SVT: Schema Validation Tool for Microsoft SQL-Server*

Ernest Teniente

Carles Farré Toni

Toni Urpí Carlos Beltrán

David Gañán

Universitat Politècnica de Catalunya Jordi Girona 1-3. 08034 – Barcelona. Catalonia (Spain) teniente@lsi.upc.es

Abstract

We present SVT, a tool for validating database schemas in SQL Server. This is done by means of testing desirable properties that a database schema should satisfy. To our knowledge, no commercial relational DBMS provides yet a tool able to perform such kind of validation.

1. Introduction

Database schema validation is related to check whether a database schema correctly and adequately describes the user intended needs and requirements. The correctness of the data managed by database management systems is vital to the more general aspect of quality of the data and thus of their usage by different applications.

This is an increasingly important problem in database engineering, particularly since database schemas are becoming more complex. Indeed, detecting and removing possible flaws at schema design time will prevent those flaws from materializing as run time errors or other inconveniences at operation time.

As an example, assume we have two tables containing information about categories and employees: *Category* (*name, salary*) and *Employee*(*ssn, name, catName*), where underlined attributes correspond to primary keys and where *catName* is a foreign key for the *Employee* table. We could define those tables as follows:

CREATE TABLE Category (name char(10) PRIMARY KEY, salary real NOT NULL, CONSTRAINT chName CHECK (name <> 'ceo'), CONSTRAINT chMinSal CHECK (salary > 50000), CONSTRAINT chMaxSal CHECK (salary < 45000))

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004 CREATE TABLE Employee (ssn int PRIMARY KEY, name char(30) NOT NULL, catName char(10) NOT NULL, CONSTRAINT chCatName CHECK (catName < 'ceo'), CONSTRAINT fkCat FOREIGN KEY (catName) REFERENCES Category(name))

Syntactically, those tables are correctly defined. However, a deeper analysis of the schema allows determining that they may not contain any tuple. The reason is that it is impossible for a category to have a salary lower than 45000 and higher than 50000 as stated by constraints *chMinSal* and *chMaxSal*. Moreover, since employees must belong to categories it is also impossible to insert any employee in the previous database.

We could also realize that the constraint *chCatName* is redundant because it may never be violated. Clearly, employees must belong to categories that may not be named 'ceo' (constraints *fkCat* and *chName*). Then, we do not need to enforce this condition again in the definition of the Employee table by means of *chCatName*.

The designer could fix the previous problems by removing *chCatName* and defining the range of the salary correctly (for instance, by stating that it may range from 45000 to 50000). Now, the database schema could be populated and would allow tuples like *Category(a,30000)* and *Employee(1,john,a)*.

The designer could also be interested to define some views on the new (modified) schema. For instance, a view to retrieve employees' salary (*EmpSalaries*) and another one to retrieve those employees that are not assigned to any category yet (*EmpWithoutCat*):

CREATE VIEW EmpSalaries AS (SELECT ssn, Employee.name, salary FROM Employee, Category WHERE catName = Category.name)

CREATE VIEW EmpWithoutCat AS (SELECT ssn, Employee.name FROM Employee WHERE catName NOT IN(SELECT name FROM Category))

^{*}Research partially supported by Microsoft Research, grant 2003-192. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Some weaknesses may appear also during view definition. As entailed by fkCat, there is no database state where we may find an employee without a category. Therefore, the second view is ill-defined since it may never contain any tuple.

Those are just some examples to illustrate the kind of flaws that may appear during the definition of a database schema. Unfortunately, no relational DBMS provides any tool to perform validations like the ones we have just seen [TG01]. In this demonstration we present a tool, the Schema Validation Tool (SVT), to address this problem.

SVT allows a database designer to perform several tests to check desirable properties of database schemas defined in SQL Server. Among them we have: schema satisfiability, liveliness, integrity constraint redundancy, reachability, etc. SVT is able to check whether a given property is satisfied or not. In the first case, it provides also an example of a database state satisfying the property.

SVT accepts schemas defined by means of a subset of the SQL language provided by SQL Server. It accepts the definition of:

- Primary key, foreign key, boolean check constraints.

- SPJ views, negation, subselects (exists, in), union.

- Data types: integer, real, string.

The current implementation of SVT assumes a set semantics of views and queries and it does not allow null values neither aggregate nor arithmetic functions.

2. Testing Desirable Properties

The goal of this section is to define the set of desirable properties implemented in SVT (inspired on [DTU96]) to validate SQL Server database schemas and to explain the method used by SVT to check them.

2.1 Property Definition

State-satisfiability:

A database schema is state-satisfiable if there is at least one, non-empty, database state where all integrity constraints are satisfied. For instance, the schema containing the initial tables *Category* and *Employee* is not state-satisfiable.

