Demonstration

TINTIN: a Tool for INcremental INTegrity checking of Assertions in SQL Server

Xavier Oriol
Universitat Politècnica de Catalunya
xoriol@essi.upc.edu

Ernest Teniente
Universitat Politècnica de Catalunya
teniente@essi.upc.edu

Guillem Rull
Universitat de Barcelona
Barcelona, Spain
grull@ceipac.ub.edu

ABSTRACT

We present TINTIN, a tool to perform efficient integrity checking of SQL assertions in SQL Server. TINTIN rewrites each assertion into a set of standard SQL queries that, given a set of insertions and deletions of tuples, allow to incrementally compute whether this update violates the assertion or not. If one of such queries returns a non empty answer, then the assertion is violated. Efficiency is achieved by evaluating only those data and those assertions that can actually be violated according to the update. TINTIN is aimed at two different purposes. First, to show the feasibility of our approach by implementing it on a commercial relational DBMS. Second, to illustrate that the efficiency we achieve is good enough for making assertions to be used in practice.

Keywords

Integrity checking, SQL, Assertions

1. INTRODUCTION

In standard SQL, users can specify general constraints using the `CREATE ASSERTION` statement. The basic technique for writing assertions is to specify a query that selects those tuples that violate the desired condition. By including this query inside a `NOT EXISTS` clause, the assertion will specify that the query result must be empty. Thus, the assertion is violated if and only if the query result is not empty [2].

Assertions were initially defined in SQL-92 [1] and they serve as a means for expressing global integrity constraints not tied to a particular table, but ranging over several ones. They are sufficient for expressing most constraints since almost the full expressiveness of SQL can be used to define the query inside the `NOT EXISTS` clause. It is also well known that many integrity constraints can only be expressed via assertions since the other constructs provided by SQL are not powerful enough. Thus, assertions provide an elegant way to define general constraints in SQL.

However, assertions are still not supported by any of the most well-used commercial RDBMS (Oracle, MySQL, SQL Server, PostgreSQL, DB2). It might be argued that assertions can be emulated via manually writing a set of triggers, which is a widely supported feature of RDBMS. However, its manual definition is error prone and the whole set of necessary triggers to write might not be evident when given a complex constraint, thus, compromising the integrity of the data if just one trigger is missing or ill-defined. Hence, it is better to delegate this complex checking code to RDBMS capabilities [8], as we do in TINTIN[1].

TINTIN is a tool that provides incremental integrity checking of assertions in SQL Server. Given an SQL Server DB, and a set of SQL assertions written on its schema, TINTIN automatically builds all the necessary procedures/queries to efficiently check whether any update satisfies the assertions.

As an example, consider the schema of the well-known TPC-H benchmark [7] shown in Figure 1, a benchmark for illustrating decision support systems that examine large volumes of data, execute complex queries, and give answers to critical business questions.

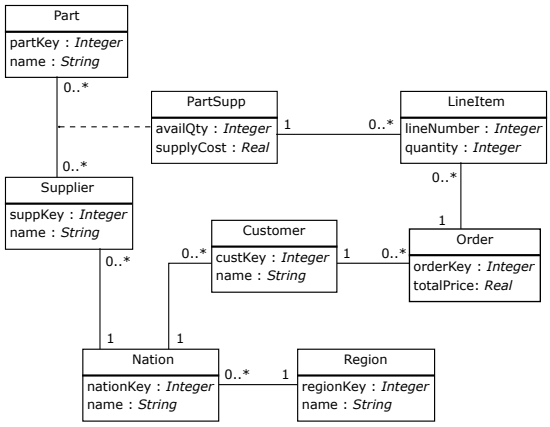

Figure 1: The TPC-H Schema.

Now, assume that we want to define a general constraint over the previous schema stating that all orders have at least one line item. This constraint could be specified by means of the SQL assertion shown below:

```
CREATE ASSERTION atLeastOneLineItem CHECK(
  NOT EXISTS(
      SELECT * FROM ORDERS AS o
      WHERE NOT EXISTS (
        SELECT * FROM LINEITEM AS l
        WHERE l.L_ORDERKEY = o.O_ORDERKEY)));
```

[1]http://www.essi.upc.edu/~xoriol/tintin/

TINTIN allows checking the assertion *atLeastOneLineItem* efficiently in data sets consisting of 1GB to 5GB of data and with 1MB to 5MB of tuple insertions/deletions, with times ranging from 0.01 to 0.04 seconds depending on the scenario. These results are much better than the time required for directly executing the query inside the assertions on the database, ranging from x89 to x2662 times faster.

