SQL-Sampler: A Tool to Visualize and Consolidate Domain Semantics by Perfect SQL Sample Data

Van Tran Bao Le1  Sebastian Link2  Flavio Ferrarotti1

1 School of Information Management, Victoria University of Wellington, New Zealand
Email: van.t.le@vuw.ac.nz, flavio.ferrarotti@vuw.ac.nz
2 Department of Computer Science, University of Auckland, Auckland, New Zealand
Email: s.link@auckland.ac.nz

Abstract

SQL database designs can result from methodologies such as UML or Entity-Relationship modeling. Description Logic specifications, or relational normalization. Independently from the methodology, the use of good sample data is promoted by academia and commercial database design tools to visualize, validate and consolidate the database designs produced. Unfortunately, advice on what constitutes good sample data, or support to create good sample data are hard to come by. Armstrong databases provide a right notion of sample data that perfectly represent the domain semantics encoded in the form of SQL constraints. We present a tool that computes Armstrong sample tables for different classes of SQL constraints, and different interpretations of null markers. Armstrong tables illustrate the perceptions of an SQL database design about the semantics of an application domain. The tool exemplifies the impact of various design choices on Armstrong tables. These include the expressiveness of the classes of SQL constraints considered, and the semantics of null markers. Armstrong tables complement existing database design methodologies. In particular, they provide data samples that guide the transfer from relational approximations of an application domain to an actual real-life SQL table design.

1 Introduction

Classical database design comprises a variety of methodologies, including conceptual approaches with UML or Entity-Relationship modeling, Description Logic specifications or relational normalization. The output of these approaches is usually a database schema within Codd’s relational model of data. The ultimate classical goal, however, is to design an SQL database schema. Relational database schemata constitute only approximations of the target SQL database schema. The reason is that SQL provides features not available in the relational model. In SQL tables it is possible that duplicate and partial information can occur. This makes data processing more expensive as duplicate removal is often considered to be too expensive, and the occurrence of null markers provides simple yet efficient means for partial information to enter the database. Due to these features, the interaction of SQL constraints is delicate, difficult to comprehend for database designers, and even more difficult to communicate to other stakeholders of the target database. Since SQL database design is a challenging and essential task, academic and commercial database design tools, e.g. ERWin (CA Technologies 2011), promote the use of good sample data to visualize, validate and consolidate the database designs they produce. Unfortunately, advice on what constitutes good sample data, or support to create good sample data are hard to come by. Armstrong databases provide a right notion of perfect sample data (Beeri et al. 1984, Fagin 1982, Hartmann, Kirchberg & Link 2012, Mannila & Räihä 1986). They constitute single database instances that satisfy the SQL constraints currently perceived semantically meaningful by the team of database designers, and, for a given class of SQL constraints under consideration, violate all those constraints currently perceived meaningless. Hence, Armstrong databases are visualizations of abstract sets of SQL constraints.

2 Motivating Example

We will now examine an example that illustrates how Armstrong databases can be used to transfer a relational approximation of a target database schema into an SQL table definition. The example showcases the benefit in using good sample data to complement current database design methodologies. For this purpose we revisit a classical example, originally used to show that there are Boyce-Codd normal form decompositions that cannot preserve all functional dependencies (FDs) (Beeri & Bernstein 1979). Suppose the design team has obtained the relation schema CONTACT with columns Address, City, and ZIP, and FD set Σ with Address → ZIP and ZIP → City. This schema is in Third normal form, but not in Boyce-Codd normal form (Beeri & Bernstein 1979). Normalization algorithms stop here, and cannot provide any further guidance on how to implement the relation schema within an SQL table definition. An inspection of an Armstrong relation for Σ such as

