Building Consistent Sample Databases To Support Information System Evolution and Migration

Jesus Bisbal, Bing Wu, Deirdre Lawless, and Jane Grimson

Computer Science Department, Trinity College Dublin, Ireland

e-mail: *Firstname.Lastname*@cs.tcd.ie

Abstract. Prototype databases are needed in any information system development process to support data-intensive applications development. It is common practice to populate these databases using synthetic data. This data usually bears little relation to the application's domain and considers only a very reduced subset of the integrity constraints the database will hold during operation.

This paper claims that in situations where operational data is available, as is the case in information system evolution and migration, a sample of this data must be used to create a prototype database. The paper outlines a method for consistently sampling a database. This method uses a new concept, the Insertions Chain Graph, to assist in selecting instances so that the resulting Sample Database reaches a consistent state, a significant task of database sampling.

1 Introduction

When developing data-intensive applications there is a need to prototype the database that the applications will use during operation. Such a database can be used to support several stages of the development process [12]: (1) Validation of database and/or application requirements: different designs can be checked for completeness and correctness [10]. (2) User training: applications will need a database on which to operate with during training. (3) Testing: as is needed in, for example, system's functional test [4].

In this paper, the term Prototype Database refers to any database constructed to support the development of data-intensive applications. Common practice is to populate a Prototype Database with data from predefined domains (e.g. [10, 15, 11]) instead of using 'real' data. In addition, usually only a very reduced set of integrity constraints (hereafter constraints) that the database will actually hold during operation is considered (e.g. [2]). The resulting Prototype Databases cannot fully support the applications development process.

This paper presents an approach which proposes that existing operational data be used to construct Prototype Databases. This will result in Prototypes specific to the application's domain. In cases where a Prototype Database is a sample of operational data it will be called *Sample Database*. In contrast, a Prototype Database constructed from predefined domains will be called *Test Database*.

The Sampling method presented in this paper is being developed as part of the MILESTONE project, which is developing a methodology for legacy system migration. A Sample Database has been identified as essential to support the migration process [14].

The remainder of the paper is organised as follows. The next section provides a brief overview of related work. Section 3 states the problems consistently sampling a database can rise. Section 4 defines the concept of *Insertions Chain Graph*, required by the method outlined in Sect. 5. The final section summarises the paper and indicates a number of intended future directions for this research.

2 Related Work

Most of the work on Prototype Databases found in the literature is based on the construction of Test Databases. The general mechanism for test data generation involves inserting data values into the database and then testing-and-repairing this data by adding/deleting data so that it meets the specified constraints. Most solutions proposed for test data generation deal with a very reduced set of constraint types (e.g. [11, 15, 2]), tipically functional dependencies and inclusion dependencies only.

A notable exception is found in [10] where a subset of First-Order-Logic (FOL) is used to define the set of constraints that the generated test data must meet. However using an approach based on FOL the whole set of constraints must be explicitly uncovered, a non-trivial task when dealing with large databases. This set of constraints must then be expressed using a FOL language which can result in an obscure set of formulas. This set of formulas must be consistent, otherwise no database can satisfy this set of constraints. It is well-known that to prove consistency in FOL is a semi-decidable problem [6]. It is also necessary to perform some kind of logical inference in order to maintain consistency. Neufeld [10] proved that such approach does not scale.

A different approach is presented in [15]. This method firstly checks for the consistency of an Extended Entity-Relationship (EER) schema defined for the database being prototyped, considering cardinality constraints only. Once the design has been proved consistent, a Test Database is generated. To guide the generation process, a so-called general Dependency Graph is created from the EER diagram. This graph represents the set of referential integrity constraints that must hold in the database and is used to define a partial order between the entities of the EER diagram. The test data generation process populates the database entities following this partial order. Löhr-Richter [9] further develops the test data generation step used by this method, and recognises the need for additional information in order to generate application's domain relevant data.The most significant drawback of this approach is that it considers cardinality and referential integrity constraints only.

3 The Problem of Consistent Sample Database Extraction

This section outlines the main desirable properties to be satisfied by Prototype Databases and describes the issues involved in consistently sampling a database.

3.1 Properties of Prototype Databases

Considering the three uses of Prototype Databases, i.e. validation of requirements, user training and testing, such databases should satisfy two main properties:

P1. Faithfully represent operational data. For both user training and testing purposes, applications should be subjected to conditions as similar as possible to those which they will encounter during operation. Thus a Prototype Database must be 'similar' to the operational data. Testing will be more effective as errors which would otherwise have occurred after the applications were placed in production are more likely to be detected during testing.

