UQJG: Identifying Transactions that Collaborate to Violate an SQL Assertion

Toon Koppelaars
Oracle America, Inc.
toon.koppelaars@oracle.com

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

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

Sergi Curto
Universitat Politècnica de Catalunya
sergi.curto@estudiantat.upc.edu

Eduard Pujol
Universitat Politècnica de Catalunya
eduard.pujol@upc.edu

ABSTRACT

An SQL assertion is a declarative statement about data that must always be satisfied in any database state. Assertions were introduced in the SQL92 standard but no commercial DBMS has implemented them so far. Some approaches have been proposed to incrementally determine whether a transaction violates an SQL assertion, but they assume that transactions are applied in isolation, hence not considering the problem of concurrent transaction executions that collaborate to violate an assertion. This is the main stopper for its commercial implementation. To handle this problem, we have developed a technique for efficiently serializing concurrent transactions that might interact to violate an SQL assertion.

CCS CONCEPTS

• **Information systems** → **Integrity checking**; **Data locking**.

KEYWORDS

SQL assertions; integrity checking; transactions serialization

1 INTRODUCTION

Autonomous databases, recently launched by Oracle as the next generation of DBMSs [1], use machine learning and semantic reasoning to automate database tuning, security, updates, and other management tasks traditionally performed by database administrators or developers. Unlike a conventional database, an autonomous database performs all these tasks automatically without human intervention.

A key element of an autonomous database is *transaction processing*. One of the main open issues in transaction processing is that of allowing an autonomous database with the ability of automatically checking SQL assertions. Assertions were defined in the SQL92 standard [12] but still no relational DBMS incorporates them.

An assertion is a named general constraint. That is, a boolean query with embedded queries, probably among several tables, whose evaluation must be true for any possible database state. Currently, the code to check these conditions has to be implemented manually in a host language, since no DBMS supports assertions. Having to manually program these checks is clearly a very difficult and error prone task, and this is the reason of aiming to include them in the Oracle Autonomous Database.

Checking SQL assertions requires solving two different problems: efficiently determining whether a given transaction violates an assertion; and identifying the transactions that would not violate the assertion when applied in isolation but that could do it when executed together with other transactions.

Figure 1: *WorksIn* schema

Some incremental checking techniques for SQL assertions have been proposed (see for instance [9, 11]), or might be adapted from similar areas such as incremental pattern matching [4]. However, none of them is yet able to identify whether two transactions can be executed concurrently when they do not collaborate to violate an SQL assertion. Therefore, they force all transactions to be serialized to correctly detect possible SQL assertion violations. This is a clear impediment for any commercial implementation of SQL assertions.

To illustrate the concurrency problem, consider the simple schema in Figure 1 and an SQL assertion forcing all employees to live in the city where their department is located:

```
CREATE ASSERTION 'Same city as Dept' CHECK (
    NOT EXISTS(
        SELECT 'A foreign employee'
        FROM Employee E, Department D
        WHERE E.dep=D.dep_id AND E.city<>D.city))
```

The database contents from Tables 1 and 2 satisfies this assertion.

Table 1: Department

Dep_ID	Dep_Name	City	Budget
D1	Sales	Madrid	150,000
D2	Data Analysis	BCN	200,000

Table 2: Employee

Emp_ID	Emp_Name	Salary	City	Dep
E1	Josh	20,000	BCN	D2

Now, consider two transactions: t_1 inserts a new employee $E2$ from *Madrid* working in $D1$, and t_2 relocates $D1$ to *BCN*. In isolation, they do not violate the assertion. However, if they were to be applied simultaneously the resulting database would violate the *'Same city as Dept'* assertion since, in this case, there would be an employee $E2$ living in one city (*Madrid*) but working in another one

The final publication is available at ACM via http://dx.doi.org/10.1145/3459637.3482210

(*BCN*). Therefore, we need t_1 to block t_2 (or vice versa). However, traditional lock strategies based on locking the affected tables or the affected rows are not able to identify this situation since both transactions affect different tables and rows.

