Bruno Miguel de Melo Gonçalves Areal

Licenciado em Engenharia Informática

Building Anonymised Database Samples

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador :	Prof. Doutor José Alferes, Prof. Catedrático, Universidade Nova de Lisboa
Co-orientador :	Prof. Doutor Miguel Goulão, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente:	Doutor José Alberto Cardoso e Cunha
Arguente:	Doutor Vitor Manuel Beires Pinto Nogueira
Vogal:	Doutor José Júlio Alves Alferes

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

December, 2011

Building Anonymised Database Samples

Acknowledgements

First of all, I would like to thank my advisors, José Júlio Alferes and Miguel Goulão, for their patience and support. And also to my employers, and specially to Lina Matos and Paulo Castro, for bending the rules and giving me some of my work-hours for developing this work. I would also like to thank all my colleagues and friends, who are always ready to party but were also capable of not disturbing me when i really needed to get this work done. A special thanks to Bruno Veigas, who stayed with me across the summer weekends (when everybody else was on vacations or at the beach), for shopping and cooking for me while I was working, and also for his ideas when I was lost in my code, looking for bugs. I'm forever indebted to Inês Matos, Sonia Furtado and Mário Fonseca for the long nights online, helping me to surpass my lack of inspiration for writing this thesis. Last, but definitely not least, I would like to thank my parents and my grandparents for their continuous encouragement and motivation. A special thanks to Sofia for her love, support and ability to get me back to work, even from the other side of the ocean.

Abstract

In this work we propose Anonym Database Sampler (ADS), a flexible and modular system capable of extracting an anonymised, consistent and representative sample from a relational database. ADS was envisioned for use in testing and development environments. To this end, a sample specification input is requested from the user, that is used by ADS's sampling engine to perform a stratified random sample. Afterwards a *First-choice hill climbing* algorithm is applied to the sample, optimising the selected data towards the specified requisites.

Finally, if some restrictions are still to be met, tuples and/or keys modifications are performed, ensuring that the final sample fully complies with the initial sample specification. While having a representative and sound database that developers can use in these environments can be a great advantage, we assume that this representativeness does not need to comply with a truly statistical representativity, which would be much harder to obtain. Thereby, ADS samples are not appropriate for any kind of statistical data analysis. After the sample being successfully extracted, due to the sensitivity of the data contained in most organisation databases, a data anonymisation step is performed. The sampled data is consistently enciphered and masked, preventing data privacy breaches that could occur by delivering to developers a database containing some real operational data.

Keywords: anonymous sampling, database sampling, test databases, database, sampling algorithm

Resumo

Neste trabalho, propomos ADS, um sistema flexível e modular capaz de extrair uma amostra anónima, consistente e representativa de uma base de dados relacional. ADS foi concebido com o objectivo de ser utilizado em ambientes de teste e desenvolvimento aplicacionais.

Para tal, é requerida ao utilizador a especificação da amostra que deseja obter. Especificação essa que será usada pelo motor de amostragem para produzir uma amostra estratificada da base de dados original, onde em seguida será aplicado um algoritmo de *trepa-colinas* de forma a optimizar os dados seleccionados de forma estes se aproximem dos requisitos especificados.

Finalmente, se algumas restrições continuarem por cumprir, são realizadas modificações de tuplos e/ou chaves, assegurando-se assim que a amostra obtida é completamente compatível com a especificação da amostra. Não obstante do facto de que uma base de dados representativa e consistente nos seus ambientes de testes e desenvolvimento será uma grande vantagem para os programadores, assumimos que esta representatividade não necessitará de ser uma representatividade estatística no puro sentido da sua definição, algo que seria muito mais difícil de obter. Consequentemente, as amostras produzidas pelo ADS deverão apenas ser encaradas como dados para testes e desenvolvimento, não sendo apropriados para qualquer tipo de análise estatística de dados. Após a amostra ter sido extraída com êxito, devido à sensibilidade dos dados contidos nas bases de dados da maioria das organizações, é executada uma anonimização de dados. Os dados amostrados são cifrados e mascarados de forma consistente, prevenindo assim potenciais violações de privacidade que poderiam ocorrer através da apresentação de dados operacionais reais aos programadores.

Palavras-chave: amostragem anónima, amostragem de bases de dados, bases de dados de teste, bases de dados, algoritmos de amostragem

Contents

List of Figures

List of Tables

Listings

List of Acronyms

SQL	Structured Query Language
DBMS	database management system
JDBC	Java Database Connectivity Driver
JavaCC	Java Compiler Compiler
ADS	Anonym Database Sampler

Preface

Six months before I started this thesis, I left from my job at a small IT company to start an internship at one of the biggest Portuguese financial institutions, which is also the company where I work today. So when the time came, I was looking at the available Master Thesis proposals, wondering how would I be able to successfully finish any of these proposals while keeping my eight-hour-per-day work.

After some time talking to my friends, colleagues, bosses and professors, I came up with what I thought that was a brilliant idea: creating my own master-thesis proposal!

This decision ended up by being one of the most challenging adventures I have ever put myself into (and I am the extreme sports lover kind-of-guy...).

Almost all of my colleagues started working on their Master Thesis without having a clue about what they were doing. They started reading some papers their advisors gave them, doing some writing, and most of them only really understood what they were working on when they already were half-way through their thesis. Their advisors knew what they were doing, and that was more than sufficient. In my case, I was the one who had to know what I needed to do, as I was the one who proposed my topic - but I *really* had no clue.

I simply had an idea for a solution to a problem we had at my company - we really needed to have representative development environment databases in which we could trust while working on our projects - some sort of operational database sample; and I knew that, given the multiple platform environments we work on, that solution should be easily adaptable to as many projects as possible. I have convinced my bosses and my advisors that this could be a cool problem to solve, and that it could result in a Master Thesis. But that was all I had.

Eventually, whith my advisors help (and patience), we reached a consensus about what my work was supposed to be, and how it should work. But even then, every detail that we thought that could be solved by using some external tool (and many in fact were), turned up not to be as easy to combine into a single system as we hoped. And I soon discovered that a big Portuguese financial institution, where only "half of the Internet" is available for "security reasons", is not the most friendly environment to develop some academic and innovative IT work. And my laptop got stolen mid-summer. And Oracle had bugs that where already patched, but those patches where not available for our Oracle-Express free-version. And...

Past these hardships, this is the result. My Master Thesis was successfully completed! My proposed solution does work! And I have learned a valuable lesson: Don't you ever, ever, *ever* propose to do something serious without knowing exactly what you are putting yourself into, and how do you plan to accomplish it.

And if by any chance you are a student, reading this while looking for a Master Thesis proposal and wondering if you should create your own, please read the above paragraph again. All of it! :-)

1 Introduction

1.1 Motivation

When performing development operations over applications, developers usually find themselves in the need of using some kind of database to store the data used by these applications, usually relying in some sort of database management system (DBMS).

Taking into account that most of the development operations in organisations are related to maintenance or evolutionary operations [KKMS09], the possibility of using the real data contained in production environments databases would provide a valuable advantage to developers, giving them the chance of making decisions based on real data. Nevertheless, given the amount of data contained in most application databases, the option of providing such possibility would also imply higher infrastructure costs. In addition, due to the sensitivity of the data contained in most application databases, providing access to such data would result on an unthinkable - and in many situations illegal - breach of their clients privacy.

As a result, development environments databases are usually populated with randomly generated data, or by manually inputting data requested by developers in order to create all the test cases that they are able to identify. Albeit these methods can provide data that developers can use in their work, it should be clear that the data contained in these development environment databases are far from being a suitable representation of the reality. Actually, this lack of data representativeness tends to be a bigger problem than it may seem at first glance. Since most developers have no access to production data, they are often misled by the data they possess, leading them to perform code optimisations that are not necessary and, on the other hand, neglecting some cases that seem sporadic but actually happen recurrently.

In order to surpass this lack of data representativeness in development environment databases, in this thesis we propose Anonym Database Sampler (ADS), a database sampling system, capable of extracting an anonymised representative sample from a database.

Having a representative database in a development or testing environment can be a huge advantage for developers, as it provides a richer environment, where developers need not to rely on their own or clients' best guesses of possible scenarios. In such representative databases, the relationships between data can be faithfully kept, generating a more realistic environment in which developers can detect a greater number of borderline and uncommon scenarios. Nevertheless, this representativeness does not have to take the form of a truly statistically representative database, which would be much more expensive to obtain in every respect. For this usage, what needs to be kept are some properties of the original data that tend to be crucial for testing applications, such as maintaining minimal and maximal values for a given attribute, maintaining some values averages or maintaing proportionality among data in relations. And most importantly, maintaining database constraints, such as primary and foreign keys.

To extract a sample from a database, one could simply select a random sample from the whole database, while ensuring its consistency using the key restrictions from the original database. However, delivering to developers a randomly obtained sample database, would likely fail to solve the data representativeness problem, given that a uniform sampling method can not deliver a representative sample for populations with non-uniform probability distributions [GHYZ04], i.e. performing a random sample over a non-uniform distribution will likely fail to deliver a representative sample.

Therefore, and given that each database attribute will probably have its own distribution, to obtain a representative sample one needs first to define from what data he wishes to obtain such representation (i.e., which attributes from the database he wishes to ensure representativeness, and to what extent). Given this, to guarantee that a database sample meets the needs of the developers who will use it, ADS provides a sample specification tool, allowing developers to manifest the needs for their developments. Using the provided specification, ADS can ensure that, at least for the specified data, the information contained in the sample database reflects the characteristics of the data contained in the original database. Moreover, by performing data anonymisation operations on the obtained sample without damaging data consistency and representativeness, ADS also prevents data privacy breaches.

As a result, ADS can provide to developers and organisations the advantages of using real data in their development environments while lowering infrastructure costs, without compromising data privacy.