To achieve good similarity with operational data, a Prototype Database must be constructed with two features in mind: (a) it must satisfy the same constraints as the operational data; and (b) it must include all the 'data-diversity' found in the operational data, i.e. not being restricted to only one consistent part of the database.

P2. Efficiently constructed. For requirements validation and testing purposes, several Prototype Databases may be required in order to test different possible designs, use different testing data, etc. Therefore the Prototype Database construction process should be efficient.

3.2 The Faculty Database Example

Fig. 1 shows a possible EER schema [3] for a Faculty Database example, which will be used throughout the paper to illustrate the sampling process. This database stores data about Courses, with Students who follow them and Lecturers who lecture these Courses. Every Course must have only one Lecturer, and a given Lecturer can teach several Courses. Each student must follow between two and eight Courses. The faculty is composed by Persons, which can be either Students or Lecturers. The entity Full Time Students represents a subset of Students, those that follow four or more Courses.

Fig. 2 represents a logical relational design[1] for the database example that could be developed from Fig. 1 (see [3]).

3.3 Challenges in the Database Sampling Process

The most important aspect to consider when sampling from an existing database regards the correctness of the resulting database. The Sample Database must be

[1] Primary keys are underlined and foreign keys marked with asterisk (*).

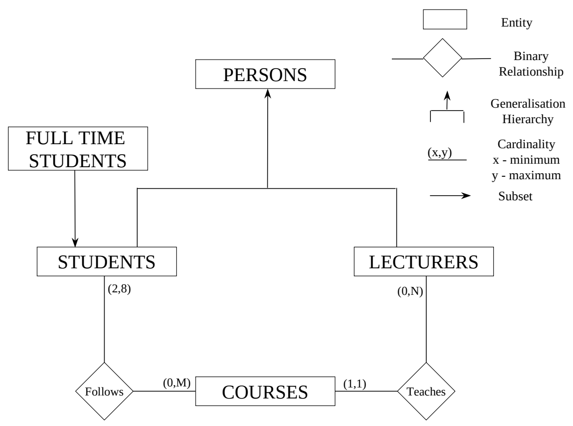

Fig. 1. Extended Entity-Relationship Schema for the Faculty Database

consistent with the same constraints held by the Source Database (term used to refer to the database being sampled). The extraction of such a sample requires knowledge about the set of constraints that the Sample must meet since this information is used to guide the sampling process. Therefore, the first challenge is to extract the set of constraints held by the database (There is a large body of research on database reverse-engineering, e.g. [8], while few database prototyping methods have been reported). This set of constraints is buried within both the database schema (hereafter, schema) and the applications.

Persons(IDPerson, Role, ...)
Students(IDStudent* REFERS TO Persons(IDPerson), ...)
Lecturers(IDLecturer* REFERS TO Persons(IDPerson), ...)
Courses(IDCourse, IDLecturer* REFERS TO Lecturers(IDLecturer), ...)
Follow(IDStudent* REFERS TO Students(IDStudent),
IDCourse* REFERS TO Courses(IDCourse), ...)
FullTimeStudents(IDStudent* REFERS TO Students(IDStudent), ...)

Fig. 2. Logical Relational Design for the Faculty Database

The second challenge posed by consistent database sampling is that due to constraints the inclusion of one instance[2] in the Sample may require the inclusion of other instances[3] before the resulting Sample Database reaches a consistent

[2] Hereafter, terms *entity* and *instance* will be used in descriptions which do not refer to a concrete data model. Terms *table* and *tuple* will refer to the relational model used in the Faculty Database example.

[3] It is well known that updates must be propagated [7] to keep any database in a consistent state after any data manipulation. The same discipline also applies to the construction of a Sample Database.

state. This chain of insertions can be represented by a directed graph, called here Insertions Chain Graph.

4 Insertions Chain Graph

This section formally defines the concept of Insertions Chain Graph (ICG) and presents an example to illustrate this concept.

