Evaluation of an Implementation of Cross-Row Constraints Using Materialized Views

José María Cavero Barca
Rey Juan Carlos University
C/ Tulipán s/n
28933 Móstoles (Spain)
josemaria.cavero@urjc.es

Belén Vela Sánchez
Rey Juan Carlos University
C/ Tulipán s/n
28933 Móstoles (Spain)
belen.vela@urjc.es

Paloma Cáceres García de Marina
Rey Juan Carlos University
C/ Tulipán s/n
28933 Móstoles (Spain)
paloma.caceres@urjc.es

ABSTRACT

SQL assertions are a powerful means used to specify cross-row constraints, and have been available in the SQL standard since 1992. Unfortunately, assertions are not supported in commercial database management systems. Although triggers and application programs can be efficiently used to constrain database content, they are more complex to write and more error-prone. The objective of this paper is to analyze whether the use of materialized views could be a viable solution as regards the automatic implementation of SQL assertions. Materialized views are views that physically store the result of a query and are periodically updated. The method consists of defining a materialized view which contains the number of tuples that violate the condition expressed in the assertion. The materialized view will contain a CHECK constraint that guarantees that the number of tuples that violate the assertion is equal to zero. The proposed method is an easy and automatic means of implementing the integrity constraints described using assertions. We have carried out a series of tests, and although triggers perform better than materialized views in most situations, there are some in which materialized views would be an efficient option. They are easily automatable and less error prone than triggers.

1. INTRODUCTION

In relational databases, an integrity constraint is basically a Boolean expression that must be evaluated as TRUE [1]. A database integrity constraint, therefore, constrains the values that can appear in a given database.

The standard language for relational databases (Structured Query Language, SQL) provides several means to deal with integrity constraints [2]: table constraints, column constraints, domain constraints and assertions. The first three include UNIQUE, PRIMARY KEYS, FOREIGN KEYS and CHECKS, and are supported in most commercial Relational Database Management Systems (RDBMSs). The last (assertions) are the most general form of integrity constraint, and they are a simple and easy method by which to enforce cross-row constraints (that is, constraints across related rows, possibly in different tables). In short, their structure includes a condition that must be fulfilled. The RDBMS is responsible for ensuring that the condition is satisfied in every state of the database.

Unfortunately, although assertions have been part of the SQL standard since 1992 [3,4], commercial RDBMSs do not support assertions, and many cross-row integrity constraints are, therefore, usually implemented by means of triggers (which are used "to detect some conditions that happen in a database and then react to the database" [5]), or are included in the applications used to access the database. Recently, a few works related with constraints that affect collectively several tables have appeared. For example, in [6] the authors propose adding more functionality to core SQL by means of package queries to support constraints that the set of result rows to a query must satisfy. However, if a RDBMS supported assertions, it would be easy to control integrity constraints, thus eliminating the need to use triggers or application programs to control the integrity of the database or any other mechanism. While assertions only specify the condition that must be fulfilled to satisfy the constraint, triggers and application programs must be programmed by taking into account every possible situation that could violate the constraint.

In this paper we show an automatic implementation of assertions using materialized views. Materialized views are database objects that contain the result of a query at a specific time, are updated from time to time, and are used to increase the speed of queries in very large databases. As will be shown in the following sections, the implementation of the method requires a RDBMS that supports materialized views and that can include CHECK constraints, along with an automatic procedure that can be used to refresh the view. We have chosen the Oracle database [7] since we have prior expertise in using it, and because it supports all the conditions that must be satisfied in order to carry out the implementation and even some additional features. This approach has some basic advantages that

could make it useful in certain environments, although in many situations it performs worse than when using triggers.

The organization of the remainder of this paper is as follows. In Section 2, we summarize SQL assertions and the example that will be used in the rest of the paper. Section 3 focuses on materialized views and how they can be used to implement assertions. Section 4 shows the results of some tests in which triggers are compared with materialized views. Finally, Section 5 discusses the results and presents some conclusions.