This research was motivated by a real necessity of the company for which the author works to. There, due mostly to privacy issues, the developers struggle to build their own data for development based only on the knowledge they can obtain from the users of their applications. In such context, the goal of this work is to deliver more suitable databases for testing and development, based in real operational data. ADS is a proof of concept prototype that achieves this goal, and will be subject to further improvements when applied to software deployment.

1.2 Solution proposed

ADS is a flexible and modular system capable of extracting an anonymised representative sample from a relational database, envisioned for application's development, testing or user training environments.

On the database domain, sampling is commonly employed in many areas where the use of the whole database proves to be unnecessary or inefficient. In fact, many examples of applications of database sampling can be found, from data mining [JL96] to query optimisation [LNS90]. Consequently, most of the recent DBMS already maintain statistics about their contained data, allowing them to automatically achieve some performance optimisations.

Instead of using some random sampling algorithm to select which data should be extracted to the sample, ADS requires a sample specification as an input. Combining this option with the statistical information about the database contained data, the system provides the possibility of performing a "fine-tuning" of the desired sample, by adjusting the data's relevance given the intended usage.

Since different situations may require different data privacy rules, the sample specification allows the definition of different sensitivity levels for each attribute contained in the original database. As such, it is the developers' responsibility to define these parameters, as they know better the data representativeness needed for their work. Moreover, since these developers will also be the ones who use the extracted sample, multiple sample specifications can be further included. This option offers the possibility of imposing minimal privacy levels for the most sensitive data by ensuring the most restrictive one.

It is worth mentioning that the anonymisation applied to the obtained samples is not adequate for public exposure. As explained later, data anonymisation not only implies some sort of encryption but also some other techniques that, when fully applied, reduce data representativeness in order to minimise data disclosure. As a consequence, since one of the system's main goals is to deliver a suitable representative sample, ADS data anonymisation serves essentially as a deterrent for data disclosure. The use of the obtained samples should be restricted to related personnel, who already deal with sensitive data in a daily basis and therefore have already signed some kind of non-disclosure agreement. That said, the proposed architecture was designed so that stronger forms of anonymisation can replace the encryption support used in the prototype produced in the scope of this dissertation.

In summary, the main contributions of this thesis are:

- A simple and expressive sample specification language to define the characteristics of the desired sample.

- ADS sampling engine and subsequent operational logic that makes possible the extraction of an anonymised, consistent and representative sample from a relational database.
- ADS itself, which architecture plays an important role in the whole system flexibility and modularity.

1.3 Structure of this document

The rest of this document is organised as follows: In Chapter 2 we present an overview of the areas related with our work, along with a brief discussion about how our work relates with the studies in those areas. In Chapter 3 we define the created sample specification language and, subsequently, in Chapter 4 we discuss the architecture followed during ADS's implementation, while presenting the most relevant implementation details and the system execution logic. Afterwards, we proceed by explaining the developed sampling engine algorithm in Chapter 5, followed by the performed system tests and results analysis in Chapter 6. Finally, in Chapter 7 we conclude this document with some final considerations and identify future challenges for this work.

2 Related Work

In this chapter, we present an overview of the various areas related with database sampling and anonymisation. This way, we begin by presenting in section 2.1 a quick review about the most relevant statistical sampling concepts. Afterwards, we discuss in section 2.2 the state of the art regarding the use of sampling in databases, while also including an overview of the database sampling tools commercially available. Finally, in section 2.3 we present a short overview over the existent data anonymisation techniques.

2.1 Statistical sampling

Sampling is an area of statistics, concerning the selection of a representative subset from a given population, thus allowing to learn characteristics of a large group without having to observe every contained element. As a consequence, sampling is commonly used in many areas to gain a better knowledge about the characteristics of a population, given the high costs (of money and/or time) involved in gaining such knowledge about every item in a population.

Sampling is a largely studied and well established area. Consequently, the explanations presented in this section can be found in most statistical books, like [Coc77].

Population The word population is used to denote the aggregate from which the sample is chosen. Thereby, the population to be sampled (i.e. the group of elements to be sampled) should always match precisely the aggregate about which information is wanted (i.e. target to be studied). For example, applying this definition to our work, if we wish to obtain a sample of the data related with bank clients that use online-banking, we should restrict the collection of data from the database to tuples that are related with clients which have an online-banking id, excluding all the other tuples.

Simple random sample Simple random sampling is a method that consists in selecting n (the desired sample size) units out of N (the size of the target population) such that every one of the ${}_NC_n$ distinct samples has an equal chance of being drawn. At any draw, the process used to perform this kind of sample must give an equal chance of selection to any of the elements in the population who have not already been drawn.

Stratified sample In stratified sampling, the population of N units is firstly divided into sub-populations of $N_1, N_2, ..., N_L$ units respectively, according to well defined criteria, so that the sub-groups are homogeneous. These sub-populations (known as *strata*) are non-overlapping, and together they comprise the whole of the population, so that $N_1 + N_2 + ... + N_L = N$. Once the *strata* have been determined, a random sample is drawn from each, so that the drawings are made independently on each *stratum*. This technique is particularly useful if data of known precision is wanted for certain subdivisions of the population, being advisable to treat each subdivision as a "population" in its own right. Stratification may also produce a gain in precision in the estimates of characteristics of the whole population. It may possible to divide a heterogeneous population into sub-populations, each of which is internally homogeneous. If each stratum is homogeneous, in that the measurements vary little from one unit to another, a precise estimate of any stratum mean can be obtained from a small sample in that stratum. These estimates can then be combined into a precise estimate for the whole population.

2.2 Sampling from databases

In the recent years database management systems (DBMSs) became commonplace tools in most organisations, being used to store most of their applications data, many times creating databases with terabytes of information. In this sense, sampling proved to be a powerful technique to deal with such large databases, for which Frank Olken's work in [Olk93] stands as one of the major contributions to this area. In fact, the recognised value of sampling databases has come to the point where most of the modern DBMS already include sampling mechanisms [CMN98, Ora11b] in order to achieve automatic performance optimisations, mainly related with query size estimation or intermediate query result sizes for query optimisation algorithms [LNS90, HK04]. Oracle DBMS even supports the possibility of performing a query over a sample of a given table [Ora11c] which is, as we discuss later, one of the reasons that made us choose this DBMS for the development of this work.

Database sampling has also been applied to other database-related areas, like data warehousing and data mining [CHY96]. Here, it is used to ease the process of defining a solution for a specific problem, and then to apply that solution to the entire database [JL96], like performing data mining of association rules [Toi96, ZPLO97]. In addition, database sampling has also been proposed to maintain a warehouse of sampled data, allowing quick approximate answers to analytical queries, while also easing data auditing and interactive data exploration [BH06].

Even though several works have been proposed towards achieving a better sample representativeness, these studies are mainly focused in new sampling algorithms, aiming to solve the lack of capability of conventional uniform sampling methods for extracting a representative sample from data with nonuniform probability distributions [GHYZ04, PF00]. And to the best of our knowledge, not much work has been done in order to obtain representative and consistent samples from a database to be used specifically in application development environments.

In fact, one of the few studies we were able to find that is associated with our work is the one presented in [BG00], which focuses on the advantages of using prototype databases populated with sampled operational data, along with a proposal of a consistent sampling database process. However, although agreeing with the argumentation presented regarding the advantages of using sample databases, in this thesis we present a different approach for the sampling process. By ensuring the same schema on both original and sampled databases, while delivering to developers the creation of a sample specification, we simplify the adjustment of the sampling process for different sample requests or different database instances, without jeopardising the system modularity and the adaptation for other DBMS. On the other hand, by requesting statistical data about tuples and their relations as an input (which can be easily found stored in modern DBMS), we achieve an agile sample process, given that the sample data selection becomes simplified.

2.2.1 Commercial applications for sampling from databases

Notwithstanding the lack of academic studies on sampling databases for development environments, there are a few commercial applications which announce to support such functionality such as *IBM Optim* [Man11]. These applications are usually designed to be integrated data management systems, and include many other features. For example, *IBM Optim* is developed to archive, classify and manage the data within many database instances and many DBMS (and particularly *IBM DB2 DBMS* [DB211]). Although these features are not relevant within the context of this work, they tend to play a major role in how a sample database is obtained by these commercial applications. These data management tools are focused in being capable of applying some kind of operation to a given set of tuples, while ensuring that the data remains consistent by applying the same operation to all the related tuples. The same concept is applied to build test databases based on operational data, where the data operations represent a consistent data extraction from the original database.

This way, the selection of the data that should be present in the sample database is performed by a user, defining a specific set of existent tuples. Although such sampling method guarantees a consistent sample database, it should be clear that no data representativeness is ensured. In fact, the operation being performed is more similar to a *strata* selection than to a sample selection. Consequently, albeit the fact that we have an exact representation for all the data related with the selected tuples, probably such representation will not reflect the real data distribution.

2.3 Data anonymisation

Data anonymisation is a largely studied area, with a broad range of work being developed in an everyday basis. Accordingly, given that the data to be anonymised is usually stored in some form of database, many of these studies focus in database content anonymisation.

It is a common pitfall to think that data anonymisation is achieved by masking, encrypting or removing person-specific data with all explicit identifiers, such as name, address and email. Assuming that data anonymity is maintained because, by looking directly to the obtained data, it seems impossible to associate those information's to the persons from which the data is related is incorrect. Using the remaining data we can, in most cases, re-identify individuals by looking at unique characteristics found in the released data, or by linking or matching the data to other data. For example, in a study [Swe00] using 1990 U.S. Census it was reported that 87% of the United States population had characteristics that made them unique based only on {*postcode*, *gender*, *date of birth*}, three characteristics that are usually not considered as a privacy invasion.

In [Dal86] a different approach was proposed to identify which are the attributes that must be considered private, denoted *quasi-identifiers*. *Quasi-identifiers* are those attributes that, like in the example stated above, seem safe for public disclosure but, when linked to other data, can uniquely identify the object intended to be anonymised. In [Swe02] a different anonymisation model is proposed, named k-anonymity. The concept behind the k-anonymity model is to ensure that each sequence of *quasi-identifier* values within a table appears with at least k occurrences. Consequently, by using this model, it is ensured that an attacker can not link any of the anonimysed data to match less than k individuals.