4.1 Definition of ICG

An Insertions Chain Graph can be formally defined as a quadruple ICG=(Q,I,T,δ), where: **Q** is the set of entities in the database; **I** is the set of arrow identifiers; and **T** is the set of arrow types that describe the type of consequence of inserting an instance into the Sample, as described in Table 1 (Partial arrows subsume all other arrow types. However, the other arrow types are kept because they simplify the resulting ICG). δ is the Insertions Function. For each entity in the

Arrow Type	**Required Information**	**Semantics**	**Constraint Type**
Total	None	One insertion in the target entity is required every time an instance is inserted into the source entity	Referential Integrity Constraints
Quantified	Number of additional instances	The specified number of refered intances must be present in the target entity after inserting the referring instance into the source entity	Cardinality Constraints
OR	Target Entity Names: P$\in 2^Q$, and Criterion to select between them	One instance must be inserted in (some of) the specified entities, after inserting one instance into the source entity	Generalisation Constraints
Partial	Condition	Only when the condition evaluates to true will the insertion be required	General Constraints

Table 1. Arrow Types in an Insertions Chain Graph

database, it describes the set of arrows in the ICG that start at this entity. It is defined as

$$\delta : Q \rightarrow 2^{I \times 2^Q \times T \times C} \tag{1}$$

and thus each arrow in the graph can be described by a quadruple: <**i**, **P**, **t**, **c**> where **i** $\in$ **I**, **P** $\in 2^Q$, **t** $\in$ **T**, and **c** $\in$ C = $\mathbb{N} \bigcup$ **P**redicate $\bigcup$ **S**elector-**P**redicate. **c** represents the additional information required for some arrow types, as detailed in Table 1. It must be interpreted in different ways depending on which arrow type it is associated to (see Sect. 4.2 for examples): (1) QUANTIFIED arrows: c $\in \mathbb{N}$; (2) PARTIAL arrows: c $\in$ **P**redicate must express a condition to be satisfied by the databases before a new insertion into the target entity is required. For each entity E in the database, there are two entity names that may appear in this predicate, Source-E and Sample-E, referring to this entity in the Source and Sample databases respectively. Using the Faculty Database example, and thus the relational model, the language used to express this predicate is assumed to be a well-formed SQL predicate (see [5] for a possible syntax of such a predicate); (3) TOTAL arrows: c has no meaning in this case; and (4) OR arrows: c $\in$ **S**elector-**P**redicate is a criterion to select which of the possible target entities a new instance must be inserted into. It consists of a list of predicates, defined as for Partial arrows, each one associated with an entity in the Sample Database. When a predicate evaluates to true, a new insertion into its associated entity will be required. The syntax to express this criterion is:

Selector-**P**redicate	::=	**P**redicate → Name
		|**P**redicate → Name, **S**elector-**P**redicate
Name	::=	**Sample**-*EntityName*

The rationale behind these four arrow types is that they are able to express much of a Source Database's semantics. The constraint type each arrow type expresses is shown in Table 1 in column Constraint Type.

4.2 Example of ICG

Fig. 3 represents an ICG that includes all semantics of the Faculty Database. Each node of this graph represents one table of the logical design (Fig. 2), and its edges represent database semantics relevant to the sampling process[4].

Arrows starting or ending at entity Persons represent the generalisation hierarchy shown in Fig. 1. A partial arrow from Students to Full Time Students ensures that the semantics of this entity are kept consistent, which also requires a quantified arrow from Full Time Students to Follow to ensure that all Full Time Students actually follow at least four courses. When a tuple from table Follow is sampled, related tuples from Course and Student tables must also be sampled. A quantified arrow from Students to Follow represents the fact that all students must follow at least two courses. Finally, since all Courses must have one lecturer, a total arrow is required from table Courses to Lecturers.

Table 2 contains the definition of all conditions used in Fig. 3. C_1 is associated with a partial arrow and thus it must return a boolean value. It queries the Sample Database to count how many courses a given student follows. If (s)he follows more than four courses (s)he is a full time student. C_2 is associated with an OR arrow and is used to decide whether a given person is a student or lecturer. It queries the Source Database and checks whether the identifier of a given person also appears in table Students or Lecturers (being x the identifier).

Instead of using a graphical representation, an ICG could also be described using its formal definition, in terms of a quadruple ICG=(Q,I,T,δ).

[4] Maximum cardinality constraints have not been represented as their absence cannot lead to an inconsistent Sample Database.

