TINTIN: Comprobación Incremental de Aserciones SQL

Xavier Oriol[1], Ernest Teniente[1], and Guillem Rull[2] *

[1] Universitat Politècnica de Catalunya
{xoriol,teniente}@essi.upc.edu
[2] Universitat De Barcelona
grull@ceipac.ub.edu

Abstract. Ninguno de los SGBD actuales implementa aserciones SQL, obligando así a implementar manualmente su comprobación. Por este motivo hemos desarrollado TINTIN: una aplicación que genera automáticamente el código SQL necesario para comprobar aserciones. Dicho código captura las tuplas insertadas/borradas en una transacción, comprueba mediante consultas SQL que ninguna de ellas viole ninguna aserción, y materializa los cambios en caso de que esto suceda. La eficiencia del proceso se consigue mediante la comprobación incremental de las aserciones.

1 Introducción

Todo sistema de información tiene que poder capturar cualquier estado posible del dominio, y sólo los estados posibles de éste. Para conseguir dicho objetivo, los sistemas de información deben garantizar que sus datos siempre satisfacen ciertas condiciones, llamadas *restricciones de integridad* [1]. Por ejemplo, una restricción de integridad lógica en un sistema de información sobre libros, como el descrito en el diagrama UML de la Figura 1, sería que la fecha de publicación de toda edición de un libro tiene que ser posterior a la fecha de nacimiento de todos sus autores.

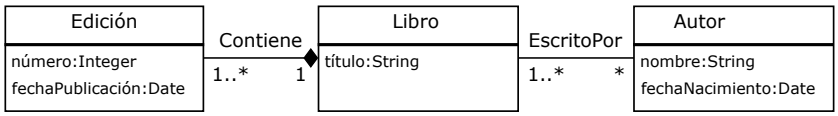

Fig. 1. Sistema de Información de libros y autores

Desde la publicación del estándar SQL-92, los sistema de información implementados con tecnología relacional pueden especificar sus restricciones de integridad mediante *aserciones* SQL. Intuitivamente, se puede definir una consulta SQL que recoja los datos que violan la restricción y una aserción que comprueba

* Este trabajo ha sido parcialmente financiado por el Ministerio de Economía y Competitividad, bajo el proyecto TIN2014-52938-C2-2-R; y la Secreteria d'Universitats i Recerca de la Generalitat de Catalunya bajo un 2014 SGR 1534 y una beca FI.

que el resultado de la consulta sea vacío. Según esta propusta, la anterior restricción se podría definir como:

```
CREATE ASSERTION 'FechasCorrectas' CHECK NOT EXISTS (
        SELECT * FROM Edicion JOIN EscritoPor JOIN Autor
        WHERE fechaPublicacion <= fechaNacimiento)
```

Sin embargo, ninguno de los SGBD relacionales actuales (como por ejemplo Oracle, MySQL, SQL Server, PostgreSQL o DB2) implementa la comprobación automática de aserciones SQL. Por lo tanto, en a la práctica, dichas aserciones acaban siendo comprobadas mediante procedimientos programados manualmente, tarea laboriosa y propensa a errores [2]. En aras de superar esta limitación, en este artículo presentamos TINTIN [3]: una herramienta para generar automáticamente el código SQL necesario para comprobar aserciones.

2 El código generado por TINTIN

Supongamos que, en nuestro ejemplo, se añade el autor *Auguste Maquet* al libro *El Conde de Montecristo* que ya tenía como autor a *Alejandro Dumas*. Asumiendo que la aserción *FechasCorrectas* era cierta en el estado previo a esta inserción, podemos asegurar que después de ella lo seguirá siendo si la fecha de nacimiento de *Maquet* es anterior a la de todas las ediciones de *El Conde de Montecristo*, sin necesidad de comprobar los datos de otros libros/autores. A esta forma de comprobar aserciones en donde se presume que los datos actuales satisfacen las aserciones y se comprueba su validez únicamente con los nuevos datos modificados se le llama *comprobación incremental*.

TINTIN consigue una comprobación incremental mediante *triggers* que capturan los datos que están siendo insertados/borrados durante una transacción y los almacenan temporalmente en unas tablas con prefijo *ins/del*. De este modo, la inserción de *EscritoPor('El Conde de Montecristo','Auguste Maquet')* sería capturada y almacenada en la tabla *ins_EscritoPor* con los valores *('El Conde de Montecristo','Auguste Maquet')*, sin ser físicamente aplicada a la base de datos.