Nevertheless, considering the known time restrictions for the development of this thesis, the fact that the anonymisation employed in the obtained database samples should be seen as a dissuasion for data disclosure by related personnel, and also due to increase of complexity that this option would imply, we chose only develop a consistent data masking method, leaving the k-anonymity implementation only to develop in future work.

3 Sample specification language

In order to create an anonymised and representative database sample, we need to specify the data relevancy and sensitivity, along with other sample's specifications, such as the desired sample size or some data details that may be omitted from the original database schema. To encode such specifications we need a language which allow us to clearly define what rules must be ensured in the sample database and thus to guarantee that the sample complies with all the requirements.

To the best of our knowledge, at the time that this work was developed, no such such language has been developed or proposed. This way, we developed an "SQL-like" declarative specification language, designed with the purpose of delivering an intuitive way to pass such input into the system.

Declarative languages have the advantage to be substantially more succinct and self-explanatory, as they state what should be the result of the computation rather than how it is to be computed. As a result, using a declarative language allows us to define specifications which are clear and quickly understandable by its users, whereas the expressions used to define the specification can be viewed as requests for the sample and the details of how the sample is obtained are left to the abstract machine. Moreover, the creation of the specification should be carried by the end users of the database sample, who know the best which proprieties the database sample should meet. Therefore, delivering the sample specification by "SQL-like" syntax expressions provides to the end user a more familiar and intuitive way of stating their requests. Accordingly, declarative languages are also verification oriented, and so, each sample specification expression not only instructs the system with information about the resulting sample database, but also serves as a post-condition and hence ensuring the sample reliability.

Anonym Database Sampler (ADS) sample specification language can be defined in three different expressions:

Sample definition which is used to define the percentage of the original database that should be present in the sample, i.e. the sample size, along with its acceptable error margin and the minimal anonymisation value that should be applied to all the sampled data.

Statistical requirements which are used to define which statistical parameters should be maintained by the sample for one or more attributes from a given table. Additionally, one can also define that those statistical parameters should have a more restrictive error margin, or that the specified attributes should have an higher anonymisation value than the one defined in the sample definition.

Data definitions which are used to instruct the system about some known information about the database that can not be found in the original database schema, like redefining some attribute data type or creating some known foreign key restriction that is omitted from the original schema.

As ilustrated in Figure 3.1, a sample specification starts with one sample definition expression, followed by as many statistical requirements or data definitions as needed, by any order. If any attribute appears in more than one expression for the same restriction, the system will ensure that the more restrictive requirement is maintained.

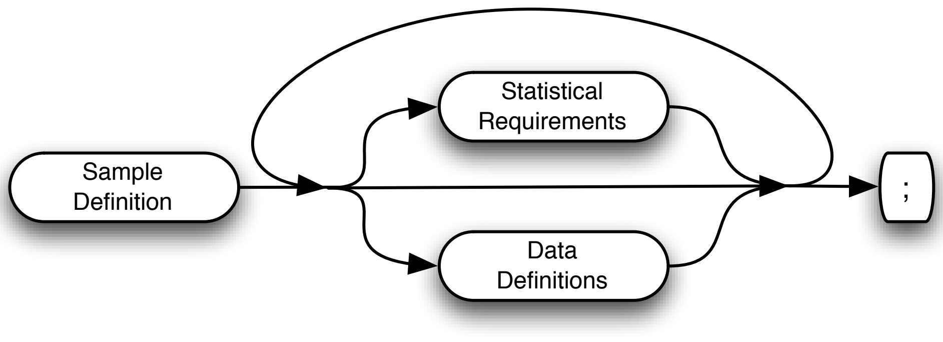

Figure 3.1: Sample specification expressions usage diagram

For the rest of this chapter, we present a detailed overview over each one of these expressions, along with a running-example of the creation of a sample from a database. In addition, the resulting sample specification example will be used in the rest of this thesis.

For our example we used the default example database from Oracle database management system (DBMS). Here one can find typical information about an organisation such as departments, employees and their job history. In Figure 3.2 we display the information contained in each one of these database tables and their connections. The connections between each table are read as follows:

- Departments have a manager and a location
- Locations have a country
- Countries have a region
- Employees have a department, a manager and a job
- Job history has a department, a job and an employee

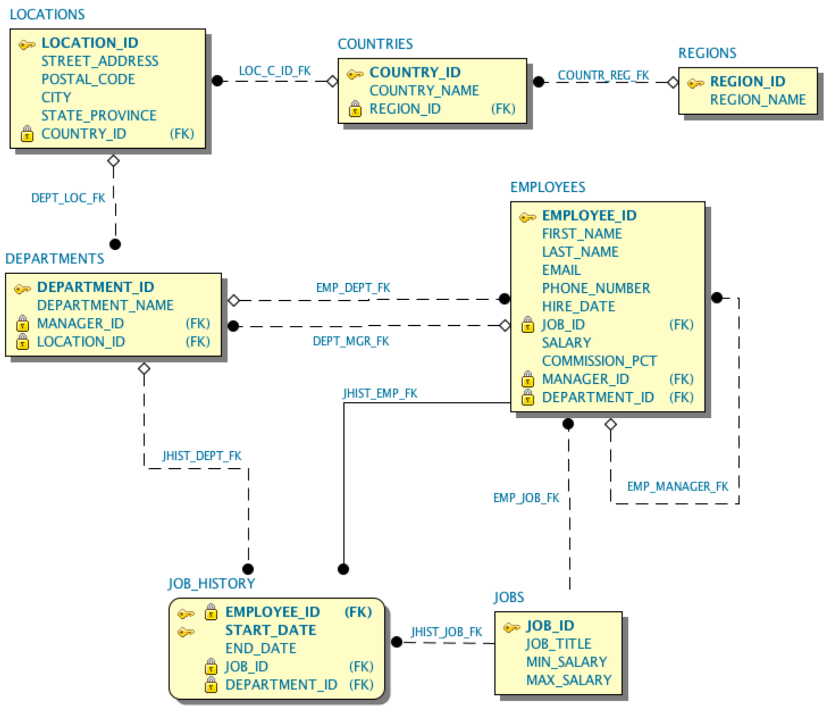

Figure 3.2: Example Database ER Diagram

3.1 Sample Definition

Every sample specification must start with a sample definition expression. These are used to specify the desired percentage value (P) of the original database that should be obtained, along with the acceptable error margin value (V) and the minimal anonymisation level that should be applied to all the sample data.

The original database size is calculated by counting the total number of tuples contained in every database table. This way, if each table of our example database contained 10 tuples, the original database size would correspond to 70 tuples. Furthermore, by requesting a 50% sample from such database, the system would try to extract a 35 tuple database, containg a combination of tuples that would fulfill all the specified statistical requirements. However, such combination can be very hard (or even impossible) to obtain, given that we also needed to ensure a sound database, by fullfilling all the original database primary and foreing key restrictions. To this end, by specifying a sample size error margin, one is relaxing this restriction, helping the system to successfully obtain his desired sample.

The error margin is defined as a percentage of the desired value, and should be read as $P \pm (P * \frac{V}{100})$. Considering the previous example, if one defined an error margin of 15%, the system would try to obtain a sample with a total number of tuples between 30 and 40.

The anonymisation value defines the sensitivity level of the sample data. In the current implementation, only one anonymisation technique was implemented: data encryption. Therefore, the current acceptable values for the anonymisation value are 0 and 1, where 0 means that no encryption should be made, and 1 indicate that the data should be encrypted. However, the language implementation was performed bearing in mind the adoption of some different data anonymisation techniques, like k-anonymity (cf. 2.3), where anonymisation levels greater than 1 could represent the k anonymity value to be ensured.

Sample definition requirement expressions can be formed as illustrated in Figure 3.3. As one can observe, the definition should start by defining the desired sample size, followed by the minimal accepted error and anonymisation values, by any order.

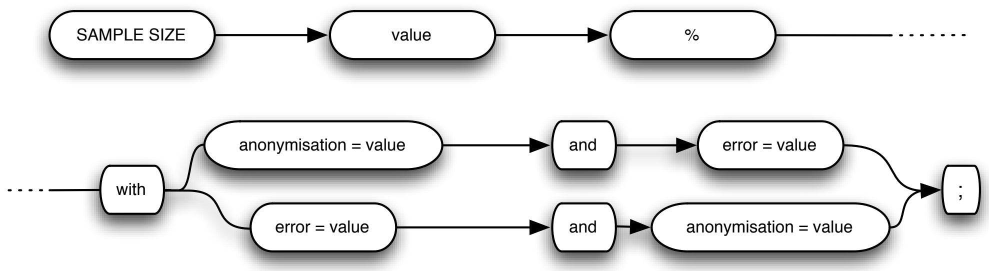

Figure 3.3: Sample definition usage diagram

As an example, to obtain a sample with 50% of the original database size, with no anonymisation level and accepting a 15% error margin, one should use the following expression:

```
sample size 50% with error=15 and anonymisation = 0;
```

3.2 Statistical Requirements

Statistical requirements are used to specify which statistical characteristics should be preserved, and for which database attributes those statistical requirements should be applied. However, it should be clear that not every statistical variable is available for an attribute. For example, it makes no sense to maintain the average value of an employee name.

Moreover, while some of these restrictions can be intuitive, and even detected based on the attribute data type, some other statistical restrictions may apply. Based on these scale types, some statistical operations can or can not apply to a given attribute, i.e. while the calculus operation could be performed, the result could not be considered correct, given that the attribute does not comply with some of the statistical variable definitions. For example, while one can calculate the median value of a nominal attribute such as *email*, using its alphabetical order, the result would be meaningless, given that such order had no relation with any attribute's characteristics.