The approach we follow in TINTIN is to automatically generate, for each assertion, several standard SQL queries which incrementally determine whether the update violates the original assertion or not. If the queries return a non-empty answer, then, the assertion is violated, otherwise, it is satisfied. The queries are incremental in the sense that they are stated in terms of the current database tables and also some automatically generated auxiliary tables containing the insertions/deletions of tuples requested by the user.

This is the crucial point for achieving efficiency of integrity checking. When a user requests an update, the tuples s/he wants to insert/delete are put in some auxiliary tables. Then, the generated queries join these tuples being inserted/deleted with the current data, and return those that violate the assertions. For each assertion, the generated queries only join those insertions/deletions of tuples that might cause its violation. Therefore, an update not including any of those insertions/deletions trivially makes the query result to be empty. For this reason, no current data of the database is accessed unless some update may cause the violation of an assertion.

The join between the current data and the tuples being inserted/deleted ensures also that only the tuples affected by the update are checked. Thus, the rest of the data, potentially the major part of the database, is skipped.

These incremental SQL queries are generated outside the database and then stored in it as views. Since we use standard SQL to define them, they could be used for checking assertions on any relational DBMS. However, and for the purpose of checking the feasibility and the efficiency of our approach, we have chosen SQL Server to implement TINTIN because of our previous expertise on this system.

Once the incremental SQL queries are defined, TINTIN builds a stored procedure called `safeCommit` to allow SQL Server to check the assertions. This procedure must be invoked at the end of each transaction, so that the procedure can check whether it violates any of the assertions. More specifically, the procedure checks whether the incremental SQL queries are empty or not. If they are, the update does not violate any assertion, so, the procedure commits the update stored in the auxiliary tables. Otherwise, the procedure shows the tuples answering the queries, i.e., the tuples violating the assertions.

2. PROBLEM AND SOLUTION

Integrity checking is the problem of efficiently determining whether a given update satisfies a set of integrity constraints, SQL assertions in our case. This is an important problem in data management since any violation of an integrity constraint would indicate an invalid state of the database. One possible way to achieve efficiency relies on, first, just checking the assertions which may actually be violated by the update and, second, considering only the relevant updated data for computing whether the assertions are violated.

We assume in this work that the update consists of a (possibly large) set of insertions and/or deletions of database tuples and that the queries defining the assertions are specified by means of the fragment of SQL that is equivalent to relational algebra. In particular, TINTIN accepts assertions to be defined through selection, projection, join, subselect (`exists`, `in`), negation (`not exists`, `not in`) and union but it does not allow functions (e.g. aggregates, arithmetic functions) for the moment.

TINTIN is aimed at providing to an SQL Server database with the capability of performing integrity checking of SQL assertions efficiently. For this purpose, TINTIN allows a user to specify assertions according to the fragment of SQL stated above, and the tool automatically builds a stored procedure in the database, called `safeCommit`, that the user will have to call at the end of each transaction. Whenever called, `safeCommit` checks whether the updates in the transaction violate any of the assertions. If no violation is found, the update is committed to the database. Otherwise, it provides the answers to the queries that detected the violation of the assertions.

The `safeCommit` procedure works by executing several SQL queries, stored as views in the database, for each one of the assertions that need to be checked. Each query captures a different situation in which some updates may lead to the violation of the assertion. These updates are explicitly stated in the query definition itself and provide the key for efficiency of integrity checking.

In the rest of this section we explain the approach we follow to obtain the SQL queries that allow checking incrementally an assertion; which is based on our previous work for handling integrity checking of OCL constraints in conceptual models [4, 5]. We only require the users to define their desired assertions. From there, all the following steps are automatically performed.

The first step consists in rewriting each SQL assertion into a logic *denial* in the same way as we did in [6]. A denial is a formula stating a condition that must not be true in any state of the database. These denials are the basis for obtaining the incremental SQL queries.

