Incremental Checking of OCL Constraints with Aggregates through SQL

Xavier Oriol and Ernest Teniente

Department of Service and Information System Engineering
Universitat Politècnica de Catalunya – BarcelonaTech
{xoriol,teniente}@essi.upc.edu

Abstract. Valid states of data are those satisfying a set of constraints. Therefore, efficiently checking whether some constraint has been violated after a data update is an important problem in data management. We tackle this problem by incrementally checking OCL constraint violations by means of SQL queries. Given an OCL constraint, we obtain a set of SQL queries that returns the data that violates the constraint. In this way, we can check the validity of the data by checking the emptiness of these queries. The queries that we obtain are incremental since they are only executed when some relevant data update may violate the constraint, and they only examine the data related to the update.

Keywords: Constraints Checking, SQL, OCL, Aggregates

1 Introduction

A conceptual schema is the formal specification of an information system in terms of the structure of the data to be stored together with the operations applicable to modify such data. The specification of the data structure includes several integrity constraints, i.e., some conditions that any state of the data should satisfy in order to be valid.

One of the most used languages for specifying conceptual schemas is UML, a standard language maintained by the OMG [1]. In UML, the structure of the data is mainly defined by a taxonomy of classes/associations (i.e., a class diagram), complemented with several textual integrity constraints written in OCL [2]. This leads to the problem of efficiently checking whether the current data of a running information system truly satisfies the OCL constraints defined in its UML schema. This is an important problem in data management since any violation of an integrity constraint would indicate an invalid state of the data.

Consider, for instance, the UML schema in Figure 1. This schema specifies an information system storing data about *movies* and the *people* participating in them as *cast members*, where each cast member exercises a *role* (e.g. director, actor, actress, etc.).

We are clearly interested to ensure that data over such schema satisfies a set of integrity constraints. For instance, there should not be any cast member playing the role of *actor* and *actress* at the same time; the sum of any movie

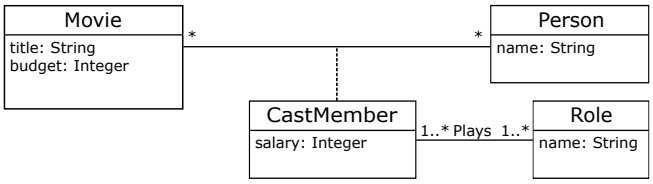

Fig. 1. Simplified UML class diagram for the IMDb Information System

budget should be higher than the salaries of its cast members; and there should be at least one cast member playing the role of *director* for each movie. These constraints are defined in OCL in Figure 2. Note that aggregate operations such as `sum` or `count` are used in the constraint definitions.

```
context CastMember inv NotActorAndActress:
self.role.name->excludes('actor') or self.role.name->excludes('actress')
context Movie inv BudgetIsHigher:
self.budget >= self.castMember.salary->sum()
context Movie inv HasSomeDirector:
self.castMember.role.name->count('director') > 0
```

Fig. 2. OCL Constraints for the simplified UML class diagram of IMDb

Several techniques have been proposed so far to efficiently checking the violation of OCL constraints. Some of them handle the problem by translating the constraints into SQL [3,4]. For example, the work in [3] builds some SQL views that return the data violating the constraints. Thus, a constraint is violated if its corresponding view is not empty. Other techniques follow a different approach aimed at incrementally checking the OCL constraints [5,6,7]. That is, assuming that no OCL constraint is violated for the current data, they determine which OCL constraints should be checked, and for which data, after some update is applied to the information system.

The method we present in this paper follows a combination of these approaches and so we take advantage of both of them. On the one hand, the idea is to translate OCL constraints into SQL queries, thus, benefiting from DBMS optimizations such as query planners, different join algorithms, indexes, caches, etc. but stating the query in terms of current data and data being inserted/deleted. In this way, the query we define is only executed when a data update matches it, hence, when the update potentially violates its corresponding constraint. On the other hand, the query only searches for violations in the current data that joins with the update. In this way, for example, we do not need to recheck all the constraints for all the movies when some update is applied in only a few of them.

As far as we know, there is only another proposal following a similar approach to ours [8]. However, the core of this work is based on the RETE algorithm, which encompasses materializing any relational algebra operation performed by

the queries. In this sense, the RETE algorithm has been criticized because of its combinatorially explosive materialized data growth and the execution time it might take to maintain such materialization [9]. On the contrary, our proposal only materializes the aggregated values accessed by the constraints (e.g. `sum`, `size`), avoiding in this way the inefficiencies caused by such intermediate materializations.

