ISSN 0361-7688, Programming and Computer Software, 2016, Vol. 42, No. 5, pp. 307–315. © Pleiades Publishing, Ltd., 2016.
Original Russian Text © Nguyen Tran Quoc Vinh, 2016, published in Programmirovanie, 2016, Vol. 42, No. 5.

# Synchronous Incremental Update of Materialized Views for PostgreSQL[1]

**Nguyen Tran Quoc Vinh**

*University of Education —The University of Da Nang (Vietnam), Ton Duc Thang, Da Nang, Vietnam*
*e-mail: ntquocvinh@ued.udn.vn*

Received August 12, 2015

**Abstract**—Materialized views are logically excess stored query results in SQL-oriented databases. This technology can significantly improve the performance of database systems. Although the idea of materialized views came up in the 1980s, only three database management systems, i.e. DB2, Oracle, SQL Server, have been successfully developed completely enough with materialized views so far. The barrier lies in building a module that can incrementally update the materialized views automatically, which corresponds to data changes in the base tables. This paper presents the algorithm to incrementally update the materialized views with inner join, focusing on one with aggregate functions, and building of a program that automatically generates codes inPL/pgSQL for triggers, which can undertake synchronous incremental updates of the materialized views in PostgreSQL.

**Keywords:** materialized views, synchronous incremental updates, triggers, automatic synthesis of source code, PostgreSQL

**DOI:** 10.1134/S0361768816050066

## 1. INTRODUCTION

Materialized view (MV) is created based on a query whose result is saved in a table of database (MV table). When we access MV, database management systems (DBMS) will not execute the query but will gain the result from MV table if possible [1–12]. Similar to cache, MV allows quick query result access. This is especially beneficial in cases where query frequency is high and the query is sufficiently complicated with a big enough amount of data so that the query "cannot" be executed again whenever DBMS receives a request. It may help to achieve 10–100× or better performance gains from the bottleneck queries. MV has been widely applied in creating quick access with low CPU, memory and disk loading, in building data warehouse, in chronicle systems such as banks, retail sales and accounting, in data visualization applications, in mobile systems, in controlling integrity constrains and in query optimization, also when realtime access to the query execution result is critical.

One of the characteristics of chronicle systems is the huge amount of data, which usually exceeds DBMS capability to execute queries with the whole database. In those systems, MV may be used to response to queries without accessing the whole chronicle system. For instance, it may be defined to calculate and store necessary summary values such as balance of single bank accounts, or the total profit of a retail store.

With every manipulation (insert, update, delete) of base tables (BT, underlying tables), the MV using those BT may fail to remain actual. To keep MV table in actual state, it needs updating whenever data in BT change. The process of keeping data in MV corresponding to data in the BT is called update (actualization). Incremental update changes the records of MV table that related to changed records in BT. Full refresh truncates MV tables and then populates data by re-executing the query. Synchronous update is performed as part of the transaction that carries out the data changes in the BT. Asynchronous update is done at another time, with the explicit request of the user or the demand for data.

MV technology has been implemented completely enough in several commercial DBMS such as Oracle, IBM DB2 and MS SQL Server, i.e. with incremental update and number of restrictions, but not in open source DBMS. One of the main barriers lies in building a module that can implement incremental updates. The work [10] shows how to implement incremental update of materialized views in PostgreSQL using triggers, but the programmers have to build the triggers from the scratch. It suggests to generate trigger functions source code automatically, but does not describe which incremental update algorithm is underlying, and especially, how to implement that algorithm pro-

[1] The article is published in the original.

Query execution and data manipulation in BT time (ms)

| BT | Configuration/SQL | SELECT | INSERT | DELETE | UPDATE |
|---|---|---|---|---|---|
| Sales | No MV | 223,853 | 13 | 82 | 110 |
| | MV | 80 | 27 | 110 | 177 |
| Customers | No MV | 223,853 | 11 | 13 | 34 |
| | MV | 80 | 19 | 61 | 88 |
| | MV – improved triggers | | 16 | 21 | 55 |
| Countries | No MV | 223,853 | 11 | 11 | 12 |
| | MV | 80 | 25 | 715 | 1799 |
| | MV – improved triggers | | 15 | 17 | 29 |

grammatically for any MV queries. This research built an incremental update algorithm that combined and improved from the previous one that had been published before, and proposing an algorithm, which automatically generates trigger source codes in PL/pgSQL language for synchronous incremental updates of MV in PostgreSQL DBMS. Hence, a program that automatically generates trigger source codes executing the incremental update algorithm mentioned above has been built. Even though it is not completed and not integrated with PostgreSQL, the program automatically generates trigger source code can support programmers. They, instead of reprogramming from the scratch, only need to adjust the generated triggers as they wish.

It is known that the PostgreSQL is a freely available DBMS that is used worldwide as a competitor to commercial databases to build enterprise information systems of all sizes. If MV technology is implemented successfully in PostgreSQL, it will bring many economic and social benefits. However, MV is just partly implemented into PostgreSQL since version 9.3.4. Even version 9.5 Alpha2 announced in Oct. 2015, the MV is updated asynchronously by a full refresh way. Somewhat experiment demonstrates this claim.

