the updating expressions and specifics for asynchronous maintenance. It is the main contribution to the field of this research.

3. We built a prototype which can synthesize the source code in an automatic manner for supporting the asynchronous incremental update and provided experiments to ensure the correctness of the solution. The total time of asynchronous update is 4–33 times smaller than synchronous ones. The accumulative update in an asynchronous manner reduces the cost by about 67–70 % of that when the updates are performed separately.

References

1. 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
2. Vinh, N. T. Q., Hao, D. T., Hang, P. D. T., Alsadoon, A., Prasad, P. C., Anh, N. V. (2019). A solution for synchronous incremental maintenance of materialized views based on SQL recursive query. Eastern-European Journal of Enterprise Technologies, 5 (2 (101)), 6–17. doi: https://doi.org/10.15587/1729-4061.2019.180226
3. 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
4. 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
5. 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
6. O'Gorman, K., Agrawal, D., El Abbadi, A. (2000). On the Importance of Tuning in Incremental View Maintenance: An Experience Case Study. Lecture Notes in Computer Science, 77–82. doi: https://doi.org/10.1007/3-540-44466-1_8
7. 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
8. 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' (VLDB Endowment, 2007, edn.), 231–242.
9. Colby, L. S., Griffin, T., Libkin, L., Mumick, I. S., Trickey, H. (1996). Algorithms for deferred view maintenance. ACM SIGMOD Record, 25 (2), 469–480. doi: https://doi.org/10.1145/235968.233364
10. Yan, W. P., Larson, P.-A. (1995). Eager Aggregation and Lazy Aggregation. Proceedings of the 21th International Conference on Very Large Data Bases, 345–357.
11. Nguyen, T. Q. V., Tran, T. N. (2014). Automatic generating C-languague-triggers modul for synchronized incremental updating materialized views in PostgreSQL. Proc. National Conference on Fundamental and Applied IT Research (FAIR).
12. Agrawal, P., Silberstein, A., Cooper, B. F., Srivastava, U., Ramakrishnan, R. (2009). Asynchronous view maintenance for VLSD databases. Proceedings of the 35th SIGMOD International Conference on Management of Data - SIGMOD '09. doi: https://doi.org/10.1145/1559845.1559866
13. Mikami, K., Morishita, S., Onizuka, M. (2010). Lazy View Maintenance for Social Networking Applications. Lecture Notes in Computer Science, 347–358. doi: https://doi.org/10.1007/978-3-642-12098-5_29
14. Chun, S., Jung, J., Lee, K.-H. (2019). Proactive Policy for Efficiently Updating Join Views on Continuous Queries Over Data Streams and Linked Data. IEEE Access, 7, 86226–86241. doi: https://doi.org/10.1109/access.2019.2923414
15. Phani, A., Tekur, C., Sai Krishna, R. K. N. (2019). Commit Time Materialized View Maintenance for Bulk Load Operations in Teradata. 2019 IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT). doi: https://doi.org/10.1109/icecct.2019.8869100
16. Almazyad, A. S., Siddiqui, M. K., Ahmad, Y., Khan, Z. I. (2009). An Incremental View Maintenance Approach Using Version Store in Warehousing Environment. 2009 Second International Workshop on Computer Science and Engineering. doi: https://doi.org/10.1109/wcse.2009.624
17. Zhuge, Y., García-Molina, H., Hammer, J., Widom, J. (1995). View maintenance in a warehousing environment. ACM SIGMOD Record, 24 (2), 316–327. doi: https://doi.org/10.1145/568271.223848