<table>
<thead>
<tr>
<th>Address</th>
<th>City</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 Hudson St</td>
<td>Manhattan</td>
<td>10001</td>
</tr>
<tr>
<td>70 King St</td>
<td>Manhattan</td>
<td>10001</td>
</tr>
<tr>
<td>70 King St</td>
<td>San Francisco</td>
<td>94107</td>
</tr>
<tr>
<td>15 Maxwell St</td>
<td>San Francisco</td>
<td>94129</td>
</tr>
</tbody>
</table>

does also not help, as SQL features like duplicate rows and null markers are not featured in relations. We may therefore ask for an Armstrong table (Hartmann,
An inspection of the table $t$ shows that a specification of FDs does not exclude occurrences of duplicate rows in SQL tables. In fact, $\Sigma$ does not imply any uniqueness constraints (UCs) over SQL tables (Hartmann, Kirchberg & Link 2012). At this stage, the design team decides that the FD $\text{Address, City} \rightarrow \text{ZIP}$ should be replaced by the stronger UC $u(\text{Address, City})$, meaning that there cannot be any different rows with matching total values on both $\text{Address}$ and $\text{City}$. Furthermore, the interpretation of the null marker $n_1$ is no information, i.e., a value may not exist, or it may exist but is currently unknown. This is the interpretation that SQL uses (Zaniolo 1984). The occurrence of $n_1$ in the table above indicates that the column $\text{City}$ is nullable. An Armstrong table $t'$ for the revised constraint set is

<table>
<thead>
<tr>
<th>Address</th>
<th>City</th>
<th>ZIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>03 Hudson St</td>
<td>Manhattan</td>
<td>10001</td>
</tr>
<tr>
<td>03 Hudson St</td>
<td>Manhattan</td>
<td>10001</td>
</tr>
<tr>
<td>70 King St</td>
<td>Manhattan</td>
<td>10001</td>
</tr>
<tr>
<td>70 King St</td>
<td>San Francisco</td>
<td>94107</td>
</tr>
<tr>
<td>15 Maxwell St</td>
<td>San Francisco</td>
<td>94129</td>
</tr>
<tr>
<td>46 State St</td>
<td>n1</td>
<td>60609</td>
</tr>
</tbody>
</table>

Looking at the last two rows of the table $t'$, the design team notices that the UC $u(\text{Address, ZIP})$ is still not implied by the constraints specified so far. As the UC is considered to be meaningful, the designers decide to specify this constraint as well. Inspections of further sample data do not reveal any additional meaningful constraints. Thus, the design team finally arrives at the following SQL table implementation:

```sql
CREATE TABLE Contact (Address VARCHAR, City VARCHAR, ZIP INT,
UNIQUE(Address,City), PRIMARY KEY(Address,ZIP),
CHECK(Q = 0));
```

where the state assertion $Q$ enforces the FD $\text{ZIP} \rightarrow \text{City}$ by

```sql
SELECT COUNT(*)
FROM Contact c1
WHERE c1.ZIP IN (SELECT ZIP
FROM Contact c2
WHERE c1.ZIP = c2.ZIP
AND (c1.City = c2.City)
OR (c1.City IS NULL AND c2.City IS NOT NULL)
OR (c1.City IS NOT NULL AND c2.City IS NULL));
```

on a data or middle tier.

### 3 Contribution

In this article we showcase a new tool, called SQL-Sampler, to generate Armstrong tables for different classes of SQL constraints including:

- NOT NULL constraints,
- uniqueness constraints, and
- functional dependencies,

and the following two interpretations of null marker occurrences:

- $n_1$, that is, no information, and
- unk, that is, value unknown at present.

The tool complements existing database design methodologies by creating sample data with provably good properties. The creation of such sample data is promoted by leading database design tools, but has not enjoyed any support yet. Our tool enables the effective visualization, consolidation and communication of database designs produced by currently available design methodologies. In particular, it can be used to transfer relational approximations of a target schema into an SQL implementation.

### Organization

We discuss related systems and the novelty of our tool in Section 4. In Section 5 we give the necessary definitions of the SQL table model and the classes of constraints under investigation. An overview of the functionality of SQL-Sampler is given in Section 6. The use of the graphical user interface is illustrated by a simple example in Section 7. Details on the implementation of SQL-Sampler are given in Section 8. Finally, we conclude in Section 9.

### 4 Related Systems and Novelty

SQL-Sampler helps design teams identify those SQL constraints that encode the semantics of the given application domain. The help comes in form of Armstrong tables that provide an exact, sample-based representation of the SQL constraints currently perceived meaningful. Design teams can consolidate their current perceptions about the application domain’s semantics by inspecting Armstrong tables together with domain experts.