The experimental query uses BT countries, customers, sales and costs with the record count of 23, 55.500, 918.881 and 82.112. The popular query is to calculate totals from each customer: SELECT countries.country_id, country_name, country_region_id, country_region, customers.cust_id, cust_first_name, cust_last_name, SUM(quantity_sold*unit_price) as total, COUNT(*) as cnt FROM countries, customers, sales, costs WHERE countries.country_id = customers.country_id AND customers.cust_id = sales.cust_id AND 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, country_name, country_regionid, country_region, customers.cust_id, cust_first_name, cust_last_name. The MV is created with the data populated by the command "create materialized view", and then it is updated via the command "refresh materialized view". PostgreSQL provides both commands. The MV creating and updating processes take the same time (about 4 minutes on the system configured in scope of this research).

Regarding the incremental update algorithm (Section 2.2.3) for MV with aggregations, this research inherits and develops ideas of some research published previously [1–3, 6, 11, 12], e.g. the idea to separate updating into inserting and deleting [2, 3]; or the idea of relevant and irrelevant updates in BT for SPJ query [1] is developed for query with aggregations. This research improves the developed algorithm for some cases of query (Sections 2.2.4–2.2.7). The implementation in PostgreSQL is another main contribution (Section 3). The suggestions to transform query (Section 2.2.1) also distinguishes this research compared with prior researches in incremental updates for MV.

This research does not deal with MV created basing on nested query, recursive query, and outer join query [7, 8]. We are going to examine the MV types that focus in inner joins.

## 2. INCREMENTAL UPDATE ALGORITHM

### *2.1. MV Based on SPJ Query*

SPJ query is the one that includes only the SELECT, FROM, WHERE clauses, but not the grouping and aggregate functions. From clause may contain inner joins. In this article, for SPJ MV case, the author built a program that generates trigger codes doing incremental updates based on an algorithm that improved the ones previously published [1] and create its software for the automatic synthesis of triggers source code for incremental update of MV considering that:

—query result does not contain duplicates;

—any one key of each BT is added automatically into the SELECT clause;

—consider the modification operation in the form of equivalent operations as delete and subsequent insert;

—exclude the data manipulation in BT that does not affect the MV;

—the improvement from Sections 2.2.5–2.2.7 is applied.

This research focuses more on queries with aggregate functions.

### *2.2. Queries Including Aggregate Functions*

It is assumed that MV is created based on the query with aggregate functions $Q$ ($S$, $F$, $J$, $W$, $G$), in which:

$S$: a set of the selected columns in select clause. $S$ can include columns or aggregate functions with the expressions (E) using columns from the BT such as SUM(E), COUNT(E), AVG(E), MIN(E) and MAX(E). E does not consist of aggregate functions.

$F$: a set of BT $T_1$, $T_2$ ... $T_n$ used in the query.

$J$: a set of criteria of joins.

$W$: criteria for records chosen to process in WHERE clause.

$G$: a set of grouping columns in GROUP BY clause.

Besides, the following symbols are also used:

$P = S \backslash G$—a set of aggregate functions in select clause;

$T_i^{\mathrm{i}}$ ($T_i^{\mathrm{d}}$)—a bag of records (from now on, bag of records means a "set" that may contain duplicate elements) is added to (deleted from) BT $T_i$.

To simplify, we use some symbols: $C = W$ AND $J$.

**2.2.1. MV creation.** The information about a query transformation to create MV, or specifically, the information about the columns in MV table, helps clarify the update algorithms. The purpose of the transformation is to improve the optimization of MV usage and the execution of incremental update algorithm for MV. $S$ consists of columns saved in MV table. Regarding to aggregate functions, only cases of SUM(E), COUNT(E), AVG(E), MIN(E) and MAX(E) are considered. Based on the select query syntax regulations in SQL language as well as the inferences from relational algebra, it is possible to state that, "column" in S can be considered as (i) a column from BT, as (ii) an algebraic expression of aggregation functions, or (iii) an algebraic expression of the BT columns and aggregation functions. Expression (E) does not include aggregate functions, which means that aggregate functions cannot be a parameter of aggregate function. For the case (iii), in which the columns get involved in expressions in a similar order to aggregate functions (such as the column a1 in the query: select ..., (a1 + SUM(a2)) AS some sum ... group by ..., a1) they can be considered as other normal columns involved in $G$. Therefore, instead of saving the result of algebraic expressions between aggregate functions or between aggregate functions and BT columns, we save the result of each of the operands.

This research consequently suggests several query transformations during the process of creating MV:

COUNT(*) is added automatically because it helps us determine when to delete a record from MV table. If no record further joins the group, the record associated to that group will be removed from MV table. Indeed, MAX(E), MIN(E), SUM(E) cannot determine this, except COUNT(*).

In terms of algebraic expression among aggregate functions or grouping columns (ii) and (iii), we separate and save them as individual columns. The original expression can be calculated from columns stored in MV table if necessary.

In terms of AVG(E) function, we propose to save SUM(E) and COUNT(E) individually, and compute AVG(E) = SUM(E)/COUNT(E) when needed. Thus, we recommend to split AVG(E) into SUM(E) and COUNT(E) in MV query. Instead of saving AVG(E), we store SUM(E) and COUNT(E). If E, determined from metadata, is not NULL, COUNT(E) and COUNT(*) are equivalent; therefore, it is unnecessary to save COUNT(E) but just COUNT(*).