This paper extends our previous work in [10], where we outlined how to perform incremental checking of OCL constraints for a specific fragment of OCL, in the following directions:

- *Dealing with Aggregation Operations.* We extend our approach to be able to deal with distributive aggregation functions of OCL (i.e., `sum`, `count`, `size`).
- *Expressiveness Analysis.* We analyze the expressiveness of the OCL our method can deal with and we show that, because of dealing with aggregation, we can handle constraints beyond the OCL_{FO} fragment of OCL [11].
- *Experimentation.* We have made several experiments using real data extracted from the public interface of IMDb[1] information system. With these experiments we show the scalability of our approach in realistic conditions.

2 Basic Concepts and Notation

Terms, atoms and literals A *term* t is either a variable or a constant. An *atom* is formed by a n-ary *predicate* p together with n terms, i,e., $p(t_1, ..., t_n)$. We may write $p(\bar{t})$ for short. If all the terms $\bar{t}$ of an atom are constants, we call the atom to be *ground*. A literal l is either an atom $p(\bar{t})$, a negated atom $\neg p(\bar{t})$, or a built-in literal $t_i\ \omega\ t_j$, where ω is an arithmetic comparison (i,e., $<$,$\leq$,$=$,$\neq$).

Derived/base/aggregate predicates A predicate p is said to be *derived* if the boolean evaluation of an atom $p(\bar{t})$ depends on some derivation rules, otherwise, it is said to be *base*. A *derivation rule* has the form: $\forall \bar{t}.\ p(\overline{t_p}) \leftarrow \phi(\bar{t})$ where $\overline{t_p} \subseteq \bar{t}$. In the formula, $p(\overline{t_p})$ is an atom called the *head* of the rule and $\phi(\bar{t})$ is a conjunction of literals called the *body*. We restrict all derivation rules to be safe (i.e., any variable appearing in the head or in a negated or built-in literal of the body also appears in a positive literal of the body) and non-recursive. Given several derivation rules with predicate p in its head, $p(\bar{t})$ is evaluated to true if and only if one of the bodies of such derivation rules is evaluated to true.

An aggregate predicate p_a (aka aggregate query/rule [12,13,14]) is a predicate defined over some predicate p that aggregates one of the terms of p with some aggregation function f. An aggregate predicate is defined by means of a rule: $\forall \bar{t}.\ p_a(\overline{t_p}, f(x)) \leftarrow p(\bar{t})$ where $\overline{t_p} \subseteq \bar{t}$ and $x \in \bar{t}$. An atom $p_a(\overline{t_p}, x_f)$ evaluates to true if and only if x_f equals to aggregating all values x in $p(\bar{t})$ by means of f. E.g. given the aggregate predicate $sumSalaries(e, x)$ defined by $sumSalaries(e, sum(s)) \leftarrow salary(e, s)$, $sumSalaries(e, x)$ evaluates to true if and only if x is equal to the sum of all salaries s such that $salary(e, s)$.

[1] www.imdb.com

We also extend the notion of base/derived/aggregate predicate to atoms and literals. I.e., when the predicate of some atom/literal is base/derived/aggregate, we say that such atom/literal is base/derived/aggregate respectively.

Logic formalization of the UML Schema. As proposed in [15] we formalize each class C in the class diagram with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $c(Oid)$ together with n atoms of the form $cA_i(Oid, A_i)$, each association R between classes $\{C_1, \ldots, C_k\}$ by means of a base atom $r(C_1, \ldots, C_k)$, and, similarly, each association class R between classes $\{C_1, \ldots, C_k\}$ with attributes $\{A_1, \ldots, A_n\}$ by means of a base atom $r(Oid, C_1, \ldots, C_k)$ together with n atoms $rA_i(Oid, A_i)$.

Structural events. A *structural event* is an elementary change in the population of a class or association of the schema [16]. That is, a change in the contents of the data. We consider six kinds of structural events: class instance insertion/deletion, association instance insertion/deletion and attribute instance insertion/deletion. Attribute updates are simulated by means of a simultaneous deletion and insertion of the old and new value respectively.

We denote insertions by ι and deletions by δ. Given a base atom $p(\overline{x})$, insertion structural events are formally defined by $\forall \overline{x}.\ \iota p(\overline{x}) \leftrightarrow p^n(\overline{x}) \wedge \neg p(\overline{x})$, while deletion structural events by $\forall \overline{x}.\ \delta p(\overline{x}) \leftrightarrow p(\overline{x}) \wedge \neg p^n(\overline{x})$, where p^n stands for predicate p evaluated in the new data state, i.e., the one obtained after applying the change.

Aggregate events. An *aggregate event* is a change in the value of some aggregate predicate. Aggregate events are determined by the structural events applied in a data state. Similarly as before, we denote increases of aggregate events by means of ι and decreases by means of δ.

