Database Sampling to Support the Development of Data-Intensive Applications

Jesús Bisbal

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

October 2000

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Jesús Bisbal

Dated: October 8, 2000

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Jesús Bisbal

Dated: October 8, 2000

To the memory of my father

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor, Prof. Jane Grimson, without whose continuous encouragement and constructive supervision this dissertation would have never been completed. I am also indebted to my family who has always supported me in all my decisions, and encouraged me to achieve my goals.

I would like to thank numerous people I have been working with for the last four years during the development of this research. Special thanks must go to Deirdre Lawless and Dr. Bing Wu with whom I worked very closely in the Milestone project. Their friendship, advice and understanding made of a small project a big success. Gaye Stephens was extremely helpful when I became a member of the SynEx project. All other members of the Dublin implementation group for the SynEx project contributed immensely to make of this a very enjoyable and fruitful experience, Damon Berry, Eoghan Felton, Sebastien Pardon, Brid Brennan, and Benjamin Jung. I must thank particularly to Cliff Redmond who has been a great support during the last few months of this research.

I would like to thank several people that have been a good influence during these years. With Glòria Santamaria Pérez I shared the unforgettable experience of living in a country without speaking the local language. Rob McGrath and Owen Wallace became my teachers of English language and Irish culture, particularly regarding to pub culture. I would like to thank Meera Gopaul and Thiri Aung for sharing their madness with me. Sebastià Pérez-Chuecos Vallés and Ramón Muñoz Campos, despite the years and the distance, always made sure I would not forget where I come from. I would like to thank Nuria Barceló i Peiró for sharing her disappointments and successes with me, even across continents.

Finally I would like to express my appreciation to Gaye Stephens and Eoghan Felton who

kindly proof-read a draft of this dissertation.

Jesús Bisbal

University of Dublin, Trinity College

October 2000

Abstract

A prototype database is a model of a database which exhibits the desired properties, in terms of its schema and/or data values, of an operational database. Database prototyping has been proposed as a technique to support the database design process in particular, and the whole data-intensive application development process in general (e.g. requirements elicitation, software testing, experimentation with design alternatives). Existing work on this area has been widely ignored in practice mainly on the grounds that its benefits, i.e. an *increase* in the quality of the resulting software systems, do not justify the costs of developing and using a *satisfactory* prototype of the database under construction.

Increasingly more software development projects consist of extending or enhancing existing systems, as opposed to developing new ones. Legacy information systems migration and WEB-enabling existing systems are examples where operational data can be expected to be available at development time. In these types of projects, using the entire operational database may not be cost-effective and a carefully selected subset may be more appropriate. The availability of operational data and a database schema can significantly reduce the effort required to build an appropriate prototype database to support the project at hand. The benefits of using such a prototype, with domain-relevant data and semantics, can also be expected to be higher than in those cases where software is developed from scratch and thus an operational database may not be available from which to build a prototype database.

This thesis investigates how a prototype database can be constructed from an existing database. A prototype database which is populated with data from an operational database is referred to as a *Sample Database*; when it is populated using synthetic data values, it is called a *Test Database*. The context in which prototype databases in general, and Sample

Databases in particular, can be used is analysed. Existing database prototyping approaches are reviewed and a framework to evaluate them is developed. The thesis studies the process of extracting a sample from a database, giving special consideration to the semantic content of the resulting sample. Semantic information commonly used in practice, and how it can be included in the sampling process, is investigated. A formal framework is also developed as a more abstract study of database sampling. A prototype of a database sampling tool, CoDaST, was implemented to test the concepts developed in this thesis.

Publications Related to this Ph.D.

[1] J. Bisbal and J. Grimson. Database prototyping through consistent sampling. In *the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR'2000).* Scuola Superiore Guglielmo Reiss Romoli (SSGRR), August 2000.

[2] J. Bisbal and J. Grimson. Database sampling with functional dependencies. *Information and Software Technology*, June 2000. Submitted.

[3] J. Bisbal and J. Grimson. Generalising the consistent database sampling process. In B. Sanchez, N. Nada, A. Rashid, T. Arndt, and M. Sanchez, editors, *Proceedings of the Joint meeting of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000) and the 6th International Conference on Information Systems Analysis and Synthesis (ISAS'2000)*, volume II - Information Systems Development, pages 11–16. International Institute of Informatics and Systemics (IIIS), July 2000.

[4] J. Bisbal, D. Lawless, R. Richardson, B. Wu, J. Grimson, V. Wade, and D. O'Sullivan. An overview of legacy information systems migration. In Bob Werner, editor, *Proceedings of the Joint 1997 Asia Pacific Software Engineering Conference and International Conference in Computer Science (APSEC'97/ICSC'97)*, pages 529–530. IEEE Computer Society Press, December 1997.

[5] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, September/October 1999.

[6] J. Bisbal, B. Wu, D. Lawless, and J. Grimson. Building consistent sample databases to support information system evolution and migration. In G. Quirchmayr, E. Schweighofer, and T. J.M. Bench-Capon, editors, *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98)*, volume 1460 of *Lecture Notes in Computer Science*, pages 196–205. Springer-Verlag, 1998.

[7] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, D. O'Sullivan, and R. Richardson. Legacy systems migration – a method and its tool-kit framework. In Bob Werner,

editor, *Proceedings of the Joint 1997 Asia Pacific Software Engineering Conference and International Conference in Computer Science (APSEC'97/ICSC'97)*, pages 312–320. IEEE Computer Society Press, December 1997.

[8] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, D. O'Sullivan, and R. Richardson. Legacy system migration : A legacy data migration engine. In Petr Cervinka, editor, *Proceedings of the 17th International Database Conference (DATASEM'97)*, pages 129–138. Czechoslovak Computer Experts, October 1997.

[9] 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 B. Werner, editor, *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems (ICECCS'97)*, pages 200–205. IEEE Computer Society Press, September 1997.

Contents

Acknowledgements **v**

Abstract **vi**

List of Tables **xv**

List of Figures **xvii**

Chapter 1 Introduction **1**

1.1 Motivations 1

1.2 Consistent Database Sampling (CoDaS) 3

1.3 Research Contributions of this Thesis 3

1.4 Thesis Organisation 4

Chapter 2 Context: Applications of Database Prototyping **6**

2.1 Introduction 6

2.2 Terminology 6

2.3 Applications of Test Databases 8

2.3.1 Information System Development 10

2.4 Applications of Sample Databases 11

2.4.1 Information System Development 11

2.4.2 Legacy Information System Migration 12

2.4.3 Data Mining 12

2.4.4 Approximate Query Evaluation 13
2.4.5 Data Mining with Consistent Database Sampling 13

Chapter 3 State of the Art **16**
3.1 Introduction 16
3.2 Software Systems Prototyping 16
3.3 Database Prototyping 17
3.4 Database Prototyping Evaluation Framework 21
3.5 Summary 24

Chapter 4 Database Sampling - The Process **25**
4.1 Introduction 25
4.2 Consistency in Database Sampling 26
4.3 Consistent Database Sampling Process (CoDaSP) 28
4.4 Dealing with Integrity Constraints 31
4.5 Summary 31

Chapter 5 Database Sampling - Applying the Process **33**
5.1 Introduction 33
5.2 School Reference Database 34
5.3 Definitions and Notation 36
5.4 Information to Guide the Sampling Process 39
5.5 Insertions Chain Graph (ICG) 40
5.5.1 Motivations 40
5.5.2 Definition of Insertions Chain Graph 41
5.5.3 Example of Insertions Chain Graph 43
5.5.4 Consistent Database Sampling using an ICG 46
5.5.5 Expressiveness of ICG 49
5.5.6 Sampling Example 50
5.6 Consistent Database Sampling with Functional Dependencies 51
5.6.1 Motivation 51

5.6.2 Defining the Problem 52
5.6.3 Agreements Table 52
5.6.4 Consistent Database Sampling using an Agreements Table 55
5.6.5 Sampling with Subsets or Supersets of Σ 58
5.6.6 Sampling Example 59
5.7 Random Sampling 61
5.8 Analysis of Sampling Algorithms 62
5.8.1 Complexity 62
5.8.2 Termination and Correctness 64
5.9 Summary 66

Chapter 6 Database Sampling - Formal Framework 67

6.1 Introduction 67
6.2 Background 67
6.2.1 Denotational Semantics and λ-Calculus 68
6.2.2 Undecidable Problems and Problem Reducibility 70
6.3 Semantics of Consistent Database Sampling 71
6.3.1 Domains 72
6.3.2 Domains as *Complete Lattices* 72
6.3.3 Supporting Functions 74
6.3.4 Meaning Function 75
6.4 *Sampling-Relevant* Integrity Constraints 76
6.4.1 Characterisation of Sampling-Relevance 78
6.4.2 Non-Decidability of Sampling-Relevance 78
6.5 Summary 81

Chapter 7 Prototype of a Consistent Database Sampling Tool (CoDaST) 82

7.1 Introduction 82
7.2 Requirements of CoDaST 83
7.3 Consistent Database Sampling Protocol 83

7.4 Database Sampler Integration Mechanism (DaSIM) 85
7.5 Design of CoDaST 86
7.5.1 Design of *SamplerIntegrator* 93
7.6 Implementation of CoDaST 97
7.6.1 Defining the Consistency Criteria 97
7.7 Summary 99

Chapter 8 Conclusions **100**
8.1 Review of this Thesis 100
8.2 Summary of Thesis Contributions 103
8.3 Future Work 104
8.3.1 Integrity Constraints 104
8.3.2 CoDaST 105
8.3.3 Theoretical Development 105
8.3.4 Objective Evaluation 106

Bibliography **107**

Appendix A Trial **115**
A.1 Introduction 115
A.2 A Consistent Instance of the Reference Database 115
A.3 Consistency Criteria Definitions in the School Database 117
A.4 A Consistent Sample of the Reference Database 122
A.5 A Consistent Sample using *SamplerIntegrator* 125

List of Tables

5.1 Arrow Types in an Insertions Chain Graph 42

5.2 Conditions Table for the ICG Example of Fig. 5.3 45

5.3 Insertions Function Example 46

5.4 Sampling Information for ICG of Fig. 5.3 50

5.5 Instance of **Courses** (rep.) 53

5.6 Agreements Table Example 54

5.7 Sample with FD of **Courses** 60

A.1 Instance of **Students** 116

A.2 Instance of **Courses** 116

A.3 Instance of **Teachers** 116

A.4 Instance of **Depts** 117

A.5 Instance of **Rooms** 117

A.6 Instance of **Exams** 118

A.7 Instance of **Pre-requisites** 119

A.8 Instance of **Timetable** 119

A.9 Instance of **FullTimeStudents** 120

A.10 Instance of **Persons** 120

A.11 Sample with ICG of **Students** 123

A.12 Sample with ICG of **Courses** 123

A.13 Sample with ICG of **Teachers** 123

A.14 Sample with ICG of **Depts** 124

A.15 Sample with ICG of **Rooms** 124
A.16 Sample with ICG of **Exams** 124
A.17 Sample with ICG of **Pre-requisites** 124
A.18 Sample with ICG of **Timetable** 124
A.19 Sample with ICG of **FullTimeStudents** 124
A.20 Sample with ICG of **Persons** 125

List of Figures

2.1 Applications of Prototype Databases 9
2.2 Data Mining with Consistent Database Sampling 14
3.1 Framework for Evaluation of Prototype Database Construction Methods . . . 22
4.1 Database Sampling Context Diagram 26
4.2 Information Sources Involved in Consistent Database Sampling 27
4.3 Consistent Database Sampling Process (CoDaSP) 29
5.1 School Reference Database EER Diagram 35
5.2 Normalised Schema for the Reference Database 36
5.3 Insertions Chain Graph (ICG) Example 44
5.4 Consistent Sampling with ICG 48
5.5 Consistent Sampling with Functional Dependencies 56
6.1 Non-Decidability of Sampling-Relevant Constraints 80
7.1 Design of Random Samplers 87
7.2 Design of Database Sampler with Functional Dependencies 88
7.3 Design of Database Sampler with Insertions Chain Graph 90
7.4 Set of Basic Database Samplers 91
7.5 Design of Interface to a Database 92
7.6 Design of Sampler Integrator 93
7.7 Sampler Integrator Execution Example 96

7.8 Using LEX/YACC to Generate an ICG Parser 98

A.1 Hierarchy of Database Samplers 127

Chapter 1

Introduction

1.1 Motivations

Software prototyping is a risk-reduction technique commonly used in software development projects when the requirements of the system under development are not well defined. For the purposes of this thesis, a software prototype is a partial implementation of the system to be built (see Section 3.2). This software prototype is exposed to user comment and then refined to meet additional requirements or correct errors. This process is repeated until an adequate prototype has been developed, thus resulting in a better understanding of the requirements for this part of the system. Whether software prototyping is to be used as the main software development paradigm or as part of a wider framework has been widely discussed elsewhere [57, 55, 31, 34] and is out of the scope of this research. Prototyping is, nonetheless, a technique applicable to a wide range of software development projects, which ultimately leads to more usable software.

This thesis is concerned with the process of prototyping data-intensive applications. Thus in addition to the prototype application, a *prototype database* is also required. It has been recognised [66, 31] that, whenever possible, operational data, as opposed to synthetic data, should be used to populate prototype databases. However, the work presented here is the only research found in the literature that studies the process of creating or generating a prototype database from operational data. Existing work populates the resulting prototype databases

mainly with synthetic data values (e.g. 'name1', 'name2', ... as values for an attribute representing persons names). This thesis analyses the database prototyping process and, in particular, *Database Sampling*, that is, the construction of prototype databases populated with operational data. At the simplest level, some applications of sampling are only concerned with the origin of the data, namely that it should be operational data. However, many applications require the resulting database to contain, in addition, semantic information similar to that of the operational database (see Chapter 2). Thus in *Consistent Database Sampling* the objective is to select data items so that the resulting *Sample Database* satisfies predefined criteria, generally a set of integrity constraints.

Software Prototyping, as described above, is a technique closely associated with user requirements analysis. Although this is the main motivation for prototyping, other application areas have also been identified, including user training and software testing [57]. In addition, database prototyping, and in particular Database Sampling, is also applied to other areas outside software development, including data mining and approximate query evaluation (see Section 2). This research was initially motivated in the context of Legacy Information Systems Migration [18], where the need to identify a sample of an existing operational database is essential to the success of any migration project [71], in particular during testing of the transformation of the data from the legacy to the target schema (see Section 2.4.2).

It has been reported that as much as 60% [28] of total software development costs are devoted to enhancing existing applications, to add or modify functionality, rather than developing new applications [33]. Therefore in 60% of projects it is reasonable to expect that an operational database exists from which the sample data can be extracted. Legacy Migration and WEB-enabling existing applications are examples of projects in which a database would be available.

In all the applications outlined above it is likely that it would be too costly to use the entire database and therefore a sample may be required. This justifies the research work reported in this thesis.

1.2 Consistent Database Sampling (CoDaS)

The purpose of database sampling is to extract a sample that *faithfully represents* the operational database. How the *representativeness* of the sample is evaluated depends on the sampling application at hand. For example, in random sampling the representativeness of the sample is evaluated in terms of its size so that a sample with at least a predefined number of instances is considered to represent its operational database faithfully enough. Additionally, there is the underlying assumption that the relationships between instances are not relevant to the representativeness of the sample.

This thesis, however, is concerned with evaluating the representativeness of a sample database in terms of the set of integrity constraints it satisfies. In this context, a sample is considered representative if it is consistent with a predefined set of integrity constraints, which is, in general, a subset of those satisfied by its operational database. Therefore, in Consistent Database Sampling (see Section 2.2), the representativeness criteria are identified with consistency criteria.

For the purposes of this research work the most significant challenge that must be addressed when consistency is used to evaluate the representativeness of a sample database is that the inclusion of one instance in the sample may require the inclusion of other instances before the resulting sample database reaches a state consistent with the specified set of integrity constraints. These insertions may, in turn, require additional insertions. How this chain of insertions is appropriately enforced in a generic and efficient way is the main focus here.

1.3 Research Contributions of this Thesis

This thesis contributes in a number of ways to the research in the area.

- It presents the state of the art in prototype databases construction methods, with a focus on supporting data-intensive applications development.
- Database prototyping approaches can be compared according to an evaluation frame-

work introduced in this thesis. Such framework also identifies the context in which each approach will be better suited.

- It defines the problem of consistently sampling from a database, proposes that database sampling should be used as a database prototyping technique, and distinguishes this approach from that of using synthetic data to populate prototype databases.

- It demonstrates that database sampling is a practical approach by developing sampling methods that consider several types of integrity constraints to be satisfied in the resulting sample.

- It develops a formal framework for database sampling as a benchmark for sampling tools and to help reasoning about database sampling.

- A prototype of a consistent database sampling tool (CoDaST) which incorporates several sampling strategies was designed and implemented. Each strategy is implemented by a sampling module which samples a database according to particular representativeness criteria. This prototype also defines a practical framework where sampling modules can be seamlessly integrated in order to construct a new module that samples a database according to several criteria simultaneously. The thesis describes CoDaST and shows that this incremental construction of complex sampling modules is an appropriate way of sampling a database considering all its semantic complexity.

1.4 Thesis Organisation

This thesis is organised as follows. The next Chapter identifies the range of applications where prototype databases are needed, classifying them based on whether they require the prototype database to be populated with synthetic or operational data.

Chapter 3 reviews existing methods for building prototype databases and presents an evaluation framework that can be used to classify these methods and identify to which type of applications, as in Chapter 2, they are better suited.

The process of extracting a sample from a database is studied in Chapter 4. This description is kept as abstract as possible, without assuming a particular data model (e.g. relational, hierarchical, semi-structured) in the database being sampled or that a specific type of consistency criteria is used to evaluate the resulting sample.

Chapter 5 specialises the description of the sampling process given in Chapter 4 for the case of the relational data model and particular types of integrity constraints. Concrete examples that illustrate the concepts developed throughout this thesis will be based on the School Reference Database. This database is described in this Chapter.

A formal framework for database sampling is developed in Chapter 6. It defines precise semantics for consistent database sampling as specialised in Chapter 5. Chapter 6 also studies a particular type of integrity constraints, referred to as *Sampling-Relevant*, which are of special interest in the context of database sampling.

Chapter 7 details the design and implementation of a prototype of a sampling tool, CoDaST, that has been built as a proof of concept for the developments of this thesis, implementing the methods and algorithms analysed in Chapter 5.

The final Chapter summarises the findings of this thesis and identifies a number of possible future directions for this research.

Appendix A presents a realistic database instance for the School Reference Database of Chapter 5. A Sample of this instance which could result when using CoDaST is also given here.

Chapter 2

Context: Applications of Database Prototyping

2.1 Introduction

This Chapter sets the construction of prototype databases in context by identifying the most relevant applications where prototype databases are required. Section 2.2 introduces the terminology that will be used in the reminder of this thesis, using different terms to refer to prototype databases depending on the origin of the data used to populate them, i.e. synthetic or operational. The applications where each of the resulting prototype database type are best suited are analysed in Sections 2.3 and 2.4 respectively.

2.2 Terminology

For the purposes of this thesis the term *Prototype Database* is defined as follows:

Prototype Database: Any database used to *model* (part of) the data and/or the semantics of another database.

Note that this definition is not concerned with how a prototype database is built, but rather is only concerned with what it is used for.

The main contribution of this research is the investigation of the process that builds a prototype database populated with domain-relevant data from an operational database. The resulting prototype database is called here *Sample Database*.

Sample Database: A Prototype Database populated with data from an existing database, according to predefined data selection criteria. The existing database being sampled may also be referred to as *Source Database*, in order to stress the fact that the sampling process deals with two separate databases, i.e. the Source and the Sample.

Data can be sampled from a database according to many different criteria. For example, data items to be included in a Sample Database could be randomly selected from the Source Database. Frequently, however, it is important to include semantic information in the Sample Database, such as for example satisfying a set of integrity constraints. In these cases the resulting Sample Database is referred to as *Consistent Sample Database*.

Consistent Sample Database: A Sample Database where the data selection is performed following predefined criteria used to evaluate the consistency of the resulting database.

In order to simplify the terminology to be used, a *Sample Database* may also be referred to as simply *Sample*.

Finally, when it is not possible or necessary to use data from an existing database the resulting prototype database is called a *Test Database*.

Test Database: A Prototype Database which is populated using synthetic data, i.e. data values not relevant to the database application domain.

A prototype database can, in fact, have elements of both Test Database and Sample Database. This would be the case, for example, if a subset of the data of an operational database (e.g. the set of first names and last names of teachers in a database storing information about a School) is used as the domain for the values of the attributes in a prototype database. The resulting database can be seen as a Sample Database because each individual data value has been taken from an existing database. It can also be seen as a Test Database,

since the database as a whole is not a direct result of sampling an existing database (e.g. if first and last names are combined randomly the resulting full names may not correspond to any existing name in the Source Database). This may be an appropriate approach in case of an operational database containing variable quality data, as is commonly the case when dealing with legacy databases [18]. With this approach, the resulting prototype database could be of better quality than its Source Database, while at the same time containing data values with which the users would be familiar. However, if the focus is on achieving high similarity with the original database (including its semantics), a common requirement in many prototyping applications, then this prototype database construction method would not be appropriate.

Another mixed approach to database prototyping which could not be regarded exactly as producing either Test Databases or Sample Databases would result if part of the prototype database is populated with data from an existing database, and another part using synthetic values. For simplicity, all these *mixed* approaches to database prototyping are also referred to as Test Databases in this thesis, because the focus here is on building Consistent Sample Databases as defined above.

From the discussion given in this Section, terms *Test Database* and *Sample Database* can be seen as two extremes of a spectrum of possible types of prototype databases, depending on which type of data has been used to populate them. This view is exploited in Section 3.4 when describing a framework for evaluating prototype database construction methods.

The next two Sections analyse the application areas that require the construction of database prototypes. They are classified in terms of which type of prototype database, namely Test Database or Sample Database, can be used in each type of application. Fig. 2.1 summarises the different application areas described here.

2.3 Applications of Test Databases

Database prototyping is commonly used to support several stages of the information systems development process. These applications of database prototyping can, to some extent, be supported using synthetic data, that is, Test Databases. Other applications of database

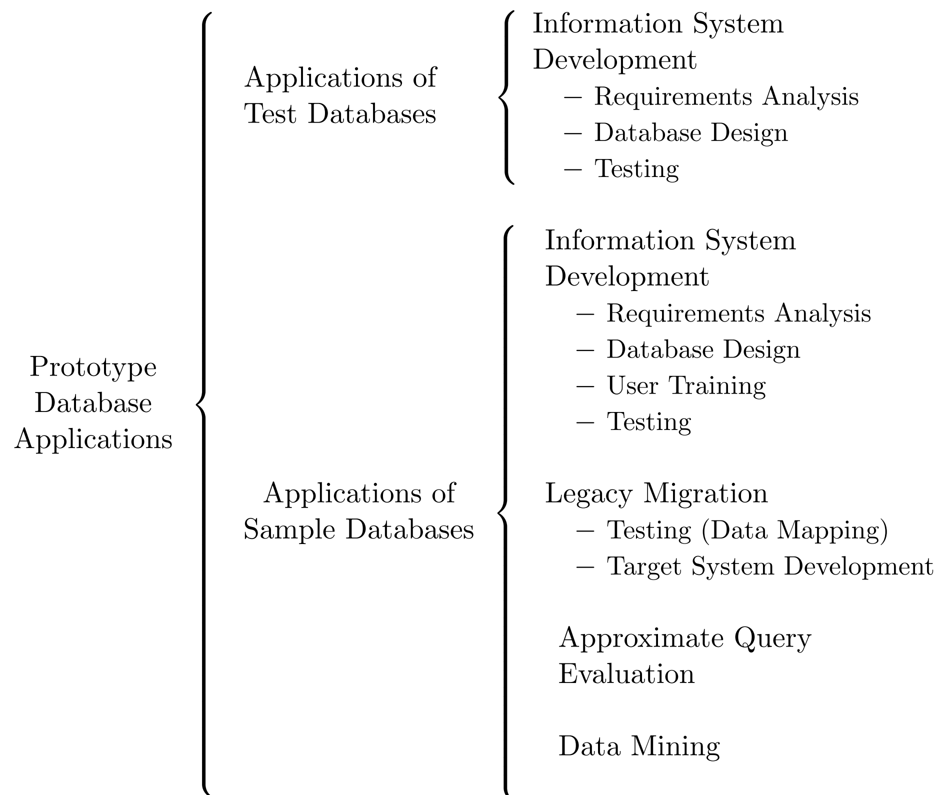

Fig. 2.1: Applications of Prototype Databases

prototyping are associated with the availability of operational data, as will be discussed in Section 2.4.

2.3.1 Information System Development

All prototype database constructions methods available in the literature, with the exception of [19, 16, 14, 15] which resulted from the research presented here, populate the resulting database using synthetic values (see Section 3.3). Although Sample Databases support the information system development process better than Test Databases (refer to Section 2.4.1), there may not always be an operational database from which to sample. Therefore, a Test Database may be the only alternative.

When prototyping data-intensive applications, in order for it be realistic, a prototype should ideally interact with a database with the same schema as the one it will have when in production. A Test Database may fulfill these requirements, depending on both its semantic contents and the semantic needs of the prototyping application. A Test Database can support the following stages of the information system development process.

Requirements Analysis This is regarded as the most common motivation for prototyping an application [74, 33]. In this context, a prototype is exposed to user comment to gain a better understanding of user requirements. Providing the user with a realistic prototype, in terms of behaviour and look-and-feel, may be imperative in this context. When prototyping data-intensive applications, a prototype database is needed to model functional aspects of the application.

Database Design Experimenting with different design alternatives facilitates a better understanding of their completeness and correctness [49, 63, 50]. A design is *complete* if it contains all the desired semantics, and it is *correct* if it does not include any undesired semantics. A partially implemented database, even if solely populated with synthetic values, could support the evaluation of different design options, by exposing missing and undesirable relationships between data items.

Testing At the later stages of software testing, applications must be tested in conditions as similar as possible to those they will encounter during operation [66]. As outlined above, a Test Database with the appropriate schema could lead to a realistic prototyping environment. Examples of such stages of software testing include system functional test [12, 13], performance evaluation [30], and back-to-back testing [57].

2.4 Applications of Sample Databases

An alternative to populating prototype databases with synthetic values is using data sampled from an existing operational database. The applications where a Sample Database would be helpful are analysed next.

2.4.1 Information System Development

All stages of the information system development process analysed in Section 2.3.1 can also be supported using Sample Databases. In fact, it is generally recognised [31, 66] that Test Databases are not as useful and that Sample Databases, when available, should be used instead. Section 2.3.1 identified the need for a prototype to interact with a database with a schema as similar as possible to that of the operational database. If, in addition to the same schema, a prototype database contains domain-relevant data values, this will make the prototype more realistic and hence improve its usefulness. In particular, requirements analysis and testing can be expected to be more effective if operational data is used instead of synthetic data. Users will be able to identify the data items that the prototype is manipulating, understanding their semantics, and thereby providing more useful feedback to the developers. Also, testing will detect those errors that are more likely to occur during operation [66]. Additionally, user training can be particularly well supported by a Sample Database.

User Training Users should only be trained with domain-relevant data; synthetic data would be of limited use. They need to be familiar with the actual data they will view when using the production application and therefore operational data should be used to populate a prototype database in this context.

2.4.2 Legacy Information System Migration

The term *legacy system* is generally used to refer to any software system that significantly resists modification and evolution. A particular type of information system development project is that of Legacy Information System Migration. This can be defined as follows [18, 17]:

Legacy Information System Migration consists in moving an information system to a more flexible environment which allows information systems to be easily maintained and adapted to new requirements, retaining original system data and functionality without having to completely redevelop them.

In this context operational data is certainly available for sampling, as it itself is a part of the system to be migrated. This process will require a prototype database to support the development of the target system, as described in Sections 2.3.1 and 2.4.1. Additionally, in this particular type of project, the data must be migrated from the legacy environment into the target environment. This is a crucial part of any migration project, and some methodologies have been proposed to support this process [21, 41, 71, 69, 70, 53]. The actual mapping from the legacy to the target database schemas must be developed and tested thoroughly. In this testing process, a prototype database as similar as possible to the operational database, e.g. a Sample Database, will be invaluable. Using the entire legacy database for testing purposes only would probably be too costly and unnecessary.

2.4.3 Data Mining

Data mining is the extraction of useful knowledge from large amounts of data. The data analysis should ideally be performed using as much information as possible as this can potentially lead to discovering all useful and relevant patterns in the database. However, data mining algorithms are computationally very expensive. Much research has been undertaken during the last decade to develop faster data mining methods. One proposal in [38] is to mine the information using only part of the database, instead of the entire set of data. This approach achieves a trade-off between the accuracy of the information extracted and the computational

cost of the procedure to mine it.

In this context a prototype database could be extracted from the database to be mined and the mining process performed on this prototype database. The data items used to populate the prototype database would be selected randomly. Kivinen [38] investigates how much data needs to be included in the prototype database as a function of the confidence one expects in the information extracted from this database. See Section 2.4.5 for an alternative sampling strategy for data mining.

2.4.4 Approximate Query Evaluation

Improved performance in answering aggregate queries [35, 51] to relational databases can be achieved by giving approximate answers to them. For example, if the goal is to compute the average value of a particular attribute, a random Sample of the database can be extracted and the result computed for this Sample. The result for the entire database can be inferred form the result for the Sample. It has also been proposed [52] that a *sampling operator* could be included in database management systems, so that only a random Sample of the actual result is returned. This approach can be justified when computing the entire result is unnecessary or too expensive. Note that this operator could be used to implement the approximate answers to queries outlined above.

Random sampling, either uniform [51, 56] or biased [54, 3], has mainly been used in this context. It does not seem likely that Consistent Sampling as defined in Section 2.2 would provide better results than random sampling in these types of applications.

2.4.5 Data Mining with Consistent Database Sampling

Section 2.4.3 briefly outlined the use of Sample Databases without semantic information in data mining. In this context, only approaches based on random sampling have been found in the literature. By randomly sampling a database, some patterns can be lost as related data items may not always be sampled together. Uniform random sampling has been criticised in the context of data mining on the grounds that it may completely miss small clusters of data. Biased sampling has been proposed instead [54]. This Section outlines an alternative

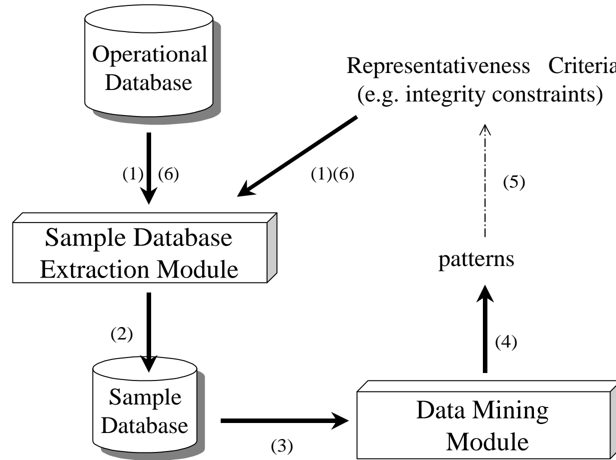

Fig. 2.2: Data Mining with Consistent Database Sampling

approach based on the use of Consistent Database Sampling.

Consistent sampling is expected to lead to Sample Databases which, as a whole, are more representative of the database being analysed than if random sampling is used, thereby increasing the possibility of finding interesting patterns. This idea has been initially proposed by this research work [16].