Armstrong databases are user-friendly representation formats of abstract constraints (Beeri et al. 1984, Fagin 1982, Hartmann, Kirchberg & Link 2012, Mamnita & Räihä 1986). Several prototypes were developed that compute Armstrong databases for a given set of FDs, and the paradigm design-by-example was established (De Marchi et al. 2003, Mamnita & Räihä 1986, Silva & Melkanoff 1979). However, all these tools produce relations. These are just the idealized special case of SQL tables where neither duplicate rows nor null values occur. Thus, relations never show what delicate interactions between NOT NULL constraints, UCs, and FDs are possible over SQL tables. Hence, previous tools do not help with SQL table design - and were not intended for this task.

SQL-Sampler is designed to create Armstrong databases for different classes of SQL constraints, and for different interpretations of null marker occurrences within SQL tables. The classes considered include NOT NULL constraints, UCs, and FDs. The interpretations of null marker occurrences include the no information (Zaniolo 1984) and the unk interpretation (Codd 1979). The choice of a class and an interpretation determines the semantics of SQL tables.
and thus also the Armstrong tables produced. SQL-Sampler is the result of implementing algorithms for different combinations of these SQL constraints, and different interpretations of null marker occurrences, published in our recent work (Ferrarotti et al. 2011, Hartmann, Kirchberg & Link 2012, Le et al. 2012b). Our tool is the first implementation of these algorithms. The tool has been exploited in comprehensive experiments that confirmed its usefulness in identifying semantically meaningful constraints that were incorrectly perceived as meaningless before its use (Le et al. 2013). This research has extended usability studies from the relational model of data (Langeveldt & Link 2010).

5 SQL tables and constraints

In this section we define the syntax and semantics for the different classes of constraints under different interpretations of null markers.

Let $\delta = \{H_1, H_2, \ldots\}$ be a countably infinite set of symbols, called columns. A table schema $\Sigma$ is a finite non-empty subset $T$ of $\delta$. Each column $H_i$ of a table schema $T$ is associated with an infinite domain $dom$ of the possible value types that can occur in column $H_i$. To encompass partial information every column may contain occurrences of a null marker, $ni \in dom(H_i)$.

For column sets $X$ and $Y$ we may write $XY$ for $X \cup Y$. If $X = \{H_1, \ldots, H_m\}$, then we may write $H_1, \ldots, H_m$ for $X$. In particular, we may write $H$ to represent $\{H\}$. A row over $T$ is a function $r : T \rightarrow \bigcup_{H \in T} dom(H)$ with $r(H) \in dom(H)$ for all $H \in T$. For $X \subseteq T$ let $r(X)$ denote the restriction of the row $r$ over $T$ to $X$. An SQL table $t$ over $T$ is a finite multi-set of rows over $T$. For rows $r_1$ and $r_2$ over $T$, $r_1$ subsumes $r_2$ if for all $H \in T$, $r_1(H) = r_2(H)$ or $r_2(H) = ni$. For example, the row

$$t = (03\text{Hudson St, Manhattan, 10001})$$

subsumes the row

$$t = (03\text{Hudson St,ni, Manhattan,10001})$$

For a row $r$ over $T$ and a set $X \subseteq T$, $r$ is said to be $X$-total if for all $H \in X$, $r(H) \neq ni$. Similarly, an SQL table $t$ over $T$ is said to be $X$-total, if every row $r$ of $t$ is $X$-total. An SQL table $t$ over $T$ is said to be total if it is $T$-total.

A null-free sub-schema (NFS) over the table schema $T$ is an expression $nfs(T_s)$ where $T_s \subseteq T$. The NFS $nfs(T_s)$ over $T$ is satisfied by an SQL table $t$ over $T$, denoted by $\models_{T_s} \neg nfs(T_s)$, if and only if $t$ is $T_s$-total. In practice, the NFS consists of those columns declared NOT NULL in the SQL table definition.

An SQL functional dependency (SFD) over a table schema $T$ is an expression $X \rightarrow Y | X, Y \subseteq T$. An SQL table $t$ over $T$ satisfies the SFD $X \rightarrow Y$ if for all rows $r, r'$ in $t$ having the following holds: if $r(X) = r'(X)$ and $r, r'$ are both $X$-total, then $r(Y) = r'(Y)$ (Lien 1982). An SQL uniqueness constraint (SUC) over table schema $T$ is an expression $u(X)$ where $X \subseteq T$.