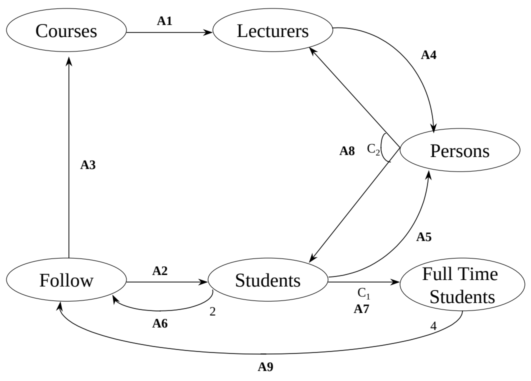

Fig. 3. (Complete) Insertions Chain Graph for the Faculty Database

4.3 Expressiveness of ICG

ICG can be seen as a language to express integrity constraints (it expresses the consequences of these constraints rather than the constraints themselves). As for any language, its expressiveness could be assessed [1]. Since it is based on the query language used by the Source Database, its expressiveness depends on that language. More precisely, the example given above is SQL-based, so its expressiveness is limited by the expressiveness of SQL (which is, in turn, a proper subset of FOL).

Condition Name	Definition
C_1	**Select** count(*) **From** **Sample**-Follow **Where** IDStudent = x) ≥ 4
C_2	**Exists** (**Select** IDLecturer **From** **Source**-Lecturers **Where** IDLecturer = x) $\rightarrow$ **Sample**-Lecturers, **Exists** (**Select** IDStudent **From** **Source**-Students **Where** IDStudent = x) $\rightarrow$ **Sample**-Students

Table 2. Conditions Table for the ICG Shown in Fig. 3

In spite of the fact that some constraints cannot be expressed (e.g. instance-based constraints), as is the case for any other language [1], ICG allows for the definition of a wide range of commonly used constraints.

5 Consistent Database Sampling Method

This section describes a method for consistently sampling a database based on the use of an ICG. For testing purposes it is commonly agreed that random sampling will lead to adequate test data [13]. For this reason instances will be selected randomly[5] from the Source Database.

5.1 Algorithm

The following information is required as input to the sampling process:

1. Insertions Chain Graph for the database being sampled.
2. Minimum number of instances for each entity that should appear in the Sample Database. This information provides a condition to stop sampling.
3. The number of different entities that each entity points to and is pointed by in the ICG, referred to as Fan-out and Fan-in values (see *SelectEntity* below).

Fig. 4 outlines the algorithm for extracting a Sample Database consistent with an Insertions Chain Graph introduced as input parameter. In order to

```
/*NotFullEntities represents the set of entity names which still have not reached
the minimum number of instances in the Sample Database. Initially it would be
the set of entities with a minimum number greater than zero.
SampleDatabase represents the contents of the Sample Database being extracted.*/
begin
     while $\neg$ (NotFullEntities = $\emptyset$ ) do
          entity := SelectEntity (); /* Based on Fan-in and Fan-out values */
          instance := SampleInstance (entity); /* Randomly */
♣         InsertInstance(SampleDatabase,entity,instance,NotFullEntities);
♣                   /* It also modifies NotFullEntities */
♣         TargetEntities := ICG.$\delta$(entity)$|_P$;
♣                   /* Set of entities to be considered as */
♣                   /* a consequence of the current insertion*/
♣         for each target_entity $\in$ TargetEntities do
♣              new_instance := SelectInstance ( target_entity, instance );
♣              KeepConsistent ( target_entity, new_instance, NotFullEntities );
♣         endfor
     endwhile
end
/* OUTPUT: SampleDatabase */
```

Fig. 4. Algorithm for Consistent Sample Database Extraction

simplify the description of the algorithm only those parameters essential to the meaning of each function call are shown.

[5] It must be decided which probability distribution to use. For simplicity, it can be used a uniform distribution. Another option is to use a so-called Operational Distribution [13], where instances frequently accessed when the database is in production will be more likely to become part of the Sample.

• *SelectEntity*: using the ICG, and restricted to the entities in *NotFullEntities*, select the entity with the biggest Fan-out value; if equal, select the entity with smallest Fan-in. If equal again, choose randomly. This heuristic aims to lead to the maximum possible consequences and additional insertions, which will keep the Sample Database consistent.

• *SampleInstance*: randomly samples an instance of the input entity.

• *InsertInstance*: inserts the sampled instance into the Sample Database. It also checks if this entity has already reached the minimum number of instances required in the Sample Database. If so, its name is removed from *NotFullEntities*.

• $ICG.\delta(entity)|_P$: ICG.$\delta()$ is the Insertions Function, and $|_P$ denotes the restriction the resulting quadruple to its second element, P.