Given this, as we explain in further detail in Chapter 4, we used Oracle's DBMS statistical engine to obtain most of the statistical variables values, and reflected the Oracle statistical restrictions in the language implementation. However, those restrictions are only based on the attribute data type, and does not validate any of the scale type restrictions. Therefore, in the current ADS's implementation, it should be the user to know what statiscal variables she can apply to a given attribute.

In table 3.1 we present ADS supported statistical variables and their identifiers, along with the attribute types each variable can be applied to. We separate these attribute types into two major groups: numerical and non-numerical. Numerical attributes are represented by long, double, float or integer numbers. All the other possible attribute types are considered non-numerical, including date. For more information about these restrictions, see [Ora11a]

Statistical Variable	ADS variable id	Numerical	Non-numerical
Maximum	max	×	×
Minimum	min	×	×
Average	avg	×	
Standard Deviation	sd	×	
Median	median	×	×
Mode	mode	×	×
Ratio	ratio	×	×

Table 3.1: Available statistical variables and possible attribute types

It is noteworthy that we introduced a non-standard statistical variable: ratio. This variable should be used when a developer wishes to maintain the distribution of a given attribute instead of some other statistical variable of that attribute value. By specifying that the ratio for the attribute A from table T should be preserved, one is requesting that, for each value V in the domain of A, the tuple percentage whith the value V should be the same in both the original and the sampled table T. For example, by using the following expression, one is requesting a sample that contains the same percentage of departments at any location in the sampled and the original departments table. The same percentage of departments at Lisbon, the same percentage of departments at Viseu, the same percentage of departments at Evora, etc.

```
preserve ratio for LOCATION_ID from DEPARTMENTS;
```

Furthermore, by requesting to preserve the ratio of an attribute that is a foreign-key, one can ensure that the ratio of the relations between the primary and the foreign table are ensured, thereby allowing a correct data distribution between those two tables.

Statistical requirement expressions can be formed as illustrated in figure 3.4. As one can observe, in addition to defining one or more statistical variables for one or more attributes from a table, we can also add new error margins and new anonymisation restrictions for these attributes. In these expressions, error margins and anonymisation restrictions are handled as in sample definition expressions (exposed in Section 3.1), but applied to the table attributes that were specified. Additionally, if an attribute is specified in more than one expression, the system will always consider the most restrictive error margin and anonymisation value.

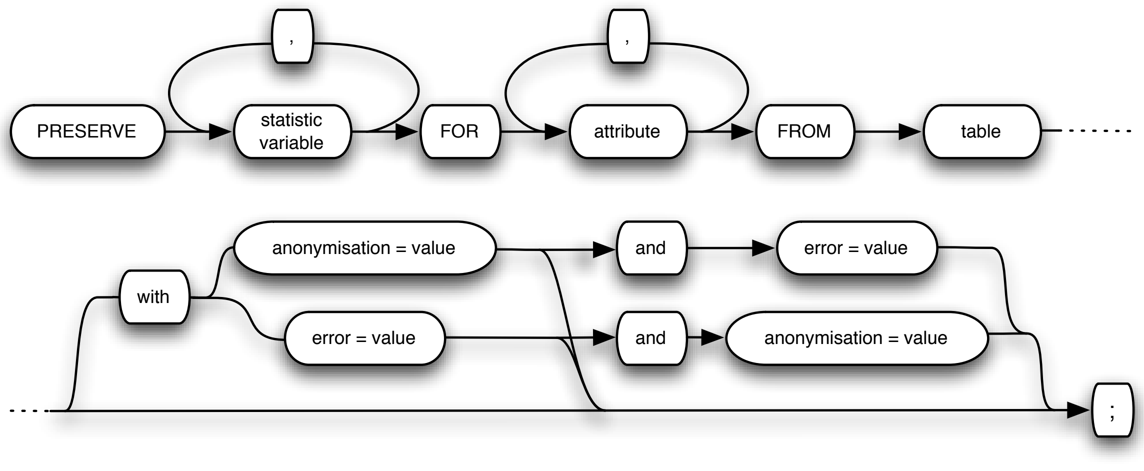

Figure 3.4: Statistical requirements usage diagram

As an example, consider that we wish to obtain a sample that maintains the maximum, minimum and average salary values from employees with a 10% error margin, while also maintaining the median value for employees hire date with a 5% error value. In addition, assume that we also wish to ensure the distribution of departments by country. This sample specification can be defined using the following expressions:

```
preserve max, min, avg for SALARY from EMPLOYEES with error=10;
preserve median for HIRE_DATE from EMPLOYEES with error=5;
preserve ratio for COUNTRY_ID from LOCATIONS;
preserve ratio for LOCATION_ID from DEPARTMENTS;
```

3.3 Data Definitions

ADS relies on a well-designed database to successfully extract a sample from that database, given that the system uses the foreign-key relations to ensure data consistency and representativeness. Furthermore, attribute data types also play a major role in sample specification, given that some statistical variables can only be applied to numerical data types.

However, one frequently encounters "real-world" databases that, to keep up with the evolution of the applications they interact with, have been repeatedly altered over time. Typically, such alterations result in sub-optimally designed databases, in which both attribute restrictions and the relationships between attributes are handled by the application's code. Furthermore, attributes are also frequently assigned to a type that does not match their characteristics. For instance, although an employee number is an integer, it makes no sense to calculate the average employee number.

To overcome these limitations, data definition expressions were included in the specification language of ADS. These expressions enable developers to instruct the system regarding relations and data restrictions that can not be collected from the database schema, and can be used as illustrated in figure 3.5. Given that the data type re-definition functionality, in the current implementation, serves only as a way to instruct ADS that a given attribute should be used as a numerical or a non-numerical attribute, we decided to only define two possible data types for this operation: Number and String. Using this option, one can instruct the system about which statistical characteristics can be requested for a given attribute. However, this conversion is only used by the system, and is not passed into the sample database.

As an example, consider that we wish to instruct the system that the attribute *phone_number* from the *employees* table should be used as a string, given that it would make no sense to request to maintain the average value of such attribute. And that *region_id* from the *countries* table is related with the *region_id* from the *regions* table. These definitions could be stated using the following expressions, can be formed as summarized in Figure 3.5:

```
define PHONE_NUMBER from EMPLOYEES as STRING;
define REGION_ID from COUNTRIES = REGION_ID from REGIONS;
```

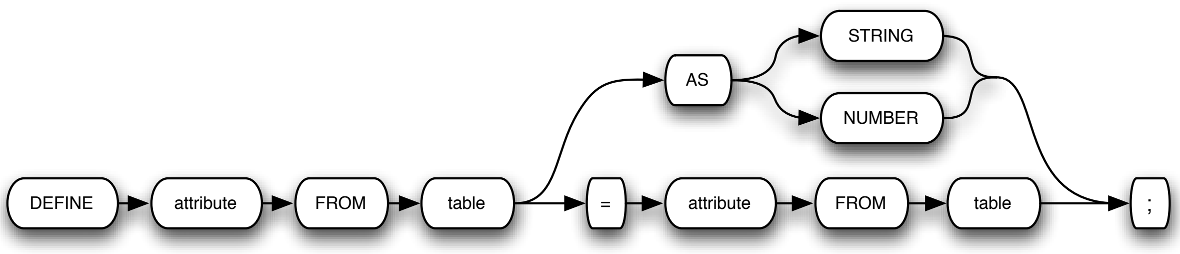

Figure 3.5: Data definitions usage diagram

3.4 Summary

In summary, ADS sample specification language is an inovative and intuitive way for developers to explicitly describe their specific needs for any given database sample. As presented, the language is simple enough for requesting a simple sample without effort, but is also expressive enough for requesting a more specific database sample, where one which to maintain information about the relations between the tables, or to provide a better definition for some attributes.

Furthermore, as a declarative language, the sample specification is composed by a set of restrictions that should be satisfied. Therefore, the user can concentrate in her work and only needs to think about which data characteristics she needs to have in her development environment, without worrying about how such sample can be obtained.

4

System architecture

The database samples produced by ADS are envisioned to be used in development and test environments. To this end, the system was designed using a modular architecture and can easily be adapted to any DBMS, offering the possibility of replacing the components that directly interact with the database.

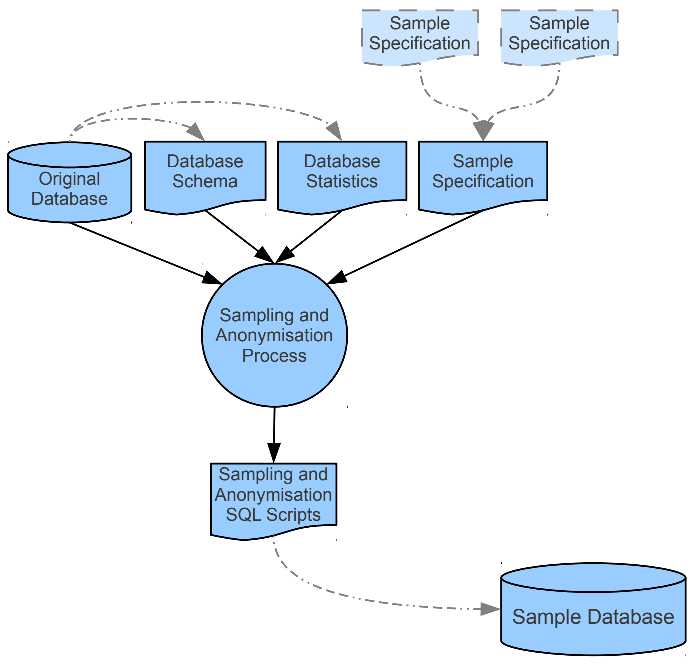

Figure 4.1: Anonym Database Sampler Process Diagram

As described in Figure 4.1, ADS input is fourfold: the relational database from which we need a sample; the table schema for the data contained in that database system; the statistical information about the data present in the database; and the specification of the intended sample, which can be composed by more than one sample specification file, as we explain later. Each one of these inputs is handled by a different component, which is used to collect the desired information about the database into the system. As it is further explained in Section 4.1, in the current system implementation both statistical data and schema reader components use some DBMS features to collect the desired information. Consequently, both of these components connect to the database through the database connector component.