An SQL table $t$ satisfies the SUC $u(X)$ if for all rows $r, r' \in t$ the following holds: if $r(X) = r'(X)$ and both $r$ and $r'$ are $X$-total, then $r = r'$. For example, both SQL tables $t_1$ and $t_2$ from the introduction satisfy ZIP $\rightarrow$ City. While the table $t_1$ violates every SUC, the table $t_2$ satisfies $u(\text{Address, City})$ but violates $u(\text{Address, ZIP})$.

Let $C$ be a class of constraints, for example, the combined class of NOT NULL constraints, SUCs and SFDs. We say for a set $\Sigma \cup \{\varphi\}$ of constraints from $\Sigma$ over table schema $T$ that $\Sigma$ implies $\varphi$, denoted by $\Sigma \models \varphi$, if for every SQL table $t$ over $T$ that satisfies every constraint in $\Sigma$, $t$ also satisfies $\varphi$.

For example, the table $t_2$ from the introduction shows that the set $\Sigma$ consisting of $ZIP \rightarrow \text{City}$, $u(\text{Address, City})$, and the NFS $nfs(\text{Address, ZIP})$ does not imply $u(\text{Address, ZIP})$.

For a set $\Sigma$ of constraints in $\Sigma$ over table schema $T$, we say that an SQL table $t$ over $T$ is $\Sigma$-Armstrong for $\Sigma$ if $t$ satisfies every constraint in $\Sigma$, and violates every constraint in $\Sigma$ over $T$ that is not implied by $\Sigma$.

For example, the table $t_2$ from the introduction is Armstrong for the set $\Sigma$ containing $ZIP \rightarrow \text{City}$, $u(\text{Address, City})$, and $nfs(\text{Address, ZIP})$. By inspecting table $t_2$, we know that $\Sigma$ does not imply $\text{City} \rightarrow \text{Address}$ nor $u(\text{Address, ZIP})$, but does imply $\text{ZIP, City} \rightarrow \text{Address, City}$.

Constraints can also be defined on tables that feature the Codd null marker unk, instead of ni. In that case we speak of Codd tables. For a Codd table $t$ over $T$, the set $\text{Poss}(t)$ of all possible worlds relative to $t$ is defined by:

$$\text{Poss}(t) = \{t' \mid t'$ is a table over $T$ and there is a bijection $b : t \rightarrow t'$ such that $\forall r \in t, r$ is subsumed by $b(r)$ and $b(r)$ is $T$-total$\}$$

A Codd functional dependency (CFD) over table schema $T$ is an expression $\varphi(X \rightarrow Y)$ where $X, Y \subseteq T$. A Codd table $t$ over $T$ satisfies $\varphi(X \rightarrow Y)$ if there is some $p \in \text{Poss}(t)$ such that for all rows $r, r' \in p$ the following holds: if $r(X) = r'(X)$, then $r(Y) = r'(Y)$. A Codd uniqueness constraint (CUC) over table schema $T$ is an expression $\varphi(u(X))$ where $X \subseteq T$. A Codd table $t$ satisfies $\varphi(u(X))$ if there is some $p \in \text{Poss}(t)$ such that for all rows $r, r' \in p$ the following holds: if $r(X) = r'(X)$ and both $r$ and $r'$ are $X$-total, then $r = r'$. The notions of implication and Armstrong tables, defined in the context of SQL tables above, are defined analogously in the context of Codd tables. The use case in Section 7 discusses CUCs and CFDs on the same example from the introduction, thereby illustrating the differences between both semantics.

Algorithms to compute $\Sigma$-Armstrong tables were recently developed for the classes of NOT NULL constraints and i) SUCs in (Le et al. 2012b,a), ii) SFDs in (Hartmann, Kirchberg & Link 2012), iii) SUCs and SFDs in (Hartmann, Kirchberg & Link 2012), and iv) CUCs and CFDs in (Ferrarotti et al. 2011). Our tool implements all of these algorithms.
Figure 2: Main Interface of SQL-Sampler

Figure 3: Screenshot of Selecting the Context
6 System Overview

SQL-Sampler was developed in C#. The desktop version runs in Windows 7 (64 bit) and can be downloaded at armstrongtable.sim.vuw.ac.nz/ArmstrongData.zip and the web-based tool is available at armstrongtable.sim.vuw.ac.nz.