• *SelectInstance*: selects from the Source Database the appropriate instance of the *target_entity*. It must be an instance not yet in the Sample.

• *KeepConsistent*: (recursive function) The instance selected when calling to *SelectInstance* must be inserted into the Sample, and this insertion must also be kept consistent. This function implements the part of Fig. 4 marked with ♣.

The reader should note the randomness involved in function *SampleInstance*. Although the starting entity selected in *SelectEntity* is based on the ICG, in *SampleIntance* concrete instances are selected randomly from the Source Database. This procedure will, potentially, lead to the selection of all interesting portions of the Source Database, which refers back to Sect. 3.1, property P1(b).

5.2 Termination and Correctness

The termination of the alrgorithm shown in Fig. 4 can be guaranteed: (1) The outer loop terminates because the number of entities in the set *NotFullEntities* will decrease (function *InsertInstance*) as new instances are inserted into the Sample. Eventually, it becomes the empty set and the algorithm terminates. (2) The chain of recursive calls (function *KeepConsistent*) also terminates due to the workings of function *SelectInstance*. It ensures that infinite loops cannot be created because it only selects instances not yet in the Sample.

This algorithm is correct with regard to the set of constraints represented in the ICG given as input parameter: the Sample Database satisfies those constraints. The size of the resulting Sample is predefined by the user. Additional insertions may be required in order to satisfy the entire set of constraints.

6 Conclusions and Future Research

This paper has analysed the construction of databases used to support the development of database applications. A method for consistent Sample Database extraction has been developed based on an alternative representation of database constraints, the Insertions Chain Graph (ICG). This graph has been formally defined and its role in the sampling process clearly stated.

Future work will focus on adding new constructs to the language (ICG) so that it gains in expressiveness, particularly regarding to the limitations outlined in Sect. 4.3. As seen in Sect. 4.2 not all constraints need to be expressed in order to arrive at a consistent Sample. A characterisation of such constraints will be investigated. Another direction is concerned with the existence of a Minimal Insertions Chain Graph. That is, whether given an ICG an 'equivalent' ICG may be constructed such that it has the minimum possible number of arrows, without omitting any constraint. A minimal ICG will allow for a more efficient sampling process, because only those arrows that actually lead to additional insertions in the Sample Database would be part of the graph.

References

[1] S. Abiteboul and V. Vianu. *Theoretical Studies in Computer Science*, chapter Expressive Power of Query Languages, pages 207–251. Academic Press, 1992.

[2] C. Bates, I. Jelly, and J. Kerridge. Modelling test data for performace evaluation of large parallel database machines. *Distributed and Parallel Databases*, pages 5–23, January 1996.

[3] C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design: an Entity-Relationship Approach.* The Benjamin/Cummings Publishing Company, 1992.

[4] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, second edition, 1990.

[5] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proceedings of the 16th Very Large Databases Conference*, pages 566–577, 1990.

[6] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[7] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley Publishing Company, 1990.

[8] J-L. Hainaut, J. Henrard, J-M. Hick, D. Roland, and V. Englebert. Database design recovery. In *Proceedings 8th Conference on Advanced Information Systems Engineering (CAiSE'96)*, volume 1250 of *Lecture Notes in Computer Science*, pages 272–300. Springer-Verlag, May 1996.

[9] P. Lohr-Richter and A. Zamperoni. Validating database components of software systems. Technical Report 94–24, Leiden University, Department of Computer Science, 1994.

[10] A. Neufeld, G. Moerkotte, and P. C. Lockemann. Generating consistent test data: Restricting the search space by a generator formula. *VLDB Journal*, 2(2):173–213, April 1993.

[11] H. Noble. The automatic generation of test data for a relational database. *Information Systems*, 8(2):79–86, 1983.

[12] I. Sommerville. *Software Engineering.* Addison-Wesley, fifth edition, 1995.

[13] E. J. Weyuker and B. Jeng. Analyzing partition testing techniques. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[14] B. Wu, D. Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, and D. O'Sullivan. The butterfly methodology: A gateway-free approach for migrating legacy information systems. In *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems*, pages 200–205, 1997.

[15] A. Zamperoni and P. Lohr-Richter. Enhancing the quality of conceptual database specifications through validation. In *Proceedings of the 12th International Conference on Entity-Relationship Approach*, volume 823 of *Lecture Notes in Computer Science*, pages 85–98. Springer-Verlag, 1993.