From the above example, it may look like, if any transaction modifies the Employee table, we should always lock the Department table. However, this policy is too restrictive. Assume now another two transactions: t_3 inserting in the original database a new employee from *BCN* in department $D2$, and t_4 updating department $D1$ to relocate it to *BCN*. In this case, both transactions can be applied concurrently since they do not *collaborate* to violate the assertion. This is true, regardless of the contents of the database, because each transaction affects a different department.

In this paper, we propose a method we have developed and implemented to identify those transactions that can be safely executed concurrently since they do not collaborate to violate an SQL assertion. Intuitively, two transactions can be executed concurrently if the effects of the first do not compromise the integrity revalidation of the second. This is a challenging problem because no transaction can be aware of the effects being executed by another concurrently executed uncommitted transaction.

Our technique builds, for each SQL assertion, its Universal Quantification Join Graph (UQJG). Intuitively, the UQJG is a graph stating, for given a transaction, the attribute values it should lock in order to block the execution of another transaction that, in case it was executed concurrently, might compromise the integrity revalidation of the assertion. The UQJG is directly built from the SQL assertion definition, and the lock strategy depends only on the insertions and deletions [1] of the transactions together with the structure of the UQJG. Therefore, our technique does not need to check the database contents nor to apply any other expensive computation. To our knowledge, this is the first approach on general transactions serialization for SQL assertion revalidation.

The underlying goal of this work is to provide a real implementation of SQL assertions in Oracle Autonomous Database. However, the proposed techniques are generic and applicable to all databases.

2 PRELIMINARIES

We say that a transaction is an *assertion triggering transaction* if it can cause a violation of an assertion (either alone, or in collaboration with another one). For instance, $t1$ in Example 1 is an assertion triggering transaction for the assertion *'Same city as Dept'*.

Our lock strategy applies only to assertion triggering transactions since the other transactions can be executed concurrently without further analysis. Identifying assertion triggering transactions is an already well-studied problem which we leave out of the scope of this paper. In this sense, we encourage the interested reader to take a look to the notions of table insert/delete polarity [2] to get an idea of how it can be done.

Our locking strategy is aimed at identifying which pair of assertion triggering transactions can collaborate to violate an assertion. We say that a pair of transactions collaborate when they do not violate the assertion in isolation, i.e. without taking into a account the database modifications of the other transaction; but they violate the assertion when both transactions are commited.

Previous transactions t_1 and t_2 collaborate to violate the assertion. Given the initial database, neither t_1 nor t_2 violates the assertion by itself but, when taking into account the changes they both make, we get into a database state where the assertion is violated.

[1] Modifications are considered to be a deletion of a tuple together with an insertion of the same tuple with different values.

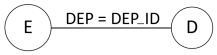

Figure 2: UQJG for 'Same city as Dept'

3 THE UQJG

The Universal Quantification Join Graph (UQJG) is the data structure we implement to support our serialization technique. Loosely speaking, given an SQL assertion, its UQJG specifies all possible serialization scenarios for two transactions that concurrently update two tables involved in the assertion.

3.1 UQJG definition

UQJG nodes represent table references and edges represent concurrency problems that may occur when two assertion triggering transactions update at the same time the contents of the referred tables. Since an assertion can have multiple references to the same table, a UQJG can have multiple nodes representing the same table. Moreover, a UQJG can have self-edges. Self-edges specify situations where modifying the contents of the same table through two concurrent transactions can lead to the violation of the SQL assertion.

Figure 2 provides the UQJG of the assertion *'Same city as Dept'* from our motivating example. Since the assertion only refers to two tables, *Employee* and *Department*, we have only two nodes in the UQJG. Furthermore, since concurrently updating both tables might cause a constraint violation, as previously discussed, there is an edge between the two. However, note that there is no self-edge. This means that two transactions can update the very same table, concurrently, without causing a constraint violation.

However, not all possible concurrent updates over *Employee* and *Department* will necessarily cause a problem. In fact, all problematic concurrent updates meet one condition: their department value is the same. Therefore, in a UQJG edges can be decorated with tags. A tag describes a condition that is always satisfied by the concurrent updates that collaborate to violate the assertion over such edge.

For instance, in Figure 2, the tag *dep = dep_id* states that any two assertion triggering transactions, when updating *Employee* and *Department*, satisfy that the value *dep* used to update *Employee* is equal to the value *dep_id* used in *Department*.