Dependencies. A *tuple-generating dependency* (TGD) is a formula of the form $\forall \overline{x}, \overline{z}.\ \varphi(\overline{x}, \overline{z}) \rightarrow \exists \overline{y}.\ \psi(\overline{x}, \overline{y})$. A *denial constraint* is a special type of TGD of the form $\forall \overline{x}.\ \varphi(\overline{x}) \rightarrow \bot$, in which the conclusion only contains the $\bot$ atom, which cannot be made true, and the premise may contain positive, negated and built-in literals. An *event dependency constraint* is a special type of denial constraint containing at least one positive event atom.

3 Our Approach

We follow a two-steps approach to translate, in compilation time, a set of OCL constraints to SQL queries. In the first step, each OCL constraint is translated into a set of event dependency constraints (EDCs). An EDC is a logic rule identifying a particular situation where some structural events applied to a certain state of the data will cause the violation of the OCL constraint.

In the second step, each EDC is translated into a different SQL query. We assume a mapping from the EDC predicates to SQL tables for this purpose. Roughly speaking, base predicates representing UML classes/associations are mapped to SQL tables containing their instances, structural event predicates

are mapped to auxiliary SQL tables containing the structural events being applied, and aggregate predicates are mapped to auxiliary SQL tables containing a materialization of the relevant aggregated values. Given this mapping, the translation from an EDC to SQL results into a query joining the three: current data, current aggregated values, and structural events.

At runtime, our method works as follows: (1) an actor places the structural events he/she wants to apply in the auxiliary structural event tables; (2) our method executes the queries for checking the violation of OCL constraints; (3) if the queries return the empty set, there is no violation and the method commits the structural events in the tables representing the data, and it incrementally updates the materialized aggregated values. On the contrary, if the query returns some data, there is some constraint violation and thus, the update is rejected.

The key for incrementality is the join in the SQL queries between structural events, current data and current aggregated values. First of all, any SQL query joining a structural event which is not applied (i.e., whose SQL table is empty) is immediately discarded. Therefore, we only check those constraints that can be violated according to the ongoing structural events. Second, the data considered by an SQL query during its execution is necessarily the data joining the structural events applied, thus, avoiding to look through all the database. Third, the materialized aggregated values avoid the need to recompute from scratch the necessary aggregations required to check a constraint, which might be expensive. Finally, our method guarantees that the materialized aggregated values can be updated incrementally, thus, with negligible time penalty.

In our previous work [10] we tackled efficient integrity checking for $\text{OCL}_{\mathsf{UNIV}}$ constraints [2], a specific fragment of OCL. We extend here our approach to handle OCL distributive aggregates, which extends the expressiveness of the OCL we can deal with beyond OCL_{FO} [11]. For the sake of self-containment of the paper, we start by reviewing the translation from $\text{OCL}_{\mathsf{UNIV}}$ constraints to SQL queries. Then, we extend our approach dealing with aggregates and we show how this extension allows us to deal with any OCL_{FO} constraint and beyond.

3.1 $\text{OCL}_{\mathsf{UNIV}}$ Translation into SQL

First, we encode each $\text{OCL}_{\mathsf{UNIV}}$ constraint as a logic denial according to [15]. Logic denials are written over the logic formalization of the UML schema which has been defined in Section 2. The logic formalization corresponding to our UML schema in Figure 1 is:

$$\begin{aligned}&\mathit{movie(m)}, \mathit{mTitle(m,t)}, \mathit{mBudget(m,b)}, \mathit{person(p)}, \mathit{pName(p,n)},\\ &\mathit{castMember(c,m,p)}, \mathit{cmSalary(c,s)}, \mathit{plays(c,r)}, \mathit{role(r)}, \mathit{rName(r,n)}\end{aligned}$$

We assume, without loss of generality, that instances of *Role* and *Person* can be identified by its *name*. Thus, we will use the name attribute as their OIDs. In this way, we can omit the *rName* and *pName* predicates.

[2] The name is due to the observation that any OCL expression in $\text{OCL}_{\mathsf{UNIV}}$ can be written as a logic formula where all variables are universally quantified.

Given the previous logic formalization, the translation of the OCL_{UNIV} constraint *NotActorAndActress* is encoded into the following denial constraint:

$$plays(c, Actor) \land plays(c, Actress) \rightarrow \bot$$

In the rest of this section, we explain how to obtain the EDCs from denial constraints and how to transform them into SQL queries.

Obtaining EDCs for OCL_{UNIV} Constraints An event dependency constraint (EDC) identifies a particular situation in which the original OCL_{UNIV} constraint would be violated in a data state D^n resulting from applying some set of structural events to some initial data state D. Therefore, each denial constraint obtained in the previous step will be translated into several EDCs, each one corresponding to a different way in which the constraint may be violated.