2. SQL ASSERTIONS

In standard SQL, users can specify general constraints via the CREATE ASSERTION statement. An SQL assertion is "a CHECK constraint at the database level that is allowed to contain queries" [8]. One basic technique employed to write assertions is that of specifying a query that selects the tuples that violate the condition. By including this query in a NOT EXISTS clause, the assertion will specify that the query result must be empty. Any constraint that can be expressed in terms of a query that selects the tuples that violate the desired constraint could, therefore, be expressed using an assertion in the following way:

```
CREATE ASSERTION Assertion_Name
CHECK (NOT EXISTS (Query that selects the tuples that violate the constraint));
```

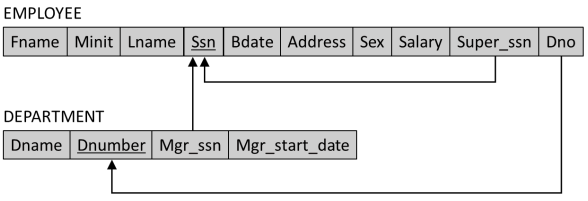

Fig. 1. Relational database schema.

In the remainder of the paper, we use a simplified version of an example regarding the storage of information concerning employees and departments from [9]. Every Employee belongs to one Department, and every Department has an Employee who works as a Manager. The schema is shown in Figure 1.

The following assertion constrains "that the salary of an employee must not be greater than the salary of the manager of the department that the employee works for":

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS
       (SELECT *
       FROM EMPLOYEE E,
            EMPLOYEE M,
            DEPARTMENT D
       WHERE E.Salary > M.Salary
            AND E.Dno = D.Dnumber
            AND D.Mgr_ssn = M.Ssn));
```

This simple assertion guarantees that none of the employees violate the condition in any state of the database. The RDBMS should reject insertions, deletions and updates that make an employee's salary greater than that of the manager of his/her department.

None of the most frequently used commercial RDBMS support assertions, although some attempts have been made to build support for assertions, particularly in Oracle. For example, back in 2016, this possibility was considered in Oracle's Database Ideas [8]. Implementation alternatives were proposed, and complexity and performance problems that could appear were also discussed. It would, however, appear that the idea was eventually discarded owing to its complexity.

As mentioned previously, no commercial product supports assertions. A previous assertion is, therefore commonly implemented in the application programs, or by using a set of triggers that should be executed in the case of events that could provoke a violation of the constraint. A very simple assertion like that previously described could be implemented by, for example, using four triggers that will be fired when an employee is inserted or updated, when the manager of a department is updated, etc. Sometimes, a lot of situations have to be taken into account when programming triggers to support constraints, that is, sometimes, as stated in [10] "transforming integrity constraints into triggers for verifying database consistency produces a serious and complex problem". Moreover, "manually checking integrity constraint enforcement at the application level is usually difficult, as the code base to be examined could be large" [11].

Therefore, although triggers and application programs can be efficiently used to constrain database content, they are more complex to write and more error-prone. In the following section, we show how to implement constraints in a RDBMS by simulating assertions and using materialized views.

3. IMPLEMENTING ASSERTIONS USING MATERIALIZED VIEWS

An assertion checks that the condition that follows the keyword CHECK holds true for every database state. The NOT EXISTS (query) clause returns TRUE if the query returns no tuples. This is equivalent to saying that the result of applying the COUNT function to the query must be zero. That is, the previous condition is

equivalent to saying that the following query returns zero:

```
SELECT COUNT (*) AS Invalid_Tuples
FROM
       (SELECT *
       FROM EMPLOYEE E,
              EMPLOYEE M,
              DEPARTMENT D
       WHERE E.Salary > M.Salary
              AND E.Dno = D.Dnumber
              AND D.Mgr_ssn = M.Ssn));
```

In a RDBMS, a view is a virtual table that represents the result of a query. If a view contains the result of the previous SELECT, a simply CHECK column constraint (for example, CONSTRAINT CK_ASC CHECK (Invalid_Tuples = 0)) should therefore be sufficient to guarantee this constraint. Unfortunately, traditional views are not allowed to include CHECK constraints. Simple integrity constraints stated in a view could imply very complex constraints in the base tables (base tables are the tables from which the view obtains the data). Nevertheless, CHECK constraints can be included in a kind of views called materialized views, as will be shown in the following paragraphs.

Some RDBMSs also implement "materialized views". Materialized views were first implemented in the Oracle Database. A materialized view (also called a snapshot) is a database object that contains the result of a query at a specific time, and is periodically updated on the basis of certain criteria. A materialized view eventually enables efficient access to the cost of some data that may potentially be out-of-date. In Oracle, the content of a materialized view may be updated manually or automatically: for example, every day by using the clause START WITH SYSDATE NEXT SYSDATE + 1, or when a commit occurs in the base tables by using the ON COMMIT clause. Materialized views are very useful in data warehousing environments, in which frequent queries regarding historical data can be expensive.

In Oracle, materialized views can include CHECK constraints, as common tables (or can even be stored in a previously created table). It would, therefore, be possible to define CHECK constraints in order to constrain the values stored in the materialized view. This would, in turn, constrain the values of the base tables used in the query defined in the materialized view.

Oracle therefore satisfies all the conditions that must be fulfilled in order to automatically implement assertions using materialized views. The basic procedure is as follows: a materialized view has to be created for each assertion. The materialized view will contain only one column, Invalid_Tuples, which will contain the number of tuples that violate the constraint specified in the assertion. Moreover, a CHECK column constraint constrains that the value of the column must be equal to zero.

The refresh method used will be ON COMMIT. This guarantees that for every commit the materialized view will be updated and, therefore, if the CHECK column constraint is violated, the operation that provoked the commit will be rejected. We therefore guarantee that when a/some invalid tuple/s appear/s, the materialized view is immediately updated. This obviously implies that if a particular operation in a transaction violates the constraint, the whole transaction will be rolled back instead of committed

We, therefore, first create the materialized view with a REFRESH ON COMMIT option:

```
CREATE MATERIALIZED VIEW
  ASSERTION_SALARY_CONSTRAINT
  REFRESH ON COMMIT
  AS SELECT COUNT(*) AS Invalid_Tuples
      FROM
      (SELECT *
      FROM EMPLOYEE E,
            EMPLOYEE M,
            DEPARTMENT D
      WHERE E.Salary > M.Salary
            AND E.Dno = D.Dnumber
            AND D.Mgr_ssn = M.Ssn);
```

We then modify the materialized view using an ALTER TABLE sentence, adding a CHECK constraint that guarantees that the number of invalid tuples is always zero:

```
ALTER TABLE
ASSERTION_SALARY_CONSTRAINT
       ADD CONSTRAINT CK_ASC
          CHECK (Invalid_Tuples = 0);
```

If we make some updates in the database that cause an employee to have a salary that is greater than that of the manager of the department that the employee works for, we will obtain the following error message when a commit occurs:

```
Error SQL: ORA-12008: error in
materialized view refresh path

ORA-02290: check constraint
(SYSTEM.CK_ASC) violated
```

The main disadvantage of the REFRESH ON COMMIT option is that the time required to complete the commit will be longer because of the extra

processing involved. Although this should not be an issue in a data warehouse environment, because it is unlikely that concurrent processes will be attempting to update the same table [12], this could be a problem in a transactional environment. Oracle has an option (the FAST REFRESH option) that can be used to improve the performance of the refreshing. The FAST REFRESH option performs an incremental refresh and requires the creation of a series of materialized view logs that store the changes made to the base tables since the last commit. The FAST REFRESH option also has some restrictions, such as the type of SELECT, aggregations, remote tables, etc [13].

4. RESULTS

We have carried out various experiments with the example shown in Sections 2 and 3, in an Oracle database, version 12c. Each execution of the example consisted of creating the tables and inserting the test data from scratch, in order to avoid malfunctions owing to data kept in the cache. We have carried out various tests applied to the previous example in order to compare the use of a set of triggers with that of materialized views (one normal and one using the FAST REFRESH option). We have specifically developed four triggers, because there are four situations in which the constraint can be violated:

- The first three are fired when a new employee is inserted, when the salary of an existing employee is modified, or when an existing employee is assigned to a different department. These triggers share virtually the same code, and need only compare the salary of the new or existing employee with the salary of the manager of the department.

- The fourth trigger is fired when the manager of an existing department is modified. In this case, the trigger needs to check that the salary of every employee in the department is not greater than the salary of the new manager. Note that this trigger does not need to be fired when a new department is inserted, because in this case no employees are still assigned to it.

Table 1. Inserting 100 employees (seconds)

	ONE FINAL COMMIT			ONE COMMIT AFTER EACH INSERT		
	MATERIALIZED VIEW	MATERIALIZED VIEW (FAST REFRESH)	TRIGGERS	MATERIALIZED VIEW	MATERIALIZED VIEW (FAST REFRESH)	TRIGGERS
10,000 EMPLOYEES	0.0765	0.1190	0.0264	0.9057	2.2282	0.0253
50,000 EMPLOYEES	0.0886	0.1182	0.0247	2.2106	2.4138	0.0271
100,000 EMPLOYEES	0.1098	0.1229	0.0253	3.7018	2.8495	0.0278
200,000 EMPLOYEES	0.1314	0.1266	0.0261	6.7539	3.4744	0.0293
300,000 EMPLOYEES	0.1650	0.1357	0.0254	9.8549	4.1711	0.0277

Table 2. Updating 100 managers (seconds)

	ONE FINAL COMMIT			ONE COMMIT AFTER EACH UPDATE		
	MATERIALIZED VIEW	MATERIALIZED VIEW (FAST REFRESH)	TRIGGERS	MATERIALIZED VIEW	MATERIALIZED VIEW (FAST REFRESH)	TRIGGERS
10,000 EMPLOYEES	0.0564	0.1102	0.0701	0.9656	2.9299	0.0934
50,000 EMPLOYEES	0.0666	0.1118	0.3174	2.2747	3.2582	0.3738
100,000 EMPLOYEES	0.0836	0.1178	0.6468	3.8715	3.8114	0.7125
200,000 EMPLOYEES	0.1182	0.1256	1.3163	7.1413	4.4121	1.4020
300,000 EMPLOYEES	0.1445	0.1370	1.9195	10.4417	5.3521	2.0144

Table 1 shows the average results obtained after repeating the insertion of 100 new employees into the EMPLOYEE table 10 times with 10,000, 50,000, 100,000, 200,000 and 300,000 tuples (in seconds). The first three columns show the results when only one commit is executed after the 100 insertions. The last three columns show the results when one commit is forced after each insertion.

The performance of the triggers is similar in both cases (with or without commits), and the size of the table has no significant effect, because the trigger only compares the salary of the employee with the salary of the manager of the department. Moreover, the triggers clearly perform better than materialized views, especially in the case of forcing a commit after each insert. In the case of

materialized views, the FAST REFRESH option only has a clear effect when the tables affected have a lot of tuples and a lot of commits have to be done. It seems reasonable that in almost empty tables the fact of having the additional task of managing a set of materialized view logs does not have any effect, or even a negative effect.