Consistent Database Sampling in data mining would lead to an iterative data mining process as follows. It would start by sampling the operational database (arrow (1) in Fig. 2.2) using a given consistency criteria, e.g. a known set of integrity constraints. Then the resulting Sample Database (2) would be mined (3). Patterns discovered during mining (4) could suggest (5) additional criteria to be considered when extracting a new sample of the database (6). This process would be repeated until the Sample Database is considered to reflect the consistency criteria sufficiently comprehensively and enough patterns have been extracted from the database.

The process described in this Section would result in a better understanding of the semantics of the database, an important issue in legacy information systems [18], and in a Sample Database highly similar to the operational database, creating a more realistic prototype (see Section 2.3.1). The mining process is also expected to be more efficient than if the entire database was used; the rationale being that it mines several small databases as opposed to

one large database.

The use of Consistent Sampling in Data Mining described above can seem paradoxical. It relies on initial criteria to extract a Sample Database, which is then mined precisely to extract these criteria. There is no such paradox due to the fact that it describes an iterative process. An initial Sample could be extracted according to very basic constraints, like for example a set of referential integrity constraints only. This Sample would trigger the process describe above.

Chapter 3

State of the Art

3.1 Introduction

This Chapter reviews existing literature related to database prototyping from two different points of view. Firstly, Section 3.2 presents an overview of existing research in the area of software systems prototyping, the context in which database prototypes are most commonly used. Secondly, Section 3.3 reviews existing approaches to database prototyping within the context of data-intensive applications prototyping, the main motivation for the research work presented here. Finally, a framework to evaluate existing methods to construct prototype databases is presented in Section 3.4.

3.2 Software Systems Prototyping

A software prototype is a *model* of the software to be built which exhibits the *desired properties* of the final product [34]. A significant number of software prototyping methodologies have been proposed over the years. Each of them addresses different needs that arise during the software development process (e.g. requirements analysis, understanding design alternatives), are based on different techniques (e.g. mock-ups, executable specifications, design and coding in a target language), and are focused on different domain types (e.g. database design, user interface), etc. Reviewing relevant system prototyping approaches is out of the scope

of this thesis. Wood [68] provides a very extensive literature review, including a unifying taxonomy and terminology for the existing prototyping solutions.

The vast majority of software system prototyping approaches available in the literature are associated with requirements engineering, particularly user-friendliness [74], by exposing a prototype to the user who can then comment on required modifications and missing functionality. In some particular application domains, such as data and knowledge intensive systems [31], prototyping is considered an integral part of the whole development process. This is an example of a very young application domain with high uncertainty throughout the development process, and (application and database) prototyping is seen as the most appropriate risk-reduction technique, by allowing design alternatives to be explored. In this context, Guida [31] describes a prototyping framework, termed *Prototyping Hierarchy*, to support the prototyping of data and knowledge intensive applications. This framework consists of several layers, each of which corresponds to a logical paradigm for data representation (i.e. relational, extended-relational, deductive, object-oriented, and active). Software prototypes are built on top of one of these layers, depending on which one is more appropriate to represent the particular application domain. Guidelines are given as to how additional layers could be added to the Prototyping Hierarchy if required. Guida explicitly recognises the importance of using operational data to populate prototype databases as '[...] users can interactively experiment with data structure, while viewing familiar data being manipulated by the prototype.' How operational data is to be used for prototyping purposes is, however, not addressed. The Consistent Sample Database approach being investigated here could become part of such prototyping framework.

3.3 Database Prototyping

Most of the existing approaches to database prototyping generate Test Databases, that is, they populate the resulting database with synthetic values. In the context of the data-intensive applications prototyping, existing approaches can be classified into those for database performance evaluation, and those for requirements analysis. The main goal of the former is to generate large amounts of data, without including specific semantics. The objective of

the latter is to produce a database which is highly similar to the database it models, and therefore they focus on the semantic contents of the resulting database.

Jim Gray [30] reports on a database prototyping method for database performance evaluation. His paper describes sequential and parallel algorithms to populate a database with large amounts of data. The data values are strictly randomly generated using several probability distributions. The approach presented in [9] is also concerned with database performance evaluation. It shows that generic database benchmarking tests prove unsatisfactory for high-performance databases because their underlying schemes are too simplistic and the data volumes being considered too small[1]. Bates [9] paper proposes an alternative, more realistic, benchmark model suitable for large parallel databases. It describes a toolkit used to generate database prototypes based on more real requirements, in terms of workload, semantic information, and data values. The domain values for some attributes are taken from predefined domains (e.g. files with female names, male names, and family names), thus following one of the *mixed* approaches to database prototyping outlined in Section 2.2. As a benchmark, the database schema and semantic information are static, referring to the finance domain. Although a very flexible approach for database performance evaluation, adapting this approach to be used in a different context (e.g. requirements analysis) and in a different application domain (e.g. not for a financial application) does not seem feasible. A similar approach can be found in most papers describing data mining (see Section 2.4.3) algorithms. In this context performance evaluation of the algorithm being proposed is imperative and populating a test database with purely synthetic data is, with few exceptions, the norm. Although each individual paper reports on its own particular synthetic database, the one described in [5] is becoming a common reference Test Database. The databases used in this context are designed for performance evaluation only. Although they do contain semantic information, as mining this information is precisely the objective of the algorithm under test (i.e. finding patterns in the data), the way in which these semantics are included in the data is always

[1] Benchmarks have been updated since this paper was published. The latest version of the benchmark referred to, TPC-C Version 3.5 at http://www.tpc.org/, is effective since 25 October 1999. This new version increases the data volumes according to technological advances. However, the business model still enforces a significantly simpler set of integrity constraints than that used in [9], so the claim of excessive simplicity made by Bates can still be considered to hold.

domain-specific.

Some approaches have been proposed to build Test Databases with significant semantic information, always in terms of the set of integrity constraints being satisfied by the resulting database. 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 items so that it eventually meets the specified constraints. Most solutions proposed for test data generation deal with a very reduced set of constraint types. Noble [50], for example, describes one such method, considering referential integrity constraints and functional dependencies, and populating the database mainly with synthetic values, although the user can also enter a list of values to be used as the domain for an attribute.

A notable exception is found in [49] where a subset of First-Order-Logic (FOL) is used to define the set of constraints that the generated test data must meet, thus allowing complex constraints to be defined. 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-based language to yield a set of formulas, which must be consistent. It is well-known, however, that to prove consistency in FOL is a semi-decidable problem [23]. It is also necessary to perform some kind of logical inference in order to maintain consistency. Neufeld [49] proved that such an approach does not scale.

A different database prototyping approach is presented in [73]. This method firstly checks for the consistency of an Extended Entity-Relationship (EER) diagram [10] 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 which 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 [44] further develops the test data generation step used by this method, and recognises the need for additional information in order to generate relevant data for the application domain. The

most significant drawback of this approach is that it considers cardinality and referential integrity constraints only.

Tucherman [63] presents a database design tool which prototypes a database also based on an Entity-Relationship diagram. The tool automatically maps this design into a normalised relational schema. Special attention is paid in this contribution to what are called *restrict/propagate* rules, that is, to enforcing referential integrity constraints. When an operation is going to violate one such constraint, the system can be instructed to either block (restrict) the operation so that such violation does not occur or to propagate it to the associated tables by deleting or inserting tuples as required. No explicit reference is made as to how the resulting database would be populated for prototyping, although it identifies the possibility of interacting with the user when new insertions are required, as the user may be the only source for appropriate domain-relevant data values.

Mannila [45] described a mechanism for efficiently populating a database relation so that it satisfies exactly a predefined set of functional dependencies (and thus being an Armstrong relation as will be defined in Section 5.3). Such a relation can be used to assist the database designer in identifying the appropriate set of functional dependencies which the database must satisfy. Since this relation satisfies all the required dependencies and no other dependency, it can be seen as an alternative representation for the dependencies themselves. A relation generated using this method is expected to expose missing or undesirable functional dependencies, and the designer can use it to iteratively refine the database design until it contains only the required dependencies. Special attention is paid in [45] to the size of the relation being generated. A designer might not be able to identify missing or undesirable dependencies if s/he was presented with a relation containing a large number of tuples. For this reason, the goal is to produce a relation that, when satisfying exactly a given set of functional dependencies, contains the smallest possible number of tuples. Refer to [47] for a more extended treatment of this approach.

All database prototyping approaches described above populate the database using data values not related to the application domain. This limits their applicability, as was described in Section 2.4. It must be noted that the need for a Sample of a database arises in a context

significantly different from that which motivated existing database prototyping approaches. Methods that produce synthetically generated databases are needed when developing completely new applications. Currently, there is an increasing need to develop or extend applications based on existing software and data (maintenance, migration, etc.) which motivates the need to address the traditional problem but with an additional constraint, the availability of existing applications and related data. Refer to [18] for a more extended discussion on the challenges posed by existing (legacy) applications.

The research reported in this thesis addresses the limitations of the approaches outlined above by building prototype databases that are populated with operational data. Section 5.5 [19] describes a Sample Database construction method which can enforce sets of commonly used integrity contraint types (including cardinality constraints, generalisation dependencies, inclusion dependencies and subset dependencies). From the point of view of the semantic information included in the resulting database this method can be seen as having similar objectives as [49, 50, 73, 63] described above. Section 5.6 [15] describes a method to construct a Sample Database with a single database relation which is an Armstrong relation for a given set of functional dependencies, and thus is comparable to the approach presented in [45]. In Chapter 4 [16] the database sampling process is analysed from a more abstract point of view, without focusing on any particular type of integrity constraint or specific data model (e.g. relational, object-oriented, hierarchical, semi-structured) to be sampled, and thus identifying the issues that must always be addressed when consistently sampling from a database.

3.4 Database Prototyping Evaluation Framework

When the need for constructing a Prototype Database arises, it is necessary to identify the appropriate method to use. The requirements of the database prototyping application (e.g. the application areas described in Section 2) will determine the most suitable method(s). Existing approaches can be classified according to two orthogonal criteria: (1) the origin of the data used to populate the Prototype Database; and (2) the amount of semantic information the resulting Prototype Database contains. Based on these two concepts, it is possible to select the appropriate method for constructing the prototype database for the application at

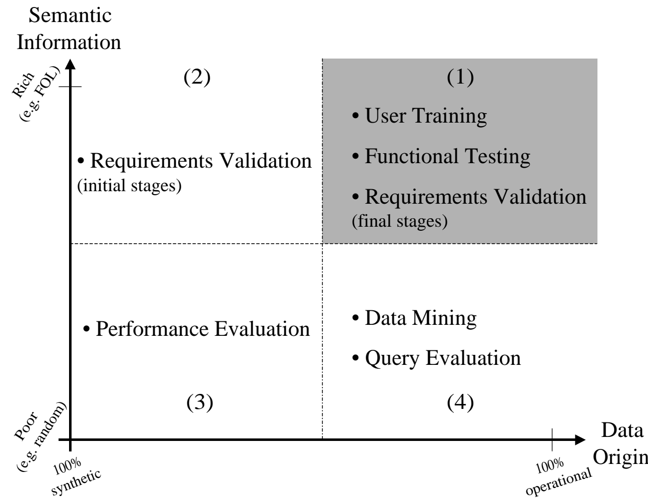

Fig. 3.1: Framework for Evaluation of Prototype Database Construction Methods

hand.

Synthetic vs. Operational Data This issue has already been discussed in Section 2.2.

Poor vs. Rich Semantics Existing solutions enforce different sets of integrity constraints in the resulting prototype database, as analysed in Section 3.3. Random values generation [30] and random sampling [51, 38], for example, would result in databases with poor semantic information as no particular constraints would be enforced. In contrast, an approach which includes a highly expressive language (e.g. first-order-logic or FOL) used to define the set of integrity constraints being enforced would produce databases with richer semantic information.

The kind of support that a prototype database built using a particular method provides to the information system development process can be assessed by identifying where this method falls according to the two classifying criteria given above. These criteria can, therefore, be seen as a framework for the evaluation of prototype database construction methods. Fig. 3.1 shows a graphical representation of such framework, where the two criteria, data origin and semantic content, have been displayed along each of the axes. Given one concrete prototype database construction method, the relative semantic richness it can enforce will provide a

value for its X coordinate in Fig. 3.1. Similarly, the percentage[2] of operational data used to populate a prototype database will define a value for its Y coordinate. Fig. 3.1 has been divided into four quadrants that indicate which methods are more appropriate for the various stages of the information system development process.

Methods in quadrant (1) (e.g. [16]) lead to databases highly similar to those the information system will use in production. For this reason the resulting prototype databases will be useful for user training, software testing and at late stages of requirements analysis. Methods that fall in quadrant (3) (e.g. [30]) do not enforce complex integrity constraints and do not need to query other data sources to populate the constructing prototype database. Such methods can, therefore, efficiently generate large volumes of data, which makes them particularly appropriate for performance evaluation of information systems. Quadrant (2) represents a trade-off between efficiency (quadrant (3)) and faithfulness (quadrant (1)). For this reason methods in this quadrant (e.g. [49]) are appropriate for initial database requirements analysis where alternative designs must be explored (semantic information would be useful) and therefore several prototype databases may need to be generated (need for efficiency). Finally, methods in quadrant (4) (e.g. [51, 38]) would not generally be useful to support information systems development. This quadrant indicates the use of operational data and the inclusion of little semantic information in the prototype database being populated. Using operational data results in prototype databases very similar to the production database; not including semantic information leads to a databases less similar to the operational database, which contradicts the use of operational data within information system development. Methods within this quadrant are, however, the only ones used in applications of sampling such as data mining[3] and approximate query evaluation.

The above discussion places Sample Databases and Test Databases in the context of information systems development. Sample Databases are those produced by methods that fall in quadrants (1) and (4), and Test Databases by those in quadrants (2) and (3). This thesis is concerned with the construction of *Consistent* Sample Databases, which result when

[2]A method could combine operational with synthetic data to populate a prototype database, as discussed in Section 2.2.

[3]Refer to Section 2.4.5, where *Consistent Sampling* is applied to data mining.

using methods in quadrant (1), grayed in Fig. 3.1. It is clear from this Figure that this type of prototype databases can be used to support more stages of the development process than those that result from using methods in other quadrants. Therefore Consistent Sample Databases are the preferred prototype database type to be used when operational data is available.

3.5 Summary

This Chapter has reviewed existing literature in the area of database prototyping to support the development of information systems.

Firstly, it has provided an overview of software systems prototyping, identifying one particular area, data and knowledge intensive systems, where database prototyping and database sampling are considered an integral part of the development process. Then this Chapter has focused on existing approaches to build prototype databases, both for performance evaluation and for requirements analysis. Finally a framework for evaluating existing approaches, and identifying their applications areas, has been described.

Chapter 4

Database Sampling - The Process

4.1 Introduction

This Chapter analyses the process of Consistent Database Sampling and the main issues it involves. The process is described in abstract terms so that it can be applied in the context of disparate data models and a wide range of integrity constraints. Concrete examples of how this process can be specialised to particular data models and constraints are given in Chapter 5.

For the purposes of this Chapter, abstract terms like *entity* and *instance* will be used in descriptions which do not refer to a concrete data model. An *instance* is associated with a data item in a database, and an *entity* with a collection of instances. In terms of the relational model [24], for example, these concepts would be identified with a *tuple* and a *relation* (*table*) respectively.

The next Section identifies the most significant challenges that must be addressed when consistently sampling from a database. Then Section 4.3 defines the Consistent Database Sampling process that underlies the work presented in this thesis. Finally Section 4.4 classifies integrity constraints into two types, namely locally satisfiable and global integrity constraints, as each of these two types of constraints sets different requirements for the way a database must be sampled.

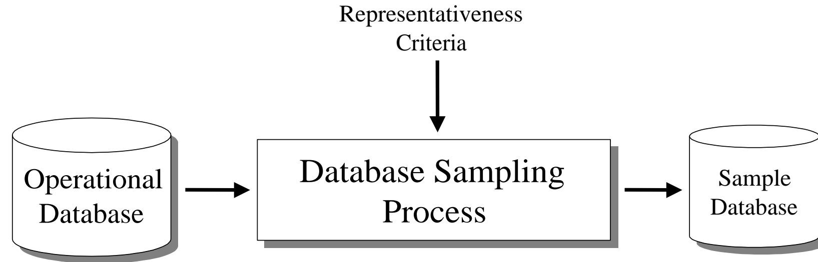

Fig. 4.1: Database Sampling Context Diagram

4.2 Consistency in Database Sampling

Considering any of the applications of database prototyping described in Chapter 2, the purpose of database sampling is to extract a Sample that *faithfully represents* a Source Database, as illustrated in Fig. 4.1. This Figure describes the sampling process in terms of its inputs and its outputs, i.e. a Source Database and criteria used to evaluate the representativeness of the resulting Sample Database. How the *representativeness* of the Sample is evaluated depends on the sampling application at hand. For example, in random sampling the representativeness of the Sample is evaluated in terms of its size so that a Sample with at least a predefined number of instances is considered to represent its Source Database faithfully enough. Additionally, there is the underlying assumption that the relationships between instances are not relevant to the representativeness of the Sample.

This thesis, however, is concerned with evaluating the representativeness of a Sample in terms of the set of integrity constraints it satisfies. In this context, a Sample is considered representative of its Source Database if it is consistent with a predefined set of integrity constraints, which is, in general, a subset of those satisfied by its Source Database. Therefore, in Consistent Database Sampling (see Section 2.2), the representativeness criteria shown in Fig. 4.1 are identified with consistency criteria. Fig. 4.2 illustrates the sources of information required to consistently sample from a database. The Source Database itself is not sufficient to produce a consistent Sample Database. A suitable representation of the set of integrity constraints to be satisfied must be used to guide the sampling process in order to build a consistent Sample. Refer to Chapter 5 for examples of such representations particularly well

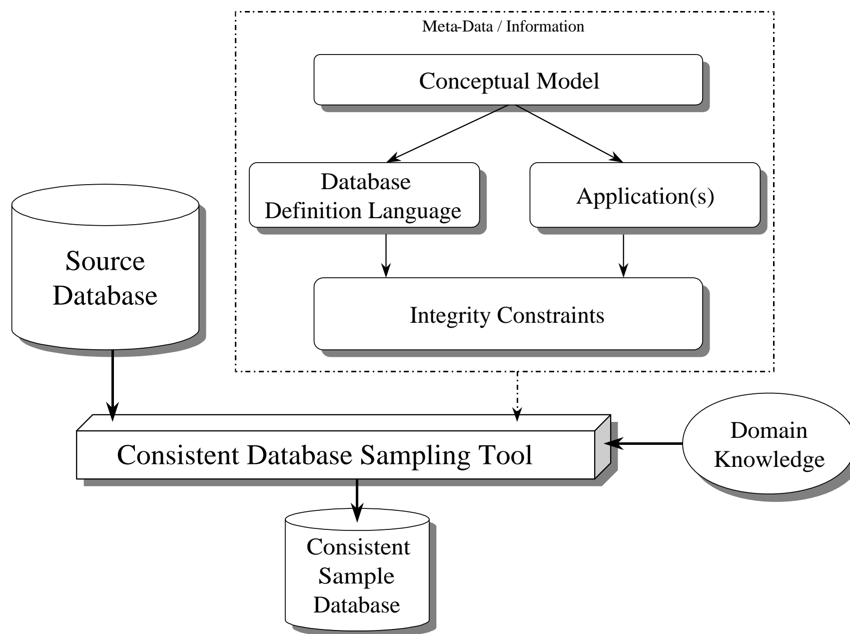

Fig. 4.2: Information Sources Involved in Consistent Database Sampling

suited for sampling purposes. As shown in Fig. 4.2, the set of integrity constraints satisfied by the Source Database is buried within both the Database Definition Language (DDL) and the applications' logic. These constraints could also be explicitly identified in a conceptual model. Finally, domain knowledge is required in this process at different levels. It can be used, for example, to decide which of all the integrity constraints satisfied by the Source Database actually need to be considered in the database sampling application at hand. For example, when maintaining a database schema, database sampling may be used to extract a prototype database of manageable size which will reveal constraints that no longer should be satisfied or that should be added as a consequence of a change in the environment in which the database operates. In this context, a database designer may only require the resulting Sample to satisfy a set of functional dependencies (see Section 5.6). Also, when the size of the resulting database is to be considered (i.e. in isolation or combined with consistency criteria) the required Sample size is likely to be context dependent. In the same context as above, the smallest possible database satisfying the predefined set of functional dependencies is likely to be the most appropriate one.

Although these are, conceptually, the sources of information required for sampling, they may not always be available. For example, in the context of legacy information systems as referred to in Section 2.4.2, a conceptual model is unlikely to be available, and the data definition may not be separated from the logic of the applications. Therefore, this raises the problem of how the set of integrity constraints satisfied by the Source Database can be identified. There is a large body of research on database reverse-engineering, e.g. [32, 6, 58, 4]. Although it could be claimed that no satisfactory solution to this problem has been found to date, this issue is out of the scope of this research work. The appropriate set of integrity constraints to be satisfied by the resulting Sample is considered to be given as one of the inputs to the sampling process. It must be noted, however, that the process of consistent sampling itself is likely to increase the understanding of the semantics of the Source Database (see Section 2.4.5).

For the purposes of this research work, the most significant challenge that must be addressed when consistency is used to evaluate the representativeness of a Sample Database is that the inclusion of one instance in the Sample may require the inclusion of other instances before the resulting Sample Database reaches a state consistent with the specified set of integrity constraints. These insertions may, in turn, require additional insertions. It is well known that updates must be propagated to keep any database in a consistent state after any data manipulation [25]. The same applies to the construction of a Sample Database. How this chain of insertions is appropriately enforced in a generic and efficient way is the main focus of this thesis work.

The particular steps required in order to consistently sample from a database can be described by abstracting the details of a concrete database model or integrity constraints type. This is done next in Section 4.3.

4.3 Consistent Database Sampling Process (CoDaSP)

A generalised description of the Consistent Database Sampling Process is used here to identify the key issues that must be addressed during sampling and therefore provides a better understanding of the process itself. The Consistent Database Sampling Process, or CoDaSP,

```
void DatabaseSampler::ExtractSample(){
    InitialiseProcess();   //Needed information.
    while(!StopSampling()){
        currentInstance = SelectInstance();  //From sourceDB.
        SampleDB.insertInstance(currentEntity, currentInstance);
        Synchronise();       // Inter-Sampler consistency.
        UpdateProcess();  // Intra-Sampler consistency.
    }
}
```

Fig. 4.3: Consistent Database Sampling Process (CoDaSP)

underlying the work reported in this thesis is shown in Fig. 4.3. It is general enough to be applied to any database model and with any arbitrary collection of integrity constraints for that model as it does not make any assumptions in this respect. This Figure shows, in fact, the structure of any iterative algorithm: initialisation, condition, action, and preparation for next iteration. However, it makes explicit the fact that the process is extracting a (consistent) Sample from a database. The algorithmic description of the CoDaSP given here (strongly influenced by the syntax of the C++ programming language [60]) is presented as a method of a class called `DatabaseSampler`. This description has been chosen for consistency with the sampling tool analysed in Chapter 7, which implements the Process presented in this Chapter and the algorithms described in Chapter 5.

The initialisation step is represented by a call to the `InitialiseProcess()` method. As outlined in Section 4.2, a suitable representation of the set of integrity constraints being considered must be used to guide the sampling process (see Fig. 4.2). Initialising the process refers to setting up this representation. Examples of such representation include an Insertions Chain Graph (see Section 5.5) and an Agreements Table (Section 5.6). Initialiation also involves creating an empty Sample Database with the appropriate schema; in general, the same as the Source Database.

The next step of the Process involves a condition that determines when to stop sampling, and it is represented by a call to method `StopSampling()`. According to the previous Section, sampling can stop when the current Sample *faithfully* represents its Source Database. Using the examples given in Section 4.2, this condition will be satisfied when the Sample Database

satisfies the required set of integrity constraints, or a predefined Sample size has been reached. These two criteria could, in fact, be combined leading to more realistic representativeness criteria according to which a Sample is representative of its Source Database only when it is consistent and also satisfies the specified size requirements. Section 5.5 applies this combined criterion.

Sampling from a database is about selecting the appropriate set of instances according to the given criteria. In general, two different types of selections must be performed. The first one is needed to initiate the process itself. This selection may not be directly related to the consistency criteria used to evaluate the resulting Sample Database. That is, the chain of insertions, as referred to in Section 4.2, that occur in consistent sampling must be initiated by some insertion which is not related to previous insertions. The second type of selections is needed to keep the Sample Database consistent with the last inserted instance.

These two types of selections are represented in Fig. 4.3. The first one, which does not depend on previous selections, is represented by a call to `SelectInstance()`. The criteria used to make this initial selection would depend on the database sampling application at hand. Random selection would be one possible strategy to start up the consistent sampling process. Another one arises from the fact that a common requirement in database sampling is to extract a *small*[1] Sample satisfying the predefined criteria (refer, for example, to Section 3.3 for a review of the database prototyping method described in [45]). Consequently, the initial insertion must be peformed in such a way that, when kept consistent, it leads to a small Sample. In this case, initial selections, although indirectly, can also be related to the consistency criteria that guides the sampling process. Chapter 5 will provide specific details on how initial selections can be performed, considering, for example, functional dependencies, referential integrity constraints and cardinality constraints.

Any selected instance must be inserted into the Sample Database, which is represented by a call to `InsertInstance()` in Fig. 4.3. This step has been included only to indicate that an appropriate interface to the Sample Database is required.

[1] The definition of *smallness* is domain dependent. In general, the required minimum size would be known at the beginning of the sampling process. Achieving consistency may require additional insertions after the minimum Sample size has been reached.

The rest of the required selections, which depend on previous selection and the consistency criteria, are represented by method `UpdateProcess()`. As described above, these selections are performed using the information that, based on the set of integrity constraints, guides the sampling process. This information must be updated after each insertion as this will determine which other selections are required. `UpdateProcess()` is also responsible for updating this information.

Method `Synchronise()` is central to the Sampling Protocol underlying the Sampling Process and is described in detail in Chapter 7. This Protocol allows for different sampling criteria to be satisfied simultaneously.

4.4 Dealing with Integrity Constraints

The previous sections of this Chapter identified the problems that must be solved when consistently sampling from a database. The most significant issue regards how the set of integrity constraints under consideration is used to identify the database instances that, at any given moment during sampling, are required in order to maintain consistency in the Sample Database under construction.

Integrity constraints can be classified based on the type of information that is needed to identify the next instance that must be inserted into the Sample. In that respect, integrity constraints are either *locally satisfiable*, if only the last selected instance needs to be considered, or *global*, when all previous selections must be considered. This classification is relevant to sampling due to the fact that enforcing locally satisfiable constraints is expected to have lower complexity than enforcing global constraints. Section 5.5 investigates the use of locally satisfiable constraints during sampling and Section 5.6 considers global integrity constraints.

4.5 Summary

This Chapter has investigated the process of consistently sampling from a database.

The discussion has been kept as abstract as possible to make the process presented here applicable to different circumstances, such as including disparate data models and consistency

criteria used to evaluate the resulting Sample. Firstly, it has identified the most relevant problem that must be solved when a consistent Sample is to be extracted from a database, namely the chain of insertions triggered by the consistency criteria. Then the sampling process underlying this work has been detailed, identifying the issues that are common to any sampling activity, independent of which consistency criteria is being used. Finally integrity constraints have been classified, for sampling purposes, into two categories: those that require the sampling process to maintain global information in order to enforce them in the Sample Database and those that can be enforced using local information only.

Chapter 5

Database Sampling - Applying the Process

5.1 Introduction

This Chapter specialises the abstract description of the database sampling process given in Chapter 4 to concrete types of integrity constraints. The description will focus on the relational data model [24] and will address the problem of consistent database sampling considering inclusion dependencies, cardinality constraints, functional dependencies, generalisation dependencies and subset dependencies to evaluate the consistency of the resulting Sample.

The next Section introduces the School Reference Database, used throughout the thesis when concrete examples are required. Section 5.3 describes the terminology and notation to be used in the context of relational databases. They are all based on standard definitions, e.g. [72, 1]. This same terminology will also be used in Chapter 6. Then Section 5.4 discusses the need for a representation of the set of integrity constraints to be satisfied by the resulting Sample Database which can be used to guide the sampling process. Sections 5.5 and 5.6 describe two such representations, termed *Insertions Chain Graph* and *Agreements Table* respectively. The first one can be used to guide the sampling according to a range of integrity constraints types commonly used in practice. The second representation has been designed to guide the sampling process when the consistency criteria is the satisfaction of a set of

functional dependencies. Random sampling in the context of consistent database sampling is analysed in Section 5.7, as this is the only sampling strategy found in the literature and therefore is used as the reference strategy to which the work reported in this thesis must be compared. Finally Section 5.8 analyses the different sampling algorithms presented here.

5.2 School Reference Database

The School Reference Database will be used in the reminder of this thesis when concrete examples are needed to illustrate the concepts being developed. This database was taken from [31], and extended to include some integrity constraints not present in its initial form. Fig. 5.1 shows the Extended Entity-Relationship Diagram (EER) for the reference database, following the graphical conventions used in [10]. Boxes denote *entities* while diamonds represent *relationships* between entities. *Attributes* are denoted with bullets, and may be associated with both entities and relationships; particularly, black bullets represent primary key attributes for a given entity. *Cardinality constraints* are denoted using pairs of integers $(\min, \max)$, using the *look-across* semantics, that is, an instance of the entity associated with this pair of integers may participate in the relationship in a minimum of *min* times and a maximum of *max*. For example, relationship `Teaching` in Fig. 5.1 states that a `Course` can be taught by one and only one `Teacher`, while a `Teacher` must teach at least one course, but no more than three. The relationship between `Student` and `FullTimeStudent` is denoted with an arrow to represent that full time students are a *subset* of students. In this case, assume that a Student is considered a full time student if it has taken at least five exams. Only full time students are assigned a tutor, but all students take exams. Finally, a *generalisation hierarchy* is represented in this Figure using an arrow with several starting entities. In Fig. 5.1 this is used to describe that `Person` generalises both `Student` and `Teacher`.

Fig. 5.2 shows a possible mapping of the semantic model of Fig. 5.1 into a normalised relational schema. Refer to [10] for details of such mapping. This Figure shows the set of tables and the corresponding attributes for each table. The semantics of each attribute should be clear from Fig. 5.1. Primary key attributes are underlined in Fig. 5.2.