For instance, the assertion `atLeastOneLineItem` of our running example would be rewritten as:

$$order(o) \wedge \neg lineIt(l, o) \rightarrow \bot \quad (1)$$

Clearly, the previous denial states that if there is an order o without any line item l, an inconsistent state will be reached, which is exactly the condition to be avoided by `atLeastOneLineItem`.

Then, for each denial, TINTIN obtains its corresponding *Event Dependency Constraints* (EDCs, for short). Each EDC is a logic rule identifying a particular situation where some update applied to a certain state of the database D causes the violation of the denial, i.e., of the corresponding assertion. The main idea for obtaining EDCs is to replace each literal in the logic rule obtained from the assertion by the expression that evaluates this literal in the new state of the database D^n, i.e., the state obtained after applying the update. Positive and negative literals in the denial are handled in a different way according to the following formulas:

$$\forall \overline{x}.\ p^n(\overline{x}) \leftrightarrow (\iota p(\overline{x})) \vee (\neg \delta p(\overline{x}) \wedge p(\overline{x})) \quad (2)$$

$$\forall \overline{x}.\ \neg p^n(\overline{x}) \leftrightarrow (\delta p(\overline{x})) \vee (\neg \iota p(\overline{x}) \wedge \neg p(\overline{x})) \quad (3)$$

Rule 2 states that a literal $p(\overline{x})$ will be true in the new state of the database D^n if it has been inserted or if it was already true in the initial state D and it has not been deleted.

In an analogous way, rule 3 states that $p(\overline{x})$ will not hold in D^n if it has been deleted or if it was already false and it has not been inserted.

By applying the substitutions above to all logic denials, we get a set of EDCs which states all possible ways to violate the assertions by means of insertions and/or deletions of tuples.

In particular, we get the following EDCs for the denial 1 of our running example:

$$\iota order(o) \wedge \neg lineIt(l,o) \wedge \neg \iota lineIt(l,o) \rightarrow \bot \quad (4)$$

$$\iota order(o) \wedge \delta lineIt(l,o) \wedge \neg aux(o) \rightarrow \bot \quad (5)$$

$$order(o) \wedge \neg \delta order(o) \wedge \delta lineIt(l,o) \wedge \neg aux(o) \rightarrow \bot \quad (6)$$

$$aux(o) \leftarrow \iota lineIt(l,o)$$

$$aux(o) \leftarrow lineIt(l,o) \wedge \neg \delta lineIt(l,o)$$

Intuitively, EDC 4 states that *atLeastOneLineItem* will be violated if a new order o is inserted and there was no line item for o in the initial state of the database and no line item for o has been inserted by the transaction. EDC 5 behaves in a similar way, while EDC 6 determines that the assertion will be violated if a line item for an existing order o has been deleted and neither a new line item has been inserted for o nor the database contains any other line item for o (given by the rules defining $aux(o)$).

Note that, in this example, EDC 5 can be safely discarded assuming that the foreign key constraint from *lineitem* to *order* is satisfied in the current state of the data. TINTIN incorporates some semantic optimizations like this one that allow obtaining a reduced and simplified number of EDCs which allow performing integrity checking more efficiently.

The idea of obtaining EDCs to identify the different situations that may lead to the violation of a constraint is grounded on the concept of *event rules* [3], which were aimed at performing integrity checking in deductive databases.

Finally, each EDC is translated into an SQL query as proposed in [4]. Roughly, each positive literal in the EDC is translated into a table reference placed in the `FROM` clause of the query, possibly with a `JOIN` condition with some previously translated literal that shares a common variable with it. Built-in literals and constant bindings are directly translated to the `WHERE` clause, and negated base and derived literals are translated as correlated subqueries.

In our running example, we would translate EDC 4 as:

```
CREATE VIEW atLeastOneLineItem1 AS
SELECT *
FROM ins_orders AS T0
WHERE NOT EXISTS(SELECT *
    FROM lineitem AS T1
    WHERE T1.l_orderkey = T0.o_orderkey)
  AND NOT EXISTS(SELECT *
    FROM ins_lineitem AS T1
    WHERE T1.l_orderkey = T0.o_orderkey)
```

We have defined the query as a view to store it into the database. It is worth noting the usage of the auxiliary tables storing the insertions and the deletions of tuples for each table of the database, as it happens with *ins_orders* and *ins_lineItem* in the previous view. TINTIN automatically builds them, together the necessary triggers to capture the insertions/deletions of tuples to place them into such auxiliary tables. Thus, the existence and maintenance of these auxiliary tables is fully transparent to the database users.