Convert $W$ and $J$ into conjunctive canonical forms: (... OR ...) AND ... AND (... OR ...). The changes are to: (i) compare clauses $W$ and $J$ among queries to determine the ability to use MV; and (ii) identify whether updates in BT are related to MV or not. (i) The comparison of two logical expressions is complicated. In this research, the author proposes the comparison of two Boolean expressions using the following rule: converting them into conjunctive canonical forms, then sorting by dictionary order and comparing them. (ii) In this canonical form, both W and J only have true value if all constituted clauses are true. If a record does not satisfy any component clauses, it will not have any impacts on $Q$ query execution results.

**2.2.2. Incremental update algorithm.** Events to alter data in BT are divided into three types of operations: insert, update and delete a set of records. Records in tables cannot be duplicated. However, the tuples of values corresponding to a set of columns may be duplicated. These sets are considered bags. Most published researches related to incremental update for the MV considered updating equivalent to deleting a bag of records and then adding a new one [1, 2]. However, this separation is not always a better way. If the query does not include MIN(E), MAX(E) aggregate functions, it is unnecessary to divide update into insert and delete. Otherwise, the division is obvious. Especially, when the updated columns are not included in $G$.

For insert, data in the BT is not related because the inserted value can replace the current MIN(E), MAX(E) values by comparing it to the value in MV table (Tmv). As for delete, we have to look for new MIN(E), MAX(E) values from BT if we delete records having fields with current MIN(E) or MAX(E) values.

Regarding how BT participate in query, we may consider the two following cases, (i) some keys of $T_i$ BT get involved in queries in $G$; (ii) several columns of

$T_i$ only participate in $G$ and several columns create the key of $T_1$ and participate in $G$; (iii) $T_i$ has a special role in from clause; and (iv) all the remaining cases.

**2.2.3. Considering update equivalent to delete with subsequent insert.** The incremental update algorithm described here is synthesized from some algorithms found in published researches. It focuses on considering update the $T_i^d$ to $T_i^i$ equivalent to delete $T_i^d$ from BT and then insert $T_i^i$.

(a) The first operation case—add a new set of records $T_i^i$ into BT $T_i$.

(S1) Remove from $T_i^i$ the records that do not satisfy $C$. Obviously, the records wrong in $C$ cannot affect the executing results, so it is unnecessary to review them and they can be removed from the beginning. $J$ usually contains conditions like $T_i.A_1 = T_j.A_2$. However, in some cases, $J$ may include ordinary filter conditions. So that, condition in $C$ may be a comparing expression combined from only columns of current BT $T_i$ and constants. $C$ is in conjunctive canonical form, if a record does not satisfy any condition, it does not satisfy $C$ too. The records that are wrong in these conditions have no impact on the query executing result, they can be eliminated right from the start. At this stage, we can only filter records following the conditions in $C$, which relate to only $T_i$. The removal of records at this stage helps reduce the number of records involved in the joins in S2.

(S2) Joining achieved $T_i^i$ with the rest tables in $F$, we obtain $dQ_i^i$ results.

(S3) Removing from $dQ_i^i$ records that do not satisfy $W$.

(S4) Doing group-by operation for $dQ_i^i$ by $G$, corresponding columns are selected to participate in S, implementing expressions, we obtain new $dQ_i^i$.

(S5) With each record in $dQ_i^i$ and updating Tmv according to this principle:

—If there are no records with ($G$, $P$) forms in Tmv, add a new record with COUNT(*) taking value 1.

—SUM(E) = SUM(E) + dSUM$^i$(E). In which, E is expression. dSUM$^i$(E) is SUM(E) on $dQ_i^i$.

–COUNT(E) = COUNT(E) + dCOUNT$^i$(E). dCOUNT$^i$(E) is COUNT(E) on $dQ_i^i$. Correspondingly, COUNT(*) = COUNT(*) + dCOUNT(*).

—If MIN(E) > dMIN$^i$(E), it means that the added record will create MIN(E) value for the group, then MIN(E) = dMIN$^i$(E). Conversely, MIN(E) stays unchanged. dMIN$^i$(E) is MIN(E) on $dQ_i^i$.

—If MAX(E) < dMAX$^i$(E), it means that the added records will create new MAX(E) value for the group, MAX(E) = dMAX$^i$(E). Conversely, MAX(E) stays unchanged. dMAX$^i$(E) is MAX(E) on $dQ_i^i$.

(b) Deleting the set of records $T_i^d$ from BT $T_i$:

(S1) Removing from $T_i^d$ records that do not satisfy $C$.

(S2) Joining $T_i^d$ with the remaining tables in $F$, we obtain $dQ_i^d$ results.

(S3) Removing from $dQ_i^d$ records that do not satisfy $W$.

(S4) Doing group-by operation for $dQ_i^d$ by $G$, corresponding columns are selected to participate in $S$, implementing expressions, we obtain new $dQ_i^d$.