The main idea for obtaining the EDCs is to replace each literal in the denial constraint by the expression that evaluates it in the new state D^n. Positive and negative literals in the denial are handled in a different way according to the following formulas:

$$\forall \overline{x}.\ p^n(\overline{x}) \leftrightarrow (\iota p(\overline{x})) \lor (\neg \delta p(\overline{x}) \land p(\overline{x})) \quad (1)$$

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

Rule 1 states that an atom $p(\overline{x})$ will be true in the new state D^n if its insertion structural event has been applied or if it was already true in the initial state D and its deletion structural event has not been applied. In an analogous way, rule 2 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, we get a set of EDCs that states all possible ways to violate a constraint by means of the possible structural events of the schema. From the previous denial constraint we get:

$$plays(c, Actor) \land \neg \delta plays(c, Actor) \land \iota plays(c, Actress) \rightarrow \bot \quad (3)$$

$$\iota plays(c, Actor) \land plays(c, Actress) \land \neg \delta plays(c, Actress) \rightarrow \bot \quad (4)$$

$$\iota plays(c, Actor) \land \iota plays(c, Actress) \rightarrow \bot \quad (5)$$

EDCs are grounded on the idea of *event rules* which were defined in [17] to perform integrity checking in deductive databases. In general, we will get $2^k - 1$ EDCs for each denial constraint dc, where k is the number of literals in dc.

Translating EDCs into SQL Now, we translate EDCs into SQL. For this purpose, we require a mapping from logic predicates to SQL tables. Each literal from the EDC is translated into a table reference placed in the `FROM` clause of the query. We define an SQL `JOIN` for a table reference when the literal is positive and it has some variable in common with another previously translated literal. In contrast, we define an SQL *antijoin* (by means of a `LEFT JOIN` together a `IS NULL` condition) for negative literals. Built-in literals and constant bindings

are translated in the `WHERE` clause. Following our previous example, we would translate EDC 3 as:

```
SELECT P1.cast
FROM Plays AS P1
   LEFT JOIN del_Plays AS dP1 ON (P1.cast = dP1.cast and P1.role = dP1.role)
   JOIN ins_Plays AS iP2 ON (iP2.cast = P1.cast)
WHERE P1.role = 'actor' and dP1.cast IS NULL and iP2.role = 'actress'
```

Intuitively, the previous SQL query looks for those cast members in the table *Plays* such that: (1) their role is *actor*, (2) their role *actor* is not being deleted by the structural events, (3) some structural events are inserting the new role *actress* to them. When executing this query, the query planner specifies its start from the *ins_Plays* table, rather than *Plays*, since the cardinality of *ins_Plays* is expected to be much lower. In this way, the DBMS does not look through all current data (i.e., all data in *Plays*), but only to the data that joins the update. Moreover, if *ins_Plays* has no tuples, and thus the original OCL constraint cannot be violated, the query returns the empty set without accessing data because any join with no tuples trivially returns the empty set. In this way, our queries behave incrementally since they only look to the data related to the update, and only when the update may cause a violation.

3.2 OCL Aggregation Translation into SQL

We extend now the previous approach to deal with aggregates. There exist several kinds of aggregates according to the complexity to incrementally update them when some structural event is applied [18]. In this work, we focus on *distributive* aggregation. Intuitively, distributive aggregates are those that can be updated by taking into account the current aggregated value of the data, the aggregated value of the data inserted, and the aggregated value of the data deleted. The distributive aggregates of OCL are: `sum`, `size` and `count`.

As we did before, we first translate any OCL constraint into a logic denial constraint; then, we translate this denial constraint into several EDCs and, finally, we translate each EDCs into an SQL query.

OCL Aggregation Translation into Logic Denials Any OCL aggregation expression is defined by means of a *source* (i.e., a navigation) and an *aggregation operation* (e.g. `sum`), where the resulting aggregate value is normally used in some arithmetic comparison. We translate the *source* of the expression following the same lines as [15], and from there, we use an *aggregate predicate* to aggregate the required value. Once we obtain the aggregated required value, we can define the built-in literal encoding the arithmetic comparison.

For instance, given the *BudgetIsHigher* constraint, the source of the OCL aggregation expression is *self*.*castMember*.*salary*. This expression is translated as the following conjunction of literals:

$$castMember(c, m, p) \wedge cmSalary(c, s)$$

From there, we can aggregate the salaries (i.e., the s term) by means of defining an aggregate predicate:

$$sumSalaries(m, sum(s)) \leftarrow castMember(c, m, p) \wedge cmSalary(c, s)$$

In this way, the atom *sumSalaries(m, x)*, indicates that the sum of salaries of the movie m is x. Thus, we can use x to check whether the sum of the movie salaries is greater than its budget:

$$mBudget(m, b) \wedge sumSalaries(m, x) \wedge b < x \rightarrow \bot$$

Obtaining EDCs for Denial Constraints with Aggregates The most important issue for obtaining the EDCs in the presence of aggregate predicates relies on how to compute the aggregate value in the new data state D^n. We make use of two aggregate event predicates for this purpose: one for computing the aggregated value x_ι for the data being inserted, and another one for computing the aggregated value x_δ for the data being deleted. Since we focus on OCL distributive aggregation functions, we can guarantee that the aggregated value in the new state D^n equals to the current aggregated value x plus x_ι minus x_δ.