The information gathered by these input components is assembled into the internal information data structure component, which is used as a gathering point where all the other internal modules can refer to when in need for information to operate. Additionally, the error, anonymisation and query handlers act as functionality wrapper components, externalising the implementation of the following operations, respectively: error calculation, query generation and data anonymisation. By separating these functionalities from the information data structure, we provide the possibility of adapting ADS to different realities. Moreover, this option will also ease further developments of ADS's features.

The sampling engine uses the query and error handlers, along with the information data structure, to create a temporary sample of the original database in an internal database. To this end, given that the DBMS containing the original database is also used to store this internal database, the same database connector component is used to interact with this internal database. One can find an illustration of the described system architecture in Figure 4.2. Each component of the architecture is further explained in Sections 4.1 and 4.2.

The system's implementation was performed in Java, not only for its "write once run everywhere" characteristic, but also due to the availability of a wide range of tools and libraries which diminished the effort required to proceed with the implementation of the system. Thereby, we used the Java Database Connectivity Driver (JDBC) to handle the database interactions, and Java Compiler Compiler (JavaCC) to generate the parser for the sample specification language. The reason behind this choice relates primarily to the fact that we have already used these technologies in the past. In addition, the chosen database system was Oracle DBMS given that it provides a statistical engine, an almost-standard SQL implementation and a fully compliant JDBC driver.

The current ADS implementation should be considered as an advanced prototype, with the main purpose of providing a proof of concept for our system, leaving room to further development and/or improvements. Nevertheless, we consider that ADS's flexible and modular architecture plays a major role to the future of this prototype, since it allows the optimisation of the current features.

For the rest of this chapter we present an overview of the components that constitute

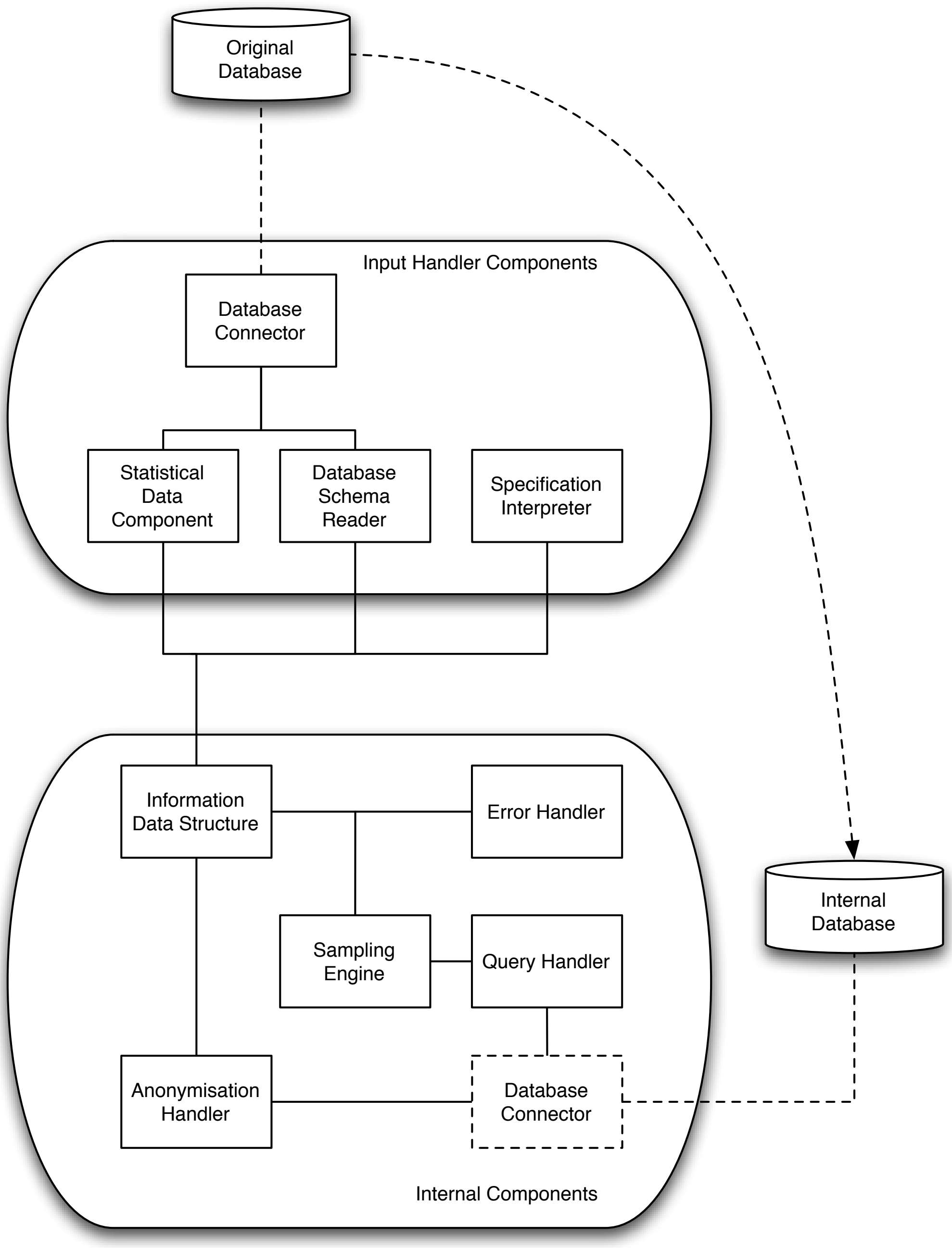

Figure 4.2: Anonym Database Sampler Architecture

the ADS architecture, while providing the most relevant details about their implementation. Given that the sample extraction process is one of the contributions from this thesis, a more complete explanation is given in chapter 5.

This way, we begin by presenting in section 4.1 the components that handle ADS requested inputs. Afterwards, in section 4.2, we discuss the system internal components. Finally, in section 4.3 we present an overview over the system's execution and possible outputs.

4.1 Input handler components

The input handler components play a major role in the system modularity. By delivering a separate and autonomous component to each input, each called by the system main process wen needed, we ensure that ADS interaction with external systems is adaptable to different technologies. For example, if one wishes to obtain a sample from a database contained in a DBMS that does not have a statistical engine, one can develop a component capable of finding the desired statistical information, or even use some external tool capable of finding the statistical data and then pass it into the system as an input file.

4.1.1 Database connector

The current ADS implementation uses JDBC to handle the database interaction. However, given that not every DBMS delivers a full JDBC implementation, we encapsulated it in a component to provide the possibility for using a different connection system.

This component is responsible for handling the connection to a given database, performing the desired queries in that database, and also to manage the database save-points providing the possibility of performing a rollback of an operation.

The database connector component consists of a single class, implementing the following interface:

```java
public interface DatabaseConnection {
  public String getDatabaseUrl();
  public String getUsername();
  public String getPassword();
  public Connection getConnection() throws SQLException;
  public void closeConnection() throws SQLException;
  public ResultSet doQuery(String query) throws SQLException;
  public int doUpdate(String query) throws SQLException;
  public void setAutoCommit(boolean on) throws SQLException;
  public void commit() throws SQLException;
  public void setSavepoint() throws SQLException;
  public void rollback() throws SQLException;
}
```

Listing 4.1: Database connector interface

4.1.2 Database schema reader

Given that Oracle DBMS fully implements the JDBC driver specification, we used of the JDBC `Metadata` object to obtain the names of tables contained in the database, along with their primary and foreign keys, which are used to create the system's data structure used to store the information about the desired sample. Furthermore, like all other input handler components, the database schema reader is designed as a self-contained autonomous module, so that it can easily be replaced by another which can obtain the database schema information in any other way.

This module contains two public methods, which should be called to obtain the database schema information into ADS data-structure:

1. `loadSchema(Database db)`, which is responsible for obtaining the schema table names, primary and foreign keys;

2. `getAndSetAttributeTypes(Database db)`, which is responsible to obtain the datatypes for all the attributes that are relevant to the sample that will be obtained - i.e. the key attributes and the attributes to which a statistical restriction was requested by the sample specification.

This separation relies on the fact that, while the schema information is collected prior to the sample specification being interpreted, the attribute data types are collected after that operation. This way one can obtain the data types for all the relevant attributes.

A `Database` object is received by each method, such an object being the database representation where the retrieved information is stored in the internal data structure. These public methods' implementation is presented in Listing 4.2 .

```
public static void loadSchema(Database db) throws SQLException,
    PrimaryKeyNotFoundException {
  loadTables(db);
  loadKeyAttributes(db);
}

public static void getAndSetAttributeTypes(Database db) {
  String allTablesString = CollectionUtils
    .collectionToCommaSeparatedQuotedString(db.getTables().keySet());
  String allAttsString = CollectionUtils
    .collectionToCommaSeparatedQuotedString(db.getAllAttributes());

  if (allTablesString != "" && allAttsString != "") {
    String query = "select table_name, column_name, data_type "
```

```
          + "from user_tab_cols where table_name in (" + allTablesString
          + ") AND column_name in ("+ allAttsString + ")";
    ResultSet rs = null;
    try {
      rs=db.getDirectConnection().prepareStatement(query).executeQuery();
      String table, att, type;
      while (rs.next()) {
        table = rs.getString("table_name");
        att = rs.getString("column_name");
        type = rs.getString("data_type");
        db.getTables().get(table).setAttributeType(att, type);
      }
    } catch (SQLException e) {
      System.out.println("Error while getting attribute data types");
      e.printStackTrace();
    } finally {
    if (rs != null) {
        try {rs.close();} catch (SQLException e) {e.printStackTrace();}
} } } }
```