(S5) For each record in $dQ_i^d$, updating Tmv according to this principle:

—SUM(E) = SUM(E) – dSUM$^d$(E). dSUM$^d$(E) is SUM(E) on $dQ_i^d$.

—If COUNT(*) = dCOUNT$^d$(*), there are no records of the group participating in the query results, delete the records with ($G$, $P$) form from Tmv. Otherwise, COUNT(*) = COUNT(*) — dCOUNT$^d$(*). dCOUNT$^d$(*) is COUNT(*) on $dQ_i^d$. COUNT(E) = COUNT(E) – dCOUNT$^d$(E).

—If MIN(E) = dMIN$^d$(E), the deleted records create MIN(E) value of the group, we recompute MIN(E) based on the BT. Conversely, the MIN(E) stays unchanged. dMIN$^d$(E) is MIN(E) on $dQ_i^d$.

—If MAX(E) = dMAX$^d$(E), the deleted records create MAX(E) value, we recompute MAX(E) based on the BT. Conversely, MAX(E) stays unchanged. dMAX$^d$(E) is MAX(E) on $dQ_i^d$.

(c) Updating the set of records $T_i^d$ from BT $T_i$ to $T_i^i$:

Considering update $dT_i$ from $T_i^d$ to $T_i^i$ equivalent to delete $T_i^d$ and then insert $T_i^i$.

(S1) Comparing $T_i^d$ and $T_i^i$, remove from $T_i^d$ and $T_i^i$ the adjusted records which are not relevant to $Q$, are not changed in relevant to $Q$ columns and do not satisfy $C$.

(S2) Doing the same as the case of deleting with $T_i^d$.

(S3) Doing the same as the case of inserting with $T_i^i$.

**2.2.4. Not considering update as being separated into delete and insert.** As reasoned above, now we only look at the SUM(E) and COUNT(E) functions because we will not apply them for queries involving MIN(E) and MAX(E) cases, and AVG(E) is handled the same as above, i.e. is calculated from SUM(E) and COUNT(E). For update, despite not considering to divide it into two new events—delete and insert separately, we still consider two sets: a set of records with

old values ($T_i^d$) and a set of records with new values ($T_i^i$). The steps for incremental update are as follows:

(S1). Eliminating the records from $T_i^d$ and $T_i^i$ that do not satisfy $C$ or all involved columns equal between new and old values.

(S2) Calculating full outer join between $T_i^i$ and $T_i^d$ with joining conditions on one key of $T_i$ and all columns of $T_i$ that participate in $G$, then doing inner join the result with rest tables in $F$, we obtain the $dQ_i$. It ensures that each record of $T_i^i$ and $T_i^d$ is joined once and the case of value changes on columns in $G$ is covered.

(S3) Doing group-by for the set of records in join result $dQ_i$ by $G$ and receiving new $dQ_i$.

(S4) Calculating SUM(E) and COUNT(E) for each group on $dQ_i$: $dSUM^i(E)$, $dSUM^d(E)$, $dCOUNT^i(E)$, $dCOUNT^d(E)$.

(S5) Implementing update according to the following principles:

(i) $SUM(E) = SUM(E) + dSUM^i(E) - dSUM^d(E)$.

(ii) $COUNT(E) = COUNT(E) + dCOUNT^i(E) - dCOUNT^d(E)$.

In this way, we can only execute trigger one time for update instead of two times for delete and insert as two separate events of update. The Tmv update operation is done once for almost cases. The join operation is archived once too, instead of twice.

**2.2.5. A key of the BT $T_i$ is in $G$.** Whether columns can create the primary key or any key at all can be identified through meta-data of the database. If all columns that create a key get involved in the query—($A_{i1}, A_{i2}... A_{ij}...A_{ik}$), particularly in $G$—when joining BT $T_i$ with other tables and then execute group-by, the value set ($A_{i1}, A_{i2}...A_{ij}...A_{ik}$) in Tmv still represents for one record from BT $T_i$. Therefore, when updating or deleting a record from $T_i$, we can do as follows.

(a) Deleting the record set $T_i^d$ from BT $T_i$:

(S1) Removing from $T_i^d$ all records which do not satisfy $C$.

(S2) Deleting from Tmv the records that reach the condition: $T_i^d.A_{ij} = \text{Tmv}.A_{ij}$, with all ($A_{i1}, A_{i2}...A_{ij}..., A_{ik}$).

(b) Updating the record set $T_i^d$ from BT $T_i$ to $T_i^i$:

Obviously, once $A_{i1}$, $A_{i2}$, ..., $A_{ij}$, ..., $A_{ik}$—the columns creating keys in BT $T_i$ participate in $G$, two cases may occur:

—None of the columns in $T_i$ participate in E from COUNT(E), SUM(E), MIN(E) and MAX(E).

(S1) Comparing $T_i^d$ and $T_i^i$, remove from $T_i^d$ and $T_i^i$ the adjusted records which are not relevant to $Q$.

(S2) Updating Tmv, update records that reach the condition $T_i^d.A_{ij} = \text{Tmv}.A_{ij}$ with all ($A_{i1}, A_{i2}...A_{ij}...A_{ik}$) to the new values: $\text{Tmv}.A_{ij} = T_i^i.A_{ij}$.