Note also that the key for incrementality is not based on batching updates for delaying the assertions checking, but on the join in the SQL queries between the update and the current data. First of all, any SQL query joining an insertion/deletion which is not being applied (i.e., whose corresponding SQL table is empty) is immediately discarded since it trivially returns the empty set. Therefore, we only check those constraints that can be violated according to the ongoing update. Second, the data considered by an SQL query during its execution is necessarily the data joining the update applied, thus, avoiding to look through all the database.

3. DEMO DESCRIPTION

The demo that we will present is intended to show the usage and efficiency of our prototype tool TINTIN by means of applying it to the checking of some assertions in the TPC-H benchmark SQL schema.

We will first request TINTIN to build the necessary auxiliary tables and triggers to capture any insertion and deletion applied to the TPC database. As a result, we will see a newly generated database event_TPC with an ins/del table for each TPC SQL table. At this point, whenever we apply an insertion/deletion of any tuple in TPC, the tuple will be captured and inserted in the corresponding ins/del table of event_TPC. In this way, the contents of TPC remains unchanged, and event_TPC contains the requested update.

Next, we will introduce in TINTIN some SQL assertions of different complexity. Consequently, TINTIN will create a procedure called `safeCommit` in TPC. This procedure, when called, will check whether applying the updates contained in event_TPC raises the violation of any assertion. If no violation is found, the procedure will commit the events into TPC; otherwise, it will report the violations. Lastly, the procedure will truncate all the events stored in event_TPC, so that a new set of events can be proposed.

At this stage, TINTIN will have created all the necessary elements to automatically check the satisfaction of the assertions when updating TPC, and will have persistently stored them in the database. Thus, TINTIN can be disconnected from SQL Server, and users might operate with the database normally with the unique consideration of invoking `safeCommit` at the end of each transaction.

To make the demonstration, we will apply some updates mixing both: updates that violate some assertion and updates that do not violate any of them. After each update, we will call the `safeCommit` procedure to see its effects, that is, we will see that it rejects the update if some violation occurs, or that it commits them if they satisfy the assertions.

With this demonstration, we will show that TINTIN enjoys the following features: 1. *Portability*: it can be easily installed in any SQL Server database —no need for special plugins nor additional technologies—. 2. *Clean installation*: all the necessary logics of the method is installed in another database—without modifying the original one—, except the `safeCommit` procedure and the triggers to capture the events, which are necessarily placed in the target database. 3. *Easy of use*: users can update the database without modifying their SQL statements/procedures. The unique mandatory requirement is to call the automatically generated `safeCommit` procedure at the end of each transaction. 4. *Efficiency*: the incremental nature of our method provides better execution times than executing non-incremental queries to perform the integrity checks.

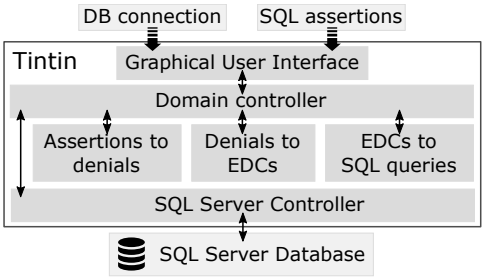

Figure 2: Tintin architecture

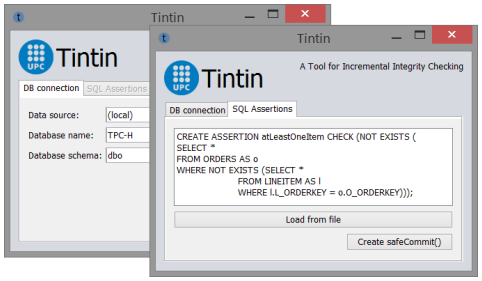

Figure 3: Graphical User Interface Design

4. PROTOTYPE

The architecture of TINTIN is shown in Figure 2, while its GUI is depicted in Figure 3. Basically, the GUI asks the user for a database (DB) connection, and the assertions that s/he wants to check in that database.