For instance, the previous denial constraint would be translated as:

$$\iota mBudget(m, b) \wedge sumSal(b, x) \wedge \iota sumSal(m, x_\iota) \wedge \delta sumSal(m, x_\delta) \wedge b < x + x_\iota - x_\delta \rightarrow \bot \quad (6)$$

$$mBudget(m, b) \wedge \neg\delta mBudget(m, b) \wedge sumSal(m, x) \wedge \iota sumSal(m, x_\iota) \wedge \delta sumSal(m, x_\delta) \wedge x_\iota \neq x_\delta \wedge b < x + x_\iota - x_\delta \rightarrow \bot \quad (7)$$

The first rule captures those violations occurring when newly inserted movie budgets with possibly some cast member salary updates are lower than the aggregated salaries in the new data sate. The second captures those violations that occur when updating the cast member salaries of those movies for which no budget update is applied.

These EDCs result from, first, replacing any base literal with the rules defined in 1 and 2 as before to obtain their evaluation in D^n. Then, we add, for any aggregate literal, two additional aggregate event literals computing the aggregated value x_ι and x_δ for the data being inserted/deleted respectively. Afterwards, we replace any occurrence of the aggregated value x for its value in D^n, that is $x + x_\iota - x_\delta$. Finally, we add a new built-in literal $x_\iota \neq x_\delta$ in the rule with no structural event literals for *mBudget* to ensure that there is, at least, some update in the aggregated value.

Now, we need to define the aggregate event predicates $\iota sumSal$ and $\delta sumSal$. Intuitively, for $\iota sumSal$ we want to sum the new salaries being added to the source *self*. *castMember*. *salary*. Again, we can compute the new instances added to the *source* by replacing their literals according to the formulas 1 and 2:

$$\iota sumSal(m, sum(s)) \leftarrow castMember(c, m, p) \wedge \neg\delta castMember(c, m, p) \wedge \iota cmSalary(c, s)$$
$$\iota sumSal(m, sum(s)) \leftarrow \iota castMember(c, m, p) \wedge cmSalary(c, s) \wedge \neg\delta cmSalary(c, s)$$
$$\iota sumSal(m, sum(s)) \leftarrow \iota castMember(c, m, p) \wedge \iota cmSalary(c, s)$$

Similarly, we can define $\delta sumSal$. In this case, we have to replace insertions by deletions since we are looking for instances which are deleted from the *source*:

$$\delta sumSal(m, sum(s)) \leftarrow castMember(c, m, p) \wedge \neg\delta castMember(c, m, p) \wedge \delta cmSalary(c, s)$$
$$\delta sumSal(m, sum(s)) \leftarrow \delta castMember(c, m, p) \wedge cmSalary(c, s) \wedge \neg\delta cmSalary(c, s)$$
$$\delta sumSal(m, sum(s)) \leftarrow \delta castMember(c, m, p) \wedge \delta cmSalary(c, s)$$

Note that, in both cases, the different rules form a partitioning of the instances being inserted/deleted in the *source* expression. In this way, we can compute the total aggregated value x_ι and x_δ by the sum of the aggregated values obtained from the various rules.

Translating EDCs with Aggregated Events to SQL Translating EDCs with aggregates into SQL queries follows the same principles as before: each literal is translated as a table reference in the `FROM` clause possibly with a `JOIN` condition. For example, the rule EDC 6 is translated as:

```
SELECT M.movie_id,M.budget,sumSalaries.X+ins_sumSalaries.X-del_sumSalaries.X
FROM ins_mBudget AS M
   LEFT JOIN sumSalaries ON(M.movie_id = sumSalaries.movie_id)
   LEFT JOIN ins_sumSalaries ON(M.movie_id = ins_sumSalaries.movie_id)
   LEFT JOIN del_sumSalaries ON(M.movie_id = del_sumSalaries.movie_id)
WHERE M.budget <sumSalaries.X+ins_sumSalaries.X-del_sumSalaries.X
```

Where *sumSalaries* is a table containing the materialized aggregation of the cast member salaries of the different movies, and *ins_sumSalaries* and *del_sumSalaries* are two views computing the aggregation of the cast member salaries being inserted and deleted for the different movies. Note that we need to use `LEFT JOIN` instead of `JOIN` in order not to lose those *movies* for which we do not have any of these aggregate values.