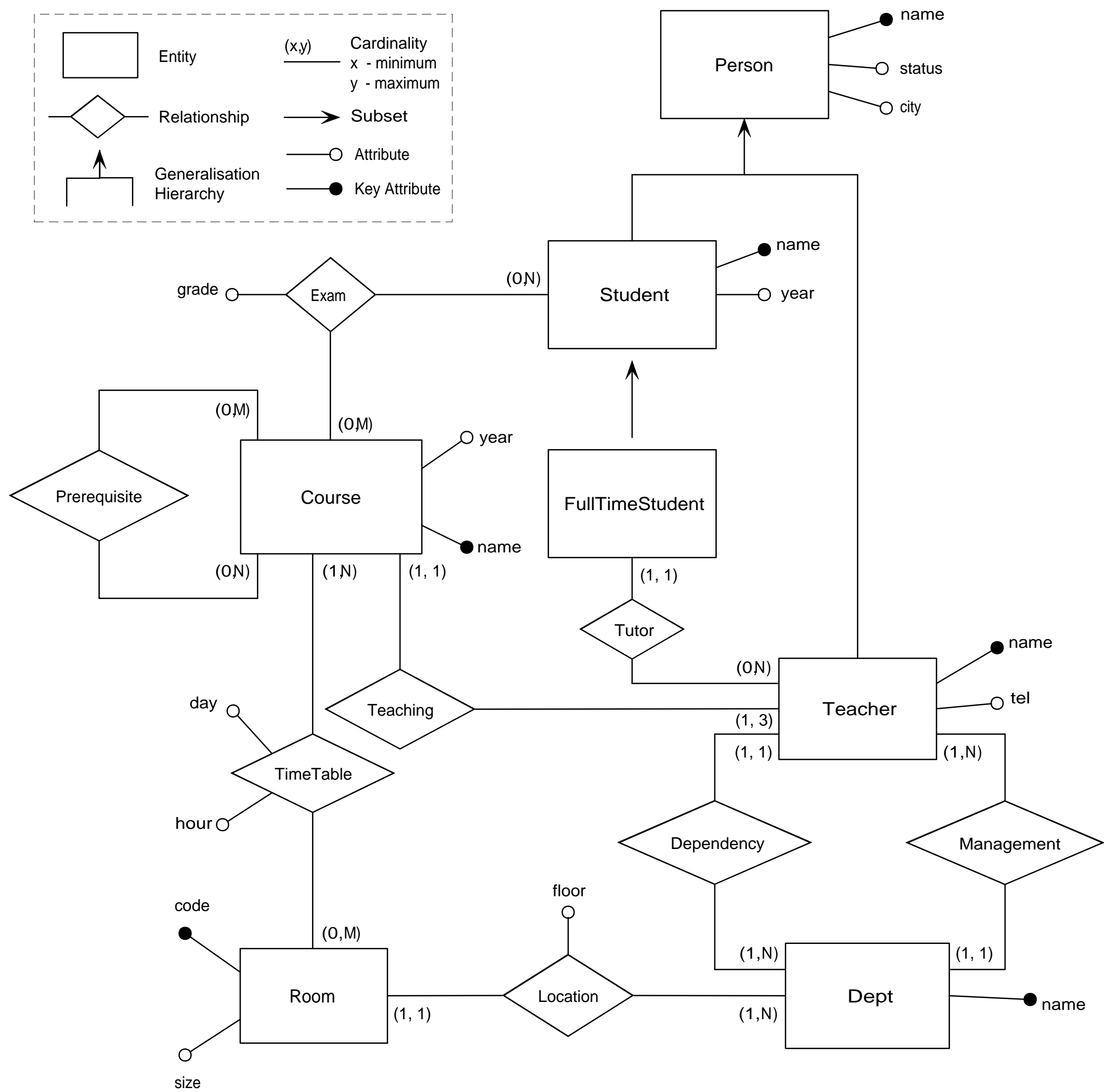

Fig. 5.1: School Reference Database EER Diagram

Students(name, year)
Courses(course, year, teacher)
Teachers(name, tel, dept)
Depts(name, director)
Rooms(room, size, dept, floor)
Exams(student, course, grade)
Prerequisits(course, previous)
Timetable(course, day, hour, room)
FullTimeStudents(student, tutor)
Persons(name, status, city)

Fig. 5.2: Normalised Relational Database Schema for the Reference Database of Fig. 5.1

5.3 Definitions and Notation

Assume there is a set of *attributes* $\mathcal{U}$, each with an associated *domain*. The domain for attribute A is denoted by $\mathcal{DOM}_A$. $\mathcal{DOM}$ denotes the union of all domains $\mathcal{DOM} = \bigcup_{A \in \mathcal{U}} DOM_A$.

A *relation schema* R over $\mathcal{U}$ is a subset of $\mathcal{U}$. A *database schema* $D = \{R_1, \ldots, R_n\}$ is a set of relation schemas. A *relation* r over the relation schema R is a set of R-tuples, where a *R-tuple* is a mapping from the attributes of R to their domains. R-tuples may also be referred as simply *tuples*. A *database instance* is a set of database relations. Only finite database instances are considered here.

Attributes in $\mathcal{U}$ are denoted with uppercase letters $A, B, C, \ldots$. R-tuples are denoted by lowercase letters, possibly with subscripts, $t_1, t_2, \ldots$. The relation r to which a R-tuple belongs will always be clear from the context. The value of an attribute, $A \in \mathcal{U}$, for a particular R-tuple, t_1, is denoted $t_1(A)$. This notation is extended to sets of attributes so that for $Y \subseteq \mathcal{U}$, $t_1(Y)$ denotes a tuple, u_1, over Y with $u_1(A) = t_1(A)$ for all attributes $A \in Y$.

An *integrity constraint* is an assertion about a database instance. A database instance I *satisfies* an integrity constraint σ, denoted $I \models \sigma$, if the assertion is true for this instance [29]. Here, integrity constraints are assumed to be Relational Calculus expressions [64, 1, 25], that is, first-order-logic formulae without functions and without free variables. Integrity constraints may also be referred to as simply *constraints*.

Section 5.6 will focus on a particular type of integrity constraint, referred to as *functional dependency*. Several concepts related to that of functional dependency will be required. For simplicity, they are all given together in this Section.

Definition 5.3.1. If $\mathcal{U}$ is a set of attributes, then a *Functional Dependency* over $\mathcal{U}$ is an expression of the form $Y \rightarrow Z$, where $Y, Z \subseteq \mathcal{U}$. A relation r over $\mathcal{U}$ *satisfies* $Y \rightarrow Z$, denoted $r \models Y \rightarrow Z$, if for each pair of tuples in r, t_1 and t_2, $t_1(Y) = t_2(Y)$ implies $t_1(Z) = t_2(Z)$. If Σ is a set of functional dependencies, then $r \models \Sigma$ means that all dependencies in Σ are satisfied in r.

Individual functional dependencies are denoted with lowercase Greek letters, possibly with subscripts, $\sigma, \gamma_1, \gamma_2 \dots$. Sets of functional dependencies are denoted using uppercase Greek letters, possibly with subscripts, $\Sigma, \Gamma_1, \Gamma_2, \dots$.

Definition 5.3.2. Let Σ and Γ be sets of functional dependencies over U. Then Σ *implies* Γ, denoted $\Sigma \models \Gamma$, iff for all relations r over $\mathcal{U}$, $r \models \Sigma$ implies $r \models \Gamma$.

Definition 5.3.3. Let Σ be a set of functional dependencies over U. The *closure of* Σ, denoted Σ^*, is defined as $\Sigma^* = \{Y \rightarrow Z | YZ \subseteq \mathcal{U} \text{ and } \Sigma \models Y \rightarrow Z\}$.

Example 5.3.1. Consider the database relation termed `Courses` defined over attributes $\mathcal{U} =$ {Course, Year, Teacher}, which stores information about the Courses taught in the School reference database. Attribute `Course` is the primary key for this relation, therefore the set of functional dependencies that must hold in this relation is $\Sigma = \{\text{Course} \rightarrow \text{Year},\ \text{Course} \rightarrow \text{Teacher}\}$. Assume that no other functional dependency must hold. According to Definition 5.3.3, it can bee seen that

$$\Sigma^* = \big\{\text{Course} \rightarrow \text{Year},\ \text{Course} \rightarrow \text{Teacher},\ \text{Course} \rightarrow \{\text{Year}, \text{Teacher}\},$$
$$\{\text{Course}, \text{Teacher}\} \rightarrow \text{Year},\ \{\text{Course},\ \text{Year}\} \rightarrow \text{Teacher}\big\}$$

Σ^* represents the set of all functional dependencies that are logical consequences of Σ. This concept allows for the definition of what are called *Armstrong Relations*, a concept initially introduced in [8], although the term itself was coined by Fagin in [26].

Definition 5.3.4. A database relation r is an *Armstrong relation for a set of functional dependencies* Σ iff r *satisfies* all functional dependencies in Σ^* and no other functional dependency. It is *minimal* if every Armstrong Relation for Σ has at least as many tuples as r.

Armstrong relations are useful in the context of database design as a set of functional dependencies and an Armstrong relation for them represent the same information. Section 5.6 exploits this property of Armstrong relations in the context of database sampling developing a method for sampling with functional dependencies. This method relies on the closure of a set of attributes under a set of functional dependencies, called *fd-closure*, which is defined next [11, 1].

Definition 5.3.5. Given a set Σ of functional depdencies over $\mathcal{U}$ and an attribute set $\mathcal{X} \subseteq \mathcal{U}$, the *fd-closure of* $\mathcal{X}$ *under* Σ, denoted $(\mathcal{X}, \Sigma)^{*,\mathcal{U}}$ or simply $\mathcal{X}^*$ if Σ and $\mathcal{U}$ are understood from the context, is the set $\mathcal{X}^* = \{A \in \mathcal{X} \mid \Sigma \models \mathcal{X} \to A\}$.

Example 5.3.2. Consider the same relation and set of functional dependencies as in Example 5.3.1. It can be seen how, for example,

$$\{\text{Course}\}^* = \{\text{Course, Teacher, Year}\},$$
$$\{\text{Teacher}\}^* = \{\text{Teacher}\}, \qquad \{\text{Year}\}^* = \{\text{Year}\},$$
$$\{\text{Course, Teacher}\}^* = \{\text{Course, Teacher, Year}\}$$

Intuitively, $\mathcal{X}^*$ represents the set of all attributes that are determined by $\mathcal{X}$ according to Σ. Some sets of attributes, referred to as *saturated* [8] (or *closed* sets in [11]), do not determine any other attributes but themselves.

Definition 5.3.6. A set of attributes $\mathcal{X}$ over $\mathcal{U}$ is *saturated* with respect to a set of functional dependencies Σ over $\mathcal{U}$ iff $\mathcal{X}^* = \mathcal{X}$.

Example 5.3.3. Consider again the same relation and set of functional dependencies as in Example 5.3.1. Following from Example 5.3.2, the set of saturated set is

$$\big\{\{\text{Teacher}\}, \{\text{Year}\}, \{\text{Teacher, Year}\}, \emptyset\big\} \quad .$$

Following the terminology of [11], it is said that two tuples *agree exactly* on a set of attributes as follows.

Definition 5.3.7. Let $\mathcal{X}$ be a set of attributes, $\mathcal{X} \subset \mathcal{U}$. A pair of tuples t_1 and t_2 over $\mathcal{U}$ *agree exactly* on $\mathcal{X}$ iff $\forall A \in \mathcal{X}, t_1(A) = t_2(A)$ and $\forall B \notin \mathcal{X}, t_1(B) \neq t_2(B)$.

Example 5.3.4. Consider tuples $t_1 = \langle$analysis 1, 1, phillip$\rangle$ and $t_2 = \langle$geometry, 1, anca$\rangle$, defined over $\mathcal{U} = \{$Course, Year, Teacher$\}$ as in Example 5.3.1. According to Definition 5.3.7, t_1 and t_2 *agree exactly* on attribute `Year`. It can be seen how t_1(Year) $= t_2$(Year) but t_1(Course) $\neq t_2$(Course) and t_1(Teacher) $\neq t_2$(Teacher), as required.

This concept is next extended to entire relations.

Definition 5.3.8. A database relation r over $\mathcal{U}$ *satisfies* $\mathcal{X}$, with $\mathcal{X} \subset \mathcal{U}$, iff there exist two different tuples in r that *agree exactly* on $\mathcal{X}$.

This last definition is central to the purposes of Section 5.6. As will be shown by Theorem 5.6.1, if relation r *satisfies* precisely all *saturated* sets then it is an Armstrong relation for the specified set of functional dependencies. Therefore, the goal in Section 5.6 is to select a set of tuples from a relation in such a way that the resulting relation satisfies all saturated sets and no other set of attributes.

5.4 Information to Guide the Sampling Process

As described in Chapter 4, consistent database sampling leads to a chain of insertions required to construct a Sample Database that satisfies the specified representativeness (i.e. consistency) criteria. The sampling process described in the previous Chapter relies on a representation of the set of integrity constraints under consideration suitable for sampling. This information is used to guide the sampling process in performing the appropriate chain of insertions. The information needed for such purposes depends on the type of integrity constraints being considered. The next two Sections describe and apply two examples of such representations.

5.5 Insertions Chain Graph (ICG)

This Section describes a graphical representation that has been designed to intuitively represent the chain of insertions required in database sampling, considering inclusion dependencies, cardinality constraints, subset dependencies, and generalisation dependencies. This representation, termed *Insertions Chain Graph* or *ICG*, was initially presented in [19] as a result of this research work. It provides a common mechanism to describe the information needed, for each type of constraint, in order to identify the set of instances that, at any given moment during sampling, must be inserted into the Sample in order to maintain consistency.

5.5.1 Motivations

As discussed in Chapter 3, most approaches to database prototyping enforce a very limited set of constraints in the resulting database. Those approaches that do address a wider set of constraints generally rely on first-order-logic (or FOL) to define the set of constraints being enforced [49]. This is indeed a powerful and well-understood formalism to express the semantics of the data, and therefore FOL-based database prototyping methods can take advantage of FOL inference mechanism to ensure the consistency of the resulting database. However, the limitations of FOL-based approaches outlined in Section 3.3 include:

- Formalising a large set of constraints in FOL results in an obscure set of formulae, where missing and undesired (wrong or redundant) constraints may be difficult to identify.
- This set of formulae must be proved consistent. It is however undecidable to prove that a set of FOL formulae is consistent [23].
- Inference in FOL has been proved not to scale in the context of prototype database population.

An Insertions Chain Graph was designed as an alternative representation for integrity constraints that overcome these problems:

- Its graphical nature makes the set of constraints being represented explicit and understandable.

- Its semantics cannot define an inconsistent set of integrity constraints (see Section 5.5.2).
- It is specially designed for database sampling purposes, therefore identifying the instances that must be inserted in order to maintain consistency is expected to be more efficient than using FOL.

An Insertions Chain Graph describes a common mechanism to represent a set of widely used integrity constraints that define semantic relationships between entities (e.g. *relation* or *table* in relational terms) in a database, as opposed to instance-level (e.g. *tuple*) relationships which will be addressed in Section 5.6. Formal and graphical definitions of an Insertions Chain Graph are given next. Section 5.5.3 describes an example of how an ICG can be used to describe a set of integrity constraints.

5.5.2 Definition of Insertions Chain Graph

An Insertions Chain Graph (or ICG) 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
- **T** is the set of arrow types that describe the type of consequence of inserting an instance into the Sample, as described in Table 5.1. It can be seen how the so-called *Partial* arrows subsume all other arrow types. However, the other arrow types are kept because they simplify the resulting ICG when used in its graphical form, which is particularly useful for sampling purposes.
- δ is called here *Insertions Function*. For each entity in the database, this function describes the set of arrows in the ICG that start at this entity. It is formally defined by Equation 5.1

$$\delta : Q \rightarrow 2^{I \times 2^{Q} \times T \times C} \tag{5.1}$$

Table 5.1: Arrow Types in an Insertions Chain Graph

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	Inclusion Dependencies
Quantified	Number of additional instances	The specified number of referred instances must be present in the target entity after inserting the referring instance into the source entity	Cardinality Constraints
OR	None	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

Thus each arrow in an ICG can be described by a quadruple $\langle \mathbf{i}, \mathbf{P}, \mathbf{t}, \mathbf{c} \rangle$ where:

- $\mathbf{i}$ is the arrow's identifier, $\mathbf{i} \in \mathbf{I}$
- $\mathbf{P}$ is the set of target entities for this arrow, $\mathbf{P} \in 2^{Q}$. It is always a singleton, except for *OR* arrows, as described in Table 5.1.
- $\mathbf{t}$ is the arrow's type, $\mathbf{t} \in \mathbf{T}$
- $\mathbf{c}$ represents the additional information required for some arrow types, as detailed in Table 5.1, with $\mathbf{c} \in \mathrm{C} = \mathbb{N} \bigcup \mathbf{P}$redicate. It must be interpreted in different ways depending on which arrow type it is associated with (see Section 5.5.3 for examples):
  1. *Quantified* arrows: $\mathrm{c} \in \mathbb{N}$
  2. *Partial* arrows: $\mathrm{c} \in \mathbf{P}$redicate must express a condition to be satisfied by the Sample Database before a new insertion into the target entity is required. Using the School Database, and thus the relational model, the language used to express this predicate is assumed to be a boolean expression which may

contain a well-formed SQL statement. Refer to [22] for a possible syntax of such a predicate.

3. *Total* arrows: c has no meaning in this case as no additional information is required
4. OR arrows: c has no meaning in this case either as no additional information is required. This arrow type may include several target entities, and this list of entities is the only information needed to enforce its semantics.

The rationale behind the four arrow types that can be defined in an ICG is that they are able to express widely used semantics of a Source Database. The constraint type each arrow type expresses is shown in Table 5.1 in column Constraint Type. The next Section illustrates all above definitions in the context of a database used in this thesis as the reference database when concrete examples are required.

It must be noted that, in case of the relational model used in this example, the *Insertions Function* (δ) must identify the attributes that are used to link the source and target entities (i.e. *tables*). This is not explicitly identified in the above definition of ICG because this is considered specific to the relational model, and the concept of an ICG has been kept as abstract as possible. If it was applied to a different data model (e.g. semi-structured, object oriented), the mechanism used to identify the link between entities is likely to differ from that used in the relational model. Appendix A provides a more complete example, including the identification of the attributes used to link between tables, as done in the sampling tool described in Chapter 7.

5.5.3 Example of Insertions Chain Graph

Fig. 5.3 represents an ICG that includes all semantics of the School Reference Database described in Section 5.2. Each node of this graph represents one table of the logical design for this database, shown in Fig. 5.2, and its edges represent database semantics relevant to the sampling process. It should be noted how some constraints, like for example maximum cardinality constraints, have not been represented as their absence cannot lead to an inconsistent Sample Database. That is, in the School reference database, each `Teacher` can teach

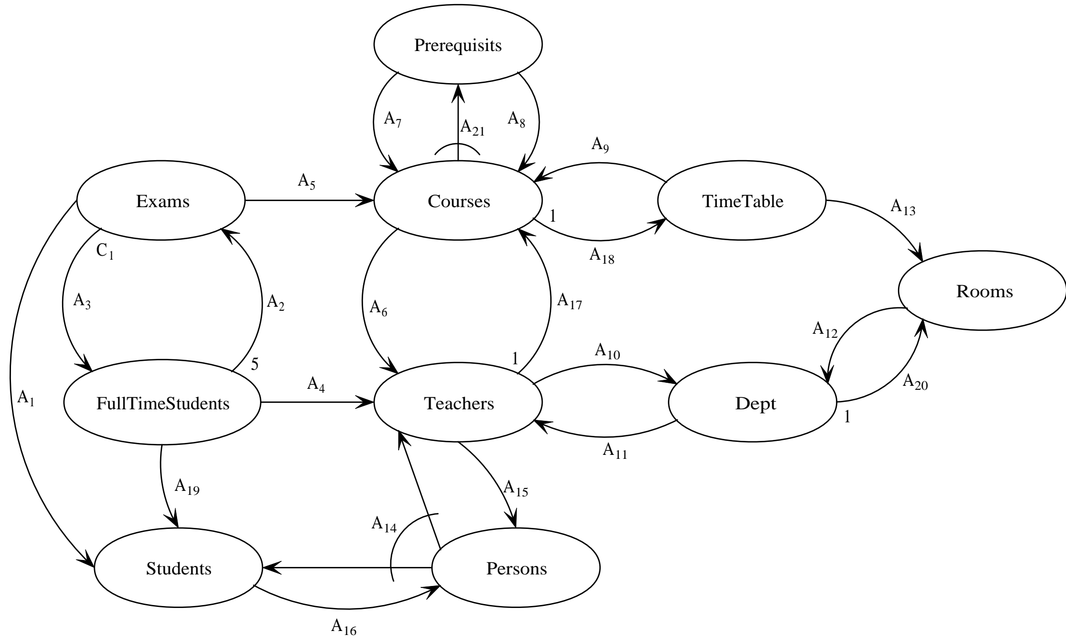

Fig. 5.3: Insertions Chain Graph for the School Reference Database of Fig. 5.1

a maximum of three `Courses`. If the Source Database satisfies this constraint (i.e. it is a consistent database instance), it is not possible that a Sample of it will contain information about a `Teacher` who is teaching more than three `Courses`. Therefore, this constraint does not need to be taken into account during sampling. Section 6.4 will study this property of integrity constraints, referred to as *Sampling-Irrelevant*, in more detail.

In Fig. 5.3, arrows that have no associated information besides their name are *Total* arrows. Arrow A_1 is an example of such type of arrow which, during sampling, will be used to ensure that whenever a tuple from table `Exams` is sampled an associated tuple from table `Students` will also be sampled. This will keep the Sample Database consistent with one of the constraints specified in Fig. 5.1 which states that any `Exam` must be taken by one and only one `Student`. Arrow A_2 is an example of *Quantified* arrow, as it has an integer number (i.e. 5) associated with it. In this case, this arrow represents the fact that, as specified in Section 5.2, any full time student in the Sample must have sat at least five examinations. This same constraint requires the addition of *Partial* arrow A_3, so that if table `Exams` records more than five exams for the same student then this student must be recorded as being a full

Table 5.2: Conditions Table for the ICG Example of Fig. 5.3

Condition Name	**Definition**
C_1	(**Select** count(*) **From** Exams **Where** name = x) $\geq$ 5

time student (and so s/he will additionally require a tutor, hence arrow A_4 is also included in the graph). As defined in the previous Section, any *Partial* arrow requires a condition to determine when an insertion is required. Condition C_1 associated with arrow A_3 is given in Table 5.2. Being 'x' the name of the student (primary key of `Students`) this condition queries the Sample Database to count how many `Exams` a given Student has taken. If s/he has sat more than five examinations s/he is a full time student. As a last example, consider *OR* arrow A_{14}, which is needed to enforce a generalisation dependency present in the reference database, with `Persons` generalising both `Students` and `Teachers`. *Total* arrows A_{15} and A_{16} enforce the same constraint but in the opposite direction, that is, if a `Teacher` (or `Student`) is sampled, then the Sample Database must record that s/he is also a `Person`.

Following this same line of reasoning, the entire ICG for the reference database could be developed, resulting in that shown in Fig. 5.3. It should be noted that such a graph can in fact be automatically created from the corresponding EER diagram. Briefly, 1 to N relationships are mapped into *Total* arrows, like for example the relationship between `FullTimeStudents` and `Teachers` in the reference database. Generalisation constraints (e.g. both `Students` and `Teachers` are `Persons`) become *OR* arrows. Cardinality constraints (e.g. a `Teachers` must teach at least one `Course`) map into *Quantified* arrows. Only the transformation of subset relationships (e.g. `FullTimeStudents` are a subset of `Students`) would require additional information, as its semantics cannot always be fully described in an EER diagram.

Instead of using a graphical representation, an ICG can also be described using its formal definition, in terms of a quadruple ICG = (Q, I, T, δ) as done in the previous Section. In the particular ICG example being developed, the formal representation is as follows.

Q = {Courses, Exams, FullTimeStudents, Teachers, Persons, Students, TimeTable, Prerequisits, Dept, Rooms}

Table 5.3: Insertions Function δ for the Insertions Chain Graph of Fig. 5.3

Source Entity	**Quadruples ⟨i, P, t, c⟩**			
	Arrow Name (i)	**Target Entity Names (P)**	**Arrow Type (t)**	**Additional Info (c)**
Exams	A_1	{Students}	Total	
FullTimeStudents	A_2	{Exams}	Quantified	5
Exams	A_3	{FullTimeStudents}	Partial	C_1 in Table 5.2
FullTimeStudents	A_4	{Teachers}	Total	
Exams	A_5	{Courses}	Total	
Courses	A_6	{Teachers}	Total	
Prerequisits	A_7	{Courses}	Total	
Prerequisits	A_8	{Courses}	Total	
TimeTable	A_9	{Courses}	Total	
Teachers	A_{10}	{Depts}	Total	
Depts	A_{11}	{Teachers}	Total	
Rooms	A_{12}	{Depts}	Total	
TimeTable	A_{13}	{Rooms}	Total	
Persons	A_{14}	{Students, Teachers}	OR	
Teachers	A_{15}	{Persons}	Total	
Students	A_{16}	{Persons}	Total	
Teachers	A_{17}	{Courses}	Quantified	1
Courses	A_{18}	{TimeTable}	Quantified	1
FullTimeStudents	A_{19}	{Students}	Total	
Depts	A_{20}	{Rooms}	Quantified	1
Courses	A_{21}	{Prerequisits}	OR	

$\mathbf{I} = \{A_1, A_2, \ldots, A_{21}\}$

$\mathbf{T} = \{\text{Total, Quantified, OR, Partial}\}$

δ : the Insertions Function for this example is shown in Table 5.3.

5.5.4 Consistent Database Sampling using an ICG

This Section describes a method for consistently sampling a database based on the use of an ICG to guide the process. 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, that is, the minimum Sample size. This information provides a condition to stop sampling. An externally specified Sample size is generally considered a plausible assumption in the random sampling literature [51, 65] since the cost of processing the sample (e.g. data mining, supporting software development) can be expected to be much higher than that of extracting the Sample itself.

3. The number of different entities that each entity points to and is pointed to by in the ICG, referred to as Fan-out and Fan-in values respectively (see `SelectEntity()` below). This information can be extracted from the ICG itself.

Fig. 5.4 outlines the algorithm for extracting a Sample Database consistent with the set of integrity constraints represented in an Insertions Chain Graph introduced as input parameter. A brief description of the supporting functions used in this algorithm follows.

- `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 will lead to a small consistent Sample Database, as the same instance can be used to force insertions in several entities, instead of requiring one new instance for each entity.

- `SampleInstance()`: randomly samples an instance from the input entity. 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* [66], where instances frequently accessed when the database is in production will be more likely to become part of the Sample.

- `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.`δ`(tEntity)`$|_P$: `ICG.`$\delta()$ is the Insertions Function (e.g. see Table 5.3), and $|_P$ denotes the restriction of the resulting quadruple to its second element, P, the set of

```
// Main Function to extract a sample from database D according to the input ICG
Function ConsistentSampleICG(input ICG : InsertionsChainGraph,
                             input D : Database)
                             input Sample : Database) output bool {
/*
ICG.NotFullEntities represents the set of entity names which still have not reached
  the minimum number of instances in the Sample Database.
entity and instance refer to a table and a tuple in this table in the
  Source Database.
*/
while (ICG.NotFullEntities ≠ ∅) do
  entity = SelectEntity(ICG); // Based on Fan-in and Fan-out values
  instance = SampleInstance (D, entity); // Randomly
  InsertInstance(Sample, entity, instance, ICG.NotFullEntities);
    // It also modifies ICG.NotFullEntities
  TargetEntities = ICG.δ(entity)|P;
    // Set of entities to be considered as a consequence of the current insertion
  for each targetEntity ∈ TargetEntities do
    newInstance := SampleInstance_local (D, targetEntity, instance );
    if (newInstance = ⟨⟩) then return false; // Inconsistent Source Database
    if NOT (KeepConsistentICG (ICG, Sample, targetEntity, newInstance, D))
      return false;
  endfor
endwhile
return true;
}
// Recursive function used by the main function.
Function KeepConsistentICG(input/output ICG : InsertionsChainGraph,
                  input/output Sample: Database, input tEntity: DBEntity,
                  input tInstance: DBInstance, input D: Database)
                  output bool {
InsertInstance(Sample, tEntity, tInstance, NotFullEntities);
TargetEntities = ICG.δ(tEntity)|P;
  // Set of entities to be considered as a consequence of the current insertion
for each targetEntity ∈ TargetEntities do
  newInstance := SampleInstance_local (D, targetEntity, tInstance );
  if (newInstance = ⟨⟩) then return false; // Inconsistent Source Database
  if NOT (KeepConsistent (ICG, Sample, targetEntity, newInstance, D))
      return false;
endfor
return true;
}
```

Fig. 5.4: Algorithm for Consistent Database Sampling with ICG

target entities for this entity.

- `SampleInstance_local()`: selects from the Source Database an instance of `targetEntity` related to `tInstance`. It must be an instance not yet in the Sample. If several are possible it will select randomly.

The Consistent Database Sampling Process described in Chapter 4 can be identified in Fig. 5.4. `InitialiseProcess()` involves the construction of the corresponding ICG, which in this case is taken as an input to the algorithm. `SelectEntity()` together with either `SampleInstance_local()` or `SampleInstance()` implement the functionality represented by `SelectInstance()` in Chapter 4. `StopSamping()` is implemented in this case by condition (`ICG.NotFullEntities`$\neq \emptyset$). Finally `UpdateProcess()` is here represented by function `KeepConsistentICG()`.

The randomness involved in function `SampleInstance()` (and `SampleInstance_local()` if necessary) should be pointed out. Although the starting entity selected in `SelectEntity()` is based on the Fan-in and Fan-out values of the ICG, in `SampleInstance()` concrete instances are selected randomly from the Source Database. This procedure will, potentially, include all the *data-diversity* found in the operational data, that is, the resulting Sample will not be restricted to one single consistent part of the database. The requirement of including data-diversity was not addressed in Chapter 4 because it can be seen as a domain-specific instantiation of the general concept of *representativeness*. However, it is reasonable to expect that in most applications of database sampling a Sample that includes as much data-diversity as the Source Database is likely to be considered a better representation of it.

5.5.5 Expressiveness of ICG

An ICG can be seen as a language to express integrity constraints. In fact, it expresses the consequences that constraints have during sampling, rather than the constraints themselves. As for any language, its expressiveness should be assessed [2]. Since it is based on the query language used by the Source Database, its expressiveness depends on that of this language. More precisely, the ICG example given above is based on the relational model and SQL, so its expressiveness is limited by the expressiveness of SQL.

Table 5.4: Sampling Information for the Insertions Chain Graph of Fig. 5.3

Entity Name	Fan-In	Fan-Out	Sample Size
Students	3	1	1
Exams	1	3	1
FullTimeStudents	1	3	0
Persons	2	1	1
Prerequisits	1	2	1
Courses	5	3	1
Teachers	4	3	1
TimeTable	1	2	1
Rooms	2	1	1
Dept	2	2	1

In spite of the fact that some constraints cannot be expressed (e.g. functional dependencies), as is the case for any other practical language [2], ICG allows for the definition of a wide range of commonly used constraints.

5.5.6 Sampling Example

As an example of how an Insertions Chain Graph is used to sample a database, consider that the School Reference Database introduced in Section 5.2 is to be sampled. Figure 5.3 shows the corresponding ICG, and Table 5.4 the associated information required for sampling as specified above, including the fan-in and fan-out values for each entity (extracted from the ICG itself) and the minimum Sample size for each entity.

With this information the algorithm of Fig. 5.4 can be applied to extract a Sample of this database. A complete description of a sampling example would require a full definition of a consistent database instance, resulting Sample and intermediate steps. Only an overview of how the execution would proceed is given here. Appendix A provides a more detailed description of this same example.

At least on instance is required in each entity, except for `FullTimeStudents` which may have none. According to the selection criteria of `SelectEntity()`, sampling could start from either entity `Exams` or entity `FullTimeStudens` as these entities have the highest fan-out and lowest fan-in values in Table 5.4. Since no tuple is required in `FullTimeStudens`, entity `Exams` will be the starting point for this sampling example. From the Insertions Chain Graph

of Fig. 5.3, each instance sampled from entity `Exams` will require the `Student` who took this examination. Once such an student is sampled, a related tuple from `Persons` will be required. The initial insertion into `Exams` will also trigger the related instance of `Courses` to be included in the Sample, and this will, in turn, require the `Teacher` who teaches this course. The Insertions Chain Graph of Fig. 5.3 will identify all other required insertions. This chain of insertions will eventually terminate because one of the required instances will already be in the Sample.