Its general workflow is depicted in Figure 1. The graphical user interface (GUI) of SQL-Sampler consists of four main modules, as shown in Figure 2.

6.1 Context Module

In the context module, users select the class of constraints they consider for their application domain. This choice also determines the interpretation of null marker occurrences within the Armstrong tables produced by the tool. Possible selections include the context of i) subsumption-free SQL tables where NOT NULL constraints and SFDs are considered, ii) SQL tables where NOT NULL constraints, SUCs, and SFDs are considered, iii) Codd tables where NOT NULL constraints, CUCs, and CFDFs are considered, and iv) SQL tables with keys where NOT NULL constraints and SUCs are considered. Note that in subsumption-free SQL tables, the class of SFDs subsumes the class of SUCs, but in arbitrary SQL tables the class of SFDs does not subsume the class of SUCs (Hartmann, Kirchberg & Link 2012). For contexts i), ii), and iv), the corresponding interpretation of the null marker is fixed to unk for no information. That is, a value may not exist or it exists, but is currently unknown. For context iii), the interpretation is fixed to unk for value exists, but is currently unknown.

6.2 Input Module

In the input module the user defines a table schema, a set of columns declared NOT NULL and a set Σ of UCs and/or FDs. The constraints are specified by a simple selection of the columns involved. Figure 4 shows part of the input module. Users can also open saved inputs from a file, and values for the domains of columns can be defined. These values are then used in the computation module to populate Armstrong tables. If no values are provided by users, generic values will be chosen. Users always have the choice of suitably replacing values in the Armstrong tables by new values. The system guarantees that the replacements always result in Armstrong tables.

6.3 Computation Module

For the computation module several algorithms established in the recent literature (Ferrarotti et al. 2011, Hartmann, Kirchberg & Link 2012, Le et al. 2012) have been implemented to compute Armstrong tables from the context and input specified. Figure 2 shows an Armstrong table produced for our example from the introduction. For users interested in the composition and structure of the Armstrong table, other computational features can be selected. These include the computation of closures of sets of columns, the computation of maximal set families, and the computation of duplicate rows (Hartmann, Kirchberg & Link 2012). In Section 7 we illustrate the definition of maximal and duplicate sets by a detailed example, and explain their instrumental role in computing Armstrong tables.

6.4 Output Module

The output module allows the user of SQL-Sampler to modify, present and save the Armstrong table produced. Figure 2 shows the legend interface where values from the table produced can be replaced manually by the user. The interface guarantees that the tables resulting from these replacements are always Armstrong tables for the context and input specified earlier. Finally, the user has the possibility to save the Armstrong table in a file.

7 Use Case

In this section we briefly illustrate the use of SQL-Sampler on a simple example.

As use case we select the relation schema Contact which consists of the columns Address, City, and ZIP. The null-free subschema nfs(Contact) is defined by Contact = {ZIP}, and as the input set of constraints we select the set

Σ = {¬u(Address, City), ¬(ZIP → City)}

that consists of a Codd uniqueness constraint and a Codd functional dependency. We illustrate how SQL-Sampler can be used to compute an Armstrong table for Σ and nfs(Contact) with respect to Codd uniqueness constraints, Codd functional dependencies and NOT NULL constraints.

7.1 Context

The use case description above tells us which context needs to be defined: We select the context Codd Table, which means that the interpretation of all null marker occurrences unk in the Codd table are fixed to “value unknown at present”. The Armstrong table is computed with respect to the combined class of CUCs, CFDFs with NOT NULL constraints. Figure 3 shows how the context can easily be selected in SQL-Sampler.