Table 2 shows the average results obtained after repeating the modification of 100 managers 10 times, again when the EMPLOYEES table has 10,000, 50,000, 100,000, 200,000 and 300,000 tuples.

In this case, the performance of the materialized views is similar to the previous one (Table 1), but the performance of the triggers has worsened. In this case, the performance of the triggers is affected by the size of the tables, because this trigger has to compare the salary of the new manager with the salary of all the employees in the department.

The main difference between both results (Table 1 and Table 2) is the performance of the triggers. The triggers that fire in the case of inserting a new employee (Table 1), or modifying an existing employee, only have to compare his/her salary with the salary of the manager of the department to which he/she belongs. Nevertheless, the trigger that fires in the case of the modification of the manager of one department (Table 2) is more complex, because it has to check that every employee in that department has a salary which is not greater than the salary of the new manager.

The following figures provide a graphic summary of the aforementioned tests. They show the performance of the triggers and materialized views after inserting 100 employees plus modifying the manager of a department 100 times (that is, the result of adding the results of Table 1 and Table 2). Figure 2 shows the results when a single commit is executed after each 100 operations and Figure 3 shows the result of forcing one commit after each operation. A linear trend line has also been added for each situation.

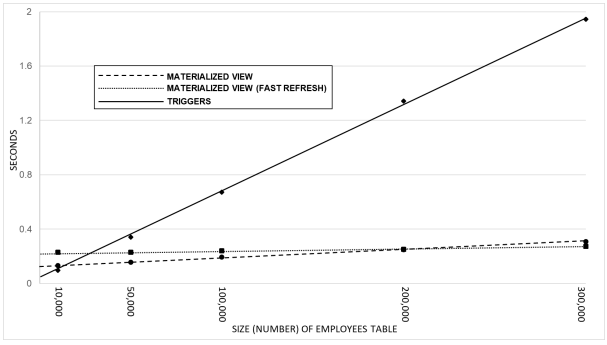

Figure 2. Inserting 100 employees and updating 100 managers, one final commit.

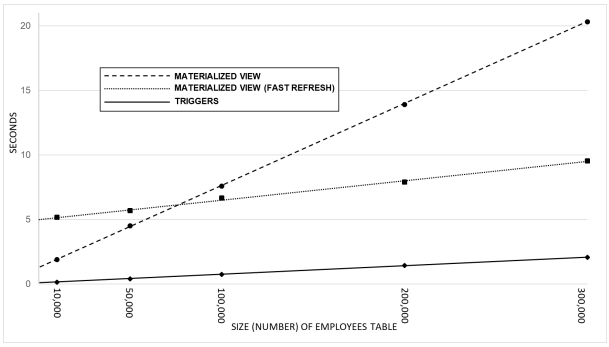

Figure 3. Inserting 100 employees and updating 100 managers, one commit after each insert/update.

As can be seen, the trend line fits very well with the results obtained. In Figure 2, when commits are executed only after the operations, triggers are the worst option when the Employees table has more than 20,000-25,000 tuples. With regard to the FAST REFRESH option, it starts to provide its benefits when the Employees table has about 200,000 tuples.

In Figure 3, when one commit is forced after each operation, triggers are always the best option. Although materialized views perform reasonably well in small tables, their performance worsens in the case of large tables, especially when the FAST REFRESH option is not used. The FAST REFRESH option starts to provide its benefits when the Employees table has about 75,000 tuples.

5. CONCLUSION

In an RDBMS, almost any constraint can be specified as an SQL query. Virtually any constraint could, therefore, be expressed using an assertion. Unfortunately, no commercial RDBMS product supports assertions. This signifies that cross-row constraints are usually implemented using a set of complex triggers, or are included in the applications. In some cases, this method is more efficient, although it is error-prone owing to the quantity of possible situations that must be taken into account. In some situations, it might therefore be better to automate this codification task and delegate it to the RDBMS.

In this paper, we have proposed an automatic and easy means of implementing assertions in RDBMSs which support materialized views and allow an ON COMMIT refresh of these views.

To the best of our knowledge, the only approach for the automatic implementation of assertions is that of Oriol et al. [14], who propose an incremental approach consisting of implementing assertions that create several triggers in order to capture the update requested by the user and placing it in auxiliary tables in SQL Server. Their method consists of generating a set of triggers and views that check the assertions, all in a semi-automatic manner (manual intervention is necessary because the user has to

manually invoke a procedure called safeCommit at the end of each transaction). The results provided by the authors in order to check the assertions range from 0.01 to 1.29 seconds, which are similar to those obtained in our experiments in the case of using triggers, signifying that they are reasonable. Unfortunately, the aforementioned authors' method does not take into account the possibility of including aggregation functions in assertions, which is a fundamental disadvantage, since it works only for very simple assertions.