Appendix A will provide further details on this sampling example.

5.6 Consistent Database Sampling with Functional Dependencies

An Insertions Chain Graph allows a set of widely used types of integrity constraints to be considered during Sampling. However, there is a type of integrity constraint, extensively investigated in the context of relational databases, which cannot be expressed using an Insertions Chain Graph: Functional Dependencies (see Section 5.3). The reason is that an ICG describes entity-level constraints, as opposed to instance-level constraints.

This section describes how functional dependencies can be included in the sampling process. For this purposes, as done in the previous Section, firstly an appropriate representation of the set of functional dependencies used to guide the sampling process will be described. Then, an algorithm will be presented which, using this representation, samples a database relation according to a set of functional dependencies.

All concepts and terminology used in this Section were defined in Section 5.3.

5.6.1 Motivation

The motivation to include functional dependencies in database sampling arise from the identification of the so-called Armstrong relations. Recall from Section 5.3 that an Armstrong relation for a set of functional dependencies Σ is a database relation that satisfies all functional dependencies that are logical consequences of Σ and no other functional dependency. From

this definition follows that a set of functional dependencies and an Armstrong relation for it are dual representations of the same information [8]. During the design and maintenance of a database, incorrect constraints are more easily identified from a list of dependencies, whereas missing dependencies are better revealed from an example of the relation. This justifies the use of Armstrong relations for database design and maintenance purposes.

5.6.2 Defining the Problem

This Section defines the objective when sampling a relation according to a given set of functional dependencies.

Given a database relation r over $\mathcal{U}$, and a set of functional dependencies Σ satisfied by r, the goal is to extract a sample of this relation which is a small[1] Armstrong relation for Σ.

Relation r will firstly be assumed to be an Armstrong relation for Σ. Then, the same problem will be addressed in Section 5.6.5 considering a subset Γ_1 and a superset Γ_2 of Σ, $\Gamma_1 \subsetneq \Sigma \subsetneq \Gamma_2$.

5.6.3 Agreements Table

In order to guarantee the minimal possible size for the resulting sample, the algorithm presented here relies on constructing what is called in this thesis an *Agreements Table* [15]. This table represents the interactions between tuples in the input relation r regarding the set of functional dependencies each pair of tuples, if included in the sample, invalidates (see below for an example). This information is used to guide the sampling process. The construction of the Agreements Table is based on the concept of a set of attributes being *saturated* (Definition 5.3.6). An Agreements Table gathers, for each tuple in the input database relation, which other tuples can be used with that one to *agree exactly* (Definition 5.3.7) on each saturated set. As an example, using the School Reference Database described in Section 5.2, consider a possible instance of table `Courses` shown in Table 5.5 and the set of functional dependencies

$$\Sigma = \{\text{Course} \rightarrow \text{Year},\ \text{Course} \rightarrow \text{Teacher}\}\ .$$

[1] The sense in which it is *small* will be addressed in Section 5.6.4.

Table 5.5: Instance of **Courses** (rep.)

	course	**year**	**teacher**
t_1	analysis 1	1	phillip
t_2	geometry	1	anca
t_3	algebra	1	robert
t_4	analysis 2	2	philip
t_5	programming	2	diana
t_6	analysis 3	3	phillip
t_7	data mining	3	mary
t_8	calculus	4	sofia
t_9	logic	4	tom
t_{10}	languages	5	diana
t_{11}	automata	5	mark
t_{12}	cybernetics	5	diana

In this case, as discussed in Example 5.3.3, the set of saturated sets of attributes is[2]

$$\big\{\{\text{Year}\}, \{\text{Teacher}\}, \{\text{Year Teacher}\}, \emptyset\big\} \ .$$

An Agreements Table has a column for each such set. Table 5.6 shows the Agreements Table for this example. It can be seen how, for example, tuple t_1 can use tuple t_4 (and also t_6) to *agree exactly* on saturated set {Teacher}. This follows from the fact that in the database relation shown in Table 5.5, $t_1(\text{Teacher}) = t_4(\text{Teacher})$, but $t_1(\text{Year}) \neq t_4(\text{Year})$ and $t_1(\text{Course}) \neq t_4(\text{Course})$, as required by Definition 5.3.7. The meaning of this relationship between t_1 and t_4 is that if both tuples are included in the Sample, no functional dependency with only attribute `Teacher` in the left-hand side will be satisfied. The algorithm described below exploits this fact in order to build a Sample of the input relation that violates (i.e. does not satisfy) all functional dependencies not in Σ^* (Definition 5.3.3).

The rationale behind an Agreements Table is that if a sample is selected in such a way that it *satisfies* all saturated attributes sets then all functional dependencies in Σ^*, and only those, hold in the resulting sample. This is formalised next.

[2] All saturated sets are considered here. However, it has been shown [8, 11] that it suffices to consider only the set of so-called *intersection generators* of this set. The algorithm presented below is valid in either case. By convention, $\mathcal{U}$ is the intersection of the empty collection of sets, thus it is not a generator (although it is saturated). For this reason it has not been included here. In the example used in this Section, the set of generators and the set of all saturated sets are the same except for $\mathcal{U}$.

Table 5.6: Agreements Table for the instance of Courses shown in Table 5.5

	{Year Teacher}*	Teacher*	Year*	∅*	**No. $\mathcal{X}^*$**
t_1		$\{t_4, t_6\}$	$\{t_2, t_3\}$	$\{t_5, t_7 - t_{12}\}$	3
t_2			$\{t_1, t_3\}$	$\{t_4 - t_{12}\}$	2
t_3			$\{t_1, t_2\}$	$\{t_4 - t_{12}\}$	2
t_4		$\{t_1, t_6\}$	$\{t_5\}$	$\{t_2, t_3, t_7 - t_{12}\}$	3
t_5		$\{t_{10}, t_{12}\}$	$\{t_4\}$	$\{t_1 - t_3, t_6 - t_9, t_{11}\}$	3
t_6		$\{t_1, t_4\}$	$\{t_7\}$	$\{t_1 - t_3, t_5, t_8 - t_{12}\}$	3
t_7			$\{t_6\}$	$\{t_1 - t_5, t_8 - t_{12}\}$	2
t_8			$\{t_9\}$	$\{t_1 - t_7, t_{10} - t_{12}\}$	2
t_9			$\{t_8\}$	$\{t_1 - t_7, t_{10} - t_{12}\}$	2
t_{10}	t_{12}	$\{t_5\}$	$\{t_{11}\}$	$\{t_1 - t_4, t_6 - t_9\}$	4
t_{11}			$\{t_{10}, t_{12}\}$	$\{t_1 - t_9\}$	2
t_{12}	t_{10}	$\{t_5\}$	$\{t_{11}\}$	$\{t_1 - t_4, t_6 - t_9\}$	4

Theorem 5.6.1. *Let r be a database relation and Σ a set of functional dependencies, both over $\mathcal{U}$. If relation r satisfies precisely all sets of attributes $\mathcal{X}$ with $\mathcal{X}^* = \mathcal{X}$, then r is an Armstrong relation for Σ.*

Proof. [3] Firstly, to prove that all functional dependencies in Σ^* hold in r, let σ be a functional dependency $\sigma \in \Sigma^*$. Without loss of generality, assume that σ is of the form $X \to A$, with $X \subset \mathcal{U}$ and $A \in \mathcal{U}$. Note that X^* is saturated. By hypothesis any two distinct tuples must agree exactly on a saturated set. If t_1 and t_2 agree exactly on X, they must agree exactly on X^*. Since $A \in X^*$, t_1 and t_2 agree also on A. It follows that $r \models \sigma$, as required.

To prove that only the dependencies in Σ^* hold in r, take a functional dependency $X \to A \notin \Sigma^*$. This implies $A \notin X^*$. By hypothesis there is a pair of tuples, t_1 and t_2, that agree exactly on X^*, therefore $t_1(X) = t_2(X)$ and $t_1(A) \neq t_2(A)$. So, by definition $r \not\models X \to A$. This proves that r is an Armstrong relation for Σ. □

In addition to one column for each saturated set, another column, termed '`No.` $\mathcal{X}^*$' (Number $\mathcal{X}^*$), is shown in the Agreements Table of Table 5.6. It represents the number of different saturated sets $\mathcal{X}$ that could be *satisfied* by selecting this tuple (i.e. number of non-empty columns in each row). This column is used to select between several possible tuples that

[3] This proof has been included here for completeness only. Refer to [11, p. 37] or [8, p. 582] for alternative proofs.

could be inserted into the sample, in order for it to be of minimal size. Selecting tuples with the highest value for '`No.` $\mathcal{X}^*$' leads to the minimum possible additional tuples required to *satisfy* all saturated sets. This heuristic has the same rationale as that in Section 5.5 where entities with higher fan-out values in an ICG were chosen first.

5.6.4 Consistent Database Sampling using an Agreements Table

The above discussion has shown how, using an Agreements Table, the resulting sample: (1) is an Armstrong relation, and (2) is expected to be small. This solves the problem as stated in Section 5.6.2. The algorithm presented in this Section extracts a Sample of *minimal* size. It is only minimal according to the information available in the Agreements Table. Tuples are selected according to their values for column '`No.` $\mathcal{X}^*$' in this table. These values represent the number of saturated sets to which each particular tuple is *directly* related (in the sense of being able to satisfy them), as opposed to considering all saturated sets to which each tuple is *transitively* related. For example, in Table 5.6, the value of '`No.` $\mathcal{X}^*$' for tuple t_1 is 3 because this tuple can be used to satisfy 3 different saturated sets: t_1 can be combined with either t_4 or t_6 to satisfy {Teacher}*, with t_2 or t_3 to satisfy {Year}*, and with t_5 or any tuple from t_7 to t_12 to satisfy $\emptyset^*$. However, if t_{12} is selected, it can also be used to satisfy {Year, Teacher}* by selecting t_{10}. Therefore, from t_1 all 4 saturated sets could actually be satisfied if one considers the sets that can be satisfied using t_1 and any of the tuples related to it (e.g. in the above example t_4 or t_6, t_2 or t_3, and t_5 or any tuple from t_7 to t_12). These two alternatives, namely considering only tuples directly related or considering also those transitively related to each tuple, can be identified with selecting tuples using either local information or global information, respectively. The algorithm that follows would not change if the second alternative was followed; however, the (time) complexity of building an Agreements Table would be significantly higher.

Fig 5.5 shows an algorithm that implements the sampling process described above. The initial call for this recursive algorithm should be

`MinimalSizeSample(`r`, `Σ`, Sample)`

where r is the relation being sampled, Σ is the set of functional dependencies being considered,

```
/* Main function to extract an Armstrong relation for Σ from relation r */
Function MinimalSizeSample(input r : relation, Σ : Set of fd,
                           output Sample : relation) output bool
/*
AgrT.SetofX is the family of sets 𝒳 with 𝒳* = 𝒳 still to be satisfied.
t_j is a single tuple from the input relation.
 */
  ConstructAgreementsTable(r, Σ, AgrT);
  while (AgrT.SetOfX ≠ ∅) // Satisfy all saturated sets 𝒳
    t_j := SelectTuple(AgrT); // Global selection criterion
    if (t_j = ⟨⟩) then /* Consistent Sampling not possible:
                       no Armstrong relation */
      return false;
    InsertTuple(Sample, t_j); // Add to the Sample
    UpdateAgreementsTable(AgrT, Sample, t_j);
      // t_j is not to be considered again
      // Some new sets 𝒳 are now satisfied
    MinimalSizeSamplerec(AgrT, Sample, t_j);
  end while
  return true;
end Function

/* Recursive procedure used by the main function. */
Procedure MinimalSizeSamplerec(input/output AgrT : AgreementsTable,
                               input/output Sample : relation,
                               input LastTuple : tuple)
/*
LastTuple is the last tuple inserted in the sample before making this
    recursive call.
t_i is a single tuple from the input relation.
 */
  while (AgrT.SetofX ≠ ∅) do // Satisfy all saturated sets 𝒳
    t_i:= SelectTuple_local(AgrT, LastTuple); // Local selection criterion
    if(t_i=⟨⟩) then return; // No tuple selected
    InsertTuple(Sample, t_i); // Add to the Sample
    UpdateAgreementsTable(AgrT, Sample, t_i);
      // t_i is not to be considered again
      // Some new sets 𝒳 are now satisfied
    MinimalSizeSamplerec(AgrT, Sample, t_i); // Keep consistency
  end while
end Procedure
```

Fig. 5.5: Algorithm for Consistent Database Sampling with Functional Dependencies

and `Sample` contains the resulting Sample relation. In addition to a Sample of relation r, this algorithm also returns whether sampling was successfully performed. The situation in which it may not be possible to extract an Armstrong relation for Σ from the input relation r is analysed in Section 5.6.5.

The algorithm of Fig. 5.5 initially selects a tuple, t_j, with the highest value for column '`No.` $\mathcal{X}$' according to the Agreements Table (if several tuples have the same value, it can select randomly between them). This tuple is then included in the sample. After that a recursive procedure, `MinimalSizeSamplerec`, is used to select tuples which, taken together with the tuple inserted last, *agree exactly* on as many saturated sets as possible. Using this last inserted tuple, referred to as `LastTuple`, function `SelectTuple_local` returns another tuple, t_i, that agrees exactly with `LastTuple` on one (or more) saturated set, $\mathcal{X}$. The information stored in the Agreements Table is used to select such a tuple. This information is updated after each insertion in order to record that tuple t_i and the newly satisfied set (or sets) $\mathcal{X}$ are not to be considered in future selections. Then `MinimalSizeSamplerec` is called again using the newly inserted tuple.

The chain of recursive calls terminates when either all saturated sets have been satisfied, or no more sets can be satisfied using tuple t_j. In the former case the algorithm will terminate and the current sample is an Armstrong relation. In the latter, a new tuple t_j will be selected and the process described above will be repeated. If no such tuple can be found, it means that there are some saturated sets that cannot be satisfied, and therefore an Armstrong relation cannot be extracted (see Section 5.6.5).

As in Section 5.5, the Consistent Database Sampling Process defined in Chapter 4 can also be identified in the algorithm of Fig. 5.5. `ConstructAgreementsTable()` is equivalent to `InitialiseProces()`. The semantics described by `StopSamping()` are implemented by condition (`AgrT.SetOfX` $\neq \emptyset$). `SelectTuple()` and `SelectTuple_Local()` implement what in Chapter 4 was referred to as `SelectInstance()`. Finally, `MinimalSamplerec()` can be identified with `UpdateProcess()`.

5.6.5 Sampling with Subsets or Supersets of Σ

The previous Section assumed that the input relation r was an Armstrong relation for Σ, and the goal was to extract a minimal Armstrong relation. This Section relaxes this assumption in both directions. First considering a subset of the whole set of functional dependencies satisfied by r, Σ, and then considering a superset.

If a subset Γ_1 of Σ, $\Gamma_1^* \subsetneq \Sigma^*$, is considered then sampling is not possible. That is, no Sample of a relation, which satisfies exactly Σ^*, will satisfy exactly Γ_1^*.

Theorem 5.6.2. *Let r be an Armstrong relation for a set of functional dependencies Σ. There cannot be a Sample s of relation r which is an Armstrong relation for a proper subset Γ_1 of Σ, $\Gamma_1^* \subsetneq \Sigma^*$.*

Proof. Let σ be a functional dependency such that $\sigma \in \Sigma^*$ but $\sigma \notin \Gamma_1^*$. Since r is an Armstrong relation for Σ there cannot be any pair of tuples in r that do not satisfy σ. Therefore, a Sample s of r cannot violate σ either. Since $s \models \sigma$ for some $\sigma \notin \Gamma_1^*$, s cannot be an Armstrong relation for Γ_1, as required. □

Consider now sampling with supersets of Σ. If the aim is also to extract an Armstrong relation for a given superset of Σ, then this may not always be possible. Refer to Section 5.6.6 for a concrete example of this situation.

Therefore, in order to sample with a superset of Σ, it is necessary to weaken the objective of sampling. Instead of requiring the resulting Sample to be an Armstrong relation, the goal is now only to ensure it satisfies all specified functional dependencies (even if it may also satisfy others).

Theorem 5.6.3. *Let r be an Armstrong relation for a set of functional dependencies Σ over $\mathcal{U}$. There exists a Sample s of r which satisfies a superset Γ_2 of Σ, $\Sigma^* \subsetneq \Gamma_2^*$.*

Proof. Let γ_1 be a functional dependency such that $\gamma_1 \in \Gamma_2$ and $\gamma_1 \notin \Sigma^*$. Without loss of generality assume γ_1 is of the form $Y \to A$, with $Y \subset \mathcal{U}$ and $A \in \mathcal{U}$. Initially, let Sample s be the entire relation r. For any pair of tuples t_1 and t_2 in s with $t_1(Y) = t_2(Y)$ and $t_1(A) \neq t_2(A)$, remove t_2 from s. No functional dependency in Σ can become unsatisfied

by removing a tuple from s (Theorem 5.6.2), therefore now $s \models \Sigma^* \cup \{\gamma_1\}$. (Note that by removing a tuple, however, a new functional dependency not in Γ_2^* may become satisfied.) This process can be repeated for any additional $\gamma_2 \in \Gamma_2 \setminus \{\gamma_1\}$ and $\gamma_2 \notin \Sigma^* \cup \{\gamma_1\}$ until $s \models \Gamma_2$, as required. □

The algorithm given in Fig. 5.5 can easily be modified to sample with supersets of Σ, or to detect when this is not possible. All that is required is to ensure that functions `SelectTuple()` and `SelectTuple_local()` do not select tuples that satisfy non-saturated sets of attributes.

The two theorems presented in this Section identify what can be expected when sampling with functional dependencies, and the minimum knowledge required about the database relation being sampled in order to do so (i.e. a set of functional dependencies for which the input relation is an Armstrong relation). If the set of functional dependencies is not the minimum required, the algorithm given in Section 5.6.4 can detect this situation, notifying the database designer and increasing the understanding of this relation.

5.6.6 Sampling Example

As an example to illustrate the concepts and algorithm presented here to sample with functional dependencies, consider, as in the previous subsection, the set of functional dependencies

$$\Sigma = \{\text{Course} \rightarrow \text{Year},\ \text{Course} \rightarrow \text{Teacher}\}$$

and the database relation shown in Table 5.5. Applying the algorithm of Fig. 5.5 to this relation and the Agreements Table shown in Table 5.6 leads to the following scenario.

1. According to the global selection criteria (function `SelectTuple`), two tuples can initially be selected since they have the highest value for '`No.` $\mathcal{X}^*$': t_{10} and t_{12}. Assume that tuple t_{10} is (randomly) chosen. Now, the recursive procedure is called using `MinialSizeSamplerec(AgrT, Sample,` t_{10}`)`.
    (a) Using the Agreements Table it can be seen how t_{12} must be selected in order to satisfy {Year Teacher*} (function `SelectTuple_local`).

Table 5.7: Sample with FD of **Courses**

	<u>course</u>	year	teacher
t_4	analysis 2	2	philip
t_5	programming	2	diana
t_{10}	languages	5	diana
t_{12}	cybernetics	5	diana

Now `MinialSizeSamplerec(AgrT, Sample,` t_{12}`)` is called.

i. Tuple t_5 is selected so that $\{Teacher\}^*$ is satisfied.

`MinialSizeSamplerec(AgrT, Sample,` t_5`)` is called.

A. In order to satisfy Year*, t_4 will be selected. Additionally, due to the interactions between t_{10} or t_{12} (already in the sample) and t_4, set $\emptyset^*$ is also satisfied.

`MinialSizeSamplerec(AgrT, Sample,` t_4`)` is called.

- All $\mathcal{X}^*$ are satisfied, so this recursive call terminates.

B. All $\mathcal{X}^*$ are satisfied, so this recursive call terminates.

ii. All $\mathcal{X}^*$ are satisfied, so this recursive call terminates.

(b) All $\mathcal{X}^*$ are satisfied, so this recursive call terminates.

2. All $\mathcal{X}^*$ are satisfied, function `MinialSizeSample` terminates.

The resulting Sample relation, shown in Table 5.7, is a minimal Armstrong relation for the input relation.

Section 5.6.5 referred to the fact that it is not always possible to obtain an Armstrong relation by sampling a database relation according to a proper superset of Σ. To see this, assume that the database relation shown in Table 5.7 is to be sampled exactly according to

$$\Gamma_2 = \Sigma \cup \{\text{Teacher} \rightarrow \text{Year}\}$$

with Σ as above. It can be seen how tuple t_5 or tuples $\{t_{10}, t_{12}\}$ must be left out of any Sample, in order to satisfy the new functional dependency $\{\text{Teacher} \rightarrow \text{Year}\}$. However, the resulting Sample (e.g. $\{t_4, t_{10}, t_{12}\}$ or $\{t_4, t_5\}$) would satisfy more functional dependencies

than only those in Γ_2^* (e.g. {Year → Teacher}), thus it cannot be an Armstrong relation for Γ_2.

As an example of successfully sampling with a superset of Σ, consider the relation shown in Table 5.7 and the set of functional dependencies

$$\Gamma_3 = \Sigma \cup \{\text{Teacher Year} \rightarrow \text{Course}\}$$

with Σ as above. Note that the relation represented by Table 5.7 is not an Armstrong Relation for Γ_3. The corresponding set of saturated sets is $\{\{\text{Year}\}, \{\text{Teacher}\}, \{\emptyset\}\}$, therefore, according to Section 5.6.5, the set that must not be satisfied is {Teacher Year} (see Table 5.6). Applying the algorithm of Fig. 5.5 with the modification suggested in Section 5.6.5, the Sample containing tuples $\{t_4, t_5, t_{10}\}$ would be extracted. This Sample is indeed an Armstrong relation for Γ_3. The same procedure will demonstrate why, in the previous example, no Sample of Table 5.7 can be an Armstrong relation for Γ_2.

5.7 Random Sampling

An alternative method for sampling a database is to select the set of tuples randomly. If such an approach is followed, the Sample size cannot be assured to be minimal, but the efficiency improvement (both in performance and memory usage) will be significant, as discussed later in Section 5.8. An approach based on random sampling is only considered here for the case of sampling with functional dependencies. There can be several different Samples of a relation that satisfy an specified set of functional dependencies (even though only a few may be of minimal size). Therefore it can be expected that random sampling will, eventually, found one. If, for example, inclusion dependencies are considered there may very well be only one tuple in one table that can be used to satisfy a particular constraint. In case of large databases, if tuples are selected randomly this may require an unacceptable amount of sampling before the right tuple is chosen. Such an approach is clearly not feasible and is not investigated any further.

Consider that tuples in a relation are to be selected randomly until a specified set of functional dependencies is satisfied by the resulting Sample. Using the same concepts as in

the previous Section, a list of *saturated sets* of attributes still to be *satisfied* by the current Sample is maintained. Whenever a new tuple is randomly selected, the algorithm checks whether any new saturated set is satisfied. If so, this tuple becomes part of the Sample and the saturated set now satisfied is deleted from the list. If not, a new tuple is sampled. This process continues until the list of saturated sets of attributes to be satisfied is empty.

These two approaches, `MinimalSizeSample` with functional dependencies and Random Sample Selection, must be compared. It is expected that in those cases where the minimal size Sample is much smaller than the initial table random sampling will not, on average, perform better than the minimal size sampling algorithm as the resulting Sample will be significantly larger than necessary.

5.8 Analysis of Sampling Algorithms

This Section analyses the sampling algorithms given in Sections 5.5.4, 5.6.4, and 5.7. The study will focus on its complexity, and termination and correctness. The complexity analysis will investigate the worst-case time complexity[4] of these algorithms. The standard O-Notation [39] will be used for such purposes.

5.8.1 Complexity

The worst-case scenario when sampling with an ICG is when the entire database instance is inter-related according to the set of constraints being considered. In this case the only consistent *Sample* would be the Source Database itself. This can be considered an extreme case and, for sampling purposes, a Source Database can be seen as being composed of several independent consistent Samples (Section 5.5.4 already referred to this view and termed it *data-diversity*). Each iteration of the outmost loop in the algorithm of Fig. 5.4 can be seen as extracting one such Sample, and minimal Sample size requirements may force several iterations. For the complexity analysis of this algorithm, let n be the number of entities in the ICG, p the number of instances each entity must contain in the resulting Sample Database as described in Section 5.5.4, and q the number of directly related instances that each instance

[4] Set operations are ignored for simplicity.

has in the database according to the set of constraints used for sampling. Note that q can be different for each instance in the database, but in each case it must be less than the maximum fan-out value of all entities in the ICG, $q \leq MAX(\text{Fan-Out})$.

The complexity of this algorithm depends on the depth of the recursive calls represented by `KeepConsistentICG()`. This, in turn, depends on the database instance being sampled, particularly on the different number of instances of the same entity that are needed to keep consistent one instance of that entity. For example, in the School Reference Database, if one `Teacher` is sampled, this will require its corresponding `Dept`, which in turn will sample back on the same entity, `Teacher`, for the head of this department. In this case, two insertions were needed in the same entity. As another example, take the instance of entity `Courses` shown in Fig. 5.5. Assume that, due to the semantics of `Pre-requisites` (see Section 5.2), any student that follows course `cybernetics` must have already passed courses `calculus`, `analysis 3`, `analysis 2`, and `analysis 1` (this scenario is consistent with the complete instance of the reference database that will be shown in Appendix A). The maximum number of required insertions in that sense will here be denoted by α. In the previous two examples its value would be $\alpha = 2$ and $\alpha = 5$ respectively. In the worst case each of these insertions would require new insertions in all related entities. Therefore this algorithm's worst-case complexity is

$$O\left(npq^{\alpha}\right) \ .$$

The algorithm to extract a minimal size Sample according to a set of functional dependencies consists in three phases: (1) Computing all saturated[5] sets of attributes; (2) Constructing the Agreements Table; (3) Extracting a minimal size Sample using the algorithm shown in Figure 5.5.

Let n be the number of attributes in $\mathcal{U}$, p the number of tuples in the databases, and q the number of functional dependencies in Σ; that is $n = |\mathcal{U}|$, $p = |r|$, and $q = |\Sigma|$.

1. Computing $\mathcal{X}^*$ can be done in linear time with the size of Σ and $\mathcal{X}$ [1], and thus checking whether $\mathcal{X}$ is saturated has complexity $O(q+n)$.

[5] Recall from Section 5.6.4 that it suffices to consider only the intersection generators.

Beeri [11] showed that some functional dependency sets exist for which any Armstrong relation will have exponential size with Σ. Therefore any algorithm must face this worst-case complexity. However, this is only the case for highly unnormalised relations, which should be rare. In case of relations in Boyce-Codd Normal Form [47], the number of saturated sets is bounded by a polynomial in n and q [45]. The degree of this polynomial depends on the number of keys in the relation. Refer to this polynomial as $pol(n, q)$. Therefore this first phase has complexity $O\left(pol(n, q)\,(q + n)\right)$.

2. For each tuple t_1 and each saturated set $\mathcal{X}$ compute which tuples t_2 agree with t_1 exactly on $\mathcal{X}$. Since $|\mathcal{X}| \leq n$ this phase's complexity is $O\left(pol(n, q)p^2 n\right)$. It can be improved, however, if the relation is sorted on $\mathcal{X}$ before each test [46]. Which results in complexity $O\left(pol(n, q)np \log p\right)$.

3. Algorithm of Fig. 5.5 has complexity $O\left(pol(n, q)p\right)$, if one ignores the construction of the Agreements Table which was analysed in Step (2).

Therefore the entire algorithm to extract a Minimal Size Sample with functional dependencies, if the relation is normalised, has complexity (note that it can be expected $p >> q$)

$$O\left(pol(n, q)np \log p\right) \ .$$

This result is comparable to the complexity of known algorithms to generate a minimal set (cover) of functional dependencies that hold in a relation [46].

Similarly, the algorithm for Random Sampling with functional dependencies has complexity

$$O\left(pol(n, q)pn\right) \ .$$

5.8.2 Termination and Correctness

The termination of the algorithm shown in Fig. 5.4 can be guaranteed: (1) The outer loop terminates because the number of entities in the set `NotFullEntities` will decrease (function `SampleInstance()`) as new instances are inserted into the Sample. Eventually, it

becomes the empty set and the algorithm terminates; (2) the chain of recursive calls represented by function `KeepConsistentICG()` also terminates due to the workings of function `SampleInstance_local()`. 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, that is, the Sample Database satisfies those constraints if the Source Database also satisfies them. 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. If an inconsistency in the Source Database is detected the algorithm terminates, and the current Sample cannot be expected to be consistent.

The correctness of the algorithm for sampling with functional dependencies shown in Fig. 5.5 has been proved by Theorem 5.6.1. Its termination can also be guaranteed. The set of saturated sets of attributes still to be satisfied, denoted by `AgrT.SetOfX` in this algorithm, is reduced at each iteration because `SelectTuple_local()` will force new saturated sets to be satisfied. It will eventually become the empty set and the algorithm will terminate. The chain of recursive calls must terminate for the same reason. If an Armstrong relation cannot be extracted for the specified set of functional dependencies this algorithm, as the previous one, can detect this situation and will terminate. The Sample will not be consistent according to the specified criteria.

Finally, the algorithm for random sampling outlined in Section 5.7 will terminate if the input relation is an Armstrong relation. It can be expected that, eventually, any of the tuples in the database will be randomly selected, therefore all required saturated sets can be satisfied. If, in contrast, the input relation is not an Armstrong relation for the specified set of functional dependencies then this algorithm, as has been described, will not detect this situation and will not terminate. It will simply continue sampling trying to satisfy additional saturated sets. Even though it is possible to modify this algorithm to ensure its termination, this will increase its time and space complexity. Since the rationale of using random sampling is its expected increased performance, this does not seem a suitable alternative. Instead, the algorithm could sample until a predefined maximum of tentative selections has been reached.

If a consistent Sample has not been reached, then it should terminate. The correctness of this algorithm can also be guaranted by Theorem 5.6.1 assuming the input relation is an Armstrong relation.

5.9 Summary

This Chapter has analysed the construction of Sample Databases considering concrete representativeness criteria, namely the satisfaction of sets of integrity constraints. Firstly a common representation of widely used sets of integrity constraints has been developed, considering inclusion depedencies, cardinality constraints, generalisation dependencies, and subset dependencies. This representation, referred to as Insertions Chain Graph (ICG), has been specifically designed for sampling purposes and can be derived automatically from an Entity-Relationship diagram for the database to be sampled. An ICG has been formally defined and an algorithm that, using such representation, extracts a consistent Sample from a database has been developed.

Then a similar problem has been addressed considering a different type of integrity constraint: Functional Dependencies. A representation suitable for sampling a database according to a set of functional dependencies has been developed. Using this representation, referred to as Agreements Table, an algorithm has been developed which samples a database relation so that the resulting Sample satisfies the specified set of functional dependencies and no other functional dependency, thus being an Armstrong relation. It has been firstly assumed that the relation being sampled was an Armstrong relation for the set of functional dependencies. Then this assumption has been relaxed, considering subsets and supersets of the initial set of functional dependencies.

For comparision purposes, the problem of sampling according to functional dependencies has also been solved using random sampling.

Finally, the complexity and correctness of the algorithms presented here have been analysed.