Tags can be composed of one equality, or a conjunction of them. In any case, tags are a key point for maximizing transaction concurrency since they allow reducing the potentially collaborating cases that require serialization.

It is worth noting that the tags only speak of values that can be obtained directly from the assertion triggering transactions. In fact, our serialization policy cannot access database values. Indeed, if the serialization policy accessed the database values, the serialization policy itself might require serializing transactions (e.g. to

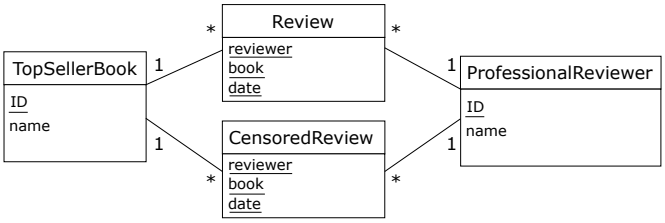

Figure 3: *BookReviews* schema

prevent not reading values that are concurrently updated by other transactions). This approach would simply not scale.

3.2 UQJG construction and rationale

Generating a basic UQJG is almost direct for any SQL assertion. I.e., for any assertion, we can create a non-optimized UQJG by adding a node for any table reference from the assertion, and place an edge, with an empty tag, between every pair of nodes (including a self-edge for every node). This is a conservative sound UQJG that would block any two transactions affecting two tables of the same assertion. Hence, the true question relies on optimizing such basic UQJG. That is, prune (self) edges, and/or add tags to edges.

In our proposal, we are able to add tags for those parts of the SQL assertion corresponding to relational algebra (i.e., first-order logics [3]) and also for some SQL aggregates (namely, Min(), Max(), Sum(), and Count(*)). This includes basic arithmetic/logic operators as well as exists clauses with nested sub-queries. For the sake of safeness, we currently do not admit nullable attributes in the tags.

Tags depend on the universal variables of the assertion. Semantically speaking, an SQL assertion can be seen as a logic constraint where some of its variables are universally quantified. For instance, in the *'Same city as Dept'* assertion, its department variable is universally quantified. This means that this assertion applies to any possible value of *department*. That is, all employees working in the *D1* department should have a city equal to the city where the *D1* department is located, for all the possible values of the *dep_id*.

Hence, given an SQL assertion, each possible value for any of its universal variables is like posing a different assertion. Clearly, two transactions triggering the violation of two different assertions do not collaborate to cause a violation of one of them. Therefore, conversely, if two transactions collaborate to cause a constraint violation, then it is sure that they agree with the value of the universal variable. Naturally, the same explanation is generalized to sets of universal variables, in case the assertion has many.

Consider now the *reviews* schema from Figure 3. Professional reviewers review top selling books, where some of these reviews may have been censored. Assume that, in this domain, all reviewers must have written at least one non-censored review for every top selling book. This constraint can be specified as follows:

```
CREATE ASSERTION 'TopSellingBooksReviews'
CHECK (NOT EXISTS (
  SELECT 'A non reviewed book'
  FROM ProfessionalReviewer PR, TopSellerBook B
  WHERE NOT EXISTS (
    SELECT 'A review for the book'
```

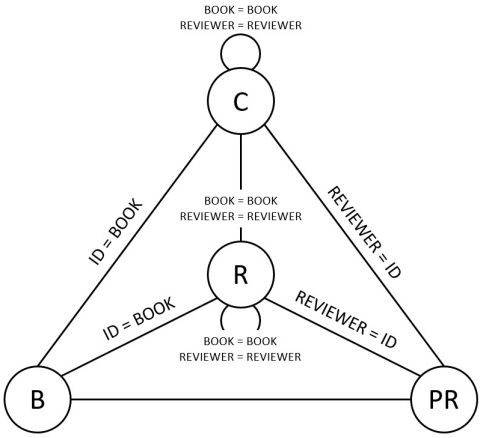

Figure 4: 'Top Selling Books Reviews' UQJG

```
FROM Review R
WHERE R.reviewer = PR.id AND R.book = B.id AND
  NOT EXISTS (
    SELECT 'Censorship for the review'
    FROM Censored C
    WHERE C.reviewer = R.reviewer AND
      C.book = R.book AND C.date = R.date))))
```

In this case, the universal variables are the pairs *Professional-Reviewers ids* and *Book ids*. Indeed, semantically, the UQJG establishes an assertion for each possible value of both SQL columns (e.g. "Mary" and "LOTR", "John" and "Harry Potter", etc).