Teniendo capturadas las diferentes tuplas a insertar/borrar en las respectivas tablas *ins/del*, TINTIN puede generar unas consultas SQL que comprueban si estas entran en conflicto con alguna aserción. Siguiendo con el anterior ejemplo, TINTIN genera la siguiente consulta (almacenada como vista):

```
CREATE VIEW 'CheckFechasCorrectas'(
SELECT * FROM ins_Edicion JOIN EscritoPor ANTI JOIN del_EscritoPor JOIN Autor
WHERE fechaPublicacion <= fechaNacimiento
        UNION ALL
SELECT * FROM ins_EscritoPor JOIN Edicion ANTI JOIN del_Edicion JOIN Autor
WHERE fechaPublicacion <= fechaNacimiento
        UNION ALL
SELECT * FROM ins_EscritoPor JOIN Edicion ANTI JOIN del_Edicion JOIN
    ins_Autor
WHERE fechaPublicacion <= fechaNacimiento
        UNION ALL
SELECT * FROM ins_EscritoPor JOIN ins_Edicion JOIN Autor
WHERE fechaPublicacion <= fechaNacimiento
        UNION ALL
SELECT * FROM ins_EscritoPor JOIN ins_Edicion JOIN ins_Autor
WHERE fechaPublicacion <= fechaNacimiento
```

Intuitivamente, el primer *select* detecta las violaciones ocasionadas por inserciones de nuevas ediciones sin añadir nuevos autores. Cabe destacar aquí la necesidad de comprobar que el autor con el que se entra en conflicto no esté siendo borrado también por la transacción, hecho que se comprueba mediante una *anti join*. De manera similar, el segundo y tercer *select* detectan las violaciones ocasionadas al insertar nuevos autores a un libro sin añadir nuevas ediciones, y el cuarto y quinto detectan las violaciones ocasionadas por insertar nuevos autores y ediciones a un libro.

Una vez se tienen estas consultas almacenadas como vistas, sólo hace faltar invocarlas para saber si una transacción provoca o no la violación de alguna aserción. Ésta es precisamente la tarea desempeñada por el procedimiento *safeCommit*, también generado automáticamente por TINTIN. En concreto, *safeCommit* (1) ejecuta las vistas para comprobar si se viola alguna aserción, (2) en caso afirmativo, devuelve un mensaje de error al usuario indicando qué aserción está siendo violada y por qué datos; alternativamente, ejecuta los cambios almacenados en las tablas *ins/del*, (3) elimina el contenido almacenado en las tablas *ins/del* para así iniciar una nueva transacción desde cero. De este modo, la simple invocación de *safeCommit* al final de una transacción es suficiente para garantizar que los cambios definidos por dicha transacción sólo se realizarán si no violan ninguna aserción de la base de datos.

En la versión actual[3], TINTIN es capaz de generar el código para comprobar aserciones descritas en el fragmento de SQL equivalente al álgebra relacional.

3 Sobre cómo TINTIN genera el código

Presentamos ahora resumidamente cómo genera TINTIN las consultas SQL que detectan las violaciones.

En primer lugar, TINTIN reescribe las aserciones SQL como *denegaciones* lógicas siguiendo la traducción definida en [4]. Una denegación es una fórmula que establece una condición que nunca debe evaluarse a cierto en un estado de la base de datos. Por ejemplo, la anterior aserción se reescribiría como:

$$\texttt{edicion}(\texttt{n}, \texttt{fP}, \texttt{l}) \wedge \texttt{escritoPor}(\texttt{l}, \texttt{a}) \wedge \texttt{autor}(\texttt{a}, \texttt{fN}) \wedge \texttt{fP} \leq \texttt{fN} \rightarrow \bot \tag{1}$$

A partir de aquí, TINTIN obtiene las correspondientes *Event Dependency Constraints* (EDCs) [5]. Cada EDC es una regla lógica que identifica una forma particular de violar una denegación mediante inserciones/borrados de tuplas sobre una base de datos D. Para obtener las EDCs, es suficiente con reemplazar cada literal en la denegación por la expresión que evalúa ese mismo literal en el nuevo estado de la base de datos D^n; o sea, el estado de la base de datos después de aplicar la transacción. En concreto, se reemplaza cada literal en función de si este es positivo o negativo mediante las siguientes equivalencias:

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

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

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

donde $\iota p(\overline{x})/\delta p(\overline{x})$ representa la inserción/eliminación de $p(\overline{x})$. Así, según la regla 2, $p(\overline{x})$ es cierto en la nueva base de datos D^n si se inserta $p(\overline{x})$, o si $p(\overline{x})$ ya era cierto en D y no se elimina. La regla 3. define análogamente el caso $\neg p(\overline{x})$.

