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 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.

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.

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.

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.

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.

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.
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:

\[
Q^x(S^x, T^x, J^x, W^x).
\]  

(1)

where:
- \(S^x = \{S_1^x, S_2^x, ..., S_n^x\}\) – set of fields that are selected and presented in SELECT predicates;
- \(T^x = \{T_1^x, T_2^x, ..., T_m^x\}\) – set of base tables that participate in FROM predicates. FROM predicate \(F^x\) is the combination of \(T^x\) and \(J^x\):
  \[
  F^x = T_1^x \bowtie_{J_1} T_2^x \bowtie_{J_2} ... \bowtie_{J_{n-1}} T_n^x.
  \]
  
- \(J^x\) – join conditions between base tables in \(T^x\);
- \(W^x\) – WHERE predicate, 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 \(C^x = J^x \land W^x\). 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:

\[
Q^x(S^x, T^x, J^x, W^x, G^x).
\]  

(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 set of a \(i^{th}\) base table \(T_i^x\) can take participation 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,

\[
\text{newF}^x = T_1^x \bowtie_{J_1} T_2^x \bowtie_{J_2} ... \bowtie_{J_{n-1}} (T_i^x \cup dnewT_i^x) \bowtie_{J_n} T_n^x = T_1^x \bowtie_{J_1} T_2^x \bowtie_{J_2} ... \bowtie_{J_{n-1}} T_i^x \cup T_i^x \bowtie_{J_i} T_i^x \cup (dnewT_i^x \bowtie_{J_i} T_i^x).
\]  

(3)

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

\[
\text{oldF}^x = T_1^x \bowtie_{J_1} T_2^x \bowtie_{J_2} ... \bowtie_{J_{n-1}} (T_i^x \setminus doldT_i^x) \bowtie_{J_n} T_n^x = T_1^x \bowtie_{J_1} T_2^x \bowtie_{J_2} ... \bowtie_{J_{n-1}} T_i^x \setminus T_i^x \setminus (doldT_i^x \bowtie_{J_i} T_i^x).
\]  

(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 \(Q^x(S^x, T^x, J^x, W^x)\) is:

\[
M^x = (S^x, T^x, J^x, W^x).
\]  

(5)

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

\[
M^x = \pi_{\{x\}} (\sigma_{W^x}(T_1^x \bowtie_{J_1} T_2^x \bowtie_{J_2} ... \bowtie_{J_{n-1}} T_n^x)).
\]

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 \(postT_i = preT_i^x \cup dnewT_i^x\),

\[
dnewT_i^x = \{preT_i^x, preT_i^x, ... dnewT_i^x, ... preT_n^x\}.
\]
\[ post^* = \{ pre^T_1, pre^T_2, \ldots, post^T_i, \ldots, pre^T_n \} = \{ pre^T_1, pre^T_2, \ldots, \{ pre^T_i \cup \text{dnew}^T_1 \} \ldots \cup \text{dnew}^T_n \}. \]

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

\[ postM^* = ( S^*, post^T_j, J^*, W^* ) = ( S^*, pre^T_j, J^*, W^* ) \cup ( S^*, \text{dnew}^T_j, J^*, W^* ); \]

\( postM^* \) is in the form of a relational algebra expression as follows:

\[ postM^* = \Pi_{\{x\}} ( D ) \left( \left( pre^T_j \bowtie \ldots \bowtie \text{pre}^T_n \right) \cup \left( \text{dnew}^T_1 \bowtie \ldots \bowtie \text{dnew}^T_n \right) \right). \]

\( \text{dnew}^M^* = ( S^*, \text{dnew}^T_j, J^*, W^* ) \) is the set of records that must be inserted into MV \( M^* \) according to the insertion of \( \text{dnew}^T_j \) into \( T^*_j \).

Suppose an MV query has two base tables \( T^*_1 \) and \( T^*_2 \) changed by insert operations between two states, 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:

\[ \text{dnewpre}^M^* = \Pi_{\{x\}} ( D ) \left( \left( \text{dnew}^T_1 \bowtie \ldots \bowtie \text{dnew}^T_n \right) \cup \left( \text{pre}^T_1 \bowtie \ldots \bowtie \text{pre}^T_n \right) \right). \]

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

\[ \text{dnewpost}^M^* = \Pi_{\{x\}} ( D ) \left( \left( \text{dnew}^T_1 \bowtie \ldots \bowtie \text{dnew}^T_n \right) \cup \left( \text{pre}^T_1 \bowtie \ldots \bowtie \text{pre}^T_n \right) \right). \]

The record set that must be inserted into 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):

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

\[ \left( \text{dnew}^T_j \bowtie \ldots \bowtie \text{dnew}^T_n \right) \cup \left( \text{pre}^T_1 \bowtie \ldots \bowtie \text{pre}^T_n \right) \]

\[ = \Pi_{\{x\}} ( D ) \left( \left( \text{dnew}^T_1 \bowtie \ldots \bowtie \text{dnew}^T_n \right) \cup \left( \text{pre}^T_1 \bowtie \ldots \bowtie \text{pre}^T_n \right) \right). \]

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 \( \text{dold}^T_j \) omitting intermediate changes of records which are deleted from/updated in/inserted into base tables and then separate it to \( \text{dold}^T_j \) and \( \text{dnew}^T_j \) to access the pre-update state of \( T^*_j \) from its post-update state.