When the user introduces the DB connection, the SQL Server Controller creates a new auxiliary database event_DB to store the different events applied to it; that is, for each table T in DB, the SQL Server Controller builds two new tables (ins_T and del_T) to store the different tuples being inserted and deleted in T. In order to capture these tuples, the SQL Server Controller creates two different `INSTEAD OF` triggers, which capture the tuple insertions/deletions and place them in the corresponding ins_T or del_T table.

Afterwards, when the user introduces the SQL assertions, they are firstly mapped into logic denials. Then, these denials are translated into EDCs and, finally, EDCs are rewritten as SQL queries. Each of these steps is implemented in a different module following the previously presented method.

The resulting SQL queries are stored as views in event_DB. Then, the SQL Server Controller builds the `safeCommit` procedure. This procedure, when called, performs the following: 1. queries the previous views. 2. if all queries are empty, it disables the triggers, applies the update (insert in the DB the tuples contained in the *ins* tables, and remove from the DB the tuples contained in the *del* tables), and enables again the triggers. 3. truncates the *ins*/*del* tables.

The prototype has been developed in Java, with the exception of the *Assertions to denials* translator component, for which we have reused a previously existing C# software.

We made some experiments to evaluate the efficiency of our tool. We have checked some assertions of different complexity with TINTIN (like the one of our running example), in data sets consisting of 1GB to 5GB of data and with 1MB to 5MB of updates, and compared its efficiency with that of a non incremental method consisting of directly querying the assertions to the database. The time TINTIN required for checking the assertions ranges from 0.01 to 1.29 seconds and it is always better than in the non incremental approach, with a benefit of orders of magnitude when considering 5MB of updates (up to x2662 times faster).

5. CONCLUSIONS

TINTIN is a tool for checking assertions in SQL Server databases. The tool takes as input a set of assertions and it automatically builds a procedure called `safeCommit` which efficiently checks whether an update violates any of the assertions and, afterwards, commits the update if no violation is found or shows the tuples causing the violation otherwise.

TINTIN works almost transparently for the database users since it only requires the users to invoke `safeCommit` at the end of each transaction. Internally, the tool builds several triggers to capture the update requested by the user and to place it in some SQL auxiliary tables. These auxiliary tables are queried with the current database tables to check whether applying the update in the current data may cause any assertion violation. This join between the update and current data is the key for efficiency.

The fundamentals of TINTIN are the Event Dependency Constraints (EDCs), which we previously used to handle integrity checking of OCL constraints in conceptual models. More details about these rules, and also on their translation to SQL queries, can be found in [4, 5].

As further work, we plan to extend TINTIN to handle aggregate functions in assertions. We also expect to make it available for other DBMSs apart from SQL Server and to exploit other DBMS capabilities such as temporary tables.

Acknowledgements This work has been partially supported by the Ministerio de Economía y Competitividad, under project TIN2014-52938-C2-2-R and by the Secretaria d'Universitats i Recerca de la Generalitat de Catalunya under 2014 SGR 1534 and a FI grant.

6. REFERENCES

[1] ANSI Standard. *The SQL 92 Standard*, 1992.

[2] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Pearson, 2014.

[3] A. Olivé. Integrity constraints checking in deductive databases. In *Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB)*, pages 513–523, 1991.

[4] X. Oriol and E. Teniente. Incremental checking of OCL constraints through SQL queries. In *Proc. of the 14th Int. Workshop on OCL and Textual Modelling*, pages 23–32, 2014.

[5] X. Oriol and E. Teniente. Incremental checking of OCL constraints with aggregates through SQL. In *Conceptual Modeling*, volume 9381 of *LNCS*, pages 199–213. Springer, 2015.

[6] E. Teniente, C. Farré, T. Urpí, C. Beltrán, and D. Gañán. SVT: schema validation tool for microsoft sql-server. In *Proc. of the 30th International Conference on Very Large Data Bases*, pages 1349–1352, 2004.

[7] Transaction Processing Performance Council. *TPC-H benchmark specification 2.17.1*, 2014. `http://www.tpc.org`.

[8] V. Tropashko and D. Burleson. *SQL Design Patterns: Expert Guide to SQL Programming*. Rampant Techpress, 2007.