Chapter 6

Database Sampling - Formal Framework

6.1 Introduction

This Chapter provides a formal framework where Consistent Database Sampling, as developed in the previous Chapters, can be precisely defined. The next Section introduces some background concepts that, although are not part of the work developed during this research, are required in latter Sections. Section 6.3 gives a semantic description of the Consistent Database Sampling Process given in Chapter 4. This description uses the Denotational Semantics formalism. Section 6.4 defines and characterises a particular type of integrity constraints, referred to as *Sampling-Irrelevant*, which can be ignored during sampling as this cannot lead to an inconsistent Sample Database. The property of being *Sampling-Irrelevant* is shown to be, in general, undecidable.

6.2 Background

A brief introduction to Denotational Semantics is given in the next Section. The objective is to outline and justify the steps involved when providing a Denotational Semantic description, as will be done in Section 6.3. Refer to [59, 7, 62] for a more extended treatment of this topic.

Then Section 6.2.2 briefly describes some concepts related to Decidability Theory in order to justify the problem that will be addressed in Section 6.4, regarding the undecidability of Sampling-Relevant integrity constraints. All concepts presented in that Section can be found in any standard study of Decidability Theory such as [43].

6.2.1 Denotational Semantics and λ-Calculus

Denotational Semantics is a formal method initially introduced to unambiguously define the semantics of programming languages. It describes how a syntactic object (i.e. a program written according to a specified syntax) can be mapped into some abstract value. The semantics of the space of abstract values is considered defined and, from it, the semantics of a program is also defined. As programs are considered to represent (or *denote*) a function, a denotational semantics of a programming language maps programs written in that language into the functions they denote.

The formalism used to define these mappings is the λ-Calculus. This is a formal system for the study of functions, their definition, application and manipulation. An expression in this formalism, a λ-expression, such as $\lambda x_1, \ldots, x_n.E$ denotes a function that takes argument values $v_1, \ldots, v_n$ and returns the result of evaluating expression E with all x_i's replaced by the corresponding v_i's, $1 \leq i \leq n$. In this formalism functions are treated as first-class objects and can, themselves, be arguments to other functions or the result of a function. In the example above, expression E may itself be another λ-expression. As a simple example consider $\lambda x.x^2$ which denotes the function which is commonly written as $f(x) = x^2$. As a more complex example with a function that returns another function consider $\lambda x.\lambda y.xy$, which represents $f(x) = g(y)$ with $g(y) = xy$; this is equivalent to $f(x, y) = xy$.

The λ-expressions used in denotational semantics are usually recursive, that is, the function being defined is used in its own definition. As an example, the factorial function written in λ-notation could be

$$Fact = \lambda n.\text{ if } n = 0 \text{ then } 1 \text{ else } n \times Fact(n-1)$$

Recursive definitions describe an equation which must be solved in order to understand what they represent. In the previous example the equation must be solved for $Fact$.

Like any other equation, recursive definitions can have one, none, or many solutions. A definition with more than one solution is ambiguous as it is not clear what it stands for. In contrast, if it has no solution it cannot be said to stand for anything at all. A theory [59] has been developed to ensure that such definitions have exactly one *natural* solution. According to the theory underlying Denotational Semantics, these equations must be defined over domains that have the structure of a *Complete Lattice*. A domain $\mathcal{D}$ is a Complete Lattice if:

1. A partial order, $\sqsubseteq$, can be defined between its elements. $\sqsubseteq$ defines a *partial order* if it is:

   (a) Transitive: $if\ x \sqsubseteq y\ and\ y \sqsubseteq z\ then\ x \sqsubseteq z$.

   (b) Antisymmetric: $if\ x \sqsubseteq y\ and\ y \sqsubseteq x\ then\ x = y$.

   (c) Reflexive: $\forall x \in \mathcal{D},\ x \sqsubseteq x$.

2. There is a *least* element, $\perp$ (bottom), according to this partial order: $\forall x \in \mathcal{D}, \perp \sqsubseteq x$.

3. Each ascending chain $x_1 \sqsubseteq x_2 \sqsubseteq x_3 \cdots$ has a *least upper bound*, denoted by $\sqcup\{x_i\}$. An element is a least upper bound of the chain if it satisfies $\sqcup\{x_i\} \sqsubseteq x_i$ for each element x_i in the chain.

It can be shown [7] that given two complete lattices $\mathcal{D}_1, \sqsubseteq_{\mathcal{D}_1}$ and $\mathcal{D}_2, \sqsubseteq_{\mathcal{D}_2}$, their direct product $\mathcal{D}_1 \times \mathcal{D}_2, \sqsubseteq_{\mathcal{D}_1 \times \mathcal{D}_2}$ can also be defined as having a complete lattice structure. Similar results can be obtained (with some restrictions) for $\mathcal{D}_1 \rightarrow \mathcal{D}_2, \sqsubseteq_{\mathcal{D}_1 \rightarrow \mathcal{D}_2}$. All these results will be used later in Section 6.3. The *natural solution* that is shown to always exist if complete lattices are used as domains in λ-expressions is defined as the *least fixed point* of the equation defined by a λ-expression. A fixed point of function $f(x)$ is a value y such that $f(y) = y$. It is a least fixed point if it is the smallest of all fixed points according to the partial order defined in the domain of x.

Denotational semantics has also been applied in contexts not related to programming languages. For example, Widom [67] defined with this formalism a semantics for the Starburst rule system. In this case the semantics denote a mapping from the set of database instances,

together with a set of rules and operations on the database, into the set of database instances. This semantics indicate how the initial instance is changed due to the operations and the set of rules. The result of this denotational semantics is a database instance, which implies that it is *understood* what a database instance is, and the objective is to define what the combination of database instance, rules and operations *means*.

A similar idea will be investigated in Section 6.3. The objective is to give a formal semantic description of Consistent Database Sampling. In order to do so, the domains involved in this description must be shown to be Complete Lattices. Only then is it possible to ensure that the λ-expressions that will be defined are consistent and not ambiguous, that is, they will have one natural solution. This solution will be the intended meaning for Consistent Database Sampling. The semantics will map sets of database instances and integrity constraints into sets of database instances. This semantics will indicate how a Sample from the initial instance can be extracted according to a set of constraints. As before, the semantics of a database instance is *understood*, and the objective is to define what it *means*, for sampling, to have a database instance with a set of constraints.

6.2.2 Undecidable Problems and Problem Reducibility

The description of the concepts presented here are traditionally described by the means of concepts like *Turing Machine* and the *language* accepted by a Turing Machine. This Section will use a terminology based on *algorithm* and the *problem* solved by an algorithm. This is in line with the Church-Turing Thesis [43] which identifies the formal concept of Turing Machine with the informal concept of algorithm.

There are some problems for which there is no algorithm that will always solve any instance of that problem. These problems are called *Undecidable*. Examples of such problems, used later in this thesis, include the consistency of a set of first-order-logic formulas [23], and the domain independence of relational calculus queries [1, p. 125].

Several techniques can be used to prove that a particular problem is undecidable. One of them is based on the concept of *Reduction*.

Definition 6.2.1. Let P_1 and P_2 be problems. A *reduction from* P_1 to P_2 is an algorithm τ

that transforms instances of problems, so that x is an instance of P_1 if and only if $\tau(x)$ is an instance of P_2.

Using reduction, a problem can be shown to be undecidable as indicated in the following Theorem.

Theorem 6.2.1. *If problem P_1 is undecidable, and there is a reduction from P_1 to P_2, then P_2 is also undecidable.*

Intuitively, if problem P_2 was decidable, it could be used to decide P_1 using this reduction. But P_1 is already known to be undecidable, a contradiction. Therefore P_2 cannot be decidable.

6.3 Semantics of Consistent Database Sampling

The formal semantics for Consistent Database Sampling presented here complements the informal description provided in Chapter 4, and it is believed can be very useful in the context of understanding and reasoning about database sampling in general, and the interaction between consistency criteria when sampling is performed according to several criteria simultaneously. The framework this semantic description provides can also be used as a benchmark reference for sampling tools, such as the one described in Chapter 7.

Denotational Semantics was initially developed as a formalism to precisely and unambiguously define the semantics of programing languages [59, 62, 7]. In this context, a denotational semantics for a conventional programming language is defined as a *Meaning Function* that takes any program in the language and produces the input-output function computed by that program [59]. Similarly, a denotational semantics for Consistent Database Sampling is defined here as a *Meaning Function* that takes any set of integrity constraints and produces a function that maps a database instance into a new database instance which is a Sample of the first one, consistent with the input set of constraints.

In order to develop this formal description, the next Section defines the domains that are used by the Meaning Function of Section 6.3.4. The definition of this Meaning Function is simplified using a set of supporting functions defined in Section 6.3.3.

6.3.1 Domains

The following are the domains used by the functions defined in the next Sections. They are based on some standard definitions given in Section 5.3.

- Let $\mathcal{DB}$ be the domain of database instances.

  If I is a database instance, then $I = \{t_1, \ldots, t_n\}$ where each t_i is a tuple. Values in a database instance are taken from domain $\mathcal{DOM}$.

- Let $\mathcal{T}$ be the domain of tuples in a database.

  If a database schema is composed by attributes $\{A_1, \ldots, A_n\}$ then $\mathcal{T} = \mathcal{DOM}_{A_1} \times \cdots \times \mathcal{DOM}_{A_n}$. The domain of database instances can be identified with the powerset of the domain of tuples, $\mathcal{DB} = \mathcal{P}\{\mathcal{T}\}$. However, to simplify the notation, both domains will be denoted differently. The concept of *database relation*, as defined in Section 5.3, is not used here. Without loss of generality, it can be assumed that each tuple t_i includes a unique identifier which also identifies t_i's relation [25].

- Let $\mathcal{IC}$ be the domain of integrity constraints.

  As done in Section 5.3, integrity constraints are assumed to be Relational Calculus expressions, that is, first-order-logic formulas without functions and without free variables.

6.3.2 Domains as *Complete Lattices*

The domains defined in the previous Section will be used in the Denotational Semantics description of Consistent Database Sampling. According to the theory of Denotational Semantics, these domains must be proven to have a structure of *Complete Lattices* in order to ensure that the semantics defined below are consistent, see Section 6.2.1.

For each of the domains defined above, its partial order, least element and least upper bound will be given. The proof that the specified relation between elements of each particular domain defines, in fact, a *partial order* will be omitted.

Theorem 6.3.1. *The domain $\mathcal{T}$ of tuples is a complete lattice [7].*

Proof. A partial order, $\sqsubseteq_{\mathcal{DOM}_{A_i}}$, and a least element, $\perp_{\mathcal{DOM}_{A_i}}$, can be assumed for each basic domain A_i [7] (e.g. integer, real, character).

Given two tuples $t_1 = \langle a_1, b_1, c_1, \dots \rangle$ and $t_2 = \langle a_2, b_2, c_2, \dots \rangle$, a partial order $\sqsubseteq_{\mathcal{T}}$ in the domain of tuples can be defined as

$$t_1 \sqsubseteq_{\mathcal{T}} t_2 \text{ iff } a_1 \sqsubseteq a_2 \text{ and } b_1 \sqsubseteq b_2 \text{ and } c_1 \sqsubseteq c_2 \cdots$$

where the appropriate partial order $\sqsubseteq_{\mathcal{DOM}_{A_i}}$ is used for each attribute A_i.

The least element of this domain can be defined as $\perp_{\mathcal{T}} = \langle \perp_{\mathcal{DOM}_{A_i}}, \perp_{\mathcal{DOM}_{A_j}}, \dots \rangle$. It can be proved that $\forall t \in \mathcal{T},\ \perp_{\mathcal{T}} \sqsubseteq_{\mathcal{T}} t$.

Any chain of tuples $\langle a_1, b_1, c_1, \dots \rangle \sqsubseteq \langle a_2, b_2, c_2, \dots \rangle \sqsubseteq \langle a_3, b_3, c_3, \dots \rangle \sqsubseteq \cdots$ defines a chain in each attribute: $a_1 \sqsubseteq a_2 \sqsubseteq \cdots$, $b_1 \sqsubseteq b_2 \sqsubseteq \cdots$, $c_1 \sqsubseteq c_2 \sqsubseteq \cdots$, etc. An element defined as $\sqcup\{\langle a_i, b_i, c_i, \dots \rangle\} = \langle \sqcup\{a_i\}, \sqcup\{b_i\}, \sqcup\{c_i\}, \dots \rangle$ is a least upper bound for this chain.

With these definitions, $\mathcal{T}, \sqsubseteq_{\mathcal{T}}$ is a complete lattice [7]. □

Theorem 6.3.2. *The domain $\mathcal{DB}$ of database instances is a complete lattice.*

Proof. In the previous Section a database instance was identified with a set of tuples, $\mathcal{DB} = \mathcal{P}\{\mathcal{T}\}$.

Its least element can be defined as $\perp_{\mathcal{DB}} = \emptyset$.

Given two database instances $I_1, I_2 \in \mathcal{DB}$, define a partial order as

$$I_1 \sqsubseteq_{\mathcal{DB}} I_2 \text{ iff } I_1 \subseteq I_2 \quad .$$

Any chain of database instances $I_1 \sqsubseteq_{\mathcal{DB}} I_2 \sqsubseteq_{\mathcal{DB}} I_3 \sqsubseteq_{\mathcal{DB}} \cdots$ has a least upper bound defined by $\sqcup\{I_i\} = \bigcup_{i>1} I_i$.

With the above definitions, $\mathcal{DB}, \sqsubseteq_{\mathcal{DB}}$ can be shown to be a complete lattice. □

Theorem 6.3.3. *The domain $\mathcal{IC}$ of integrity constraints is a complete lattice.*

Proof. Its least element is $\perp_{\mathcal{IC}} = false$.

A partial order between integrity constraints can be defined as in logical implication since a constraint is an assertion (relational calculus formula) about a database instance. Let

$\sigma_1, \sigma_2 \in \mathcal{IC}$:

$$\sigma_1 \sqsubseteq_{\mathcal{IC}} \sigma_2 \text{ iff } \forall I \in \mathcal{DB},\ I \models \sigma_1 \rightarrow I \models \sigma_2 \quad .$$

A least upper bound for a chain of constraints can also be defined based on their logic nature. Take the least upper bound of $\sigma_1 \sqsubseteq_{\mathcal{IC}} \sigma_2 \sqsubseteq_{\mathcal{IC}} \sigma_3 \sqsubseteq_{\mathcal{IC}} \cdots$ as being $\sqcup\{\sigma_i\} = \sigma_1 \vee \sigma_2 \vee \sigma_3 \vee \cdots \quad .$

With the above definitions, $\mathcal{IC}, \sqsubseteq_{\mathcal{IC}}$ can be shown to be a complete lattice. □

6.3.3 Supporting Functions

The following functions have been used to simplify the denotation of Consistent Database Sampling. Function definitions are given using the λ-Calculus [36, 59]. Refer to Section 6.2.1 for a brief introduction.

Function `ToBeSatisfied` takes a database instance and a set of integrity constraints. It considers the database instance to be the current state of the Sample Database being extracted, denoted by s. This function returns the set of integrity constraints yet to be satisfied by the Sample Database.

$$\begin{aligned} &\text{ToBeSatisfied} : \mathcal{DB} \times \mathcal{P}(\mathcal{IC}) \rightarrow \mathcal{P}(\mathcal{IC}) \\ &\text{ToBeSatisfied} = \lambda\ s, \{ic_1, ic_2, \ldots, ic_n\}.\ \{ic_i | s \not\models ic_i\} \end{aligned} \tag{6.1}$$

Function `Satisfy` adds a set of tuple(s) needed in the Sample to satisfy one particular integrity constraint. It takes a database instance that represents the operational database being sampled, denoted by d, another database instance that represents the current Sample, denoted by s, and finally the integrity constraint to be satisfied, denoted by c. It returns the Sample enlarged with additional tuples required to satisfy the specified constraint. Note that there could be several ways of consistently enlarging the current Sample.

$$\begin{aligned} &\text{Satisfy} : \mathcal{DB} \times \mathcal{DB} \times \mathcal{IC} \rightarrow \mathcal{DB} \\ &\text{Satisfy} = \lambda\ d, s, c.\ s \cup \\ &\qquad \text{Select}\Big(\{s' | s' \subset d \setminus s \wedge s \cup s' \models c \wedge \{s'' | s'' \subset s' \wedge s \cup s'' \models c\} = \{\emptyset\}\}\Big) \end{aligned} \tag{6.2}$$

This definition considers all sets of tuples not yet in the Sample ($s' \subset d \setminus s$) that can be used to satisfy constraint c ($s' \models c$). No subset of this set of tuples should also be eligible, since a small Sample is sought. Then one of these sets is selected to be part of the Sample.

The following functions are left undefined and their semantics are taken as given. They determine different parameters of the Sampling Process, like for example how to perform initial insertions which trigger a chain of insertions to maintain consistency, or how the Process chooses between tuples when several are possible.

`SelectTuple` selects one tuple from a database based on the set of integrity constraints, i.e. initial selection, or `SelectInstance()`, as referred to in Section 4.3.

$$\text{SelectTuple} : \mathcal{DB} \times \mathcal{P}(\mathcal{IC}) \rightarrow \mathcal{T} \tag{6.3}$$

`Choose` selects one integrity constraint out of a set.

$$\text{Choose} : \mathcal{P}(\mathcal{IC}) \rightarrow \mathcal{IC} \tag{6.4}$$

Finally, function `Select` selects one out of all the ways of consistently completing a Sample (e.g. the smallest one).

$$\text{Select} : \mathcal{P}(\mathcal{DB}) \rightarrow \mathcal{DB} \tag{6.5}$$

6.3.4 Meaning Function

The semantics of database sampling is denoted by meaning function $\mathcal{M}$. This function takes a set of integrity constraints and returns another function which takes an operational database and returns either a Consistent Sample of it or it does not terminate, denoted by $\perp$ (bottom). The case in which it may not terminate stems from the fact that the operational database itself may not be consistent. In this case, it may not always be possible the extract a consistent Sample. The algorithms analysed in Chapter 5 detect this possibility. However, the semantics of the Process when sampling from an inconsistent Source Database are left undefined. From a more practical point of view, interaction with the user, to repair or discard the inconsistencies found in the Source Database, is likely to be the only advisable option. This points out that a likely consequence of Consistent Sampling is a better understanding

of the Source Database, as was already outlined in Section 4.2.

$$\begin{aligned}
&\mathcal{M} : \mathcal{P}(\mathcal{IC}) \rightarrow \mathcal{DB} \rightarrow \mathcal{DB} \cup \{\bot\} \\
&\mathcal{M}[\![\{ic_1, ic_2, \ldots, ic_n\}]\!] = \lambda\ d. \\
&\qquad \text{Let } t = \text{SelectTuple}\big(d, \{ic_1, ic_2, \ldots, ic_n\}\big) \text{ in} \\
&\qquad \mathcal{M}'(\{ic_1, ic_2, \ldots, ic_n\})(d, \{t\})
\end{aligned} \tag{6.6}$$

This definition indicates how, in Consistent Database Sampling, an initial tuple, t, is selected and inserted into the Sample. Then this single-tuple Sample is consistently completed, that is, function $\mathcal{M}'$ is called. $\mathcal{M}'$ adds the necessary tuples into the Sample in order to satisfy a set of constraints. This function takes a set of integrity constraints and returns the *Least Fixed Point* of a function that takes an operational database and the current Sample Database (which may not be consistent) and returns either a consistent Sample Database or it does not terminate.

$$\begin{aligned}
&\mathcal{M}' : \mathcal{P}(\mathcal{IC}) \rightarrow \mathcal{DB} \times \mathcal{DB} \rightarrow \mathcal{DB} \cup \{\bot\} \\
&\mathcal{M}' = \lambda\{ic_1, ic_2, \ldots, ic_n\}.\ \text{Least-Fixed-Point}\Big(\lambda\ F. \\
&\quad \lambda\ d, s.\ \text{Let } R = \text{ToBeSatisfied}\big(s, \{ic_1, ic_2, \ldots, ic_n\}\big)\ \mathit{in} \\
&\quad \text{if } R = \emptyset \text{ then } s \text{ else } F\Big(d,\ \text{Satisfy}\big(d, s,\ \text{Choose}(R)\big)\Big)\Big)
\end{aligned} \tag{6.7}$$

If all constraints are satisfied, then the Sample is already consistent and the process terminates. Otherwise, it firstly selects one constraint out of all those which are not yet satisfied in the Sample, consistently completes the current Sample according to this individual constraint and then *repeats* the process, that is, its semantics are defined as its fixed point.

6.4 ***Sampling-Relevant*** Integrity Constraints

When consistently sampling from a database not all integrity constraints need to be considered. For some types of constraints, even if they are excluded from the sampling process, the resulting Sample Database will still satisfy them, assuming that the operational database being sampled is consistent according to these same constraints. Example of such integrity

constraints include maximal cardinality constraints (e.g. in the School reference database, a teacher cannot be teaching more than three courses). These type of constraints have been termed here *Sampling-Irrelevant*. If only those constraints that are actually relevant to the consistency of the resulting Sample Database are considered the sampling process can be expected to be more efficient. The representation of the set of constraints to be considered should also be simpler as it needs to express a smaller number of constraints (refer to Section 5.5.3 where this issue was already addressed).

Integrity constraints which are not Sampling-Irrelevant are referred to as *Sampling-Relevant*. The objective in this Section is to characterise these kind of integrity constraints.

From the above discussion, Sampling-Irrelevant Integrity Constraint can be defined as follows.

Definition 6.4.1. σ is a *Sampling-Irrelevant* Integrity Constraint iff

$$\forall I \in \mathcal{DB}, \forall t \in \mathcal{T}, I \models \sigma \rightarrow I \setminus \{t\} \models \sigma$$

Being I a database instance and t a tuple in this database. This definition states that if operational database I is consistent with constraint σ, then it will still be consistent after deleting a tuple from it. This definition views a Sample Database as the result of deleting, from the Source Database, all the tuples which are not in the Sample. Although this view is useful for the developments of this Chapter, it is likely to be impractical due to the size of a Source Database compared to that of a Sample Database.

Negating the expression in the previous definition provides a definition for *Sampling-Relevant* Integrity Constraints.

Definition 6.4.2. σ_q is a *Sampling-Relevant* Integrity Constraint iff

$$\exists I \in \mathcal{DB}, \exists t \in \mathcal{T},\ I \models \sigma \wedge I \setminus \{t\} \not\models \sigma$$

In this definition, I is a database instance and t a tuple in this instance. $I \setminus \{t\}$ denotes a database instance that results from removing tuple t from I. According to this definition, if σ is Sampling-Relevant there must be a database instance I and a tuple t in this database that *witness* the relevance of σ when sampling, that is, deleting t from I results in a database which does not satisfy constraint σ.

6.4.1 Characterisation of Sampling-Relevance

From Definition 6.4.2 it is possible to deduce the following sufficient, but not necessary, condition for an integrity constraint to be Sampling-Relevant.

Theorem 6.4.1. Given any integrity constraint σ,

$$\emptyset \not\models \sigma \rightarrow \sigma \textit{ is Sampling-Relevant}$$

Proof. As stated in Section 5.3, only finite database are considered here. Theorem 6.4.1 follows directly from this fact.

Consider a Sampling-Irrelevant integrity constraint σ. Let $I_0 = I$ be the initial database instance. Since σ is Sampling-Irrelevant, according to Definition 6.4.1, any tuple t_0 can be removed from it and the resulting database instance, $I_1 = I_0 \setminus \{t_0\}$, will still satisfy σ, $I_1 \models \sigma$. Repeat this process for any other tuple, t_i, leading to increasingly smaller database instances, $I_{i+1} = I_i \setminus \{t_i\}$, all of them satisfying σ, $I_{i+1} \models \sigma$. Since I is finite, eventually some database instance I_{i+1} will be empty, $I_{i+1} = \emptyset$. This proves that

$$\sigma \text{ is Sampling-Irrelevant} \rightarrow \emptyset \models \sigma$$

which is equivalent to what was to be proved. □

This theorem states that if a constraint is not satisfied by an empty database, then it must be Sampling-Relevant. It is possible, however, for a Sampling-Relevant integrity constraint to be satisfied by an empty database (see Section 8.3.3).

6.4.2 Non-Decidability of Sampling-Relevance

This Section gives a proof of undecidability for Sampling-Relevance of integrity constraints. As described in Section 6.2, one standard technique for proving that a problem is undecidable is by reducing a well-known undecidable problem to it [43]. The well-known undecidable problem used here is the Domain-independence of Relational Calculus Queries [1, 64].

A Relational Calculus Query q is an expression of the form $q = \{x_1, \dots, x_n | \varphi\}$, where all variables x_i are free in φ, being φ a first-order-logic formula without functions. The result of

a query q over database instance I, denoted $q(I)$, is the set of tuples $\langle x_1, \ldots, x_n \rangle$ for which φ is true. When interpreting a Relational Calculus Query it must be specified over which domain variables x_i range. Different answers may be obtained depending on the domains being considered [1]. For this reason a query is interpreted relative to a particular domain. Thus $q_{d_i}(I)$ denotes the result of query q for instance I if the domain of interpretation is d_i. In practice only those queries that always produce the same result, independently of the domain of interpretation, are allowed. This set of queries are referred to as *domain-independent*. Therefore, a domain-independent query can be defined as follows.

Definition 6.4.3. q is a *Domain-Independent* Relational Calculus Query iff

$$\forall d_1, d_2 \in \mathcal{DOM}, \forall I \in \mathcal{DB},\ q_{d_1}(I) = q_{d_2}(I)$$

It is known that if the entire expressiveness power of Relational Calculus is allowed to define a query, domain-independence it not decidable ([1], p. 125). Since only domain-independent queries are of interest, several syntactic restrictions have been developed to ensure the domain-independence of queries. Refer to [1, 64] for a more extended treatment of Relational Calculus Queries, justification for the need of domain-independent queries, and some syntactic restrictions (e.g. range-restricted) used to deal with their undecidability. For the purposes of this Section only its definition and its undecidability result are needed.

Negating Definition 6.4.3 results the definition of Domain-Dependent queries, which will be used in the proof presented here. It must be noted that if a problem is undecidable its complement must also be undecidable [43].

Definition 6.4.4. q is a *Domain-Dependent* Relational Calculus Query iff

$$\exists d_1, d_2 \in \mathcal{DOM}, \exists I \in \mathcal{DB},\ q_{d_1}(I) \neq q_{d_2}(I)$$

In order to simplify the notation, in the following development formulas will be given without explicitly stating the domain of variables. This should not cause confusion as variables with the same names as above will range over the same domains.

Figure 6.1 states the purpose of the proof to be developed in this Section. It is the specialisation of Theorem 6.2.1 to this particular problem.

Let q be a relational calculus query. From q, construct an integrity constraint σ_q such that σ_q is *Sampling-Relevant* if and only if q is *Domain-Dependent*

Fig. 6.1: Non-Decidability of Sampling-Relevant Integrity Constraints Problem Statement

Theorem 6.4.2. *Sampling-Relevance is an undecidable property of integrity constraints.*

Proof. Let $q = \{x_1, \dots, x_n | \varphi\}$ be a Relational Calculus Query. An integrity constraint σ_q can be constructed from q as in Equation 6.8.

$$\sigma_q = \exists d_1, d_2 \exists t' \Big(t' \in q_{d_1}(I) \wedge t' \notin q_{d_2}(I) \Big) \tag{6.8}$$

The objective is to prove that σ_q is Sampling-Relevant if and only if q is Domain-Dependent. This proof is divided into two parts.

1. σ_q is Sampling-Relevant $\rightarrow$ q is Domain-Dependent

   Since σ_q is Sampling-Relevant there must be some database instance I and tuple t that witness this fact, $I \models \sigma_q \wedge I \setminus \{t\} \not\models \sigma_q$. It suffices to consider only $I \models \sigma_q$. By construction of σ_q such database instance I witnesses the domain-dependence of q: there is a tuple t' which is in $q_{d_1}(I)$ but not in $q_{d_2}(I)$, an so $q_{d_1}(I) \neq q_{d_2}(I)$ as required.

2. q is Domain-Dependent $\rightarrow$ σ_q is Sampling-Relevant

   There is an instance I which witnesses the domain-dependence of q, $\exists t' \Big(t' \in q_{d_1}(I) \wedge t' \notin q_{d_2}(I) \Big)$. Note that t' is a tuple which results from values of tuples in I. Let I_w be a database instance that results from I by deleting any one of such tuples for each tuple t', but one[1]. For example, assume that tuples t'_1 and t'_2 are the only tuples that witness the domain-dependence of q. Further, assume that they take values from tuples t_1, t_2, t_3 and t_3, t_4 respectively. If, for example, t_1 is deleted from I, there will remain only one tuple, t'_2, which will witness the domain-dependence of q. This tuple takes its values from t_3 and t_4. Denote by t_w any of these two tuples.

   It can be seen how I_w and t_w, as constructed above, witness the Sampling-Relevance of σ_q:

[1] As in Theorem 6.4.1, only finite databases are considered.

(a) $I_w \models \sigma_q$

There is at least one tuple in I_w that witnesses the domain-dependence of q, $\exists t'\Big(t' \in q_{d_1}(I_w) \wedge t' \notin q_{d_2}(I_w)\Big)$. This makes σ_q true, as required.

(b) $I_w \setminus \{t_w\} \not\models \sigma_q$

There is one and only one t' such that $t' \in q_{d_1}(I_w) \wedge t' \notin q_{d_2}(I_w)$, and this tuple takes its values from t_w. Therefore $I_w \setminus \{t_w\} \not\models \sigma_q$, as required.

□

The motivation given in Section 6.4 to investigate the Sampling-Relevance of integrity constraints was to simplify the set of constraints being considered during sampling. Theorem 6.4.2 has shown that this goal is, in general, not achievable. Since it cannot be decided when a constraint is Sampling-relevant, all constraints must be considered. Section 8.3.3 will briefly describe how this problem could be addressed.

6.5 Summary

This Chapter has presented a formal analysis of Consistent Database Sampling.

Some standard definitions and techniques have first been given. Then a formal description for the Consistent Database Sampling Process has been analysed, using the formalism of Denotational Semantics. This formalisation has been originated from the developments presented in Chapter 5, but without making any assumptions with regard to which integrity constraints are being enforced. After that a particular type of integrity constraints, particularly relevant to database sampling, has been investigated. A characterisation of this type of constraint, referred to as Sampling-Relevant, has been developed. Finally, given any integrity constraint (expressed as a relational calculus expression) it has been proved that, in general, it cannot be decided whether it is Sampling-Relevant or not.

Chapter 7

Prototype of a Consistent Database Sampling Tool (CoDaST)

7.1 Introduction

The Sampling Process presented in Chapter 4 and the algorithms that specialise it for particular integrity constraints types investigated in Chapter 5 have been implemented and compiled into a prototype that is referred to as a Consistent Database Sampling Tool, or CoDaST. This Chapter describes the design and implementation of this tool. Some basic Object-Oriented design [20] and UML [48] (Unified Modeling Language) terminology is used.

The next Section identifies the most significant requirement for the design of CoDaST, namely the capability of incrementally constructing sampling modules that can sample a database according to increasingly more complex criteria. The mechanism that allows for such an incremental process is analysed in Section 7.4. This mechanism, however, requires that all integrated sampling modules adhere to a particular *Sampling Protocol*, as referred to here, in order to ensure that independently developed sampling modules can inter-operate with each other. Section 7.3 describes this Protocol. The design of CoDaST is detailed in Section 7.5, while Section 7.6 addresses more implementation-related issues.