Now, we need to define the SQL views *ins_sumSalaries* and *del_sumSalaries*. Such views are defined by means of translating into SQL the different definition rules of the predicates $\iota sumSalaries$ and $\delta sumSalaries$ specified in the EDCs. For instance, the first definition rule of $\iota sumSalaries$ would be translated as:

```
CREATE VIEW ins_sumSalaries1 AS
SELECT CM.movie_id, SUM(iCMS.salary) AS X
FROM castMember as CM
   LEFT JOIN del_castMember as dCM ON (CM.id = dCM.id)
   LEFT JOIN ins_cmSalary as iCMS ON (CM.id = iCMS.id)
WHERE dCM.id IS NULL
GROUP BY CM.movie_id
```

These views are obtained by translating the body of the rule into SQL following the same principles as before, then applying a `GROUP BY` with the attributes corresponding to the terms of the rule's head, and finally aggregating the corresponding term.

At this point, we only need to define a view combining all the different salary sums corresponding to the different definition rules. In the case of *ιsumSalaries* it would be:

```
CREATE VIEW ins_sumSalaries AS
SELECT iSS1.movie_id, iSS1.X + iSS2.X + iSS3.X AS X
FROM ins_sumSalaries1 as iSS1
   FULL OUTER JOIN ins_sumSalaries2 as iSS2 ON (iSS1.movie_id = iSS2.movie_id)
   FULL OUTER JOIN ins_sumSalaries3 as iSS3 ON (iSS1.movie_id = iSS3.movie_id)
```

Note the usage of `FULL OUTER JOIN` in order not to lose any of the aggregated values in the different *ins_sumSalariesX* views.

Note also that we use the views computing the aggregated value being inserted x_ι and the aggregated value being deleted x_δ to incrementally update the materialized aggregated value x. In our example, we use the views *ins_sumSalaries* and *del_sumSalaries* to update the table *sumSalaries* in case we finally commit the structural events.

3.3 Expressiveness of OCL$_{\mathsf{UNIV}}$ with Aggregation

The expressiveness of the OCL$_{\mathsf{UNIV}}$ language is already determined by a formal grammar in [19]. Incorporating distributive aggregation in OCL$_{\mathsf{UNIV}}$ extends the expressiveness of the language beyond the newly supported aggregate operations `sum`, `size`, `count` since there are many OCL operations beyond OCL$_{\mathsf{UNIV}}$ that can be rewritten in terms of aggregation (e.g. `notEmpty`). To see to what extent we are improving the expressiveness of the OCL fragment, we first discuss the expressiveness in terms of logics, and then, go back to OCL.

From the point of view of logics, dealing with aggregate predicates allows us to handle negative derived literals, that is, denial constraints with the form $\phi(\overline{x}) \wedge \neg d(\overline{x}) \rightarrow \bot$ where d is a derived predicate. Note that negative derived literals can be encoded as an aggregate predicate counting the number of instances satisfying the body of the derived literal, and comparing such number to 0.

Since the work in [15] defines a translation from OCL$_{\mathsf{FO}}$ [11] into the language of denial constraints with derived negative literals, our method can deal with any OCL$_{\mathsf{FO}}$ constraint. OCL$_{\mathsf{FO}}$ is the OCL fragment limited to first order constructs, in other words, it encompasses almost any OCL operation in exception of aggregates (`min`, `max`, `sum`, `count`, `size`) and transitive closure (`closure`).

Nevertheless, with the method proposed here we are also able to deal with some of these aggregates. In particular, we can deal with `sum`, `count`, and `size` because they are distributive. In this manner, the expressiveness of the constraints we deal with is beyond OCL$_{\mathsf{FO}}$.

4 Experiments

We have conducted some experiments to show the scalability of our approach. We have loaded the IMDb public interface available data (about $13*10^4$ movies and $2.5*10^6$ cast members) into an SQL schema corresponding to the UML schema shown in Figure 1. Then, we have measured the execution time of our method to check the OCL constraints of Figure 2 in three different scenarios: adding new movies, modifying salaries of cast members, and deleting movie directors. All these experiments have been conducted in MySQL 5.6 running on Windows 8 in an Intel i7-4710HQ up to 3.5GHz machine with 8GB of RAM.

For each scenario, we have executed our method several times increasing the number of structural events applied in each case. Note that inserting a movie requires several structural events: inserting the movie, its budget, its cast members, etc.; updating a salary requires two events: deleting the old salary and inserting the new one; and deleting a director from a movie requires three: deleting its cast membership, its role and its salary. In Table 1 we show the execution times in seconds for checking each constraint of the example regarding to the number of structural events applied to each scenario.