7.2 Input Data

Figure 5 shows a screenshot of the Input Data module of SQL-Sampler after the data from the use case was filled in.
Figure 5: Screenshot of Putting in Data

Figure 6: Screenshot of Output Data
7.3 Computing Armstrong Table

Figure 6 contains a screenshot of the Armstrong table computed by SQL-Sampler on the basis of the input data. Since no domain values had been supplied by the user, the Armstrong table was populated with generic data values. The screenshot also shows the duplicate sets computed by SQL-Sampler. The definition of maximal and duplicate sets was given in (Ferrarotti et al. 2011, Hartmann, Kirchberg & Link 2012) and is instrumental to the computation of Armstrong tables in all contexts. Here, we illustrate their instrumental role on the use case.

An Armstrong table for a given set of uniqueness constraints and functional dependencies, and a null-free subschema must violate all functional dependencies not implied by and . For every column , however, it suffices to violate those FDs , where the left-hand side is maximal, under set inclusion, with the property that is not implied. Hence, for a column the set contains all those sets that are maximal with the property that is not implied by and . The computation of maximal set families from and is detailed in (Ferrarotti et al. 2011, Hartmann, Kirchberg & Link 2012).

In our use case where , the set of maximal sets for ZIP is . Indeed, and do not imply or but do imply or . When computing an Armstrong table for and it is ensured that for each set , there are two different rows in the table that have matching non-null values on all the columns in and different values on . This ensures that the Armstrong table violates the FD , and thereby also every FD where holds. The exact construction of the Armstrong table depends on the context and is detailed in (Ferrarotti et al. 2011, Hartmann, Kirchberg & Link 2012).

In our use case, for example, the maximal set for the column ZIP is represented by the first and fourth row in the Armstrong table, as shown in Figure 6.

An Armstrong table for a given set of uniqueness constraints and functional dependencies, and a null-free subschema must also violate all uniqueness constraints not implied by and . If for some column , the FD is not implied by and , then is the subset of some set that is maximal for . Hence, the construction of the Armstrong table - as described above - ensures that the uniqueness constraint is violated by the Armstrong table. In our use case, for example, the uniqueness constraint is not implied by and or but not the CFD . If for some column , the FD is not implied by Address, the CFD is maximal for Address, and the FD is violated by the first and second row in the Armstrong table, as shown in Figure 6.

Otherwise, for every column in the Armstrong table, as shown in Figure 6.

7.4 Output Data

Figure 6 shows a screenshot of how the generic values in an Armstrong table can be replaced by real data values. This can be done by activating the “Legend” button in the “Armstrong table output” menu. The user can then manually enter the real data values that replace the generic ones.

8 Some Implementation Details

Apart from computing Armstrong tables for different classes of constraints and different interpretations of null markers, SQL-Sampler should allow users to re-use and modify previous data. This ensures the effective use of the tool in the requirements acquisition process. For that reason we implemented SQL-Sampler in C# which is a powerful programming language to build a graphical user interface application. We have stored the data in an SQL Server database, consisting of nine main tables, and provided also a web application that database designers can access from everywhere without any installation concerns. To develop the web-based application we have re-implemented the algorithms in ASP.NET. We chose ASP.NET due to its properties which allow multiple users to share the same requested data for resources concurrently. ASP.NET utilizes the C# syntax, which enabled us to re-use the code already developed for the desktop version. The web-based application is hosted under the domain of .sim.vuw.ac.nz.

Users can utilize SQL-Sampler as a desktop application or as a web application. For the desktop version, the 32-bit Windows operation system and the .NET framework 2.0 are necessary for SQL-Sampler to operate. For the web application, a system with a common Internet browser such as Firefox, Internet Explorer, or Google Chrome are required at the client side. At the server-side, SQL-Sampler requires Internet Information Services (IIS) and the ASP.NET platform 4.0 and SQL Server 2005/2008 to operate SQL-Sampler.
The algorithms have been implemented as a library of C# objects to handle four different types of Armstrong tables. For each type, relevant components have been coded to handle the sets of columns for the subsequent table, and not NULL columns, UCs, and FDs as entries in an SQL database. This enables the Web-based system to efficiently and securely perform operations on the different sets of data. It is stressed that the implementation includes authentication and authorization mechanisms, as well as access abilities for multiple users. The conceptual diagram of the SQL database is shown in Figure 7.