4.3. Proposed technique

Suppose we have a base table \( T^*_j \) with the pre-update state \( \text{old}^T_1 \). There is a record set \( \text{dold}^T_1 \) which moves \( T^*_j \) from the pre-update state \( \text{pre}^T_1 \) to the post-update state \( \text{post}^T_1 \). Certainly, records in \( \text{dold}^T_1 \) are with action information (inserted, deleted) and are ordered ascendingly by the time of the operations.

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 $T^i_\ast$, $dT^p_\ast$ contains the set of records that are deleted from and/or inserted to $T^i_\ast$, which moves $T^i_\ast$ from the pre-update state $preT^i_\ast$ to the post-update state $postT^i_\ast$.

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 $T^i_\ast$, so we can omit all the operations removing all the related records from $dT^p_\ast$.

Scenario 4.

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

For this scenario, the only last operation influences the post-update state of $T^i_\ast$. So we remove all those records from $dT^p_\ast$ relating to the first and intermediate operations, keeping only the last one.

Scenario 5.

First: Inserted.
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 $T^i_\ast$, removing it may affect the post-update state of $T^i_\ast$. All the intermediately inserted records are deleted between the pre-update and post-updates of $T^i_\ast$, the operations are self-compensated each to other. We can remove all those records from $dT^p_\ast$ 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 $T^i_\ast$, removing it may affect the post-update state of $T^i_\ast$. 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 $dT^p_\ast$ 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 $dT^p_\ast$.

Without generality, after condensing process, there are three cases of record from $dT^p_\ast$ must be considered:
- an originating record is inserted into $T^i_\ast$ represented by $t^i_\ast$;
- an originating record is deleted from $T^i_\ast$ represented by $t^i_\ast$;
- an originating record is deleted from $T^i_\ast$ and then another one is inserted into $T^i_\ast$ represented by $t^i_\ast$ and $t^i_\ast$ respectively.

Note that those records $t^i_\ast$, $t^i_\ast$ and the pair of $t^i_\ast$ plus $t^i_\ast$ 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 $t^i_\ast$ and $t^i_\ast$. It is clear that we can't re-order $t^i_\ast$ and $t^i_\ast$, which can lead to wrong post-update state of $T^i_\ast$. If we divide $dT^p_\ast$ into the set of deleted records $doldT^p_\ast$ and the set of inserted records $dnewT^p_\ast$, i.e. we have $dT^p_\ast = \{t^i_\ast | j = 1,k\}$, $dnewT^p_\ast = \{t^i_\ast | j = 1,k\}$, $doldT^p_\ast = \{t^i_\ast | j = 1,k\}$, $t^i_\ast \in dnewT^p_\ast$ and $doldT^p_\ast$ is following by $dnewT^p_\ast$, then the order of the records within each set is not important. This time,

$$postT^i_\ast = \{preT^i_\ast \setminus doldT^p_\ast\} \cup dnewT^p_\ast,$$  \hspace{1cm} (11)

and

$$preT^i_\ast = \{postT^i_\ast \setminus dnewT^p_\ast\} \cup doldT^p_\ast.$$  \hspace{1cm} (12)

At the post-update state of $T^i_\ast$, $postT^i_\ast$ 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 $dnewM^p_\ast$ from [8] ignoring the order of records in $dT^p_\ast$ in case of all $n$ base tables are updated. The expressions (14) and (13) show $doldM^p_\ast$ and $dnewM^p_\ast$ – 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.

$$dnewM^p_\ast = \begin{cases}
\begin{aligned}
&dnewT^i_\ast \bowtie j, \quad preT^i_\ast \bowtie j, \quad t^i_\ast \bowtie j \\
&\bowtie j, \quad preT^i_\ast \bowtie j, \quad preT^i_\ast \bowtie j, \quad \cdots
\end{aligned}
\end{cases}\
\cup postT^i_\ast \bowtie j, \quad dnewT^i_\ast \bowtie j, \quad t^i_\ast \bowtie j, \quad \cdots
= \pi_{\{\alpha\}} \sigma_{\{\omega\}}
\begin{cases}
\begin{aligned}
&\bowtie j, \quad preT^i_\ast \bowtie j, \quad preT^i_\ast \\
&\bowtie j, \quad postT^i_\ast \bowtie j, \quad \cdots
\end{aligned}
\end{cases}$$  \hspace{1cm} (13)

and

$$doldM^p_\ast = \pi_{\{\alpha\}} \sigma_{\{\omega\}}
\begin{cases}
\begin{aligned}
&doldT^i_\ast \bowtie j, \quad preT^i_\ast \bowtie j, \quad \cdots
\end{aligned}
\end{cases}$$  \hspace{1cm} (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 [3]. 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 tables record, 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 to build a prototype with. It now supports trigger for state- condensing process and then invoke the save code to undertake incremental maintenance of MV.

4.5. 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. 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, customers.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  
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 main-  
tenance with 10 records changed.  

|          | Incremental update time in milliseconds |  |
|----------|------------------------------------------|---|
|          | Synchronous                               | Asynchronous                   |
|          | Insert | Delete | Update | Insert | Delete | Update | Combined |
| Sales    | 22     | 90     | 138    | 29     | 55     | 75     | 108      |
| Customers| 55     | 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 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 re-  
cords. 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 per-  
fomance 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  
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 syn-  
chronous 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 condens-  
ing 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 accord-  
ting 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 incre-  
mental 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 exploit-  
ed 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 main-  
tenance. 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 Oszu, 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-language-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.2019.8899100
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/wese.2009.624
17. Zhuge, Y., Garcia-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