—If columns of $T_i$ participate in E from COUNT(E), SUM(E), MIN(E) or MAX(E), we perform updating as in the general case.

**2.2.6. A key of another BT $T_1$ is in $G$.** Several columns of $T_i$ which only participate in $G$ and several columns which create the key of $T_1$ and participate in $G$. If from clause only allows the existence of the direct inner join between two tables following $T_i.A_{ij} = T_1.A_{1j}$ conditions, not that of any joins such as ($T_i$ JOIN $T_1$ ON $T_i.A_{ij} = T_1.A_{lk}$) JOIN $T_x$ ON $T_i.A_{if} = T_x.A_{xh}$ AND $T_1.A_{lg} = T_x.A_{xy}$, and if we consider joining two tables are edges of the graph that link two vertices, i.e. two tables involved in the joins, then there may be multiple paths—the ways to execute joins to create links between two vertices—BT $T_i$ and $T_1$ in query, particularly in from clause. In this case, we apply the algorithm of finding the shortest path between two vertices to define joins of tables to create links between $T_i$ and $T_1$. The distance of two vertices—tables on one edge is defined as 1 unit. Practical methods to measure the distance based on the total cost of implementing joins are the subject of further research.

After identifying the joins—the shortest path between $T_i$ and $T_1$, we perform the incremental update Tmv as following.

(a) Deleting the record set $T_i^d$ from BT $T_i$:

(S1) Remove from $T_i^d$ all records that do not satisfy $C$.

(S2) Delete from Tmv all records reaching the condition: $T_i^d.A_{ij} = \text{Tmv}.A_{ij}$ AND $T_i^d.A_{lk} = \text{Tmv}.A_{lk}$, with all ($A_{i1}, A_{i2}...A_{ij}...A_{ik}$).

(b) Updating the record set $T_i^d$ from BT $T_i$ to $T_i^i$:

There are two cases. The first one, none of the columns in $T_i$ participate in E from COUNT(E), SUM(E), MIN(E) and MAX(E):

(S1) Comparing $T_i^d$ and $T_i^i$, remove from $T_i^d$ and $T_i^i$ the adjusted records which are not relevant to $Q$.

(S2) Update Tmv, update records reaching the condition $T_i^d.A_{ij} = \text{Tmv}.A_{ij}$ AND $T_i^d.A_{lk} = \text{Tmv}.A_{lk}$ with all ($A_{l1}$, $A_{l2}...A_{lj}...A_{lk}$) to the new values: $\text{Tmv}.A_{ij} = T_i^i.A_{ij}$.

If columns of $T_i$ participate in E from COUNT(E), SUM(E), MIN(E) or MAX(E), we perform updating as in the general case.

**2.2.7. Another case of insert is irrelevant to MV.** If $T_i$ participates in queries involving the from clause as $T_1$ JOIN $T_2$ ON...JOIN $T_i$ ON...JOIN $T_{i+1}$ ON $T_i$.key_$i$ = $T_{i+1}$.foreign_key_$i$... JOIN $T_n$..., the new

record inserted into $T_i$ will not get involved in the final join result. This means the record will not engage in execution results of the query, which creates MV. In other words, we can ignore this insert event.

## 3. PROGRAM BUILDING

### *3.1. Some Trigger Features in PostgreSQL*

Trigger is created based on the trigger function. Many triggers can share one trigger function. Trigger functions can be written in PL/pgSQL language (a language of PostgreSQL that is similar to SQL) or another language, for example, C.

In PostgreSQL, trigger for statement do not see changed records. Only trigger for each row can see the changed one and processes only one record. If n records of a table are inserted, updated or deleted by one command, triggers for each row will be fired n times.

Many triggers can be defined for one event on a table. At that time, they will be executed in alphabetical order according to the names of triggers.

Events of insert, update or delete as well as the time when trigger is fired before or after can be determined from inside the trigger functions, so we can generate one trigger function code for all events on each table.

### *3.2. Implementing Incremental Update Algorithm with Triggers*

3.2.1. Some general techniques.

(a) Variables

The record type variables are used to access triggered records. The table rowtype variables are used to access records of Tmv table and records returned by queries that have Tmv table structure.

(b) Checking of irrelevant changes

It concerns the code checking if the data change in BT is relevant. Condition in $C$ may be a comparing expression combined from only columns of current BT and constants. Because $C$ now consists of conditions in the conjunctive canonical form, the changed record does not affect the query result if it does not satisfy any condition in $C$. To generate this block of code, get all conditions combined from only columns of current BT and constants, put them to the conjunctive form and check if it yields true. If it is false, exit of the trigger function.

(c) Operations on sets of records

Inside trigger functions, no explicit operation on sets of records is implemented. It can be done implicitly by building the internal select queries and sending them to the PostgreSQL.