The UQJG we generate for this assertion is shown in Figure 4. It states, for instance, that two assertion triggering transactions modifying the tables *ProfessionalReviewer (PR)* and *TopSellerBook (B)* can always collaborate to violate the assertion (non-tagged edge between nodes *B* and *PR*). Moreover, the table reference *C* (*R*) contain a self-edge with a tag. This means that two assertion triggering transactions concurrently modifying the contents of *C* (*R*) can collaborate if they agree on the attribute values specified in the tag, that is, the universal variables. Similar conflicts exists for the other table references connected with an edge.

To show the feasibility of our approach we have implemented a prototype tool that performs this UQJG generation and we have tested it for tens of SQL assertions including (combinations of) expressions such as: joins, cross joins, left/right outer joins, (not) exists, (not) in, group by (having), min/max/count(*), union-all/minus, and/or, is (not) null, =, <, <=, <>, >=, >.

4 LOCKING

For efficiency reasons, we develop our own fine grained lock system. Indeed, table locks, as already discussed, might serialize too much and would not permit exploiting the tags; other techniques, such as index-locks, might not be available unless we build an index for the columns appearing in the tags.

We apply locks on what we call Assertion-Lock Memory-Objects (ALMOs). An ALMO does not represent a particular row, but a value (or set of values) that a transactions locks, for some table and

column(s), to avoid other transactions using it. E.g., the ALMOs of *'Same city as Dept'* are the department values. The UQJG tags specify the ALMOs each transaction must acquire according to its updates. For instance, consider the *'Top Selling Books Reviews'* assertion and assume the following assertion triggering transaction:

```
t: DELETE FROM Review
   WHERE book="LOTR" AND reviewer="Mary"
```

From the UQJG in Figure 4, we can determine that, until this transaction has not been committed, we shall not allow any other assertion triggering transaction that:

- Modifies, inserts or deletes rows of *Censored* with values Book = "LOTR" and Reviewer = "Mary" (edge between *R* and *C*). Hence, we lock the ALMO *"LOTR"+"Mary"* over *Censored*.
- Modifies, inserts or deletes rows of *Review* with values Book = "LOTR" and Reviewer = "Mary" (self-edge on *R*). Hence, we lock the ALMO *"LOTR"+"Mary"* over *Review*.
- Modifies, inserts or deletes rows of *ProfessionalReviewer* with values ID = "Mary" (edge between *R* and *PR*). Hence, we lock the ALMO *"Mary"* over *ProfessionalReviewer*.
- Modifies, inserts or deletes rows of *TopSellerBook* with values ID = "LOTR" (edge between *R* and *B*). Hence, we lock the ALMO *"LOTR"* over *TopSellerBook*.

Such ALMO locking can be easily implemented through hash values provided that their length is large enough to ensure that no accidental hash collision occurs. Moreover, they are only meant to control the serialization of transactions with regards to checking the assertions, and not for the traditional read/write problems that transactions may experience. That is, ALMO locking is a complement, but not substitute, to the current locking policies.

Moreover, it is possible to optimize the UQJG taking into account the already underlying locks of the databases. In particular, it is possible to remove some self-edges by taking into account primary keys. For instance, assume that the primary key of review is (*Book, Reviewer*). In this case, we can remove the self-edge since, in essence, the locks generated by this self-edge coincide with the already taken locks of the database that ensure primary key consistency.

Our approach is also sound when generalized to sets of transactions. That is, we correctly block a transaction Tx_1 if it might cause a violation in collaboration with a set of transactions $Tx_2, ..., Tx_n$ (rather than just with a single one Tx_2). This is so because our technique detects those pairs of atomic updates u_1, u_2 (from different transactions) that may cause a violation if applied concurrently with other unknown atomic updates U, where these updates U can come from any transaction (Tx_1, Tx_2, or even another set of transactions $Tx_i, ...Tx_l$).