From these results we can see that the time to check any constraint increases with the number of movie insertions. This is because all three constraints can be violated in this scenario. Insertions of 1000 movie had better response times than those of inserting 500 due to the cache memories of MySQL. When analyzing salary updates, we see that only the constraint *BudgetIsHigher* gets worse results when increasing the number of events considered, while the other two remain almost constant. This is because it is impossible to violate them when updating salaries. The same phenomena occurs with the constraint *NotActorAndActress* in the third scenario since it is impossible to violate it by deleting cast members.

It is worth noting that most of the experiments took less than one second and that only one of them was over 30s. Moreover, the cache memories improved the results of the last experiments with the largest number of structural events. In the case with most number of data changes (22,037 structural events), it took 12.37s to check one constraint in a database with more than 3 million rows.

We also show in Table 2 the execution time in seconds required to update the materialized aggregates for each scenario once the constraint check has been performed. Note that updating the materialized aggregates does not suppose any scalability problem since none of them takes more than 0.5 seconds.

To show the benefits of our incremental approach, we have measured the execution times of the SQL queries obtained from the OCLDresden tool [3], which translates each constraint into an SQL query, but without following an incremental approach. In this case, the execution time to check *NotActorAndActress* was 21.47s, while checking *HasSomeDirector* and *BudgetIsHigher* did not finish within two hours. We could improve these last execution times after manually rewriting the automatic translation provided by the tool, but their results were still hight: 238.33s and 79.44s. Note that, since this method is not incremental, its execution time is independent of the events applied, thus, it takes these times even when the events applied cannot violate any of the constraints.

Table 1. Time in seconds to check constraints

#Movie Insertions	**1**	**5**	**10**	**50**	**100**	**500**	**1000**
#Structural Events	28	149	272	1254	2059	10876	22037
NotActorAndActress	0.36	0.75	5.31	3.51	5.54	9.39	9.13
HasSomeDirector	0.28	0.08	0.33	0.41	0.56	1.42	0.38
BudgetIsHigher	0.41	0.90	5.37	5.54	9.82	30.34	12.37
#Salary Updates	**1**	**5**	**10**	**50**	**100**	**500**	**1000**
#Structural Events	2	10	20	100	200	1000	2000
NotActorAndActress	0.14	0.05	0.05	0.03	0.05	0.03	0.06
HasSomeDirector	0.31	0.00	0.00	0.02	0.02	0.00	0.00
BudgetIsHigher	0.17	0.09	0.30	0.69	1.16	1.00	1.44
#Director Deletions	**1**	**5**	**10**	**50**	**100**	**500**	**1000**
#Structural Events	3	15	30	150	300	1500	3000
NotActorAndActress	0.19	0.20	0.17	0.13	0.16	0.70	0.12
HasSomeDirector	0.45	0.53	0.95	1.79	2.07	13.09	3.93
BudgetIsHigher	0.30	0.47	0.37	0.44	2.38	0.60	0.48

Table 2. Time in seconds to update the materialized aggregates

	1	**5**	**10**	**50**	**100**	**500**	**1000**
Movie Insertions	0.05	0.14	0.12	0.15	0.11	0.48	0.35
Salary Updates	0.08	0.08	0.12	0.08	0.06	0.11	0.10
Director Deletions	0.05	0.05	0.08	0.06	0.42	0.06	0.14

5 Related Work

OCL constraints to SQL Similarly to our method, the work of [3] is based on translating each OCL constraint into an SQL view that will be empty if and only if the constraint is satisfied. Likewise, the work of [4] defines a translation from OCL expressions into MySQL queries/procedures that return the evaluation of the OCL expression. Both translations are able to deal with aggregates, but they are not incremental since whenever a data update occurs, the overall queries need to be recomputed from scratch. In a different way, [20] offers a translation from OCL to SQL queries that are triggered by those data updates that might violate the constraints. However, in this method queries might still look for violations trough the overall data instead of the limited data related to the update.

Incremental OCL Constraints Checking The work in [6] is based on, for each OCL constraint, mapping the different context elements for which the constraint must be evaluated to the related data required to perform such evaluation. Thus, when any of these relevant data is modified, the corresponding OCL constraint is reevaluated for such context element. However, this strategy might be prohibitive due to excessive memory usage depending on the kind of constraints involved [21]. Instead of storing all these data, the works of [5] and [7] are able to compute, given a data update, the constraints that might be violated together the context element for which they should be reevaluated. However, none of these approaches is fully incremental since they only compute the relevant context element for which to evaluate the constraint, but not the relevant data for that context element for which to perform the evaluation.