Our main objective was to analyze whether implementing assertions by means of materialized views is a viable method. We have shown that it is possible, and that it can be easily automated. Although our proposal is specific to Oracle, it would be applicable to any system that supports CHECK constraints and REFRESH ON COMMIT on materialized views. However, to the best of our knowledge, no system other than Oracle currently supports all of these aspects.

We have also carried out some preliminary tests in order to evaluate the efficiency of our approach. These tests showed that the method provides good results (even better than triggers) when making a unique commit after a set of inserts or updates. Nevertheless, it performs worse than when using a set of triggers when a commit is forced after each insertion or update. The following conclusions may be obtained from the aforementioned experiments:

- Materialized views perform similarly, independently of the cause that violates the constraint. It depends mainly on the size of the table.

- The performance of the triggers depends mainly on the particular constraint, and less on the size of the table.

- Using the FAST REFRESH option in materialized views provides benefits only in large tables.

- In general, materialized views perform better than triggers in the case of issuing a single commit for all the modifications made, as the condition is evaluated only once (when the commit is performed). In the case of the triggers, which are part of the same transaction of the firing statement, the condition is evaluated for each operation. The performance of both triggers and materialized views obviously also depends on the complexity of the condition to be evaluated and the operation that can fire the assertions.

Our method, therefore, fits better in environments with a high number of complex constraints but a low number of transactions. It is less appropriate for environments in which a high number of simple transactions are present. We are currently working on a detailed categorization of when it is worth using this method, along with its automatic implementation in a tool.

6. ACKNOWLEDGMENTS

This work has been partially supported by the Access@City research project (TIN2016-78103-C2-1-R), funded by the Spanish Ministry of Science, Innovation and Universities.

7. REFERENCES

[1] C.J. Date, 2015. SQL and Relational Theory: How to Write Accurate SQL Code. Third ed., O'Reilly.

[2] A. Behrend, R. Manthey and B. Pieper, 2001. An Amateur's Introduction to Integrity Constraints and Integrity Checking in SQL, Datenbanksysteme in Büro, Technik und Wissenschaft. A. Heuer, F. Leymann and D. Priebe, eds, Informatik aktuell. Springer, Berlin, Heidelberg, 405-423.

[3] ANSI Standard, 1992. The SQL 92 Standard. http://savage.net.au/SQL/sql-92.bnf.htm

[4] J. Melton and A. R. Simon, 2002. SQL 1999: Understanding Relational Language Components, Morgan Kaufmann.

[5] Y.I. Chang and F.L. Chen, 1997. RBE: A Rule-by-example Active Database System, Software: Practice and Experience, 27(4):365-394.

[6] M. Brucato, A. Abouzied and A. Meliou, 2017. A Scalable Execution Engine for Package Queries. SIGMOD Record, 46(1): 24-31.

[7] Oracle. http://www.oracle.com

[8] T. Koppelaars, 2016. SQL Assertions / Declarative multi-row constraints". https://community. oracle.com/ideas/ 13028.

[9] R. Elmasri and S. Navathe,2010. Fundamentals of Database Systems, Sixth ed., Addison-Wesley.

[10] H.T. Al-Jumaily, D. Cuadra and P. Martínez, 2008 "OCL2Trigger: Deriving active mechanisms for relational databases using Model-Driven Architecture", Journal of Systems and Software, 81(12):2299-2314.

[11] H. Zhang, H.B.K. Tan, L. Zhang, X. Lin, X. Wang, C. Zhang and H. Mei, 2011. Checking enforcement of integrity constraints in database applications based on code patterns", Journal of Systems and Software, 84(12):2253-2264.

[12] Oracle Database SQL Language Reference, 11g Release 2 (11.2).http://docs.oracle.com/cd/E11882_01/server.112/e41084.pdf

[13] P. Lane and P. Potineni, 2014. Oracle Database Data Warehousing Guide, 12c Release 1 (12.1). Oracle.

[14] X. Oriol, E. Teniente and G. Rull, 2016. TINTIN: a Tool for Incremental INTegrity checking of Assertions in SQL Server", 19th International Conference on Extending Database Technology (EDBT): 632-635.