7.2 Requirements of CoDaST

The Consistent Database Sampling Process (CoDaSP) described in Chapter 4 was developed as a general description of the set of activities and their relationships common to any database sampling application, independent of the criteria used to evaluate the representativeness of the resulting Sample Database. Since these activities are common to any type of sampling, they can be seen as the functionality that a *Consistent Database Sampling Tool* (CoDaST), see Fig. 4.2, should support. A CoDaST should be composed of several sampling modules which implement this funtionality. Hereafter, such sampling modules will be referred to as *(Database) Samplers*. Each Sampler extracts a Sample from a database according to some criteria (e.g. concrete integrity constraint type, random[1]). The design principle behind the general description of CoDaSP is that existing Samplers can be integrated to create more sophisticated Samplers which Sample a database according to several criteria *simultaneously*. This allows a CoDaST to be constructed *incrementally*, developing different Samplers *independently* and then integrating their behaviour in a new Sampler.

The mechanism to integrating existing Samplers is described in Section 7.4. It relies on the fact that all Samplers adhere to a well-defined Sampling Protocol which underlies the CoDaSP.

7.3 Consistent Database Sampling Protocol

The motivation for this Protocol arises from the need to allow independently developed Samplers to inter-operate. That is, database instances inserted into the Database Sample by one Sampler may need to be maintained consistent with the respresentativeness criteria of other Samplers. Each particular Sampler may also be used in isolation, without inter-operating with other Samplers. By defining a Sampling Protocol the workings of any Sampler that adheres to it will be the same, independently of whether it is integrated with other

[1]References to random sampling are included here for completeness only. As justified in Chapter 4, the preferred representativeness evaluation criteria is the satisfaction of sets of integrity constraints. Including random sampling does, however, extend the range of applications for which a Database Sampling Tool can be used.

Samplers or not.

A Database Sampler adheres to the Sampling Protocol used in CoDaST if it implements the following interface and semantics:

Interface All complying Samplers must *implement* the five methods described in Section 4.3, namely `InitialiseProcess()`, `StopSampling()`, `SelectInstance()`, `Synchronise()`, and `UpdateProcess()`. This allows Samplers to be integrated with each other without knowledge of the specific representativeness criteria they enforce.

Semantics in Isolation The semantics of these methods must be as explained in Section 4.3. That is, each individual Sampler may be used in isolation to implement the Sampling Process and extract a Sample Database according to its particular criteria.

Semantics for Integration After each insertion into the Sample Database a Sampler must always call methods `Synchronise()` and `UpdateProcess()`, in that order. See Section 7.4 for a description of the semantics of these two methods. The order in which they must be called is used to implement what can be seen as a depth-first traversal of all instances that must be inserted into the Sample Database. Even though a breadth-first traversal would also be possible, it is necessary to agree on a common strategy for all Samplers, as this needs to be exploited by the integration mechanism described in Section 7.4. Depth-first simplifies the implementation and, more importantly, has less memory requirements than a breadth-first traversal. In terms of the workings of CoDaST, a breadth-first traversal implementation would perform all insertions required by one Sampler alone. After that, for each of these insertions, each of the others related Samplers would be used to select the appropriate instances, leading to new chains of insertions, which would then require the process to be repeated. It can seen how the memory requirements of this approach are significant. A depth-first traversal, in contrast, uses one Sampler to select one instance, then another one to maintain the last instance consistent with its own criteria, and so on. Only when the first insertion is consistent with the representativeness criteria of all Samplers will, the initial Sampler, be used to keep its initial insertion consistent with its own cirteria. Therefore, using a

depth-first strategy, each Sampler only needs to store the last insertion it performed.

7.4 Database Sampler Integration Mechanism (DaSIM)

Given a set of Database Samplers the objective is to define a general Database Sampler Integration Mechanism, or DaSIM, to build a new Sampler which implements the representativeness criteria of all integrated Samplers simultaneously. This mechanism assumes that the set of Samplers to be integrated adhere to the Sampling Protocol defined in the previous Section. DaSIM is defined by this Protocol and by the semantics of method `Synchronise()` as follows (a description of `UpdateProcess()` is included for completeness only; although it was already described in Section 4.3, it complements the semantics of `Synchronise()` and therefore its inclusion clarifies the workings of DaSIM):

- `Synchronise()` A Sampler must always call method `Synchronise()` after performing an insertion into the Sample Database. This will, in case this Sampler is integrated with other Samplers, ensure that this insertion is kept consistent with the representativeness criteria of these other Samplers. For this reason `Synchronise()` can be seen as the enforcement of *Inter-Sampler Consistency* (as referred to in Fig. 4.3), as opposed to *Intra-Sampler Consistency* which is achieved by `UpdateProcess()`.

  `Synchronise()` will require information about the (instance of) Samplers to be integrated. When called, it will iterate through each of these Samplers (except for the one which performed the last insertion) and call method `UpdateProcess()` for it, *just as it should be done if this insertion had actually been performed by that Sampler*. This will keep the very last insertion consistent according to each of these Samplers. It must be pointed out that, according to the Sampler Protocol defined above, `Synchonise()` must also be called again after each of these insertions.

- `UpdateProcess()` ensures that the last insertion into the Sample Database is kept consistent with respect to the representativeness criteria enforced by the Sampler that performed this insertion.

DaSIM has been implemented and included in CoDaST as a specialised Sampler termed `SamplerIntegrator`. The design of CoDaST, including this specialised Sampler, is described next.

7.5 Design of CoDaST

The initial step when building a Consistent Database Sampling Tool (CoDaST) is the construction of a set of *Basic Database Samplers*, that is, Database Samplers which are not the result of integrating the behaviour of existing Samplers. The appropriate set of Basic Samplers to be used depends on the particular application of database sampling at hand. Each of them should sample a database according to one single criterion only. By having a set of Samplers which implement simple sampling criteria and then integrating them, as opposed to creating one single Sample that implements all criteria, Basic Samplers can be re-used and integrated in different ways for different database sampling applications. This design makes CoDaST more flexible and extensible. The current implementation of CoDaST supports the following Basic Samplers:

Random Sampler Randomly selects the specified number of tuples from a table. Several instances of this type of Sampler could be used, each of them sampling different tables. Fig. 7.1 shows the design of this Sampler as a UML class diagram [48]. It can be seen how in this design a `RandomSampler` includes (*agregates* in UML terminology) a random number generator (class `CRngCongr` in this Figure) used to generate the tuple identifiers that will be inserted into the Sample Database. Particularly, the current implementation of CoDaST uses the mixed-congruential random number generator described in [40], which has been tested for numerous empirical tests of randomness. This Sampler adapts the algorithm for efficient sequential random sampling from a file presented in [65] to the Sampling Process of Chapter 4, implementing the interface specified by the Sampling Protocol of Section 7.3. This algorithm requires the size (i.e. number of tuples) of the table being sampled and the number of tuples that have already been inserted into the Sample at any given time, in order to randomly select the next tuple.

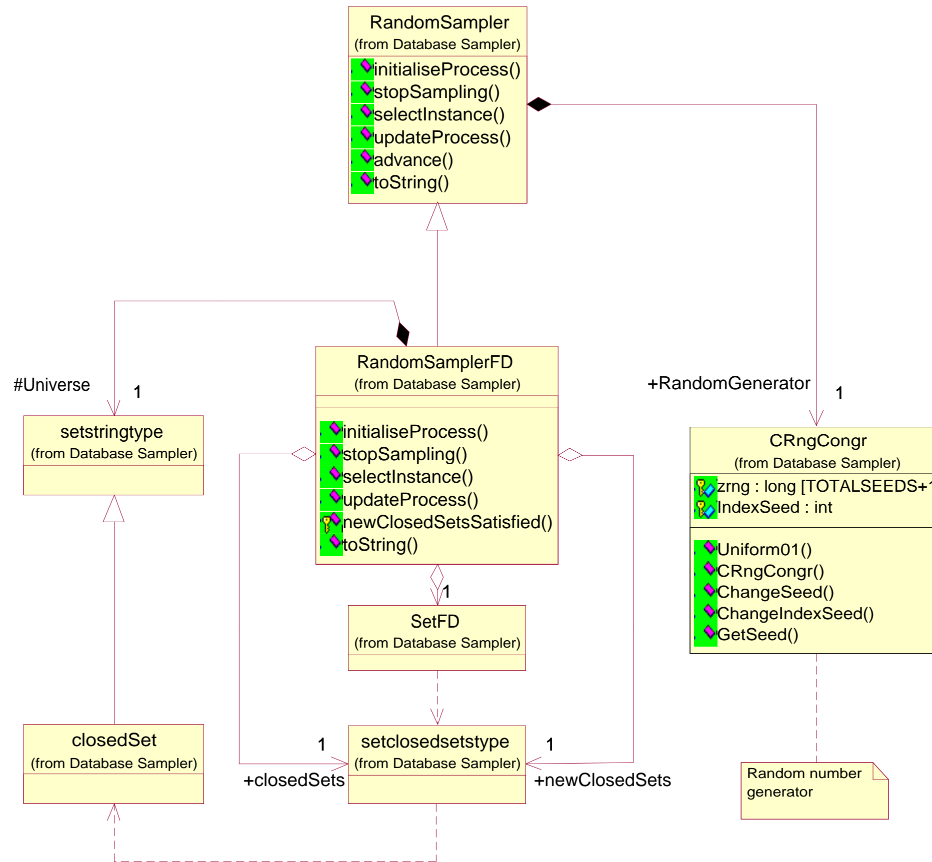

Fig. 7.1: UML Class Diagram for the Design of Random Samplers

Sampler with Functional Dependencies Selects a minimal size Sample of a specified table that satisfies the specified set of functional dependencies, as described in Section 5.6. The design for this Sampler is shown in Fig. 7.2. The most relevant elements of this design should be clear from the description given in that Section. What was there referred to as *Agreements Table* is represented in this design by class `InfoTable`. Classes `setP`, `std::set`, and `setfdtype` are needed in order to define sets of objects (e.g. strings, functional dependencies, closed sets of attributes) with the required operations and should be ignored for a general understanding of this design.

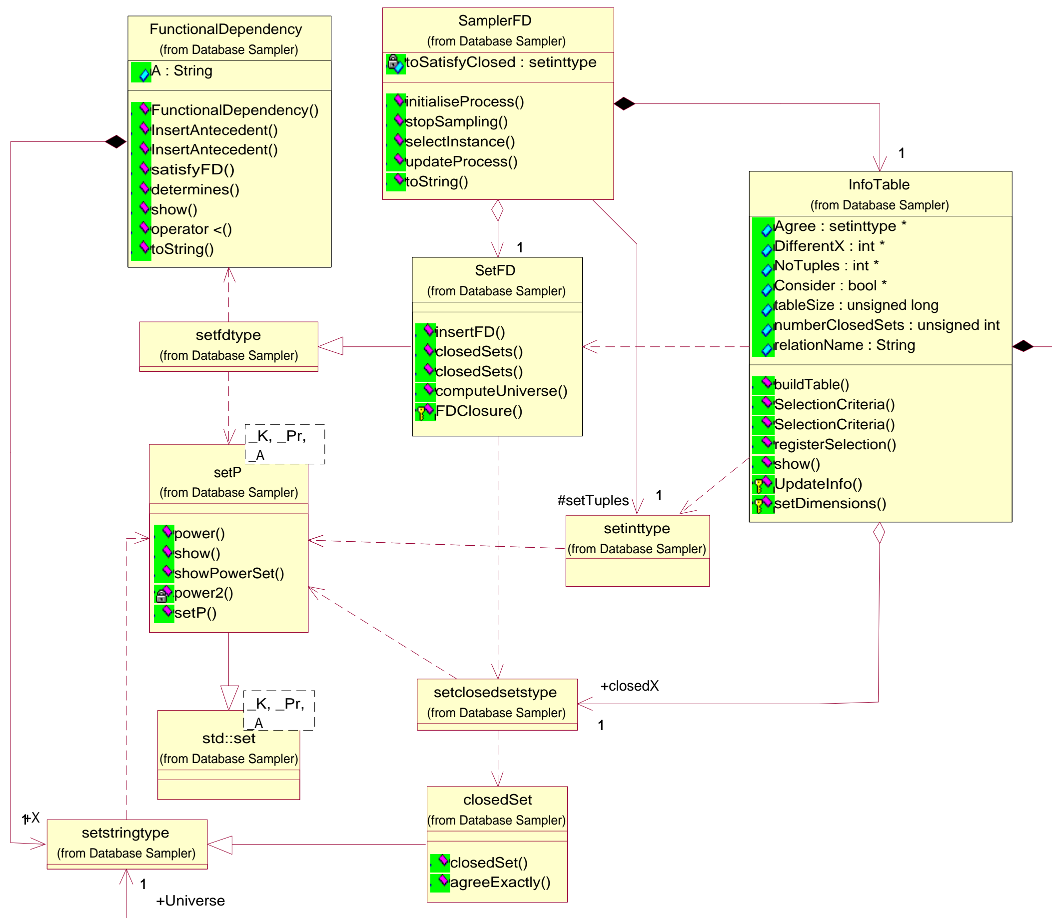

Fig. 7.2: UML Class Diagram for the Design of Database Sampler with Functional Dependencies

Random Sampler with Functional Dependencies Randomly samples a table until the resulting Sample satisfies the specified set of functional dependencies. Refer to Section 5.7 for a detailed description of the algorithm implemented by this Sampler and to Fig. 7.1 for its design as used in CoDaST. This Sampler *inherits* [20] from `RandomSampler`, as tuples are still selected randomly. Therefore, part of `RandomSampler`'s behaviour can be re-used. Other part, however, must ensure the satisfaction of a set of integrity constraints, *overwriting* [20] the definition of the interface all Samplers implement (i.e. `StopSampling()`, `UpdateProcess()`, etc).

Sampler with ICG Using an Insertions Chain Graph to represent the set of integrity constraints that must hold in the resulting Sample Database, it extracts a Sample that contains at least the specified number of instances in each entity and that satisfies this set of constraints. This Sampler implements the algorithm given in Section 5.5. The design for this Sampler together with that of an Insertions Chain Graph is given in Fig. 7.3.

Fig. 7.4 shows the general design for this set of basic Samplers. This Figure shows how all Samplers in CoDaST must inherit, directly or indirectly, from class `DatabaseSampler`. Recall from Chapter 4 that the Sampling Process was described as method `ExtractSample()` of a class named `DatabaseSampler`, which represents the abstract concept of Database Sampler (i.e. it is an *abstract class* [20]), without referring to any particular representativeness criteria. This class, in addition to implementing `ExtractSample()`, defines the interface presented in Section 7.3. All Samplers that inherit from it must implement this interface for their specific criteria, otherwise they will not be correct specialisations of `DatabaseSampler`. Finally, all Samplers can be seamlessly integrated with other Samplers using the DaSIM analysed in the previous Section. This requires both, an instance of class `SamplerItegrator` which integrates a set of Samplers (see Section 7.5.1), and the implementation of method `Synchronise()` for each of the integrated Samplers. Thanks to the use of a well-defined interface and Sampling Protocol, the implementation of this method is exactly the same for each of these Samplers (it *delegates* [20] its execution to the appropriate instance of `SamplerItegrator`) and therefore is implemented in class `DatabaseSampler`.

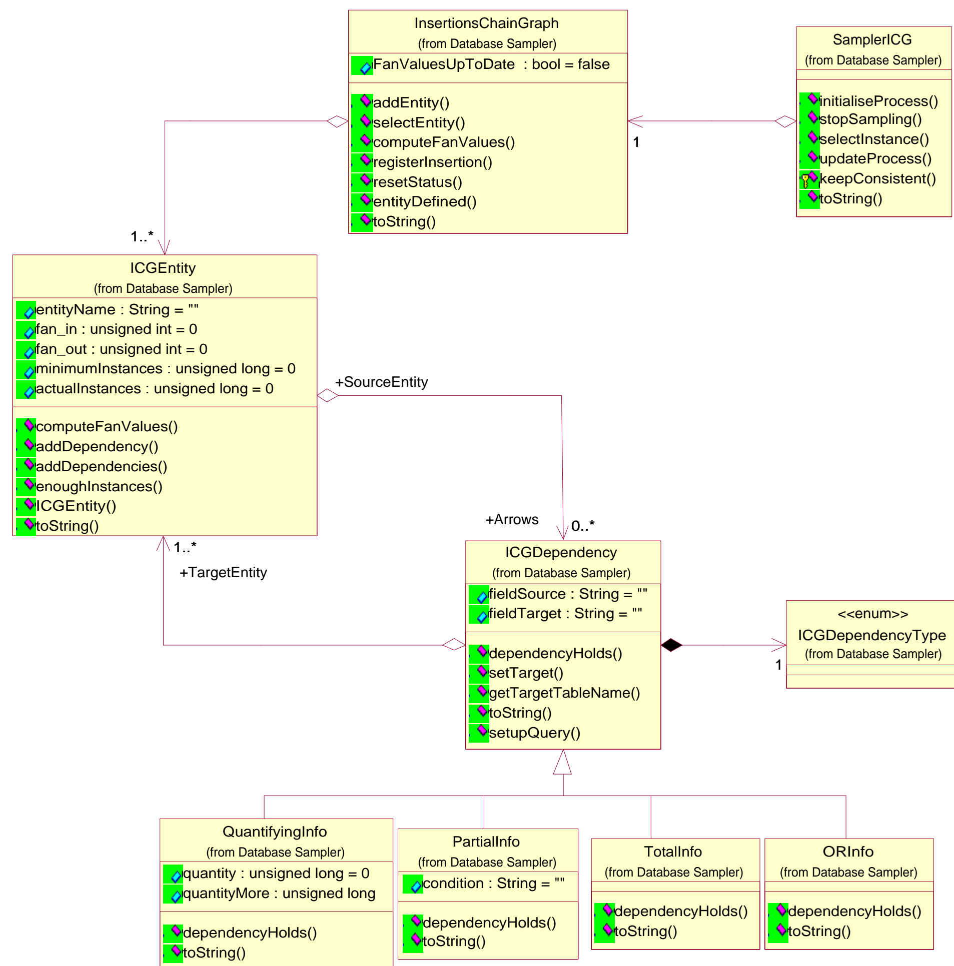

Fig. 7.3: UML Class Diagram for the Database Sampler with Insertions Chain Graph

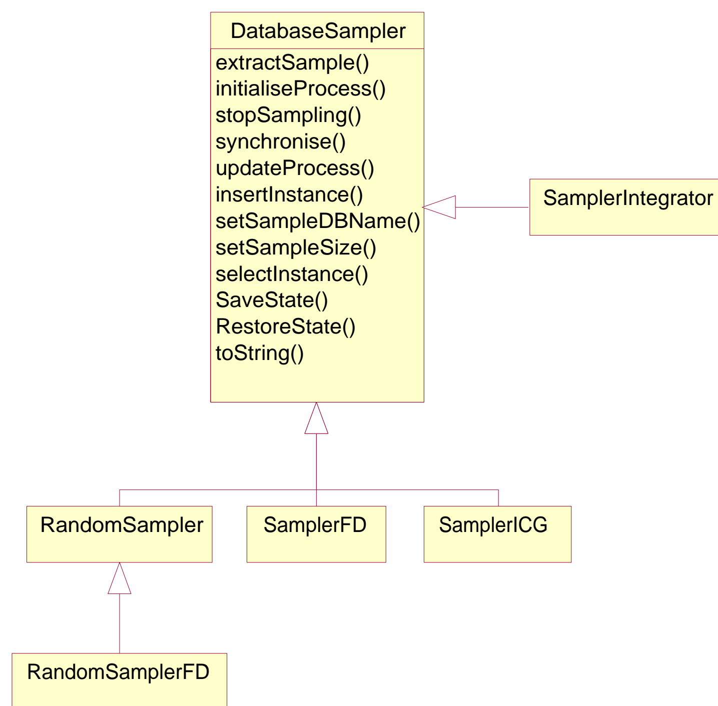

Fig. 7.4: UML Class Diagram for the Design of a Set of Basic Database Samplers

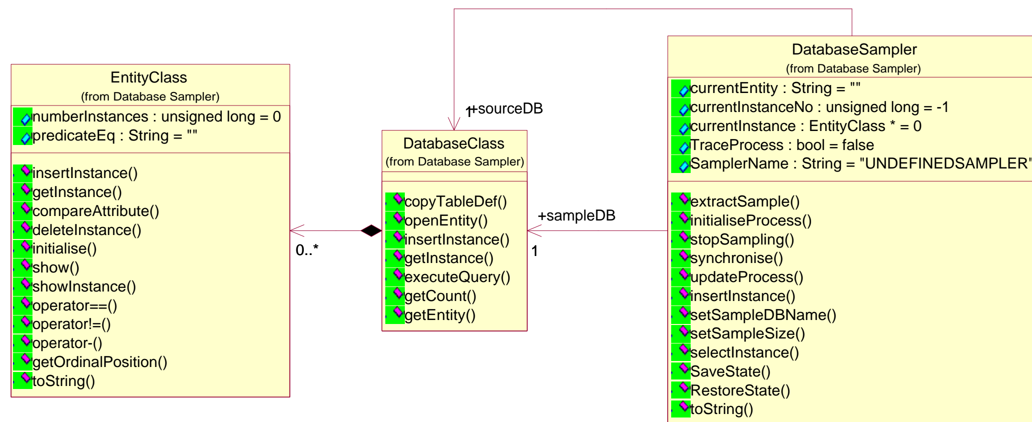

Fig. 7.5: UML Class Diagram for the Database Interface

As a final element of the design of CoDaST, recall that the description of the Sampling Process given in Section 4.3 identified the need for an appropriate interface to the database model being sampled. The design of the interface to the relational model used in CoDaST is shown in Fig. 7.5. This interface does not include terminology or concepts specific to the relational model, but only those used in CoDaSP. In case CoDaST was extended to be able to sample from other data models in addition to the relation model, this design could be seen as an interface to a generic data source. The most important element in this design regards the references (i.e. *associations* in UML terminology [48]) that `DatabaseSampler` contains to both the Sample Database (association termed `sampleDB` in Fig. 7.5) and the Source Database (termed `sourceDB` in this Figure).

Once the set of Basic Samplers has been developed, they can be integrated to create more sophisticated Samplers that Sample a database according to several criteria simultaneously. In order to achieve this integration, the Database Sampler Integration Mechanism described in the previous Section (i.e. the semantics of method `Syncrhonise()`) is implemented in CoDaST by one specific type of Sampler, called `SamplerIntegrator`, the purpose of which is not to sample a database according to any specific criteria but only to integrate the behaviour of existing Samplers. This design of this Sampler is described next.

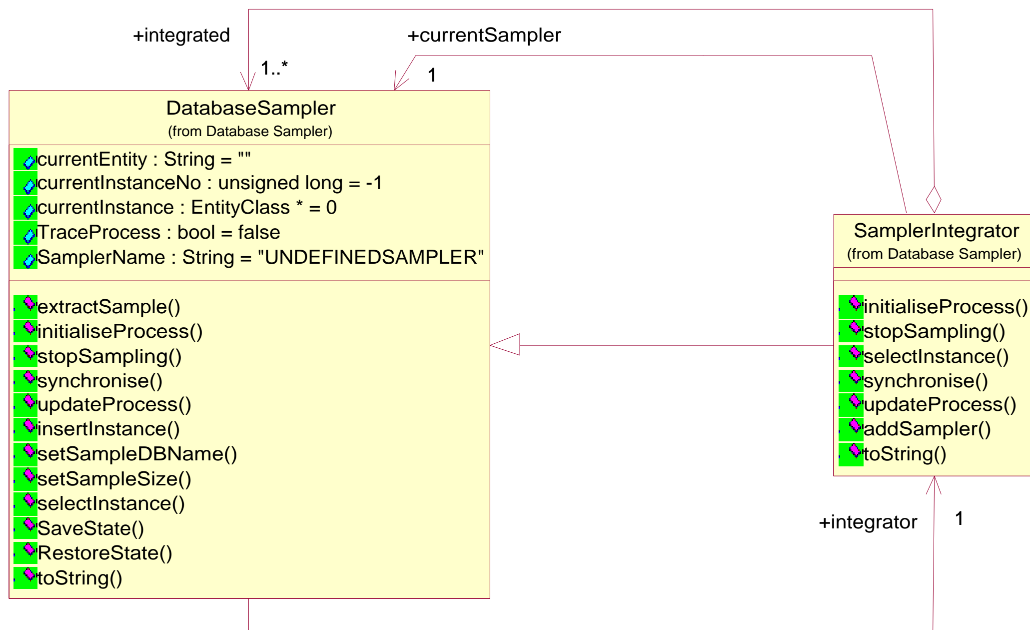

Fig. 7.6: UML Class Diagram for the Design of SamplerIntegrator

7.5.1 Design of *SamplerIntegrator*

The Database Sampler Integration Mechanism, or DaSIM, for aggregating the behaviour of Database Samplers which adhere to the Sampling Protocol analysed in Section 7.3 relies on a specific Database Sampler type called `SamplerIntegrator`. Fig. 7.6 shows a UML [48] Class Diagram for the design of `SamplerIntegrator`. This Sampler contains the list of all (instances of) Database Samplers to be integrated, and uses the fact that each of them adheres to the Sampling Protocol to reach a database which satisfies the sampling criteria of all those Samplers simultaneously. The set of Samplers being integrated could all be of the same type (e.g. several different instances of `SamplerICG`), as will be illustrated in Appendix A.

As shown in Fig. 7.6, `SamplerIntegrator` inherits from class `DatabaseSampler`, as does any other Sampler that forms part of a CoDaST. Since all Samplers, basic and integrated, implement the same interface several levels of integration are possible. That is, when an instance

of `SamplerIntegrator` integrates a set of Samplers, it does not need to know whether they are Basic Database Samplers or other instances of `SamplerIntegrator`. This design leads to a very flexible, transparent and incremental process of building comprehensive Database Samplers that reflect all the complexity of the Source Database.

In Fig. 7.6, association termed `Integrated` represents the list of Samplers to be integrated by a particular instance of `SamplerIntegrator`, using what is known as *Delegation* [61] in object-oriented systems design. In particular, it applies a design pattern known as *Command* [27]. `CurrentSampler` refers to the concrete Sampler, of association `Integrated`, that performed the very last insertion into the Sample Database, that is, the one to be kept consistent by other Samplers. Finally, association named `Integrator` represents the association with an instance of `SamplerIntegrator` that is integrating each particular instance of `DatabaseSampler` with other Samplers, if any. This association is used by `DatabaseSampler` to implement `Synchronise()`, as discussed in the previous Section.

In summary, assume a set of Samplers $Samp_1, \ldots, Samp_n$ are to be integrated. Using the design of `SamplerIntegrator` of Fig. 7.6, the resulting Sampler would iteratively construct a Sample Database as follows. Starting with an empty database, SDB_0, it would use $Samp_1$ to perform the initial insertion(s) (refer to Section 4.3), leading to a new Sample Database SDB_1. Then it would iterate through all other Samplers, $Samp_2, \ldots, Samp_n$, from association `Integrated` to maintain this insertion consistent, resulting in a set of databases $SDB_2, \ldots, SDB_n$. Each SDB_i would result from keeping the very last insertion of $Samp_{i-1}$ consistent with the criteria of $Samp_i$. This would be a recursive process, as database SDB_i would also need to be kept consistent with the very last insertion of $Samp_{i-2}$, then with $Samp_{i-3}$, and so on, each step resulting in a new database. This process leads to a set of increasingly larger databases $SDB_0 \subset SDB_1 \subset \ldots \subset SDB_m$ with the last one, SDB_m, consistent according to the criteria of all Samplers $Samp_1, \ldots, Samp_n$.

The design of `SamplerIntegrator` explained in this Section has been tested integrating two Samplers which extract a Sample of a database according to two orthogonal criteria:

Sampler with ICG and FD Samples a multi-table database so that the resulting Sample satisfies sets of functional dependencies and referential integrity constraints simultane-

ously.

Figure 7.7 shows a UML Sequence Diagram [48] with a possible execution of the Sampling Process being performed by a Database Sampler resulting from this integration, that is, integrating `SamplerICG` and `SamplerFD`. Assume, for the sake of simplicity, that the Insertions Chain Graph used by this instance of `SamplerICG` represents a set of referential integrity constraints. Each arrow in a Sequence Diagram represents a call (or return of a call) that occurs during execution. How `SamplerIntegrator` aggregates the behaviour of both samplers is shown in this diagram in arrow numbers 8 through 30. Assume that `SamplerFD` performs the initial selection (arrows 8 and 9), which is to be kept consistent by, according to the Sampling Protocol, calling method `Synchronise()` (arrow 11). This method is implemented in `DatabaseSampler`, and it results in method `UpdateProcess()` being called for the appropriate instance of `SamplerIntegrator`. Since `SamplerIntegrator` *knows* that the initial insertion was performed by `SamplerFD`, its `UpdateProcess()` method (arrow 12) calls `UpdateProcess()` in `SamplerICG` (arrow 13). This will maintain the last insertion consistent with regard to a referential integrity constraint, just as if it had been inserted by `SamplerICG` itself. After each of the insertions performed by `SamplerICG` it must also call `Synchronise()` (arrows 15 and 25). This will lead `SamplerIntegrator` to call `UpdateProcess()` in `SamplerFD` (arrow 18), in order to maintain `SamplerICG`'s insertions consistent with `SamplerFD`'s criteria. Now the process is repeated, since whenever `SamplerFD` inserts a tuple (e.g. arrow 19) it will also call `Synchronise()` (arrow 20). This will lead `SamplerIntegrator` to call `UpdateProcess()` of `SamplerICG` (arrow 23) which takes the process back to the same state it was in arrow 13. This process will, eventually, terminate because one of the required insertions will already be in the Sample Database. Since this insertion does not have to be performed `Synchronise()` is not called (arrows 28 and 29) and, therefore, the chain of calls terminates (arrow 30). If the minimum Sample size has been reached, the process terminates, otherwise a new initial insertion (arrow 8) will be required. Note that, in order to determine whether sampling can terminate, the implementation of method `StopSampling()` in `SamplerIntegrator` (arrow 31) must delegate this decision to both samplers, `SamplerFD` (arrow 32) and `SamplerCIG` (arrow 34), and only if both of them

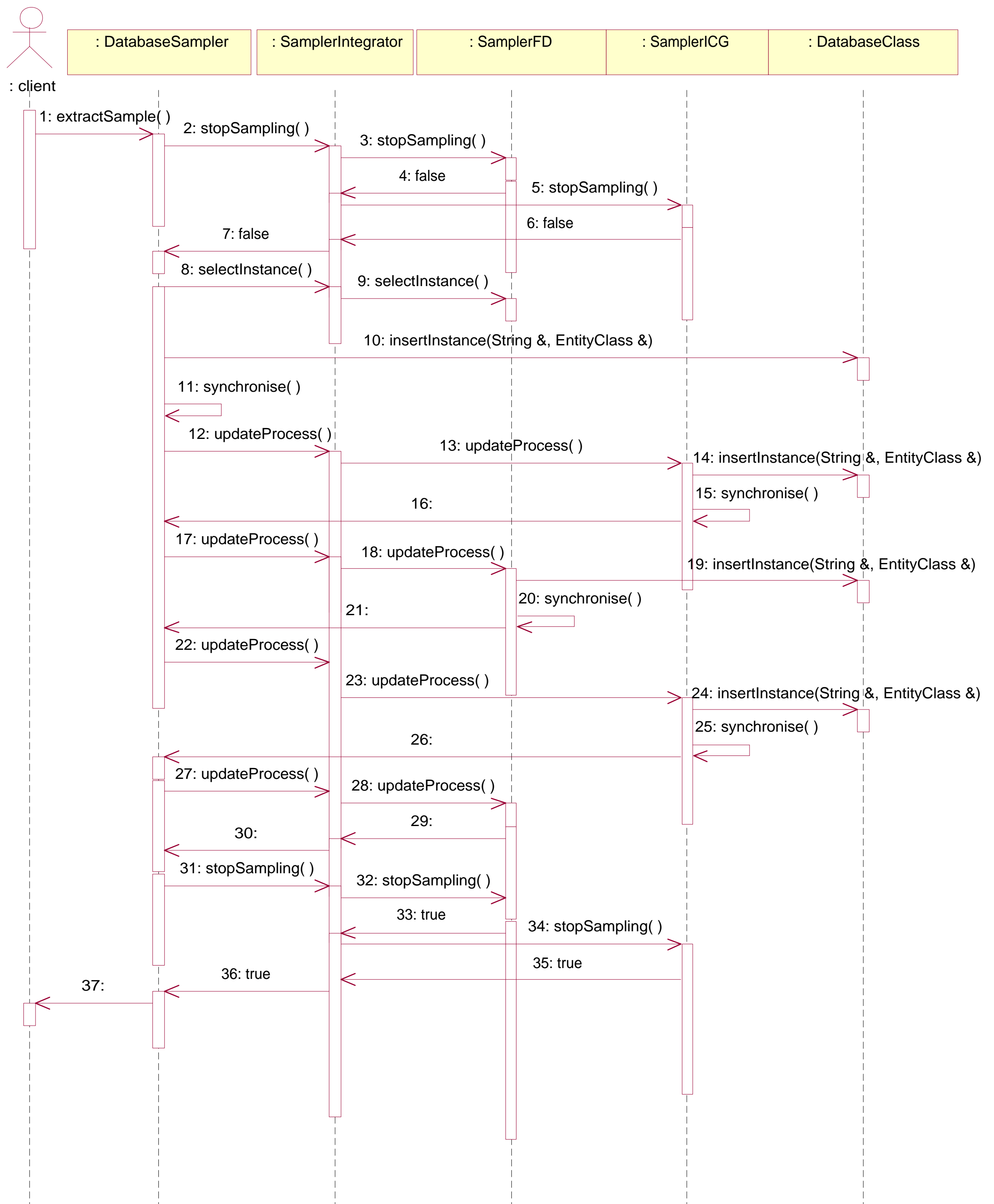

Fig. 7.7: Sampler Integrator Execution Example

are finished according to their stopping criteria (arrows 33 and 35) will sampling stop (arrow 36). In order to simplify the resulting diagram, only calls to `Syncrhonise()` (and its consequences) have been shown as these are the most relevant to Sampler integration. According to the Sampling Protocol, however, for each call to `Syncrhonise()` a Sampler would also call its own implementation of `UpdateProcess()`.