Listing 4.2: Database schema reader public methods

To get the information about the tables in the database schema, as mentioned before, the JDBC `Metadata` object is used, being the table information also stored in the same `Database` object. This is made precise in Listing 4.3, where we present the method responsible for this operation.

```
private static void loadTables(Database db) throws SQLException {
  ResultSet tResultSet = db.getDirectConnection().getMetaData()
                             .getTables(null, db.getSchema(), null, null);
  String newTableName;
  while (tResultSet.next()) {
    OracleTable newTable;
    if (tResultSet.getString("TABLE_TYPE").equalsIgnoreCase("table")) {
      newTableName = tResultSet.getString("TABLE_NAME");
      newTable = new OracleTable(newTableName);
      db.createTable(newTableName, newTable);
    }
  }
  tResultSet.close();
}
```

Listing 4.3: Database schema reader loadTables method

After the table information is read, the same technique is used to obtain the primary and foreign key information into the same database representation object. To this end, we iterate over the previously created table objects, storing in each of them the corresponding primary and foreign key information, as presented in Listing 4.4 .

```java
private static void loadKeyAttributes(Database db)
    throws SQLException, PrimaryKeyNotFoundException {
  DatabaseMetaData metadata = db.getDirectConnection().getMetaData();
  String schema = db.getSchema();
  for (OracleTable table : db.getTables().values()) {
    loadPKeys(schema, metadata, table);
    loadFKeys(schema, metadata, table);
  }
}

private static void loadPKeys(String schema, DatabaseMetaData metadata,
    OracleTable table) throws SQLException {
  ResultSet pKeys = metadata.getPrimaryKeys(null, schema,
                                                        table.getTableName());
  String columnName, pKeyId;
  while (pKeys.next()) {
    columnName = pKeys.getString("COLUMN_NAME");
    pKeyId = pKeys.getString("PK_NAME");
    table.setPKeyAttribute(columnName, pKeyId);
  }
  pKeys.close();
}

private static void loadFKeys(String schema, DatabaseMetaData metadata,
    OracleTable table) throws SQLException, PrimaryKeyNotFoundException {
  ResultSet fKeys = metadata.getExportedKeys(null, schema,
                                                        table.getTableName());
  String pKeyId, fKeyTable, fKeyColumn, fKeyId;
  while (fKeys.next()) {
    pKeyId = fKeys.getString("PK_NAME");
    fKeyTable = fKeys.getString("FKTABLE_NAME");
    fKeyColumn = fKeys.getString("FKCOLUMN_NAME");
    fKeyId = fKeys.getString("FK_NAME");
    table.setFKeyAttribute(pKeyId, fKeyTable, fKeyColumn, fKeyId);
  }
  fKeys.close();
}
```

Listing 4.4: Database schema reader key related methods

4.1.3 Sample specification parser

As stated before, the sample specification parser was implemented using JavaCC. At the end of each expression, the system uses a visitor design pattern to pass into ADS internal data structure the collected information, implementing the sample specification object interface presented in Listing 4.5. This way, each line is interpreted as it is being read, and any possible errors are promptly detected and returned to the end-user. In addition, given that more than one restriction can be imposed on a single attribute, this implementation delivers the task of handling multiple restrictions to the internal components, simplifying the future substitution of this component.

```java
public interface SampleData {
  public void setSampleSize(int sampleSize) throws InvalidSizeException;
  public void setAnonim(int anonimisation) throws InvalidValueException;
  public void setError(int error) throws InvalidErrorValueException;
  public void setTableCondition(TableCondition condition)
      throws InvalidTableNameException, InvalidValueException;
  public int getSampleSize();
  public int getError();
}
```

Listing 4.5: Sample specification object interface

4.1.4 Statistical data

The statistical data component is tightly integrated with the sample specification parser, given that the possible statistical variables are incorporated in the sample specification language grammar. This integration is achieved by using the enumeration object presented[1] in Listing 4.6 which, when its correspondent value is obtained from the database, is used to return the corresponding statistical variable object. This way, although the current implementation uses Oracle's statistical engine, for querying the database about the desired attribute statistical value, this implementation can easily be overridden to encompass some other technique to obtain the desired statistics, without compromising the system integration.

```java
public enum StatVar {
  AVG("avg"), MAX("max"), MIN("min"), SD("stddev"), MEDIAN("median"),
    MODE("stats_mode" ), RATIO("");
    public String getSqlName(){ ... }
    public void setError(ErrorValue limit){ ... }
    public ErrorValue getError(){ ... }
    public StatisticVar getStatObject(Comparable<?> value){ ... }
    public StatisticVar getStatObject(){ ... }
}
```

Listing 4.6: Statistical variables enumeration object

[1]To help with the code readability, we omitted the private methods and the implemented code from Listing 4.6, presenting only an "Interface" for the implemented Enumeration

This enumeration is then used by the statistical data object to obtain and refresh the statistical data present in ADS internal infrastructure. To perform this operation, the statistical object obtains a list containing every attribute for which its restriction that had not been collected or accomplished, and obtains the desired values for the statistical attributes, which are directly stored in the internal data-structure.

The Oracle DBMS already provides some statistical functions, we used them to obtain the desired values from the database using a query. However, this is not the case for the ratio variable, which is not automatically provided by Oracle's statistics.

When we want to preserve the ratio of a given attribute in a table, we must compute, for each value V in the domain of that attribute, the percentage of tuples in that table that have V. This computation is easily done in SQL.

For example, if the sample specification includes:

`preserve ratio for department_id for Employees;`

Then the list of values that must be computed is the one obtained from the SQL code in Listing 4.7.

```sql
select count(*) into n from Employees;
select department_id, count(*)/n as r from Employees group by department_id;
```

Listing 4.7: Sample of a query for finding the ratio values for an attribute.

The computed list is passed to the corresponding StatisticVar object, and stored in ADS's internal data-structure for future operations.

4.2 Internal Components

ADS internal components can be viewed as three major groups: information data-structure, handlers and the sampling engine which, as stated before, is covered in the next chapter.

4.2.1 Information data structure

This is the component used to store the information collected from ADS inputs. Therefore, this data structure is designed to contain the database information and its schema representation: the database tables, and their primary and foreign keys which are used to represent the links between the tables. We also use this schema representation to store the sample specification, spreading its information within their referenced object representations. Likewise, the necessary statistical data is also stored in the objects that represent the attributes from where they were collected. Given that this module is used by all other internal modules to obtain the information they need, we also implemented methods to request some specific information directly, like asking for all the attributes from a give table that have any associated statistical variable restriction, or all the attributes from all tables for which the desired restrictions are yet to be fulfilled. On the other hand, no metadata is stored in this data structure, given the high memory cost that such option would imply.

To obtain information from the information data-structure component, one must use the methods provided by the `Database` interface, which can be found in Listing 4.8.

```
public interface Database {
  public String getSchema();
  public DatabaseConnection getConnection();
  public Connection getDirectConnection() throws SQLException;
  public int updateDBSize() throws SQLException;
  public int getDBSize();
  public void createTable(String newTableName, OracleTable table);
  public Iterator<OracleTable> getTablesByMaxTotalNumberOfpKeys();
  public Iterator<OracleTable> getTablesByMinTotalNumberOfpKeys();
  public String getDBLinkName() throws SQLException;
  public String toString();
  public void setTableCondition(TableCondition condition)
      throws InvalidTableNameException, InvalidValueException;
  public boolean isTableCreated(String tableName);
  public Map<String, OracleTable> getTables();
  public void setTables(Map<String, OracleTable> tables);
  public Map<String, List<Attribute>> getConditionedTables();
}
```

Listing 4.8: Internal data-structure database object interface

4.2.2 Query handler

Given that most DBMS *SQL* implementations are inconsistent with the standard and usually incompatible between vendors, we use a new component to handle the query generation. The centralisation of these operations allows the system to maintain its adaptability, without compromising the system's integration and efficiency. To ensure this, we define a query as a string variable that contains tags marking the parts that should be replaced with correct parameters (and according to the query's execution context). Then, each query has an associated operation that generates the final query given these variables. As a result of this design, in order to adapt this module to a different DBMS, one only needs to replace the current prototypes with their equivalent, using the destiny DBMS *SQL* specification.

```
    insert into {#table_name#} with Q as (
        select * from {#table_name#}@{#db_link#} {#condition#}
    ) select * from Q sample({#sample_size#})
```

Listing 4.9: Example of a string used to generate a query

As an example, consider the string presented in Listing 4.9. This string is used to obtain the query that will perform a copy of the data contained in a given table at the original database into the corresponding table in the sample database, being the original data restricted by a given condition. The generation of each query is handled by a method that not only performs a direct substitution of the tags with their corresponding values, but also converts the attribute's list of conditions into their corresponding *SQL* statements. One can find several examples of queries generated by this module in Chapter 5.

4.2.3 Error handler

This component is used to check that the data restrictions are ensured in the obtained sample, by verifying that the requested statistical values are within the specified error margin. To meet this goal, this component starts by identifying the correct object type for a given attribute restriction, obtaining the original value from the data structure component along with its defined error margin. Afterwards, the original value is compared with the current respective restriction value from the sample database. The difference between both values is returned if it is greater than the defined error margin. Otherwise, if the difference is within the error margin, the value is considered to be acceptable, and zero is returned. It is noteworthy that this method is specially tailored for numerical value comparison. Therefore, when confronted with non-numerical restriction comparisons (such as the median value for a date or a string), the given values and error margins are converted to numbers before comparison.

The error handler component consists of a single class, implementing the following interface:

```
public interface ErrorChecker {
    public int getDBSizeError(Database original, Database destiny);
    public int getStatVarError(StatisticVar<T> original,
        StatisticVar<T> destiny);
}
```

Listing 4.10: Error handler object interface

Both methods return an integer, which represents the distance from the current value to the original. This way, if the value is lower than the current, a negative number is returned. Likewise, if the value is higher than the current, a positive number is returned. In addition, these methods also verify the permitted error margin for the restriction being evaluated. If the value meets the error margin restrictions, then an acceptable value has been reached, and consequently 0 is returned.