Liveliness:

A table or a view R is lively if there is one consistent database state where at least one fact about R is true. A state is consistent if no integrity constraint is violated on it.

Hence, tables or views that are not lively correspond to relations that are empty in each consistent state of the database. This may be due to the presence of some integrity constraints, to the view definition itself or to a combination of both.

Such predicates are clearly not useful and possibly illspecified. For instance, the view *EmpSalaries* of the previous section was lively while *EmpWithoutCat* was not.

Integrity constraint redundancy:

An integrity constraint (or a subset of constraints) is redundant if database consistency does not depend on it. In other words, it is redundant when it is already guaranteed that the database instances that it wants to avoid will never occur.

SVT distinguishes two different types of redundancy. A constraint is absolutely redundant if it may never be violated. A constraint is relatively redundant (wrt a set of constraints) if it may never be violated when none of the constraints in the set is violated.

In our example, *chCatName* is relatively redundant wrt to *fkCat* and *chName*.

Reachability:

A database designer may be interested also in more general properties like checking whether certain desirable states may be satisfied according to the current schema. This is usually known as checking reachability of partially specified states.

SVT provides two different ways to check reachability: wizard and query reachability. In the first case, the designer may specify by means of a wizard a set of tuples that tables and views should contain. Using this facility, he could define questions like: may the database contain Employee(1,maria,sales) and EmpSalaries (2,joan,47000)? Then, SVT would provide a positive answer and an example database, like for instance Employee(1,maria,sales), Employee(2,joan,sales) and Category(sales,47000), that satisfies the previous question.

In the second case, the desired state is specified by means of an SQL query. For instance, SVT would determine that the state defined by the following query is not reachable:

SELECT Employee.name, salary FROM Employee, Category WHERE Employee.catName = Category.name and salary > 80000 and salary < 90000

Query containment:

A query Q1 is contained into another query Q2 if the answers that Q1 obtains are always a subset of the answers of Q2, independently of the database state.

2.2 Checking Desirable Properties

SVT uses the CQC Method [FTU03, Far03] to effectively and efficiently check desirable properties of SQL Server database schemas. The main goal of this method is to perform query containment tests, i.e. to check whether the answers that a query obtains are a subset of the answers obtained by another query for every database.

Intuitively, the aim of the CQC Method is to construct a *counterexample* that proves that the query containment

relationship being checked does not hold. The method uses different *Variable Instantiation Patterns*, according to the syntactic properties of the queries and the databases considered in each test. The aim is to prune the search of possible counterexamples by generating only the relevant ones but at the same time without diminishing the requirement of completeness.

The CQC Method requires two main inputs. The first one is the definition of the *goal to attain*, which must be achieved on the database that the method will try to obtain by constructing a database state. The second one is the set of constraints to enforce, which must not be violated by the constructed database.

The initial goal to attain may be any conjunction of literals expressing a certain property. Consequently, the CQC Method can check any property that may be formulated in terms of a goal to attain under a set of constraints to enforce. In particular, it is suited to check the database schema validation properties we have previously defined since all of them are aimed to prove that a certain goal is satisfied provided that it does not violate a set of integrity constraints.

The CQC Method is sound and complete in the following terms:

Failure soundness: if the method terminates without building any counterexample, the specified property holds

Finite success soundness: if the method builds a finite counterexample then the property does not hold when queries contain no recursively-defined derived predicates.

Failure completeness: if the property holds between two queries then the method terminates reporting its failure to build a counterexample when queries contain no recursively-defined derived predicates.

Finite success completeness: if there exists a finite counterexample, the method finds it and terminates when either recursively-defined derived predicates are not considered or recursion and negation occurs together in a strict-stratified manner.

Query containment is undecidable for the general case. Therefore in some cases, e.g. in the presence of solutions with infinite elements, it may happen that the CQC Method does not terminate. However, the previous completeness results guarantee that if either there exist one or more finite states for which the property does not hold or there is no state (finite or infinite) satisfying the property, the CQC Method terminates.

3. SVT System Description

In the SVT System we implemented the ideas presented in the previous sections. Its internal architecture consists of several components as it is shown in figure 1.

The GUI component allows using the SVT in an easy and intuitive way. To perform the different tests available in SVT, users go along the following interaction pattern:

1. Select the database schema to be tested.

- 2. Select one among the available tests: Schema satisfiability, Relation liveliness, Query reachability, Wizard reachability, Integrity Constraint Redundancy, Query Containment. Moreover, users must specify one of the two usual semantics regarding to integrity constraint enforcement: Immediate or Deferred.
- 3. Fill the required data that the selected test requires.
- 4. Execute and obtain the test results. To perform the same test with other input data, users may go back to step 3. To perform a different test on the same schema, users should go back to step 2.