Utilizando estas sustituciones, en el anterior ejemplo se obtienen las reglas:

$$\texttt{ins\_ed(n,f,l)} \land \texttt{ins\_esc(l,a)} \land \texttt{ins\_aut(a,fN)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (4)$$

$$\texttt{ins\_ed(n,f,l)} \land \texttt{ins\_esc(l,a)} \land \texttt{aut(a,fN)} \land \neg\texttt{del\_aut(a)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (5)$$

$$\texttt{ins\_ed(n,f,l)} \land \texttt{esc(l,a)} \land \neg\texttt{del\_esc(l,a)} \land \texttt{ins\_aut(a,fN)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (6)$$

$$\texttt{ins\_ed(n,f,l)} \land \texttt{esc(l,a)} \land \neg\texttt{del\_esc(l,a)} \land \texttt{aut(a,fN)} \land \neg\texttt{del\_aut(a)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (7)$$

$$\texttt{ed(n,f,l)} \land \neg\texttt{del\_ed(n,l)} \land \texttt{ins\_esc(l,a)} \land \texttt{ins\_aut(a,fN)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (8)$$

$$\texttt{ed(n,f,l)} \land \neg\texttt{del\_ed(n,l)} \land \texttt{ins\_esc(l,a)} \land \texttt{aut(a,fN)} \land \neg\texttt{del\_aut(a)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (9)$$

$$\texttt{ed(n,f,l)} \land \neg\texttt{del\_ed(n,l)} \land \texttt{esc(l,a)} \land \neg\texttt{del\_esc(l,a)} \land \texttt{ins\_aut(a,fN)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (10)$$

$$\texttt{ed(n,f,l)} \land \neg\texttt{del\_ed(n,l)} \land \texttt{esc(l,a)} \land \neg\texttt{del\_esc(l,a)} \land \texttt{aut(a,fN)} \land \neg\texttt{del\_aut(a)} \land \texttt{f} \leq \texttt{fN} \rightarrow \bot \quad (11)$$

Si bien la cantidad de EDCs generada es exponencial con la longitud de la denegación, TINTIN incorpora ciertas optimizaciones para eliminar parte de estas reglas. Así, las EDCs 6 y 10 son eliminadas puesto que, para violarse, la base de datos debería violar primero la integridad referencial *escritoPor(autor)* $\rightarrow$ *autor(nombre)*. Adicionalmente, TINTIN elimina la EDC 11 ya que que únicamente se viola en una base de datos que previamente violara la misma denegación.

A partir de aquí, TINTIN traduce las EDCs a consultas SQL siguiendo las pautas establecidas en [6]. La validez y completitud de las vistas está garantizada puesto que el concepto de EDC se basa en las reglas de eventos [7], las cuales han sido demostradas como válidas y completas en bases de datos deductivas.

4 Conclusiones

Hemos presentado TINTIN, una herramienta para generar automáticamente el código SQL necesario para comprobar aserciones SQL. En el futuro, pretendemos extender TINTIN para (1) hacer frente a los agregados distributivos, (2) generar un número lineal de vistas extrayendo factor común entre las EDCs generadas.

Referencias

1. Olivé, A.: Conceptual Modeling of Information Systems. Springer, Berlin (2007)
2. Tropashko, V., Burleson, D.: SQL Design Patterns: Expert Guide to SQL Programming. Rampant Techpress (2007)
3. Oriol, X., Teniente, E., Rull, G.: TINTIN: a tool for incremental integrity checking of assertions in SQL server. In: Proceedings of the 19th Int. Conference on Extending Database Technology, EDBT. (2016) 632–635
4. Teniente, E., Farré, C., Urpí, T., Beltrán, C., Gañán, D.: SVT: schema validation tool for microsoft sql-server. In: Proc. of the 30th Int. Conference on Very Large Data Bases. (2004) 1349–1352
5. Oriol, X., Teniente, E., Tort, A.: Computing repairs for constraint violations in UML/OCL conceptual schemas. Data & Knowledge Engineering **99** (2015) 39 – 58
6. Oriol, X., Teniente, E.: Incremental checking of OCL constraints with aggregates through SQL. In: Conceptual Modeling. Volume 9381 of LNCS. Springer (2015) 199–213
7. Olivé, A.: Integrity constraints checking in deductive databases. In: Proceedings of the 17th Int. Conference on Very Large Data Bases (VLDB). (1991) 513–523