In Figure 4.3 we present the error function implemented by both `ErrorChecker` public methods, which rely in some private methods implementing the calculation of the minimum and maximum values for the restricion being checked, considering the error

$$f(original, current) = \begin{cases} 0 & \text{if } min \leq current \leq max \\ original - current & \text{otherwise} \end{cases}$$

Figure 4.3: Error function

margin defined in the sample specification. These calculus are performed as shown in Figure 4.4.

$$min(original, error) = original - (original * \tfrac{error}{100})$$

$$max(original, error) = original + (original * \tfrac{error}{100})$$

Figure 4.4: Minimum and maximum error functions

4.2.4 Anonymisation handler

The anonymisation component is responsible for enciphering the data contained in the final sample. To this end, it uses a temporary table in the internal database to store each distinct value contained in the sample database, pairing it with its encrypted value. This table is then used at the end of the execution process, to perform a substitution of each identified value by its correspondent cypher.

The anonymisation handler component consists of a single class, implementing the following interface:

```
public interface Anonymiser {
    public void anonymise(Database destiny);
}
```

Listing 4.11: Error handler object interface

The `anonymise(Database destiny)` method is responsible for passing through every table contained in the internal data-structure database representation object, collecting the information about which attributes should be anonymised. Afterwards, every attribute's distinct value is collected into a temporary table, along with its corresponding cipher. Finally, this temporary table is used to perform a coherent substitution in every tuple where the original value is found.

4.3 System execution

As the system starts, it connects to the database from which the sample is to be extracted, collecting its schema information. This information is then used to create the schema representation in the information data structure component, by building objects to represent each table, its contained attributes and data types, along with their primary and foreign keys.

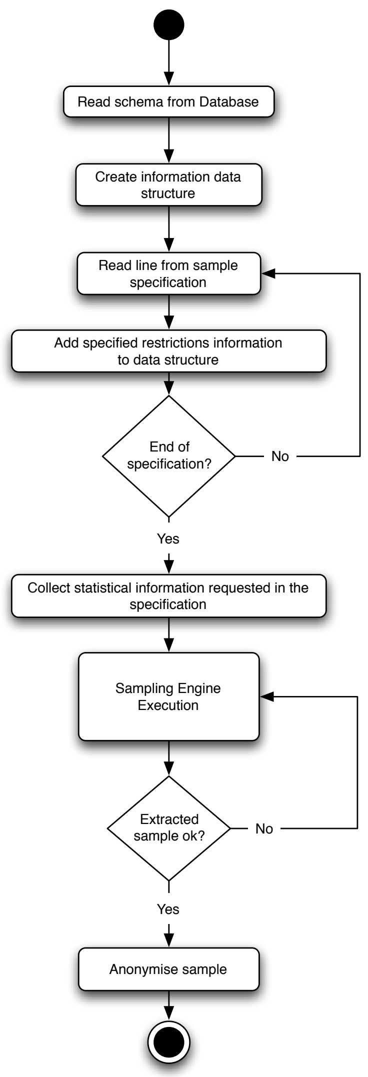
Read schema from Database
Create information data structure
Read line from sample specification
Add specified restrictions information to data structure
End of specification?
No
Yes
Collect statistical information requested in the specification
Sampling Engine Execution
Extracted sample ok?
No
Yes
Anonymise sample

Figure 4.5: Anonym Database Sampler Execution Diagram

Afterwards, ADS reads the sample specification, storing the restrictions definitions obtained in each line in its correspondent attribute representation in the data structure component. When the end of the sample specification is reached, the statistical variable values defined in the specification are obtained from the database, which are stored in their correspondent attribute restriction representation.

After all the data is collected, ADS proceeds to the sampling engine execution, performing the necessary actions to obtain a representative sample from the database, which are described in Chapter 5.

When this execution is concluded, ADS presents the user with the statistical information about the obtained database sample, highlighting the possible differences that can occur and asking what action should be performed next. If the user is not satisfied with the obtained sample, she can restart the sampling operation. If the obtained sample meets her needs, the system proceeds to the anonymisation operation and returns the sample database creation and data insertion scripts to the user. The above system execution description is sumarised in Figure 4.5.

5 Sampling engine

The sampling engine is the component that handles the sample extraction process, i.e. the process that selects and copies data from the original database into the sample's database.

In order to implement such a sampling engine, one could try to simplify by randomly sampling each of the database's tables, and subsequently verifying if all the specified restrictions for each table were fulfilled. However, this solution would only work if all of the database attributes had a normal distribution, since every tuple from each table have the same probability of being selected. And even in that situation, such a sample would hardly maintain every foreign-key restriction present in the original database, i.e. the sample database thus obtained would hardly be consistent.

In alternative to simple random sampling, ADS offers another sampling method denoted stratified random sampling [MM44]. This sampling works by dividing the population in disjoint groups (*strata*), and then performing an independent random sample on each of them. Moreover, the same percentage of tuples is selected from each *strata*, ensuring that the desired sample size is achieved. This sampling technique allows a better performance than simple random sampling in non-normal distributions and remains equally efficient with normal distributions. This way, it is ensured that the proportions of each original *strata* are maintained in the extracted sample, therefore reducing the sampling error and helping to create a base sample where the data distribution is closer to the original. Even so, one can not guarantee to comply with all the restrictions specified by the developer, mainly due to the fact that every tuple from a given *strata* has the same probability of being selected. And, like in the simple random sampling technique, foreign-key restrictions could hardly be maintained.

Given this, since we need to ensure all the developer's specified restrictions and, even more importantly, the sample database consistency, we needed to implement some complementary technique. This technique, when combined with the previously described method, should ensure that all the restrictions were fulfilled. Therefore, we developed a *First-choice hill-climbing* [RN03] algorithm to optimise the sample data by achieving the desired values for the sample's statistical data, while also ensuring that the resulting database sample was sound.

As a local search algorithm, rather than considering the multiple possible paths to achieve a current state, the *First-choice hill-climbing* operates by considering a single state. Then the algorithm only moves to neighbours of that state, i.e. states that can be reached from the current one by performing some sort of action. Using such a local search algorithm has two main advantages: (1) a very little amount of memory is required; and (2) it is often possible to find reasonable solutions in large or infinite (continuous) state spaces for which other systematic algorithms are unsuitable. Both of these advantages are particularly suitable to our problem, given the typically large size of our standard databases.

In a nutshell, a local search algorithm runs as a loop that continually moves in the direction of increasing value, i.e. uphill, and terminates when it reaches a "peak" where no neighbour has a higher value. Since there is no need to maintain a search tree, the data structure for the current node only needs to record the state and the value of the goal function.

Moreover, as previously mentioned, we use a stochastic hill climbing variation (*First-choice hill-climbing*), that randomly chooses a movement from the set of all available uphill moves. This usually converges more slowly than the classical *hill-climbing* algorithm (also known as steepest ascent), but in some state landscapes, it achieves better solutions [RN03]. Since a state may have many (e.g. thousands of) successors, this variation generates successors randomly, until one of the generated is better than the current state.

Nevertheless, while combining these two techniques explained above we can achieve an accurate data combination sample, such combination may be impossible to obtain. On one hand, the foreign key restrictions can force us to break some statistical restriction. On the other hand, the data distribution for a given attribute can even be so characteristic that it is not possible to maintain some statistical value for that attribute without breaking the sample size restriction. For example, imagine that we have a two person population to sample, father and son. It would be impossible to obtain a sample of such population that maintains its average age, unless the sample matches the population.

Given this, the implemented sample process is three-folded. First, ADS uses a stratified random sampling technique to produce an initial sample. Afterwards, a *First-choice hill-climbing* optimisation algorithm is used to iterate over this base sample, swapping the base sample tuples in order to make them closer to the desired values. Finally, if some restrictions are yet to be fulfilled, a tuple and/or key modification is performed to ensure that the final sample fully complies with the initial sample specification. An illustration of this sample process can also be found in Figure 5.1.

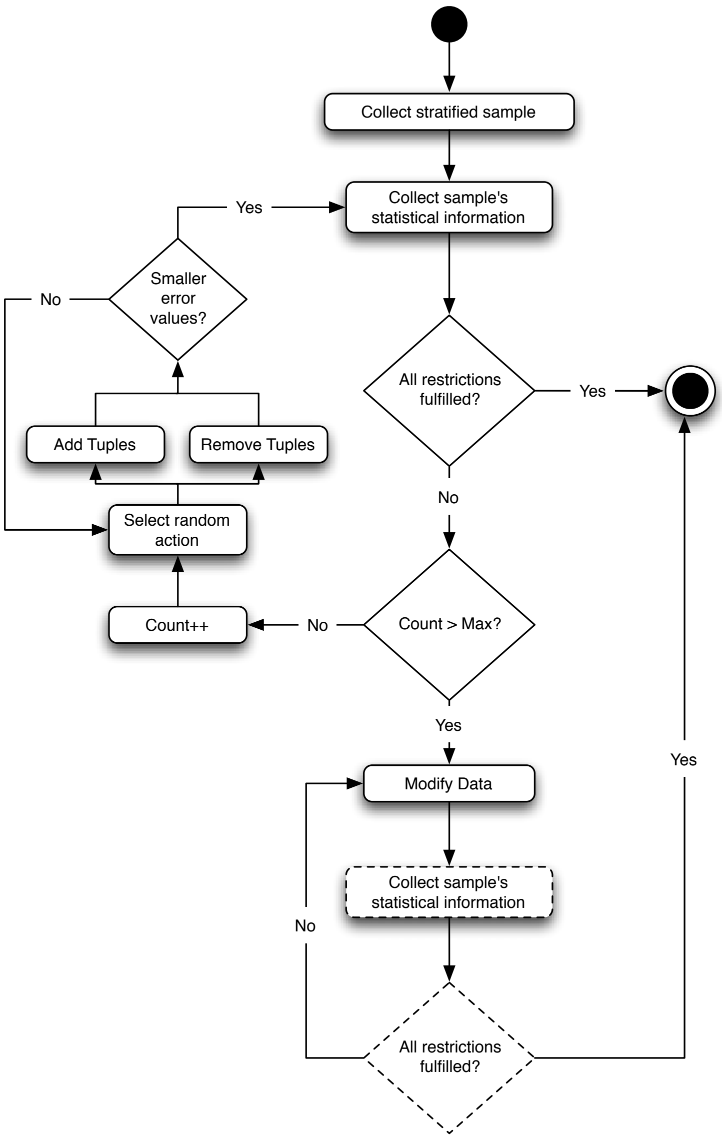
Collect stratified sample
Collect sample's statistical information
Yes
Smaller error values?
No
All restrictions fulfilled?
Yes
Add Tuples
Remove Tuples
No
Select random action
Count++
No
Count > Max?
Yes
Modify Data
Yes
Collect sample's statistical information
No
All restrictions fulfilled?