![Figure 7: SQL Server database for SQL-Sampler](image)

We explain the function of each table in the database next.

**ArmstrongTables** (ArmsID, ArmsName, ContID, CreateUser, CreateDate, LastUpdate, Notes): This table contains the name, ID, and user-related information for each Armstrong table. Most information in this table is automatically updated except for the name of the Armstrong table which is provided by the user.

**RelationSchema** (AttID, ArmsID, AttrID, AttrName, NullFree, ValueType, ValueStart, ValueEnd, ValueSets, Notes): It stores properties of columns including their name, domain type, null-free value, start value and end value which define a range of automatic data values to populate an Armstrong table. If users specify their own domain values, the values will be stored in the ValueSets column.

**FDs** (ArmsID, AttrID, LeftString, RightString, Notes): This table contains the functional dependencies the user specifies during the input data stage.

**UCs** (ArmsID, AttrIDsets, Notes): It contains the uniqueness constraints the user specifies during the input data stage.

**DataInputs** (DataID, DataName, Notes, MenuURL): It contains the categories of input data which SQL-Sampler assembles to generate each type of Armstrong table. Currently, this table consists of four rows to encode names of columns, functional dependencies, uniqueness constraints, and domain values as input data categories. Data in this table cannot be updated by users.

**Contexts** (ContID, ContName, Notes): It stores the different contexts in which Armstrong tables can be computed by SQL-Sampler. Currently, this tables contains four rows to encode subsumption-free SQL tables, SQL tables, Codd tables, and SQL tables for keys, as previously described. Data in this table cannot be updated by users.

**Inputting** (ContID, DataID, Notes): This table specifies the inputs for each context in which Armstrong tables can be computed. For example, this table has four rows to specify the names of columns, null-free subschema, functional dependencies, and uniqueness constraints as input for the context SQL table; and it has three rows to specify the names of columns, null-free subschema, and functional dependencies with nulls as input for the context Subsumption-free SQL table. Data in this table cannot be updated by users.

**Generating** (ContID, CompID, Notes): This table specifies the types of outputs available in each context of SQL-Sampler. In the context Codd table, for example, output is available as Armstrong tables, column set closures, maximal, and duplicate sets. Data in this table cannot be updated by users.

9 Conclusion and Future Work

Humans learn a lot from good examples. SQL-Sampler creates Armstrong sample data that visualizes perfectly the delicate interactions of SQL constraints. It can thus be used by design teams to communicate and consolidate their perceptions of an application domain with different stakeholders of the target database. Leading database design tools (e.g. ERWin) advertise the use of sample data to validate database schemata produced by existing database design methodologies, for examples, ER modeling and relational normalization. SQL-Sampler produces sample data with provably perfect properties, complementing existing methodologies. The inspection of its Armstrong tables leads to the recognition of many meaningful uniqueness constraints and functional dependencies (Le et al. 2013). It thus helps design teams consolidate real-world SQL table designs, and not just relational approximations of application domains.

In future work SQL-Sampler can be enhanced to handle more classes of SQL constraints such as other types of key constraints (Hartmann et al. 2011, Thalheim 1989), cardinality constraints (Hartmann, Köhler, Link & Thalheim 2012, Liddle et al. 1993), referential constraints (Fagin & Vardi 1983), or multivalued dependencies (Fagin 1977, Hartmann & Link 2012, 2006, Link 2012, 2008). It is also desirable to study integrity constraints under different representations of incomplete information, or in data models such as XML (Buneman et al. 2002, Ferrarotti et al. 2013, Hartmann & Link 2009, Vincent et al. 2004), RDF (Lausen et al. 2008, Paredaens 2012), or probabilistic models (Demetrovics et al. 1998, Link 2013a, Suciu et al. 2011).
## Acknowledgement

This research is supported by the Marsden fund council from Government funding, administered by the Royal Society of New Zealand.

## References


CA Technologies (2011), ‘ERwin Data Modeler - methods guide’, [https://support.ca.com/cadocs/0/e002961e.pdf](https://support.ca.com/cadocs/0/e002961e.pdf), page 86.


Fagin, R. (1982), Armstrong databases, Technical Report RJ3440(40926), IBM Research Laboratory, San Jose, California, USA.