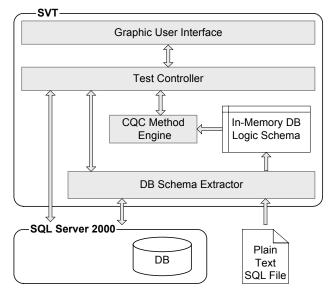


Figure 1. Architecture of SVT System.

The Test Controller component processes the commands and data provided by users through the GUI component and transfers back the obtained results. Among the tasks performed by this component, we highlight the following five ones:

- 1. To establish the connection with the required local or remote SQL Server running system.
- 2. To ask the DB Schema Extractor component to load the schema to be tested from either a given SQL Server system or a plain text SQL file.
- 3. To ask the CQC Method Engine component to perform the required test on the loaded schema.
- 4. If the CQC Method execution provides a counterexample for a given test by specifying a possible content of the database, the Test Controller generates a SQL script that contains the table insertions that allow recreating such database content. The SQL script can be displayed in the GUI component.

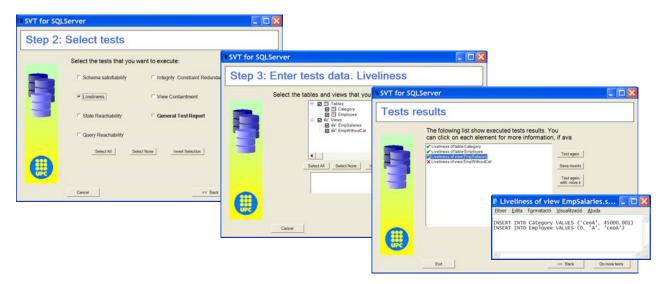


Figure 2. Screenshots of the SVT Demo

5. In any case, then the Test Controller generates also an HTML document describing the execution of the CQC method that lead to such a conclusion.

The main goal of the DB Schema Extractor component is to load an SQL DB schema from a specified source and then transform it to a format that is tractable by the CQC Method Engine. In this way, the DB Schema Extractor generates an in-memory representation of the schema where integrity constraints, views and queries are expressed in terms of deductive rules.

Finally, the CQC Method Engine implements the CQC Method in order to perform the concrete tests asked by the Test Controller component. Such tests must be expressed in terms of the goal to attain the set of constraints to enforce, as explained in previous section.

The whole SVT System has been implemented in the C# language by using Microsoft Visual .NET Studio as a development tool. Our implementation can be executed in any system that features the .NET framework and has access to a local or remote SQL Server system.

4. SVT Demo Description

The demo that we will present is intended to illustrate the main features of the SVT System. The script of the demo is as follows. First, we will define a syntactically correct SQL database schema on a local SQL Server system with the SQL Server Query Analyzer Tool, SQAT for short.

Second, we will start up the SVT system in order to validate the schema previously defined. The first test to perform will be to check whether the schema is satisfiable. In this first test, the obtained result will show that initial schema is unsatisfiable.

Third, we will modify the initial schema with the SQAT in order to make the schema satisfiable. The modification will consist on removing or changing some table constraints.

Four, returning to the SVT, we will test the satisfiability of the modified schema. In this case, the SVT will corroborate such satisfiability.

Five, we will check the liveliness property of the tables and views of the modified schema. We will see examples with or without this property.

Figure 2 shows a sequence of screenshots corresponding to the steps that we would follow if we tested liveliness for the tables and views of the example with which we illustrated Sections 1 and 2. Recall that the two database tables and the view EmpSalaries were lively whereas the view EmpWithouCat was not.

Five, we test the redundancy of the integrity constraints defined in the example. The SVT system will find several cases of redundancy.

Six, we will perform several state reachability test by means of queries or by providing the concrete rows that we want the tables to contain or the views to obtain, with examples of both successful and unsuccessful cases.

Finally, we will perform a variety of query containment tests.

References

- [DTU96] H. Decker, E. Teniente, T. Urpí. How to Tackle Schema Validation by View Updating. In *Proceedings* of EDBT'96: 535-549, LNCS 1057, Springer, 1996.
- [Far03] C. Farré. A New Method for Query Containment Checking in Databases. PhD. Thesis, Universitat Politècnica de Catalunya, 2003
- [FTU03] C. Farré, E. Teniente, T. Urpí. Query Containment With Negated IDB Predicates. *Proceedings of ADBIS* 2003, LNCS, Springer, 2003.
- [TG01] C. Türker, M. Gertz. Semantic Integrity Support in SQL-99 and Commercial (Object-)Relational Database Management Systems. *VLDB Journal* 10(4): 241-269, 2001.