Getting the result of steps (S2)–(S4) in algorithm for update event that does not consider update as being separated into delete and insert (Section 2.2.4) is special. Regularly, the full outer join between $T_i^{\mathrm{d}}$ and $T_i^{\mathrm{i}}$ (with the same key values) produces the result in form as $\{T_i^{\mathrm{d}}.A_{i1}, T_i^{\mathrm{d}}.A_{i2}, \ldots T_i^{\mathrm{d}}.A_g, T_i^{\mathrm{i}}.A_{i1}, T_i^{\mathrm{i}}.A_{i2}, \ldots T_i^{\mathrm{i}}.A_{ig}, T_i^{\mathrm{d}}.A_{ig+1}\ldots)\}$ in which, $T_i^{\mathrm{d}}.A_{i1}, T_i^{\mathrm{d}}.A_{i2}, \ldots, T_i^{\mathrm{d}}.A_{ig}$ are in $G$. For all of their operations, the instruction "case" provided by PL/pgSQL is used to produce the result in form as $\{(T_i^{\mathrm{d}}.A_{i1}, T_i^{\mathrm{d}}.A_{i2}, \ldots, T_i^{\mathrm{d}}.A_{ig}, T_i^{\mathrm{d}}.A_{ig+1}, \mathrm{NULL}\ldots)\}$, $\{(T_i^{\mathrm{d}}.A_{i1}, T_i^{\mathrm{d}}.A_{i2}, \ldots, T_i^{\mathrm{d}}.A_{ig}, T_i^{\mathrm{d}}.A_{ig+1}, T_i^{\mathrm{i}}.A_{ig+1}\ldots)\}$, $\{(T_i^{\mathrm{i}}.A_{i1}, T_i^{\mathrm{i}}.A_{i2}, \ldots, T_i^{\mathrm{i}}.A_{ig}, \mathrm{NULL}, T_i^{\mathrm{i}}.A_{ig+1}\ldots)\}\}$. The result after (S4) will be in form as $\{(G, \mathrm{dSUM^i(E)}, \mathrm{dSUM^d(E)}, \mathrm{dCOUNT^i(E)}, \mathrm{dCOUNT^d(E)})\}$. The Tmv update (S5) can be done with it.

(d) Solution with internal query results

The step "for each record in set" presents in all algorithms for all data change events. The instruction "for...in select..." is used to walk through the individual record of the set.

(e) Solution with inserted and deleted records

In PostgreSQL, trigger handling the changed records can process only one record per firing. $T_i^{\mathrm{i}}$ and $T_i^{\mathrm{d}}$ always contain one record. Thus, it is unnecessary to create temporary tables $T_i^{\mathrm{i}}$ and $T_i^{\mathrm{d}}$ and then do join to get $dQ_i^{\mathrm{i}}$ and $dQ_i^{\mathrm{d}}$. Instead of doing so, follow the step:

—Append conditions on all corresponding columns with new/old values into the $W$ clause. The $dQ_i^{\mathrm{i}}/dQ_i^{\mathrm{d}}$ contains only records involving the current inserted/deleted record of $T_i$.

—Or, replace the values corresponding to columns of the current BT being inserted/deleted in queries to calculate new values for Tmv.

The both methods give equal result. If the first option is chosen, the trigger must be called prior to the event for the delete operation. However, PostgreSQL generates and always chooses a shorter execution plan with lower cost for the second method than the first one.

**3.2.2. Inserting event**. If adding a new record to the BT does not affect the MV table as it has a special role in from clause (Section 2.2.7), the insert event is ignored and it is unnecessary to generate codes for it.

New values for all SUM(E), COUNT(E), MIN(E) and MAX(E) functions in Tmv can be calculated based on the remaining BT and inserted record sets, which allows generating trigger codes to execute incremental update Tmv for either cases—i.e. trigger is fired before or after.

**3.2.3. Deleting event.** If there are only SUM(E) and COUNT(E) functions, but MIN(E) and MAX(E), all new values in Tmv can be calculated based on the remaining tables and deleted record sets, which allows generating trigger codes to execute incremental update Tmv for either cases—i.e. trigger is fired before or after.

In contrast, if one of the aggregate functions MIN(E) or MAX(E) exists, we need to make incremental update Tmv after the event occurs. The reason is if records which are being deleted gets involved in creating MIN(E) or MAX(E) value, we need to re-calculate these values based on the final data after the event takes place.

**3.2.4. Updating event.** Without MIN(E) and MAX(E), we can use the incremental update algorithm mentioned in Section 2.2.4 and generate codes for triggers fired before or after updating.

If there are MIN(E) or MAX(E), or both, we need to generate codes according to the algorithm mentioned in Section 2.2.3—i.e. process update as delete then insert. Clearly, we have to generate individual codes for delete and then insert. Similarly, we generate incremental update codes for Tmv like in the cases of deleting events (Section 3.2.3) and inserting events (Section 3.2.2).

### *3.3. Trigger Function Generating*

**3.3.1. Generating triggers for each BT.** The input includes query that creates MV and the MV name.

(S1) Converting implicit inner join in $Q$ into explicit inner join.

(S2) Replacing aliases with real table, column names.

(S3) Converting AVG(E) in $Q$ into SUM(E) and COUNT(*) if E cannot be NULL, otherwise, into SUM(E) and COUNT(E).

(S4) Determining $S$, $F$, $W$, $J$, $G$.

(S5) Getting information of tables, columns from database.

(S6) Checking if $Q$ is correct.