OCL translation to graph patterns. The work of [8] consists in translating OCL constraints into graph patterns to benefit from graph-pattern incremental queries. Nevertheless, such method uses the RETE algorithm to achieve the incremental behavior, which materializes every relational algebra operation performed by the queries. Such materialization has already been criticized because its huge memory usage and the penalization time required to maintain such materialization. To solve such issues, other algorithms have been suggested such as TREAT [9]. However, TREAT does not handle aggregates.

6 Conclusions

Checking the satisfaction of constraints over some data state is an essential problem in order to ensure data validity. In this paper, we have proposed a method based on SQL to incrementally check the satisfaction of OCL constraints. That is, we build several SQL queries that return the data violating the constraints. Thus, we can check data validity by means of checking SQL query emptiness. Moreover, the queries we propose are defined in a way that perform incrementally. In other words, the SQL queries are only executed when some data update may have violated a constraint, and only examine the data related to the update.

Our method is based on translating each OCL constraint into a set of logic rules that we call event dependency constraints (EDCs). Each EDC is a rule that captures a different combination of events (i.e., data updates) causing the violation of a constraint given the current data state. Each EDCs is then translated into SQL. These SQL queries make use of some auxiliary tables containing the events being applied in the information system, and some other tables containing a materialization of the relevant aggregates required to check the constraint. Thus, the SQL query is a join between the three: events, current data and aggregate values. This join is the key for incrementality since it forces the query to only search for possible violations with the data related to the update.

We have shown that the expressiveness of the OCL constraints we can deal with is beyond OCL_{FO}, the first order fragment of OCL, because of dealing with the distributive OCL aggregation operations (`sum`, `count`, `size`). Furthermore, we have shown the scalability of our method by means of some experiments with real data examining both the time to execute the queries and the time to maintain the materialized aggregated data. As further work, we plan to extend our approach to deal with OCL transitive closure and non-distributive aggregate operations such as `min` and `max`.

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

References

1. Object Management Group (OMG): Unified Modeling Language (UML) Superstructure Specification, version 2.4.1. (2011) `http://www.omg.org/spec/UML/`.

2. Object Management Group (OMG): Object Constraint Language (UML), version 2.4. (2014) http://www.omg.org/spec/OCL/.
3. Heidenreich, F., Wende, C., Demuth, B.: A framework for generating query language code from OCL invariants. ECEASST **9** (2008)
4. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST **36** (2010)
5. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST **44** (2011)
6. Groher, I., Reder, A., Egyed, A.: Incremental consistency checking of dynamic constraints. In: Fundamental Approaches to Software Engineering. Springer (2010) 203–217
7. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software **82**(9) (2009) 1459–1478
8. Bergmann, G.: Translating OCL to graph patterns. In: Model-Driven Engineering Languages and Systems. Volume 8767 of LNCS. Springer (2014) 670–686
9. Miranker, D.P.: TREAT: A better match algorithm for AI production systems. In: Proc. of the 6th National Conf. on Artificial Intelligence. Volume 1 of AAAI., AAAI Press (1987) 42–47
10. Oriol, X., Teniente, E.: Incremental checking of OCL constraints through SQL queries. In: Proc. of the 14th Int. Workshop on OCL and Textual Modelling. (2014) 23–32
11. Franconi, E., Mosca, A., Oriol, X., Rull, G., Teniente, E.: Logic foundations of the OCL modelling language. In: Logics in Artificial Intelligence. Volume 8761 of LNCS. Springer (2014) 657–664
12. Afrati, F., Chirkova, R.: Selecting and using views to compute aggregate queries. In: Database Theory - ICDT 2005. Volume 3363 of LNCS. Springer (2005) 383–397
13. Cohen, S., Nutt, W., Serebrenik, A.: Rewriting aggregate queries using views. In: Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. PODS '99, New York, NY, USA, ACM (1999) 155–166
14. Consens, M., Mendelzon, A.: Low complexity aggregation in graphlog and datalog. In: ICDT '90. Volume 470 of LNCS. Springer (1990) 379–394
15. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM TOSEM **21**(2) (2012) 13
16. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
17. Olivé, A.: Integrity constraints checking in deductive databases. In: Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB). (1991) 513–523
18. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Mining and Knowledge Discovery **1**(1) (1997) 29–53
19. Oriol, X., Tort, A., Teniente, E.: Fixing up non-executable operations in UML/OCL conceptual schemas. In: Conceptual Modeling–ER 2014. (2014) 232–245
20. Al-Jumaily, H.T., Cuadra, D., Martínez, P.: OCL2Trigger: Deriving active mechanisms for relational databases using model-driven architecture. Journal of Systems and Software **81**(12) (2008) 2299–2314
21. Falleri, J., Blanc, X., Bendraou, R., da Silva, M.A.A., Teyton, C.: Incremental inconsistency detection with low memory overhead. Softw., Pract. Exper. **44**(5) (2014) 621–641