7.6 Implementation of CoDaST

CoDaST has been implemented using the C++ programming language [60] and tested on a Windows NT platform. The previous Section, together with the description of the algorithms of Chapter 5, detailed the most relevant workings of CoDaST. Only one element of its implementation is left to be analysed. It regards how the consistency criteria to be used for sampling is defined and given as input to this Sampling Tool. The next Subsection describes how this problem has been solved in this particular implementation.

7.6.1 Defining the Consistency Criteria

Parsing technology has been used to easily define the consistency criteria that the resulting Sample Database must satisfy. A language for defining an Insertions Chain Graph (ICG) has been derived from its formal definition as studied in Section 5.5.2. A complete example of the use of this language, used to sample the School Reference Database, is shown in Section A.3.

A file containing the definition of an ICG, according to this language, is an input to CoDaST. This file is parsed and the corresponding ICG is returned and used for sampling, as described in Section 5.5. The use of a parser, as opposed to hard-coding the definition for a particular ICG, makes the tool more usable since the consistency criteria definition can be easily changed. As mentioned in Chapters 4 and 6 a likely consequence of consistent sampling is a better understanding of the semantics of the Source Database, and may require some iterations in order to define the appropriate consistency criteria to be used (see also Section 2.4.5). Therefore, an easy and flexible definition of the consistency criteria is imperative.

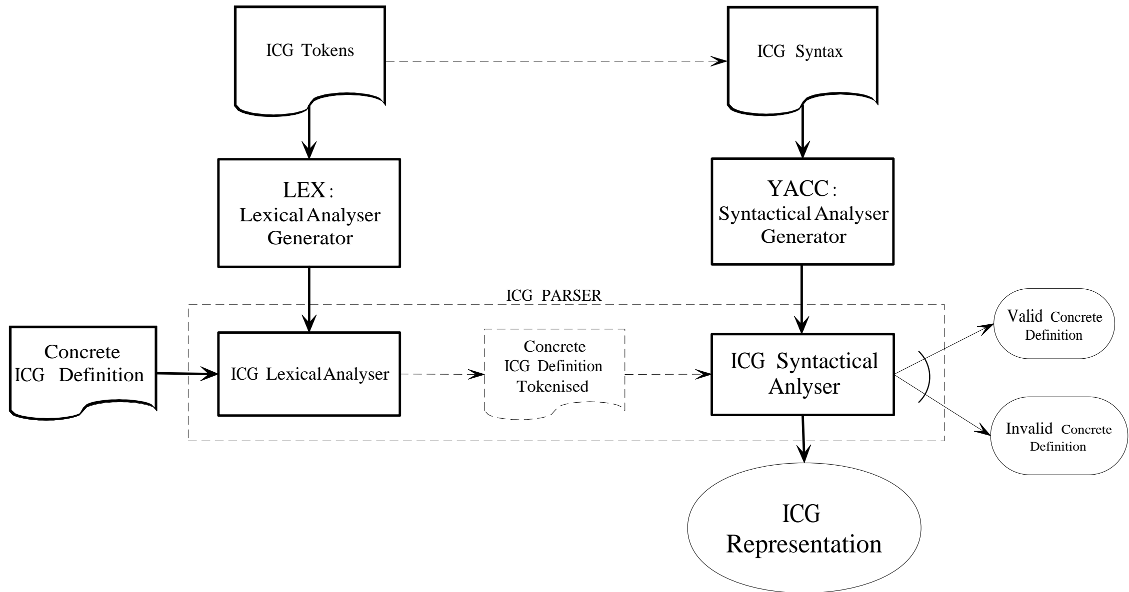

Fig. 7.8: Using LEX/YACC to Generate an ICG Parser

The parser required to implement this functionality has been generated using lexical and syntactical analyser generators, in this particular case LEX and YACC respectively [42]. Fig. 7.8 illustrates how LEX and YACC are used for this purpose. Briefly, the lexical analyser is generated by providing LEX with a definition of the *tokens* of the language, in the case of an ICG this refers to the tokens needed to define an ICG, e.g. the arrow types, the entity names involved in each arrow. The output of LEX is the ICG lexical analyser. In this case this analyser is implemented as a C++ class [60] that takes a file with a definition of an ICG and returns the list of tokens found in this definition (or reports an error if an unknown token is found). Similarly, the syntactical analyser is generated by providing YACC with a grammar [37] for the syntax that defines what is a well-formed (or valid) ICG definition. This syntax must be based on the tokens defined in the lexical analyser. The output of YACC is, as above, a C++ class that takes the list of tokens found by the lexical analyser in the input file that contains the definition of an ICG and returns whether this definition is valid according to the given syntax. Additionally, parsing can also have some other consequences, in this case the construction of the appropriate Insertions Chain Graph. This representation of the ICG in main memory is the one used for sampling in the implementation of the algorithm described

in Section 5.5. CoDaST includes the two C++ classes generated by LEX and YACC (not the ICG syntax itself).

An Insertions Chain Graph can define many different constraints, therefore developing a parser manually, without tools like LEX and YACC, would have required a very significant and error-prone effort. Additionally, the syntax for defining an ICG can now be easily changed and the analysers re-generated. Simple changes could require much effort in a parser implemented manually.

The parser described in this Section has been extended in the current implementation of CoDaST to allow the definition of several Insertions Chain Graphs simultaneously, as well as any one of the other Sampler types available in CoDaST, i.e. `SamplerFD`, `RandomSampler`, etc. This extension follows naturally from the description given here. Refer to Appendix A for concrete examples.

7.7 Summary

This Chapter has presented the design of a prototype of a Consistent Database Sampling Tool (CoDaST) which implements the concepts developed in previous Chapters.

The requirements for CoDaST has been identified. The main requirement regards the construction of sampling modules, referred to as *Database Samplers*, which sample a database according to particular criteria, and that can inter-operate with each other. The result of this integration is expected to be a new sampling module that samples the database according to several criteria simultaneously. The integration mechanism, DaSIM, has been described, as well as a Sampling Protocol which is followed by all Samplers, in order to allow for their integration. In the tool reported here, DaSIM has been implemented in one particular type of Sampler. The design of this Sampler, referred to as Sampler Integrator, has been detailed.

Chapter 8

Conclusions

8.1 Review of this Thesis

This thesis has investigated how prototype databases can be constructed using domain-relevant data values in order to support the development of data-intensive applications. A prototype database populated with domain-relevant data values has been referred to in this thesis as Sample Database. When using synthetic data values the resulting prototype database has been called Test Database.

This thesis has started by analysing the different applications where prototype databases in general, and Sample Databases in particular, could be used. Although the focus of this work has been on how prototype databases can be developed to support the software development process (e.g. requirements analysis, testing, training), other applications have also been identified (e.g. data mining, approximate query evaluation) where a Sample Database is used during production as opposed to during development. Justifying the need for prototype databases and setting a clear distinction between Sample and Test Databases have been the main goals of Chapter 2.

A literature review of existing work on prototype database constructions methods has been the focus of Chapter 3. A brief overview of the area of software prototyping has been provided, since this is the main motivation for prototyping a database. In this context a particular type of software project has been identified, the so-called *data and knowledge intensive systems*

[31]. The proposed development process for such systems considers database prototyping as central to the process, and it particularly identifies the need for Sample Databases, as opposed to Test Databases.

All database prototyping approaches found during this research result in databases populated with synthetic values, that is, Test Databases. The use of domain-relevant data values has been generally ignored by existing researchers, with rare exceptions [50] which allow for limited use of predefined sets of data values to be used as domains for some attributes. Of particular concern in this thesis has been the enforcement of complex semantics in a prototype database, in terms of the satisfaction of sets of integrity constraints. Inclusion and functional dependencies are the most commonly used sets of integrity constraints in existing approaches. More complex sets of constraints are usually not supported by existing methods, with the exception of [49] which is based on using a subset of first-order-logic (i.e. range-restricted formulas) to define the set of integrity constraints to be enforced. A framework has been developed to compare existing database prototype approaches based on these two criteria, namely the origin of the data and the semantic content enforced in the resulting prototype. This framework classifies existing approaches according to these criteria and identifies to which application areas (as in Chapter 2) each of them is better suited.

Identifying the limitations of existing approaches (i.e. synthetic values and simple semantics) and describing a framework where they can be evaluated has been the outcome of Chapter 3.

Populating a prototype database with domain-relevant data values and enforcing complex integrity constraints has been termed in this thesis *Consistent Database Sampling.* The problems involved in Consistent Database Sampling have been identified in Chapter 4. These include how to identify the chain of insertions that are required in order to maintain consistency according to each particular type of integrity constraints. An abstract Process for consistently sampling from a database has also been defined, in order to identify the issues to be addressed independently of which particular types of integrity constraints are being enforced.

The Consistent Database Sampling Process analysed in Chapter 4 underlies the study

presented in Chapter 5, which describes how to sample a database according to concrete types of integrity constraints. It has been shown that when a new integrity constraint type is to be considered for sampling: (1) a representation for this type of integrity constraint must be developed so that it can be used to sample a database, identifying the appropriate insertions to be performed; and (2) an algorithm that, using such representation, extracts a Sample consistent according to this type of integrity constraints must be developed. Firstly, a group of integrity constraint types have been represented using a common mechanism, termed Insertions Chain Graph (ICG). This is a suitable representation for sampling purposes of inclusion dependencies, cardinality constraints, generalisation constraints, and subset dependencies. Functional dependencies have been treated separately because an ICG could not describe them, and a so-called Agreements Table has been used to include functional dependencies in database sampling. Algorithms based on both representations have been described. Finally, random sampling has also been considered as a possible sampling strategy when no particular integrity constraint needs to be enforced.

A formal semantics for the consistent database sampling process has been presented in Chapter 6. The formalism used for these purposes has been that of Denotational Semantics. This Chapter has also characterised a type of integrity constraint, referred to as Sampling-Irrelevant, that can be ignored for sampling purposes as this cannot lead to an inconsistent Sample Database. This type of constraints has been shown to be undecidable.

Finally Chapter 7 has described a prototype of a sampling tool implementing the algorithms and concepts discussed in Chapter 5 and the Sampling Process of Chapter 4. This Consistent Database Sampling Tool, or CoDaST, is built up from a number of sampling modules, termed Database Samplers, each one extracting a Sample from a database according to different criteria, including the satisfaction of functional dependencies, inclusion dependencies, etc. Using what has been referred to as Database Sampler Integration Mechanism, or DaSIM, several Database Samplers can be integrated in order to extract a Sample satisfying all their criteria simultaneously. Several levels of integration are possible, that is, a Database Sampler resulting from integrating two Database Samplers can be integrated with a third one. This incremental construction of a Database Sampler that contains all semantic com-

plexity of the Source Database leads to a flexible and extensible design. The description of this incremental approach to build a complex Database Sampler is the main contribution of Chapter 7.

When CoDaST is used to extract a Sample from a database, the appropriate set of integrity constraints to be considered must be identified. This may require reverse-engineering the database and, as stated in Chapter 4, this step is out of the scope of the work presented here. However, it has also been noted that a likely consequence of consistently database sampling is a better understanding of the semantics of the database since a (small) Sample may expose missing or undesirable dependencies if required constraints are left out of the Process. Once the appropriate set of constraints has been identified, it must be given as input to CoDaST. A flexible mechanism has been developed to allow the user to specify the particular constraints to be used. CoDaST will then create a new database with the same schema as the Source Database and will populate it with data values from that database, enforcing the specified set of constraints. After that the resulting Sample Database can be used for prototyping, data mining, etc. as analysed in Chapter 2.

8.2 Summary of Thesis Contributions

The research contributions of this thesis can be summarised as follows:

- A discussion of the state of the art in database prototyping for data-intensive applications development, and the identification of a need for approaches which enforce complex constraints and populate the resulting database with domain-relevant data.
- The development of a framework for evaluating database prototyping approaches, and the identification of their application areas.
- The study of the process of sampling a database in order to enforce complex integrity constraints in the resulting database.
- The analysis of concrete integrity constraints types, widely used in practice, and how they can be enforced during sampling.

- A formal study of the process of consistently sampling from a database, and the characterisation of a type of constraint particularly relevant for sampling purposes.
- An implementation of a tool to demonstrate that the concepts developed in this thesis can be applied in practice. Additionally, this tool allows for the incremental construction of complex sampling modules by seamlessly integrating multiple sampling modules.

8.3 Future Work

The work developed in this thesis could be extended in different directions. This Section outlines some of them. First it analyses possible extensions to the work presented in Chapter 5 regarding how integrity constraints can be included in the sampling process, and also which constraint types can be considered. An alternative area for future development could focus on CoDaST, the prototype of a sampling tool that implements the work presented here and that was reported in Chapter 7. A significant amount of future work can be identified in relation to the theoretical framework described in Chapter 6. Finally, some practical consideration about the validation of this work will also be discussed.

8.3.1 Integrity Constraints

Future work could focus on adding new constructs to the language used to define an Insertions Chain Graph (ICG) so that it gains in expressiveness, particularly regarding to the limitations outlined in Section 5.5.5. Another direction for future research 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 provide a more compact representation of the semantics of the Source Database relevant for sampling. It will also 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.

As outlined in Section 5.6.4, the algorithm to extract a minimal size Sample that satisfies a set of functional dependencies (Σ) is suboptimal, in the sense that a Sample of smaller size

than the one generated by the algorithm could exist in some circumstances. This is due to the fact that the algorithm considers only local information to decide which tuples must be inserted into the Sample. How global information could be used to ensure minimal Sample size, without excessively increasing its complexity, is still to be investigated.

Theorem 5.6.3 proved that sampling with a superset of Σ is not always possible. A future line of work could investigate in which cases it is actually possible to extract an Armstrong relation for a superset of Σ.

Finally, it should be noted that the concept of Armstrong relation is not exclusively related to functional dependencies, but it has been extended to a more general type of dependencies [26]. Database sampling with this type of dependencies, in order to extract Armstrong relations for them, would be an interesting area for future work.

8.3.2 CoDaST

As more integrity constraints types are included in the sampling process (see Section 8.3.1) more sampling modules, Database Samplers, should also be added to CoDaST.

A more sophisticated user interface could be developed for CoDaST. A graphical interface for defining an Insertions Chain Graph is more appropriate than using a text file, as in the current implementation.

8.3.3 Theoretical Development

Theorem 6.4.1 stated that if a constraint is not satisfied by an empty database, then it must be Sampling-Relevant. As outlined in Section 6.4 it is possible, however, for a Sampling-Relevant integrity constraint to be satisfied by an empty database. This rises the question of how *interesting* this property is in practice, that is, how many Sampling-Relevant integrity constraints are actually not satisfied by an empty database. Finding a more practical characterisation of Sampling-Relevant integrity constraints is also an area for future research. A similar problem as minimalising an ICG could be proposed in case of this type of constraints. That is, if an appropriate characterisation of integrity cosntraints was found, it may be possible to *minimalise* a set of Sampling-Relevant integrity constraints.

Regarding to the undecidability of Sampling-Relevance. A common approach to deal with undecidability would be to search for syntactic restrictions that ensure Sampling-Relevance. For example, the domain-relevance of relational calculus queries described in Section 6.4 is also undecidable, and some syntactic restrictions of first-order-logic (e.g. range-restricted, safe) have been proposed to ensure that a query which satisfies these syntactic (therefore, decidable) restrictions is domain-independent. A similar approach could be investigated for Sampling-Relevant integrity constraints.

8.3.4 Objective Evaluation

This thesis has argued that domain-relevant data is more appropriate to populate a prototype database than synthetic data. This is essentially a qualitative argument and, although it is highly intuitive, applying the developments of this thesis to real development projects may be the only proof of whether the resulting software is of better quality.

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] S. Abiteboul and V. Vianu. Expressive power of query languages. In J.D. Ullman, editor, *Theoretical Studies in Computer Science*, pages 207–251. Academic Press, 1992.

[3] S. Acharya, P.B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the International Conference on Management of Data (SIGMOD 2000)*, pages 487–498. ACM Press, 2000.

[4] D. Aebi. Data re-engineering: A case study. In R. Manthey and V. Wolfengagen, editors, *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)*, Electronic workshops in computing, pages 305–310. Springer-Verlag, 1997.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the International Conference on Management of Data (SIGMOD 1994)*, pages 487–499. ACM Press, 1994.

[6] P.H. Aiken. *Data Reverse Engineering : Slaying the Legacy Dragon.* McGraw-Hill, 1996.

[7] L. Allison. *A Practical Introduction to Denotational Semantics.* Cambridge University Press, 1986.

[8] W.W. Armstrong. Dependency structures of data base relationships. In J.L. Rosenfeld, editor, *Information Processing 74 (Proceedings of IFIP Congress 74)*, pages 580–583. IFIP, North-Holland, August 1974.

[9] 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.

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

[11] C. Beeri, M. Dowd, R. Fagin, and R. Statman. On the structure of armstrong relations for functional dependencies. *Journal of the ACM*, 31(1):30–46, January 1984.

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

[13] B. Beizer. *Black-Box Testing : Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[14] J. Bisbal and J. Grimson. Database prototyping through consistent sampling. In *the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR'2000)*. Scuola Superiore Guglielmo Reiss Romoli (SSGRR), August 2000.

[15] J. Bisbal and J. Grimson. Database sampling with functional dependencies. *Information and Software Technology*, June 2000. Submitted.

[16] J. Bisbal and J. Grimson. Generalising the consistent database sampling process. In B. Sanchez, N. Nada, A. Rashid, T. Arndt, and M. Sanchez, editors, *Proceedings of the Joint meeting of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000) and the 6th International Conference on Information Systems Analysis and Synthesis (ISAS'2000)*, volume II - Information Systems Development, pages 11–16. International Institute of Informatics and Systemics (IIIS), July 2000.

[17] J. Bisbal, D. Lawless, R. Richardson, B. Wu, J. Grimson, V. Wade, and D. O'Sullivan. An overview of legacy information systems migration. In Bob Werner, editor, *Proceedings of the Joint 1997 Asia Pacific Software Engineering Conference and International*

Conference in Computer Science (APSEC'97/ICSC'97), pages 529–530. IEEE Computer Society Press, December 1997.

[18] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, September/October 1999.

[19] J. Bisbal, B. Wu, D. Lawless, and J. Grimson. Building consistent sample databases to support information system evolution and migration. In G. Quirchmayr, E. Schweighofer, and T. J.M. Bench-Capon, editors, *Proceedings of the 9th International Conference on Database and Expert Systems Applications (DEXA'98)*, volume 1460 of *Lecture Notes in Computer Science*, pages 196–205. Springer-Verlag, 1998.

[20] G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.

[21] M. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufann, 1995.

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

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

[24] E.T. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[25] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Publishing Company, seventh edition, 1999.

[26] R. Fagin. Horn clauses and database dependencies. *Journal of the ACM*, 29(4):952–985, October 1982.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[28] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice-Hall, 1991.

[29] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 9, pages 265–306. Kluwer Academic Publishers, 1998.

[30] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the International Conference on Management of Data (SIGMOD 1994)*, pages 243–252. ACM Press, 1994.

[31] G. Guida, G. Lamperti, and M. Zanella. *Software Prototyping in Data and Knowledge Engineering.* Kluwer Academic Publishers, November 1999.

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

[33] P. Haumer, K. Pohl, and K. Weidenhaupt. Requirements elicitation and validation with real world scenes. *IEEE Transactions on Software Engineering*, 24(12):1036–1054, December 1998. Special Issue on Scenario Management.

[34] S. Hekmatpour and D. Ince. *Software Prototyping, Formal Methods and VDM.* Addison-Wesley, 1987.

[35] J.M. Hellerstein, P.J. Haas, and H.J. Wang. Online aggregation. In *Proceedings of the International Conference on Management of Data (SIGMOD 1997)*, pages 171–182. ACM Press, 1997.

[36] R. J. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[37] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[38] J. Kivinen and H. Mannila. The power of sampling in knowledge discovery. In *Proceedings of the 1994 ACM SIGMOD-SIGACT Symposium on Principles of Database Theory (PODS'94)*, pages 77–85, 1994.

[39] D.E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997.

[40] A.M. Law and W.D. Kelton. *Simulation Modeling & Analysis*. McGraw-Hill, second edition, 1991.

[41] D. Lawless. Legacy information systems migration: A methodology and its trial implementation. Msc. thesis, Computer Science Department, Trinity College Dublin, June 1999.

[42] J.R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, second edition, 1995.

[43] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice-Hall, second edition, 1998.

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

[45] H. Mannila and K.-J. Räihä. Small armstrong relations for database design. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS'85)*, pages 245–250, 1985.

[46] H. Mannila and K.-J. Räihä. Dependency inference. In P.M. Stocker, W. Kent, and P. Hammersley, editors, *Proceedings of 13th International Conference on Very Large Data Bases (VLDB'87)*, pages 155–158. Morgan Kaufmann, 1987.

[47] H. Mannila and K.-J. Räihä. *The Design of Relational Databases*. Addison-Wesley, 1992.

[48] F. Martin and S. Kendall. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[49] 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.

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

[51] F. Olken. *Random Sampling from Databases.* PhD thesis, University of California, April 1993.

[52] F. Olken and D. Rotem. Random sampling from database files: A survey. In Z. Michalewicz, editor, *Proceedings of the Fifth International Conference on Statistical and Scientific Database Management (SSDBM'90)*, volume 420 of *Lecture Notes in Computer Science*, pages 92–111. Springer-Verlag, 1990.

[53] D. O'Sullivan, R. Richardson, J. Grimson, B. Wu, J. Bisbal, and D. Lawless. Application of case based reasoning to legacy system migration. In *Proceedings of the 5th German Workshop on Case-Based Reasoning - Foundations, Systems, and Applications*, pages 225–234. Centre for Learning Systems and Applications, Department of Computer Science, University of Kaiserslautern, March 1997.

[54] C.R. Palmer and C. Faloutsos. Density biased sampling: An improved method for data mining and clustering. In *Proceedings of International Conference on Management of Data (SIGMOD 2000)*, pages 82–92. ACM Press, 2000.

[55] R.S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fourth edition, 1997.

[56] S. Seshadri and J. F. Naughton. Sampling issues in parallel database systems. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Advances in Database Technology - Proceedings of the 3rd International Conference on Extending Database Technology (EDBT'92)*, volume 580 of *Lecture Notes in Computer Science*, pages 328–343. Springer-Verlag, 1992.

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

[58] P. Stevens and R. Pooley. Systems reengineering patterns. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering (ACM-SIGSOFT)*, volume 23 of *Software Engineering Notes*, pages 17–23. ACM Press, 1998.

[59] J. Stoy. *Denotational Semantics : the Scott-Strachey Approach to Programming Language Theory.* M.I.T Press, 1977.

[60] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, special (third) edition, 2000.

[61] C. Szyperski. *Component Software.* Addison-Wesley, 1998.

[62] R.D. Tennent. Denotational semantics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3: Semantic Structures, pages 169–322. Oxford University Press, 1994.

[63] L. Tucherman, M.A. Casanova, and A.L. Furtado. The chris consultant - a tool for database design and rapid prototyping. *Information Systems*, 15(2):187–195, 1990.

[64] J.D. Ullman. *Principles of Database System.* Computer Science Press, 1980.

[65] J.S. Vitter. An efficient algorithm for sequential random sampling. *ACM Transactions on Mathematical Software*, 13(1):58–67, March 1987.

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

[67] J. Widom. A denotational semantics for the starburst production rule language. *SIGMOD Record*, 21(3):4–9, September 1992.

[68] D.P. Wood and K.C. Kang. A classification and bibliography of software prototyping. Technical Report CMU/SEI-92-TR-13, Carnegie Mellon University, Software Engineering Institute, October 1992.

[69] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, D. O'Sullivan, and R. Richardson. Legacy systems migration – a method and its tool-kit framework. In Bob Werner, editor, *Proceedings of the Joint 1997 Asia Pacific Software Engineering Conference and International Conference in Computer Science (APSEC'97/ICSC'97)*, pages 312–320. IEEE Computer Society Press, December 1997.

[70] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wade, D. O'Sullivan, and R. Richardson. Legacy system migration : A legacy data migration engine. In Petr Cervinka, editor, *Proceedings of the 17th International Database Conference (DATASEM'97)*, pages 129–138. Czechoslovak Computer Experts, October 1997.

[71] 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 B. Werner, editor, *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems (ICECCS'97)*, pages 200–205. IEEE Computer Society Press, September 1997.

[72] M. Yannakakis. Perspectives on database theory. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 224–246. IEEE Computer Society Press, 1995.

[73] 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 (ER'93)*, volume 823 of *Lecture Notes in Computer Science*, pages 85–98. Springer-Verlag, 1993.

[74] P. Zave. Classification of research efforts in requirements engineering. *ACM Computing Surveys*, 29(4):315–321, December 1997.

Appendix A

Trial

A.1 Introduction

This Appendix describes how sampling can be applied to one concrete instance of the School Reference Database analysed in Section 5.2.

Section A.2 presents an example of a consistent instance of the reference database. Section A.3 illustrates how the set of constraints the database must satisfy can be defined for sampling purposes, using one of the mechanisms investigated in this thesis, an Insertions Chain Graph. Section A.4 shows a possible result of sampling this instance according to the specified set of constraints. The final Section provides a concrete example of how `SamplerIntegrator` can be used to integrate multiple Database Samplers, as described in Chapter 7.

A.2 A Consistent Instance of the Reference Database

Fig. 5.2 showed the logic relational design for the School Reference Database that could be obtained from the Extended Entity-Relationship schema of Fig. 5.1. An example of an instance of this database satisfying all the required constraints is given in Tables A.1 to A.10.