(S7) Generating block of PL/pgSQL script that creates MV table and populates data by executing the query $Q$. The most interest thing here is to determine data type of columns in $S$. Firstly, getting the data type of all BT columns involving in S. Secondly, setting the data type of columns in $S$. The special case is when E is an algebraic expression. If all involving BT columns in E have type of integers, then SUM(E), COUNT(E), MIN(E), MAX(E) have type of bigint. Otherwise, it has type of numeric.

(S8) Is $Q$ of SPJ type? For each BT in $F$, run procedure to generate script in PL/pgSQL to create triggers that implement incremental update for MV of SPJ type. The incremental update algorithm is mentioned in Section 2.1. Go to (S10).

(S9) $Q$ is now including aggregate functions. For each BT in $F$, run procedure to generate script in PL/pgSQL to create trigger functions and then triggers that implement synchronous incremental update for MV, which is based on queries including aggregate functions (Section 3.2.2). The incremental update algorithm is mentioned in Section 2.2.

(S10) End of procedure.

**3.3.2. Generate trigger functions for the case of MV with aggregate functions.** The triggers are generated to cover all data changes events on all BT. The triggers on different BT are independent. The process repeats the following procedure for each BT.

(S1) Generate code to check if the data change (in BT) is relevant.

(S2) If $T_i$ has no special position (Section 2.2.7) in from clause, generate PL/pgSQL code for insert event that implements the algorithm mentioned in Section 2.2.3. Otherwise, ignore it.

(S3) If all columns of the primary key or any unique key of $T_i$ are in $G$ (Section 2.2.5), generate code for delete event that implement the algorithm mentioned in Section 2.2.5. Otherwise, go to (S5).

(S4) If none of the columns in $T_i$ participates in E, generate code for update event that implement the algorithm mentioned in Section 2.2.5, then go to (S10). Otherwise, go to (S9).

(S5) If several columns of $T_i$ which only participate in $G$ and several columns which create the one key of $T_1$ and participate in $G$ (Section 2.2.6), go to (S6). Otherwise, go to (S8).

(S6) Calculate the shortest join-path between $T_i$ and $T_1$, and then generate the code for delete event (Section 2.2.6).

(S7) If none column of $T_i$ participates in E, generate code for update event (Section 2.2.6) and go to (S10).

(S8) Generate code for delete event for general case (Section 2.2.3).

(S9) If there are not MIN(E), MAX(E) generate code for update event that implements the algorithm mentioned in Section 2.2.4. Otherwise, generate code for delete event for general case—Section 2.2.3.

(S10) End of procedure.

## 4. TRIGGER SOURCE CODE GENERATOR

The built program satisfies all goals. It can generate trigger and trigger function source code in PL/pgSQL, implementing the mentioned (in Section 2) incremental update algorithm. It is written in C, so it can be integrated into PostgreSQL as a module by modifying the files matview.c and createas.c. Even though this is not completed and integrated with PostgreSQL, the program which automatically generates trigger source code can support programmers. They, instead of reprogramming from the start, only need to adjust generated triggers as they wish. This also can be used independently by integration with current MV feature of PostgreSQL. Generated triggers execute synchronously incremental update for the MV that created by the built-in command "create materialized view". The users do not need to execute the built-in command

"refresh materialized view" to do asynchronous full actualization.

The built program is tested with popular query to calculate the total amount received from each customer mentioned in Section 1. The generated triggers satisfy all requirements, fully coincide with the triggers written manually. They implement the mentioned incremental update algorithm.

## 5. EVALUATING THE GENERATED TRIGGERS AND DISCUSSION

MV can help query execution speed reaches maximum, because the original query is replaced by a simpler query that directs to the MV table. It can be done manually or automatically by rewriting module that can be integrated into PostgreSQL. Triggers on BT doing synchronous incremental updates for MV make the MV update operation becomes a part of update operation on BT. It may slow down the update on BT proportionally to the amount of data in BT, query complexity and quantity of related MV. The following evaluation estimates that slowing down time.

The environment for testing is a system with the configuration: CPU Intel Core i5 3317U, RAM DDR3 4GB, HDD SATA3 500GB 5400rpm, the operating system is Windows 8.1 64bit SP1, PostgreSQL v.9.3.4 64bit is installed as a dedicated database server. Through database administrative tool pgAdminIII v.1.18.1, the executing time of insert, update, delete record in BT for MV of three cases are measured: (i) No MV; (ii) MV and triggers implement update algorithm mentioned in Sections 2.2.3 and 2.2.4; and (iii) MV and triggers implement update algorithm with improvement (Sections 2.2.5–2.2.7). The experiment is made 30 times for each case and for each data manipulation operation. With the assumption that PostgreSQL reaches the stable state after number of 10 experiments, only the last 20 are involved in the average execution time (table). On BT sales, the experiments are provided on random records for (i) and (ii) cases of MV and triggers. On BT countries and customers, it is made on random records that average the number of joined records in BT sales for all (i), (ii) and (iii) cases. All update and delete commands are given with the conditions on object id to archive the objectivity, instead of other columns.