5 RELATED WORK

While integrity checking has been a major research trend over several years, only a few proposals have focused on how to incrementally revalidate assertions with concurrently executing transactions. To our knowledge, the only approach which addresses our same problem is the one proposed in [10]. However, this approach assumed: 1) that the transactions were already predefined at compile time under the form of preprogrammed operations, which makes the approach clearly not applicable in scenarios with transactions defined at runtime; and 2) did not exploit the attribute values to achieve a more accurate serialization. Some notions of transaction management are provided in [7], but this proposal does not show how to apply them in practice with SQL assertions.

With regards to the particular granularity of the locks, we observe that the ALMO locks we propose are to some extent similar to the predicate-locks [8] and, more specifically, to key-value locks [5]. We show the main differences in the following.

Predicate-locks were originally proposed to deal with the *phantoms* concurrency problem [8]. They were based on stating a range of rows that a transaction is locking, where this range is specified by means of some predicates (e.g., the lock applies to employee rows with a *salary* greater than 1000 and whose *dept* is 'Sales'). The tags decorating the edges of a UQJG can be seen as predicate-locks containing only conjunctions of equalities. In fact, this is not a limitation but a feature. Indeed, detecting when two general predicate-locks overlap, and hence, raising a concurrency problem, is NP-hard [6]. However, our ALMOs proposal can be implemented using hash tables, which ensures good performance in practice.

This kind of limited predicate-locks are already known in index-locks and, in particular, key-value/range locks [5]. An index-lock is a lock directly applied over an index, like a B-tree, instead of the underlying rows. Taking advantage of the inherent order of the index, it is possible to lock a given value or even a gap between values (i.e., non existing tuples). Although our proposal could easily fit with this kind of locks, it does not require creating indexes. E.g., in the *'Top Selling Books Reviews'* assertion, we need to create locks with the values for *books* and for *reviewers*. Hence, to apply index-locks we would require two indexes on this assertion. Taking into account that there can be other assertions creating more tags with different columns, it is clear that the use of index-locks may encompass the creation and maintenance of too many indexes.

6 CONCLUSIONS

We have proposed a method to identify transactions that can be safely executed concurrently since they will not collaborate to violate an assertion. Our technique is based on the Universal Quantification Join Graph (UQJG), which is automatically obtained from the SQL assertion. A UQJG provides all the information required to identify possible conflicts at run time. We have shown how to use the UQJG to serialize transactions through Assertion-Lock Memory-Objects (ALMOs). Locking ALMOs is a better policy than locking rows or tables, and it is more practical to implement than predicate-locks or index-locks. Hence, the overall result allows us to maintain a low overhead for ensuring a correct database assertion revalidation. As further work, we plan to generate tags with nullable attributes and optimize more aggregate functions.

REFERENCES

[1] www.oracle.com/autonomous-database. Last access 20-02-2021.
[2] S. Ceri and J. Widom. Deriving incremental production rules for deductive data. *Inf. Syst.*, 19(6):467–490, 1994.
[3] E. F. Codd et al. *Relational completeness of data base sublanguages.* Citeseer, 1972.
[4] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Readings in Artificial Intelligence and Databases*, pages 547–559. Elsevier, 1989.
[5] G. Graefe. A survey of b-tree locking techniques. *ACM Transactions on Database Systems (TODS)*, 35(3):1–26, 2010.

[6] H. B. Hunt and D. J. Rosenkrantz. The complexity of testing predicate locks. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 127–133, 1979.

[7] D. Martinenghi and H. Christiansen. Transaction management with integrity checking. In *Database and Expert Systems Applications (DEXA) 2005*, volume 3588 of *LNCS*, pages 606–615. Springer, 2005.

[8] H. Morgan, K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System - Eswaran, Gray, Lorie, Traiger. 19(11), 1976.

[9] X. Oriol and E. Teniente. Incremental Checking of OCL Constraints with Aggregates Through SQL. In *Conceptual Modeling*, pages 199–213. Springer International Publishing, 2015.

[10] X. Oriol and E. Teniente. Adapting Integrity Checking Techniques for Concurrent Operation Executions. In *System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0*, pages 235–248. Springer International Publishing, 2019.

[11] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *1996 ACM SIGMOD international conference on Management of data*, pages 447–458, 1996.

[12] A. Standard. The sql 92 standard, 1992.