Table A.1: Instance of **Students**

student	**year**
albert	4
alice	4
angelique	2
ann	1
benjamin	1
david	3
frank	1
jim	2
annete	5
peter	3
rosy	3
stephan	2
willow	5

Table A.2: Instance of **Courses**

course	**year**	**teacher**
analysis 1	1	phillip
geometry	1	anca
algebra	1	robert
analysis 2	2	phillip
programming	2	diana
analysis 3	3	phillip
data mining	3	mary
calculus	4	sofia
logic	4	tom
languages	5	diana
automata	5	mark
cybernetics	5	diana

Table A.3: Instance of **Teachers**

teacher	**tel**	**dept**
anca	452	mathematics
diana	551	computer science
mark	552	computer science
mary	644	statistics
phillip	453	mathematics
robert	454	mathematics
sofia	645	statistics
tom	455	mathematics

Table A.4: Instance of **Depts**

name	director
computer science	diana
mathematics	phillip
statistics	mary

Table A.5: Instance of **Rooms**

room	size	dept	floor
c1	35	computer science	1
c2	12	computer science	2
m1	20	mathematics	2
m2	15	mathematics	3
s1	30	statistics	3
s2	18	statistics	4

A.3 Consistency Criteria Definitions in the School Database

In order to define the consistency criteria needed to Sample the reference database the Insertions Chain Graph shown in Fig. 5.3 had to be given as input to CoDaST, as explained in Section 7.6. The definition used in this case was the following:

```
1
2 -- Instructions to extract a Sample from database "school.mdb"
3 -- It defines an Insertions Chain Graph (ICG) with all the constraints
4 --  in this database.
5
6 SAMPLE "c:\my documents\school.mdb" WITH SHOW
7 (
8 SAMPLERICG AS aSamplerICG:                                 -- Sampler with an Insertions Chain Graph
9
10 EXAMS(1):            [TOTAL, STUDENTS, student, name];       -- Each Exam is associated to a Student
11 FULLTIMESTUDENTS(0): [QUANTIFIED, EXAMS, student, student, 5];
12                                    -- A FullTimeStudent must have taken at least five exams
13 EXAMS:               [PARTIAL, FULLTIMESTUDENTS, student, student,
14                        (SELECT COUNT(*) FROM EXAMS WHERE student=%s)>=5];
15                                          -- If an student has followed more than five exams
16                                          --    s/he must be recorded as being fulltime
17 FULLTIMESTUDENTS:    [TOTAL, TEACHERS, tutor, name];   -- All FullTimeStudents have a Teacher as tutor
18 EXAMS:               [TOTAL, COURSES, course, course]; -- All Exams are associated with a Course
19 COURSES (1):         [TOTAL, TEACHERS, teacher, name]; -- All Courses have one and only one Teacher
20 PREREQUISITS(1):     [TOTAL, COURSES, course, course]   -- Prerequisits relate a Course with an
21                      [TOTAL, COURSES, previous, course];--    the Courses that must have been passed
22 TIMETABLE(1):        [TOTAL, COURSES, course, course];-- TimeTable entries are associated with Courses
23 TEACHERS(1):         [TOTAL, DEPTS, dept, name];        -- All Teachers belong to only one Department
24 DEPTS(1):            [TOTAL, TEACHERS, director, name]; -- Each Department has one director
25 ROOMS(1):            [TOTAL, DEPTS, dept, name];        -- Each Room belongs to one Department
26 TIMETABLE:           [TOTAL, ROOMS, room, room];        -- TimeTable entries are associated with Rooms
```

Table A.6: Instance of **Exams**

student	**course**	**grade**
albert	analysis 1	28
albert	geometry	27
albert	algebra	26
albert	analysis 2	28
albert	programming	30
albert	analysis 3	27
albert	data mining	30
alice	analysis 1	24
alice	geometry	28
alice	algebra	30
alice	analysis 2	26
alice	analysis 3	30
angelique	analysis 1	28
angelique	geometry	30
angelique	algebra	30
david	analysis 1	23
david	algebra	24
david	programming	30
jim	algebra	25
annette	analysis 1	27
annette	geometry	26
annette	analysis 2	30
annette	analysis 3	30
annette	data mining	28
peter	analysis 1	28
peter	analysis 2	30
peter	programming	27
rosy	analysis 1	26
rosy	geometry	27
rosy	algebra	30
rosy	analysis 2	28
stephan	geometry	27
stephan	algebra	24
willow	analysis 1	28
willow	analysis 2	30
willow	analysis 3	27
willow	calculus	30
willow	cybernetics	27

Table A.7: Instance of **Pre-requisites**

courses	**previous**
analysis 2	analysis 1
analysis 3	analysis 2
automata	programming
calculus	analysis 3
cybernetics	calculus
languages	programming
logic	algebra
programming	algebra

Table A.8: Instance of **Timetable**

course	**day**	**hour**	**room**
algebra	monday	9	m1
algebra	friday	11	m1
analysis 1	monday	9	m1
analysis 1	wednesday	11	m1
analysis 2	wednesday	10	m1
analysis 2	thursday	14	c1
analysis 3	wednesday	10	s1
analysis 3	thursday	15	s1
automata	wednesday	11	c1
automata	thursday	14	c1
calculus	monday	9	s1
calculus	tuesday	11	s1
cybernetics	thursday	11	s1
cybernetics	friday	9	s1
data mining	tuesday	14	s1
data mining	friday	11	s1
geometry	tuesday	10	m1
geometry	thursday	16	m1
languages	monday	14	c1
languages	wednesday	15	c1
logic	wednesday	15	c1
logic	thursday	11	c1
programming	tuesday	10	c2
programming	friday	9	c1

Table A.9: Instance of **FullTimeStudents**

student	**lecturer**
albert	anca
alice	mary
annette	robert
rosy	sofia
willow	anca

Table A.10: Instance of **Persons**

name	**status**	**city**
anca	teacher	sicily
diana	teacher	bary
mark	teacher	miland
mary	teacher	venice
phillip	teacher	bologna
robert	teacher	verona
sofia	teacher	bary
tom	teacher	rome
albert	student	rome
alice	student	naples
angelique	student	miland
ann	student	rome
benjamin	student	turin
david	student	turin
frank	student	florence
jim	student	venice
annette	student	florence
peter	student	miland
rosy	student	venice
stephan	student	miland
willow	student	turin

```
27 PERSONS(1):          [OR, (TEACHERS, STUDENTS), name, name]; -- A Person can be a Student or a Teacher
28                                                              -- or both
29 TEACHERS:            [TOTAL, PERSONS, name, name];        -- A Teacher is a Person
30 STUDENTS:            [TOTAL, PERSONS, name, name];        -- A Student is also a Person
31 TEACHERS:            [QUANTIFIED, COURSES, name, teacher, 1];-- Any Teacher must be Teaching at least
32                                                              --    one Course
33 COURSES:             [QUANTIFIED, TIMETABLE, course, course, 1]; -- Any Course must have been scheduled
34 FULLTIMESTUDENTS:    [TOTAL, STUDENTS, student, name];    -- A FullTimeStudent is an Student
35 DEPTS(1):            [QUANTIFIED, ROOMS, name, dept, 1];  -- A Department must own at least one Room
36 COURSES:             [OR,(PREREQUISITS),course,course];   -- Sample all Pre-requisits for each Course
37 );
```

First the database to be sampled is identified, and then the Sampling modules to be used during sampling must be defined. In this particular example only one Database Sampler is used. This Sampler is of type `SAMPLERICG`, as referred to in Chapter 7. Each Sampler can also have a name, which follows the keyword `AS`, in this case `aSamplerICG`. The usefulness of this name will be addressed in the next Section. The definition of the Insertions Chain Graph required by this Sampler follows after its name. In this example the ICG has been defined following the same order as in the definition of its Insertions Function in Table 5.3. The language used in this example should be clear from the formal definition of an ICG given in Section 5.5. For each edge (arrow) in the graph, its source entity is given first, followed by the minimum number of instances of this entity that should be present in the final Sample Database (see Table 5.4). If this number is not given it is assumed that it is either zero or it was given in a previous arrow definition for the same entity. Then the arrow type is identified (`TOTAL`, `QUANTIFIED`, etc). The name of the target entity is followed by the name of the attributes that link the source and the target entities respectively. For example, the first arrow definition in the above example indicates that at least one instance is required in entity `EXAMS`, and that there is a `TOTAL` arrow from `EXAMS` to `STUDENTS`. These two entities are related by attributes named `student` and `name` respectively, as it can be seen from the schema shown in Fig. 5.2. Additional information needed in some types of arrows is also specified, like for example an integer in case of `QUANTIFIED` arrows, or a condition in case of `PARTIAL` arrows.

A.4 A Consistent Sample of the Reference Database

A possible example of a consistent Sample of the database instance of Section A.2 is shown from Table A.11 to Table A.20. This Sample was extracted according to the integrity constraints defined for the reference database, as described in the previous Section using an Insertions Chain Graph. The Sample Database shown here is the result of the sampling example discussed in Section 5.5.6.

The concrete Sample being extracted depends on the random selections involved in functions `SampleInstance()` and `SampleInstance_local()`. The sequence of selections, either random or to maintain consistency, that CoDaST performed in this particular example are shown next. This will illustrate how the chain of insertions that an ICG describes has been followed in this case. For each instance inserted into the Sample Database, the name of its entity and the values for each of its attributes, in the same order as given in Fig. 5.2, are shown.

```
1
2                [EXAMS]: <annette , analysis 1 , 27.000000>
3                [STUDENTS]: <annette , 5>
4                [PERSONS]: <annette , student , florence>
5                [COURSES]: <analysis 1 , 1 , phillip>
6                [TEACHERS]: <phillip , 453 , mathematics>
7                [DEPTS]: <mathematics , phillip>
8                [ROOMS]: <m1 , 20 , mathematics , 2>
9                [PERSONS]: <phillip , teacher , bologna>
10               [TIMETABLE]: <analysis 1 , wednesday , 11 , m1>
11
12
13               [PREREQUISITS]: <calculus , analysis 3>
14               [COURSES]: <calculus , 4 , sofia>
15               [TEACHERS]: <sofia , 645 , statistics>
16               [DEPTS]: <statistics , mary>
17               [TEACHERS]: <mary , 644 , statistics>
18               [PERSONS]: <mary , teacher , venice>
19               [COURSES]: <data mining , 3 , mary>
20               [TIMETABLE]: <data mining , tuesday , 14 , s1>
21               [ROOMS]: <s1 , 30 , statistics , 3>
22               [PERSONS]: <sofia , teacher , bary>
23               [TIMETABLE]: <calculus , tuesday , 11 , s1>
24               [COURSES]: <analysis 3 , 3 , phillip>
25               [TIMETABLE]: <analysis 3 , thursday , 15 , s1>
26               [PREREQUISITS]: <analysis 3 , analysis 2>
27               [COURSES]: <analysis 2 , 2 , phillip>
28               [TIMETABLE]: <analysis 2 , thursday , 14 , c1>
29               [ROOMS]: <c1 , 35 , computer science , 1>
```

Table A.11: Sample with ICG of **Students**

student	year
annette	5

Table A.12: Sample with ICG of **Courses**

course	year	teacher
analysis 1	1	phillip
calculus	4	sofia
data mining	3	mary
analysis 3	3	phillip
analysis 2	2	phillip
cybernetics	5	diana

```
30                 [DEPTS]: <computer science , diana>
31                 [TEACHERS]: <diana , 551 , computer science>
32                 [PERSONS]: <diana , teacher , bary>
33                 [COURSES]: <cybernetics , 5 , diana>
34                 [TIMETABLE]: <cybernetics , friday , 9 , s1>
35                 [PREREQUISITS]: <cybernetics , calculus>
36                 [PREREQUISITS]: <analysis 2 , analysis 1>
```

It can be seen how sampling must start from entity `Exams` because this is the entity with the highest fan-out and lowest fan-in values in the ICG, as dicussed in Section 5.5.6. Then this initial insertion must be maintained consistent, which will trigger the first block of insertions (lines 1 to 10) shown above. After that, the Sample Database is consistent with the specified set of constraints. However, it does not satisfy the minimum Sample size requirements as defined in Section 5.5.6, according to which at least one instance of entity `Pre-Requisits` must also be present in the Sample. Therefore, one instance of this entity is randomly selected (e.g. ⟨`calculus , analysis 3`⟩). Then this instance must be maintained consistent, which will trigger the last group of insertions (lines 13 to 36).

Table A.13: Sample with ICG of **Teachers**

teacher	tel	dept
phillip	453	mathematics
sofia	645	statistics
mary	644	statistics
diana	551	computer science

Table A.14: Sample with ICG of **Depts**

name	**director**
mathematics	phillip
computer science	diana
statistics	mary

Table A.15: Sample with ICG of **Rooms**

room	**size**	**dept**	**floor**
m1	20	mathematics	2
s1	30	statistics	3
c1	35	computer science	1

Table A.16: Sample with ICG of **Exams**

student	**course**	**grade**
annette	analysis 1	27.0

Table A.17: Sample with ICG of **Pre-requisites**

courses	**previous**
calculus	analysis 3
analysis 3	analysis 2
cybernetics	calculus
analysis 2	analysis 1

Table A.18: Sample with ICG of **Timetable**

course	**day**	**hour**	**room**
analysis 1	wednesday	11	m1
analysis 2	thursday	14	c1
analysis 3	thursday	15	s1
calculus	tuesday	11	s1
cybernetics	friday	9	s1
data mining	tuesday	14	s1

Table A.19: Sample with ICG of **FullTimeStudents**

student	**lecturer**

Table A.20: Sample with ICG of **Persons**

name	status	city
annette	student	florence
phillip	teacher	bologna
mary	teacher	venice
sofia	teacher	bary
diana	teacher	bary

A.5 A Consistent Sample using *SamplerIntegrator*

This Section shows an execution example of CoDaST when several Database Samplers are used to extract a Sample from the reference database, and the specialised sampler called `SamplerIntegrator` is used to enforce all their integrity constraints simultaneously. This illustrates how the Database Sampler Integration Mechanism (DaSIM) analysed in Section 7.4 works in practice.

Assume that three different Samplers are to be integrated. For the purposes of this Section, the Database Sampler called `aSamplerICG` which was defined in the previous Section is divided into two Samplers. Refer to these Samplers as `SamplerICG_1` and `SamplerICG_2`. These two Samplers together enforce the same integrity constraints as `aSamplerICG`. Then an additional Sampler is also defined, referred to as `Sampler_Functional_Dependencies`, which samples according to a set of functional dependencies, as described in Section 5.6. The definition of these three Samplers is given next.

```
1
2 -- Instructions to extract a Sample from database "school.mdb"
3 -- It defines an Insertions Chain Graph (ICG) with HALF the constraints
4 --  in this database. Then another ICG defines the other half of the set
5 --  of constraints.
6 -- These two Samplers are integrated to achieve consistency according to
7 --  both sets of constraints.
8 -- Another Sampler is also defined, which samples according to a set of
9 --  functional dependencies. This Sampler is also integrated with the
10 --  two Samplers defined before.
11
12 SAMPLE "c:\my documents\school.mdb" WITH
13 --
14 -- Definition of the first Sampler with half the Insertions Chain Graph
15 --
16 (
17 SAMPLERICG AS SamplerICG_1:                            -- Sampler with an Insertions Chain Graph
18
```

```
19 TEACHERS(1):          [TOTAL, DEPTS, dept, name];        -- All Teachers belong to only one Department
20 DEPTS(1):             [TOTAL, TEACHERS, director, name]; -- Each Department has one director
21 ROOMS(1):             [TOTAL, DEPTS, dept, name];        -- Each Room belongs to one Department
22 TIMETABLE:            [TOTAL, ROOMS, room, room];        -- TimeTable entries are associated with Rooms
23 PERSONS(1):           [OR, (TEACHERS, STUDENTS), name, name]; -- A Person can be a Student or a Teacher
24                                                                -- or both
25 TEACHERS:             [TOTAL, PERSONS, name, name];        -- A Teacher is a Person
26 STUDENTS:             [TOTAL, PERSONS, name, name];        -- A Student is also a Person
27 TEACHERS:             [QUANTIFIED, COURSES, name, teacher, 1];-- Any Teacher must be Teaching at least
28                                                                --    one Course
29 COURSES:              [QUANTIFIED, TIMETABLE, course, course, 1]; -- Any Course must have been scheduled
30 FULLTIMESTUDENTS:     [TOTAL, STUDENTS, student, name];    -- A FullTimeStudent is an Student
31 DEPTS(1):             [QUANTIFIED, ROOMS, name, dept, 1];  -- A Department must own at least one Room
32 COURSES:              [OR,(PREREQUISITS),course,course];   -- Sample all Pre-requisits for each Course
33
34 EXAMS:;              -- Ignore these two entities
35 PREREQUISITS:;
36 )
37 --
38 -- Definition of the second Sampler with the other half of the Insertions Chain Graph
39 --
40 (
41 SAMPLERICG AS SamplerICG_2:
42
43 EXAMS(1):             [TOTAL, STUDENTS, student, name];       -- Each Exam is associated to a Student
44 FULLTIMESTUDENTS(0): [QUANTIFIED, EXAMS, student, student, 5];
45                                            -- A FullTimeStudent must have taken at least five exams
46 EXAMS:                [PARTIAL, FULLTIMESTUDENTS, student, student,
47                         (SELECT COUNT(*) FROM EXAMS WHERE student=%s)>=5];
48                                                    -- If an student has followed more than five exams
49                                                    --    s/he must be recorded as being fulltime
50 FULLTIMESTUDENTS:     [TOTAL, TEACHERS, tutor, name];   -- All FullTimeStudents have a Teacher as tutor
51 EXAMS:                [TOTAL, COURSES, course, course]; -- All Exams are associated with a Course
52 COURSES (1):          [TOTAL, TEACHERS, teacher, name]; -- All Courses have one and only one Teacher
53 PREREQUISITS(1):      [TOTAL, COURSES, course, course]   -- Prerequisits relate a Course with an
54                       [TOTAL, COURSES, previous, course];--    the Courses that must have been passed
55 TIMETABLE(1):         [TOTAL, COURSES, course, course];-- TimeTable entries are associated with Courses
56
57 PERSONS:;            -- Ignore these entities
58 TEACHERS:;
59 STUDENTS:;
60 TEACHERS:;
61 DEPTS:;
62 ROOMS:;
63 )
64 --
65 -- Definition of the Sampler with functional dependencies to be satisfied in table Courses
66 --
67 (SAMPLERFD AS Sampler_Functional_Dependencies COURSES:    -- Sampler with Functional Dependencies for
68                                                            --   table COURSES
69   course:year,                                             -- The 'course' determines the 'year'
70   course:teacher                                           -- The 'course' determines the 'teacher'
71 );
```

The language used in this case is the same as the one explained in the previous Section.

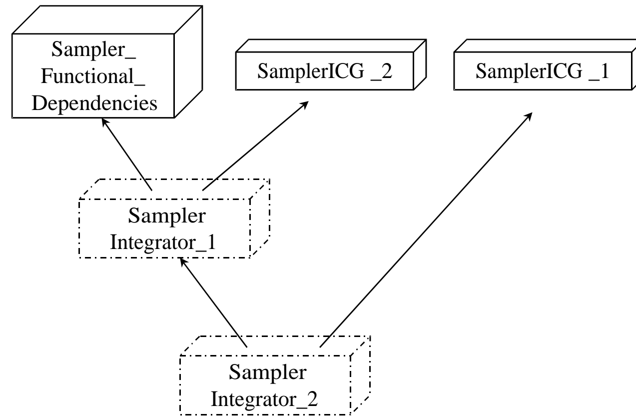

Fig. A.1: Using SamplerIntegrator to Create a Hierarchy of Database Samplers

One additional keyword has been used here (line 12), `SHOW`, which will instruct CoDaST to report on which Database Sampler is being used in each step during sampling. Also a new Sampler type has been defined (lines 67 to 71), `SAMPLERFD`, which specifies a set of functional dependencies to be enforced in the Sample as well as the table to which they refer. When parsing all these definitions, CoDaST will create the appropriate instances of `SamplerIntegrator` required to integrate these three Samplers. Section 7.5.1 stated that several levels of integration are possible. In order to illustrate this fact, assume that two instances of `SamplerIntegrator` will be used here. The first one, referred to as `SamplerIntegrator_1`, integrates samplers `SamplerICG_2` and `Sampler_Functional_Dependencies`. The second instance, referred to as `SamplerIntegrator_2`, will integrate `SamplerICG_2` and `SamplerIntegrator_1`. This leads to the hierarchy of Database Samplers shown in Fig. A.1. It must be noted that `SamplerIntegrator_2` will integrate `SamplerIntegrator_1` without knowledge of which Samplers it integrates, or whether it is also an instance of `SamplerIntegrator`.

With these definitions, the sequence of calls between the different Database Samplers that occur in CoDaST, as well as the insertions each of them performs into the Sample Database, are shown next. Lines which start with the star symbol (`*`) indicate that the Sampling Process (see Chapter 4) is on the `Synchronise()` step, and a plus symbol (`+`) indicates the execution of `UpdateProcess()` for the specified Sampler. Lines without starting symbol are associated with step `SelectInstance()` in the appropriate Sampler. In each of these calls, the entity

name of the instance being maintained consistent is also identified. As in the previous Section, for each instance inserted into the Sample Database, its entity name and attribute values are given. This example shows the interactions between all Samplers involved. This sequence of calls illustrates with one concrete example the Sequence Diagram of Fig. 7.7. In this case, however, two instances of `SamplerIntegrator` are involved, and not only one.

It can be seen how the Sampling Proces must be started by `SamplerIntegrator_2` as this is the one integrating all other Samplers (see Fig. A.1). When `SelectInstance()` is called in this Sampler, it must delegate this step to one of the Samplers it integrates, in this case `SamplerICG_1`. According to this Sampler an instance from entity `Teachers` is selected. Then `Synchronise()` is called in `SamplerIntegrator_2`, which delegates its execution to a Sampler different to the one that performed the initial insertion. Therefore `UpdateProcess()` in `SamplerIntegrator_1` is called. This in turn delegates this step to `SamplerICG_2`. This Sampler, according to the set of constraints it enforces, does not require any insertion as a consequence of the initial insertion. The same situation arises when `Sampler_Functional_Dependencies` is called. After that, `UpdateProcess()` method in `SamplerICG_1` itself is called in order to maintain consistency with the insertion it performed at the beginning of the Process. For each new insertion, all these steps will be repeated.

```
1
2 SamplerIntegrator_2
3 SamplerICG_1
4                 [TEACHERS]: <mary , 644 , statistics>
5 *SamplerIntegrator_2 (TEACHERS)
6 +SamplerIntegrator_1 (TEACHERS)
7 +SamplerICG_2 (TEACHERS)
8 +Sampler_Functional_Dependencies (TEACHERS)
9 +SamplerIntegrator_2 (TEACHERS)
10 +SamplerICG_1 (TEACHERS)
11                [DEPTS]: <statistics , mary>
12 *SamplerIntegrator_2 (DEPTS)
13 +SamplerIntegrator_1 (DEPTS)
14 +SamplerICG_2 (DEPTS)
15 +Sampler_Functional_Dependencies (DEPTS)
16 +SamplerICG_1 (DEPTS)
17                [ROOMS]: <s1 , 30 , statistics , 3>
18 *SamplerIntegrator_2 (ROOMS)
19 +SamplerIntegrator_1 (ROOMS)
20 +SamplerICG_2 (ROOMS)
21 +Sampler_Functional_Dependencies (ROOMS)
```

```
22 +SamplerICG_1 (ROOMS)
23                 [PERSONS]: <mary , teacher , venice>
24 *SamplerIntegrator_2 (PERSONS)
25 +SamplerIntegrator_1 (PERSONS)
26 +SamplerICG_2 (PERSONS)
27 +Sampler_Functional_Dependencies (PERSONS)
28 +SamplerICG_1 (PERSONS)
29                 [COURSES]: <data mining , 3 , mary>
30 *SamplerIntegrator_2 (COURSES)
31 +SamplerIntegrator_1 (COURSES)
32 +SamplerICG_2 (COURSES)
33 +Sampler_Functional_Dependencies (COURSES)
34 Sampler_Functional_Dependencies
35                 [COURSES]: <cybernetics , 5 , diana>
36 *SamplerIntegrator_1 (COURSES)
37 *SamplerIntegrator_2 (COURSES)
38 +SamplerICG_1 (COURSES)
39                 [TIMETABLE]: <cybernetics , thursday , 11 , s1>
40 *SamplerIntegrator_2 (TIMETABLE)
41 +SamplerIntegrator_1 (TIMETABLE)
42 +SamplerICG_2 (TIMETABLE)
43 +Sampler_Functional_Dependencies (COURSES)
44 Sampler_Functional_Dependencies
45                 [COURSES]: <programming , 2 , diana>
46 *SamplerIntegrator_1 (COURSES)
47 *SamplerIntegrator_2 (COURSES)
48 +SamplerICG_1 (COURSES)
49                 [TIMETABLE]: <programming , friday , 9 , c1>
50 *SamplerIntegrator_2 (TIMETABLE)
51 +SamplerIntegrator_1 (TIMETABLE)
52 +SamplerICG_2 (TIMETABLE)
53 +Sampler_Functional_Dependencies (COURSES)
54 Sampler_Functional_Dependencies
55                 [COURSES]: <analysis 2 , 2 , phillip>
56 *SamplerIntegrator_1 (COURSES)
57 *SamplerIntegrator_2 (COURSES)
58 +SamplerICG_1 (COURSES)
59                 [TIMETABLE]: <analysis 2 , wednesday , 10 , m1>
60 *SamplerIntegrator_2 (TIMETABLE)
61 +SamplerIntegrator_1 (TIMETABLE)
62 +SamplerICG_2 (TIMETABLE)
63 +Sampler_Functional_Dependencies (COURSES)
64 +SamplerICG_1 (TIMETABLE)
65                 [ROOMS]: <m1 , 20 , mathematics , 2>
66 *SamplerIntegrator_2 (ROOMS)
67 +SamplerIntegrator_1 (ROOMS)
68 +SamplerICG_2 (ROOMS)
69 +Sampler_Functional_Dependencies (COURSES)
70 +SamplerICG_1 (ROOMS)
71                 [DEPTS]: <mathematics , phillip>
72 *SamplerIntegrator_2 (DEPTS)
73 +SamplerIntegrator_1 (DEPTS)
74 +SamplerICG_2 (DEPTS)
75 +Sampler_Functional_Dependencies (COURSES)
76 +SamplerICG_1 (DEPTS)
77                 [TEACHERS]: <phillip , 453 , mathematics>
```

```
78 *SamplerIntegrator_2 (TEACHERS)
79 +SamplerIntegrator_1 (TEACHERS)
80 +SamplerICG_2 (TEACHERS)
81 +Sampler_Functional_Dependencies (COURSES)
82 +SamplerICG_1 (TEACHERS)
83                 [PERSONS]: <phillip , teacher , bologna>
84 *SamplerIntegrator_2 (PERSONS)
85 +SamplerIntegrator_1 (PERSONS)
86 +SamplerICG_2 (PERSONS)
87 +Sampler_Functional_Dependencies (COURSES)
88 +SamplerICG_1 (PERSONS)
89                 [PREREQUISITS]: <analysis 2 , analysis 1>
90 *SamplerIntegrator_2 (PREREQUISITS)
91 +SamplerIntegrator_1 (PREREQUISITS)
92 +SamplerICG_2 (PREREQUISITS)
93                 [COURSES]: <analysis 1 , 1 , phillip>
94 *SamplerIntegrator_1 (COURSES)
95 *SamplerIntegrator_2 (COURSES)
96 +SamplerICG_1 (COURSES)
97                 [TIMETABLE]: <analysis 1 , wednesday , 11 , m1>
98 *SamplerIntegrator_2 (TIMETABLE)
99 +SamplerIntegrator_1 (TIMETABLE)
100 +Sampler_Functional_Dependencies (TIMETABLE)
101 +SamplerICG_2 (COURSES)
102 +SamplerICG_1 (TIMETABLE)
103 +Sampler_Functional_Dependencies (COURSES)
104 +SamplerICG_2 (COURSES)
105 +Sampler_Functional_Dependencies (COURSES)
106 +SamplerICG_1 (PREREQUISITS)
107 +SamplerICG_2 (COURSES)
108 +Sampler_Functional_Dependencies (COURSES)
109 +SamplerICG_1 (TIMETABLE)
110                 [ROOMS]: <c1 , 35 , computer science , 1>
111 *SamplerIntegrator_2 (ROOMS)
112 +SamplerIntegrator_1 (ROOMS)
113 +SamplerICG_2 (ROOMS)
114 +Sampler_Functional_Dependencies (COURSES)
115 +SamplerICG_1 (ROOMS)
116                 [DEPTS]: <computer science , diana>
117 *SamplerIntegrator_2 (DEPTS)
118 +SamplerIntegrator_1 (DEPTS)
119 +SamplerICG_2 (DEPTS)
120 +Sampler_Functional_Dependencies (COURSES)
121 +SamplerICG_1 (DEPTS)
122                 [TEACHERS]: <diana , 551 , computer science>
123 *SamplerIntegrator_2 (TEACHERS)
124 +SamplerIntegrator_1 (TEACHERS)
125 +SamplerICG_2 (TEACHERS)
126 +Sampler_Functional_Dependencies (COURSES)
127 +SamplerICG_1 (TEACHERS)
128                 [PERSONS]: <diana , teacher , bary>
129 *SamplerIntegrator_2 (PERSONS)
130 +SamplerIntegrator_1 (PERSONS)
131 +SamplerICG_2 (PERSONS)
132 +Sampler_Functional_Dependencies (COURSES)
133 +SamplerICG_1 (PERSONS)
```

```
134 +SamplerICG_2 (COURSES)
135 +Sampler_Functional_Dependencies (COURSES)
136 Sampler_Functional_Dependencies
137                 [COURSES]: <languages , 5 , diana>
138 *SamplerIntegrator_1 (COURSES)
139 *SamplerIntegrator_2 (COURSES)
140 +SamplerICG_1 (COURSES)
141                 [TIMETABLE]: <languages , wednesday , 15 , c1>
142 *SamplerIntegrator_2 (TIMETABLE)
143 +SamplerIntegrator_1 (TIMETABLE)
144 +SamplerICG_2 (TIMETABLE)
145 +Sampler_Functional_Dependencies (COURSES)
146 +SamplerICG_1 (TIMETABLE)
147                 [PREREQUISITS]: <languages , programming>
148 *SamplerIntegrator_2 (PREREQUISITS)
149 +SamplerIntegrator_1 (PREREQUISITS)
150 +SamplerICG_2 (PREREQUISITS)
151 +Sampler_Functional_Dependencies (COURSES)
152 +SamplerICG_1 (PREREQUISITS)
153 +SamplerICG_2 (COURSES)
154 +Sampler_Functional_Dependencies (COURSES)
155 +SamplerICG_1 (TIMETABLE)
156 +SamplerICG_2 (COURSES)
157 +Sampler_Functional_Dependencies (COURSES)
158 +SamplerICG_1 (COURSES)
159                 [TIMETABLE]: <data mining , friday , 11 , s1>
160 *SamplerIntegrator_2 (TIMETABLE)
161 +SamplerIntegrator_1 (TIMETABLE)
162 +SamplerICG_2 (TIMETABLE)
163 +Sampler_Functional_Dependencies (TIMETABLE)
164 +SamplerICG_1 (TIMETABLE)
165
166
167 SamplerIntegrator_2
168 SamplerIntegrator_1
169 SamplerICG_2
170                 [EXAMS]: <jim , algebra , 25.000000>
171 *SamplerIntegrator_2 (EXAMS)
172 +SamplerICG_1 (EXAMS)
173 +SamplerIntegrator_2 (EXAMS)
174 +SamplerIntegrator_1 (EXAMS)
175 +Sampler_Functional_Dependencies (EXAMS)
176 +SamplerICG_2 (EXAMS)
177                 [STUDENTS]: <jim , 2>
178 *SamplerIntegrator_1 (STUDENTS)
179 *SamplerIntegrator_2 (STUDENTS)
180 +SamplerICG_1 (STUDENTS)
181                 [PERSONS]: <jim , student , venice>
182 *SamplerIntegrator_2 (PERSONS)
183 +SamplerIntegrator_1 (PERSONS)
184 +Sampler_Functional_Dependencies (PERSONS)
185 +SamplerICG_2 (STUDENTS)
186 +SamplerICG_1 (PERSONS)
187 +Sampler_Functional_Dependencies (STUDENTS)
188 +SamplerICG_2 (STUDENTS)
189                 [COURSES]: <algebra , 1 , robert>
```

```
190 *SamplerIntegrator_1 (COURSES)
191 *SamplerIntegrator_2 (COURSES)
192 +SamplerICG_1 (COURSES)
193                 [TIMETABLE]: <algebra , monday , 9 , m1>
194 *SamplerIntegrator_2 (TIMETABLE)
195 +SamplerIntegrator_1 (TIMETABLE)
196 +Sampler_Functional_Dependencies (TIMETABLE)
197 +SamplerICG_2 (COURSES)
198                 [TEACHERS]: <robert , 454 , mathematics>
199 *SamplerIntegrator_1 (TEACHERS)
200 *SamplerIntegrator_2 (TEACHERS)
201 +SamplerICG_1 (TEACHERS)
202                 [PERSONS]: <robert , teacher , verona>
203 *SamplerIntegrator_2 (PERSONS)
204 +SamplerIntegrator_1 (PERSONS)
205 +Sampler_Functional_Dependencies (PERSONS)
206 +SamplerICG_2 (TEACHERS)
207 +SamplerICG_1 (PERSONS)
208 +Sampler_Functional_Dependencies (TEACHERS)
209 +SamplerICG_2 (TEACHERS)
210 +SamplerICG_1 (TIMETABLE)
211 +Sampler_Functional_Dependencies (COURSES)
212 +SamplerICG_2 (COURSES)
```

As in the execution example of Section A.4, two independent blocks of insertions are performed here in order to satisfy the minimum Sample size requirements. The first one (lines 1 to 164), as analysed above, was triggered by an insertion into entity `Teachers`. Once this insertion has been maintained consistent according to all Samplers, the Sample size requirements of `SamplerICG_1` are satisfied. For this reason the second block of insertions (lines 167 to 212) is triggered by an insertion into entity `Exams`, according to the Sample size requirements of `SamplerICG_2`.