The table with the measured time shows the effectiveness of MV when the query is complex enough on the big amount of data (mentioned in Section 1). Especially, it is magnified if the query appears with high frequency, and the data manipulation commands like insert, update, delete are executed with the low one. The difference of execution time between commands update and insert/delete is very high in PostgreSQL. Comparing the two cases with and without triggers for incremental update, it primarily depends on processes of data manipulation commands as well as the numbers of records in BT, not the incremental update algorithm. The effectiveness of the implementation of the improvement to incremental update algorithm (Sections 2.2.4–2.2.7) is higher when the numbers of records involved increase. For example, the execution time with MV incremental update shortens to about 60 times when updates one record of BT countries with the number of involved records in BT sales is about 70,000.

The experimental execution to manipulate many records within one issued command delete and update (10 records in this research) shows that the difference of execution times is not much. In the future, if PostgreSQL supports triggers that handle many records, the performance of MV incremental update will be much more. Clearly, the positive effectiveness of using MV is not guaranteed for any information system operating periods or any queries. The administrator decides when and which query to use MV. It depends on the rule of MV using and data changes in BT as well as the demand to the instantaneity of the query executing result in the information system.

However, many restrictions on input query are archived. It also does not include related issues in incremental update made by triggers on the BT with foreign key integrity constrains. For example, if the foreign key integrity constrains are set between BT countries, customers and sales, deleting a record from countries may follow to delete cascade related records from customers and then sales. This may force the triggers on customers and then sales to be executed for no benefit. This problem can be resolved by adding some mechanism.

The research, however, has certain limitations and they may be suggestion for further work:

—Many limitations on the queries creating MV are archived.

—Triggers are generated in PL/pgSQL language. Clearly, manipulating data with machine language or relational algebra may be more effective.

—The impact of triggers used to update MV table has yet been assessed for DBMS performance in general. Furthermore, using standardized DBMS evaluation has not been applied, e.g. TPC standards, to evaluate obtained results. In fact, the evaluation of a DBMS by TPC standards is a separate process, requiring considerable investment.

## CONCLUSIONS

In summary, the main contributions of the research to the field include two contents. With incremental update algorithm, the research integrated some advantages of previous algorithms, and carried out four optimization cases when considering the case in which (i) several columns create the key for BT $T_i$ participating in $G$; (ii) several columns of $T_i$ only participate in $G$ and several columns create the key of $T_1$

and participate in $G$; (iii) $T_i$ has a special role in from clause; as well as (iv) performed optimization by proposing the methods including separating update into delete and insert with queries including aggregate functions MIN(E), MAX(E) and leaving it the same for queries without ones. In addition, the research also built a program that automatically generates codes for triggers for all events to all tables involved in any MV creating queries. The generated triggers implement the built incremental update algorithm synchronously as part of the transaction that does the data changes in BT. This research also tested the generated triggers for the correctness and comparison of time needed to carry out the data update in BT.

Features of synchronous incremental update is fully integrated into the source code of PostgreSQL may be more optimal plan. However, the proposed solution requires almost no code tuning for each version of DBMS because it is relatively independent with versions of the DBMS.

## REFERENCES


1. J. A. Blakeley, P.-A. Larson, and F. W. Tompa, "Efficiently updating materialized views," SIGMOD Rec. **15** (2), 61–71 (1986).
2. A. Gupta, I. S. Mumich, and V. S. Subrahmanian, "Maintaining views incrementally," *Materialized Views*, Ed. by A. Gupta and I. S. Mumick (MIT Press, 1999), pp. 177–190.
3. A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *Materialized Views*, Ed. by G. Ashish and M. Iderpal Singh (MIT Press, 1999), pp. 145–157.
4. J. Zhou, P.-A. Larson, and H. G. Elmongui, "Lazy maintenance of materialized views," *Proceedings of the 33rd International Conference on Very Large Data Bases* (VLDB Endowment, Vienna, Austria, 2007), pp. 231–242.
5. D. K. Gupta and I. S. Mumick, *Counting Solutions to the View Maintenance Problem*, Tech. Rep. AT&T Bell Laboratories, 1992.
6. K. Y. Lee and M. H. Kim, "Optimizing the incremental maintenance of multiple join views," *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP* (ACM, Germany, Bremen, 2005), pp. 107–113.
7. P.-A. Larson, "Maintenance of materialized views with outer-joins," *Encyclopedia of Database Systems*, Ed. by L. Liu and M. T. ÖZsu (Springer US, 2009), pp. 1670–1674.
8. A. Nica, "Incremental maintenance of materialized views with outerjoins," Inf. Syst. **37** (5), 430–442 (2012).
9. O. Shmueli and A. Itai, "Maintenance of views," SIGMOD Rec. **14** (2), 240–255 (1984).
10. D. Chak, Materialized views that really work, 2008. http://www.pgcon.org/2008/schedule/attachments/64_BSDCan2008-MaterializedViews-paper.pdf. Cited September 15, 2014.
11. H. Gupta and I. S. Mumick, "Incremental maintenance of aggregate and outerjoin expressions," Inf. Syst. **31** (6), 435–464 (2006).
12. D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy, "Answering Queries with aggregation using views," *Proceedings of the 22th International Conferences on Very Large Data Bases* (Morgan Kaufmann Publishers Inc., 1996), pp. 318–329.