Figure 5.1: Sampling Engine Execution Diagram

Instead of collecting information at run-time, the sampling engine process only uses the information obtained by the previous components presented in Chapter 4. Using the *select into* clause, ADS performs direct copies from the original to the internal database through a database link, relying on the system components to obtain the information used to verify the differences between the sample and the original databases. Therefore, we deliver to the DBMS the task of copying the desired data, lowering ADS memory usage.

5.1 First step: stratified random sample

In this step ADS extracts its base sample, using the statistical values defined in the sample specification to create a stratified random sample from the desired tables. If no statistical values are defined for a table, a simple random sample is extracted from that table.

In the case of the stratified random sample, the statistical variables are used to define the *strata*, i.e. the database population is divided into disjoint groups by splitting each table using all its specified statistical variables values. A random sample is then performed for each *strata* independently, collecting the same percent of data from each. This way, by defining more restrictions for a given table, one is increasing the stratification of the sample extracted from such table, thereby providing a better representation of the table's data distribution.

To handle the random tuple selection, the current implementation relies on Oracle's DBMS *select sample* clause, which allows to instruct the database to select from a random sample of data from a table, rather than from the entire table. Likewise, to perform a stratified sample, we use this same clause on a sequence of queries, each one with a different sub-query limiting the sampled table data to the tuples that fall within the range of the desired stratification, using the statistical variables values obtained by the statistical data module.

As an example, in Listing 5.1 we present the queries generated for extracting a fifty percent stratified sample from the original database. This stratification was created by requesting ADS to maintain the average value for the employees salary, the ratio for the different department locations, and the mode of the existent job positions maximum salary value, using the following sample specification:

```
sample size 50% with error=5 and anonymisation = 1;
preserve avg for SALARY from EMPLOYEES with error=10;
preserve ratio for LOCATION_ID from DEPARTMENTS;
preserve mode for MAX_SALARY from JOBS;
```

These generated queries are sequentially executed in a separated database schema, creating the base sample from which the anonymised sample database is created.

```sql
create table EMPLOYEES as with Q as (
        select * from EMPLOYEES@originalDB where SALARY<=6461.6822
    ) select * from Q sample(50)
insert into EMPLOYEES with Q as (
        select * from EMPLOYEES@originalDB where SALARY>6461.6822
    ) select * from Q sample(50)
create table JOBS as with Q as (
        select * from JOBS@originalDB where MAX_SALARY=9000
    ) select * from Q sample(50)
create table JOBS as with Q as (
        select * from JOBS@originalDB where MAX_SALARY<>9000
    ) select * from Q sample(50)
create table DEPARTMENTS as with Q as (
        select * from DEPARTMENTS@originalDB where location_id=1400
    ) select * from Q sample(50)
insert into DEPARTMENTS with Q as (
        select * from DEPARTMENTS@originalDB where location_id=1700
    ) select * from Q sample(50)
insert into DEPARTMENTS with Q as (
        select * from DEPARTMENTS@originalDB where location_id=2700
    ) select * from Q sample(50)
insert into DEPARTMENTS with Q as (
        select * from DEPARTMENTS@originalDB where location_id=2400
    ) select * from Q sample(50)
insert into DEPARTMENTS with Q as (
        select * from DEPARTMENTS@originalDB where location_id=1800
    ) select * from Q sample(50)
insert into DEPARTMENTS with Q as (
        select * from DEPARTMENTS@originalDB where location_id=2500
    ) select * from Q sample(50)
insert into DEPARTMENTS with Q as (
        select * from DEPARTMENTS@originalDB where location_id=1500
    ) select * from Q sample(50)
create table REGIONS as with Q as (select * from REGIONS@originalDB)
    select * from Q sample(50)
create table LOCATIONS as with Q as (select * from LOCATIONS@originalDB)
    select * from Q sample(50)
create table JOB_HISTORY as with Q as (select * from JOB_HISTORY@originalDB)
    select * from Q sample(50)
create table COUNTRIES as with Q as (select * from COUNTRIES@originalDB)
    select * from Q sample(50)
```

Listing 5.1: Generated queries for the stratified sample extraction

Additionally, if one specified that the maximum or minimum value of an attribute from a given table should be kept, ADS performs a direct copy of the tuple where that value is found from the original to the sample database, while excluding it from any *strata* where it could be found.

When extracting a stratified sample, the samples' original statistical values may not be maintained, depending on the sample's sheer size. Statistically speaking, if the sample is big enough, the contained data statistical variables are probably kept according to the original, but one can not ensure that it will occur. Nevertheless, developers should be able to specify the sample size that better fits their needs, thus limiting the minimum sample size does not fit in the goals of this work.

This way, and also to not interfere with the randomness of the tuple selection, during this step no "foreign-key" restrictions are ensured. Instead, these restrictions are used in the following step, to help choosing the tuples that should be selected when performing the base sample optimisation.

5.2 Second step: *First-choice hill-climbing* optimisation

In this second step, ADS uses a *First-choice hill-climbing* algorithm to optimise the sample's statistical data, i.e. to perform a set of randomly chosen actions, exchanging tuples towards the desired statistical values. The goal of this algorithm is to keep the statistical values of the sample within the error margins defined during the specification, while simultaneously checking the integrity of the sample's data and striving to ensure that all the foreign-key restrictions present in the original database are also fulfilled by the sample. To accomplish this goal, the algorithm performs a set of randomly selected actions, adding or removing tuples.

Each possible action has a corresponding query generator method, which is responsible for generating the appropriated query for executing it. This query is then executed in the sample database, adding or removing the tuples from the sample database.

As an illustration, the query presented in Listing 5.2 was generated for adding all the tuples that were missing in the primary key table but were present as foreign key in given table. More specifically, this query will copy from the original *locations* table all the tuples from which its *location_id* can be found in the sample *departments* table, but not in the sample *locations* table.

```sql
insert into LOCATIONS (
    select * from LOCATIONS@originalDB o
        where o.LOCATION_ID in (select LOCATION_ID from DEPARTMENTS)
        and  o.LOCATION_ID not in (select LOCATION_ID from LOCATIONS)
    )
```

Listing 5.2: An example of a generated query for adding tuples

Likewise, the query presented in Listing 5.3 was generated for removing all the tuples from a given table for which their foreign keys can not be found in their corresponding primary key table. More specifically, this query will remove from the *job_history* sample table all tuples for which its *employee_id* cannot be found in the *employees* sample table.

```
delete from JOB_HISTORY o where o.EMPLOYEE_ID not in
    (select EMPLOYEE_ID from EMPLOYEES)
```

Listing 5.3: An example of a generated query for removing tuples

In an attempt to keep the statistical values within the pre-defined error margins, after performing each action, ADS re-collects the sample data statistics. If this action brings the statistical values closer to their counterparts in the original database, the changes are stored as a save-point in the sample's database. Otherwise, changes are simply rolled-back. For each iteration, ADS independently performs the actions described above on all the database's tables. After an action is attempted in each table, ADS collects all the statistics and compares them with the ones previously obtained using the function presented in 5.2. If a positive result is returned - i.e. if, overall, the new statistics are closer to the original than the old ones - all save-points are committed to the sample database. If not, all save-points are discarded.

$$\sum_{restriction=1}^{n} \|previousErrorValue_{restriction}\| - \sum_{restriction=1}^{n} \|currentErrorValue_{restriction}\|$$

Figure 5.2: Cost function for the sample optimisation algorithm

In short, ADS takes the difference from the current to the desired statistical values, i.e. the current error values, as its cost function, and uses the foreign-key restrictions to define the algorithm's neighbourhood, in which tuples can be added if their foreign keys are already present in the sample database, or removed if the primary key corresponding to its foreign key cannot be found in sample database.

The algorithm stops when at least one of the following conditions occurs: (1) the sample data is sound, in the sense that it complies with the foreign-key restrictions; (2) all statistical values are within the specified error margins. But neither the first condition is necessarily the phase where the desired statistical values are met, nor the second condition corresponds to a phase where all the foreign-key restrictions are satisfied. On the other hand, the potential size of a database and the consequent number of possible combinations, along with possible incompatibilities between the desired statistical values and any tuples still to be added or removed, may lead the algorithm to end up running indefinitely or never generate a sound database. Therefore, it is also possible to configure ADS to stop this step of the sampling process after a bounded number of iterations.

5.3 Third step: Data modification

In spite of the precautions discussed in the previous Sections 5.1 and 5.2, there is always a chance that inconsistent data remains in the sample. If that is the case, ADS performs one further step at this stage, which involves two possible actions: tuple and key modification. Tuple modification ensures that an attribute's statistical values are within the predefined range. For instance, if an attribute's average is lower than desired - i.e. lower than in the original database - ADS adds the difference between the current and the desired statistical value to all values of that attribute. This is only performed to values whose final result will not surpass the attribute's predefined maximum, in an effort to maintain the original statistical distribution. Similarly, if an attribute's average is higher than desired, ADS subtracts said difference to all values, only this time paying attention not to fall behind the current minimum value.

As an example, consider that we sampled a table of employees where, among other restrictions, we specified that the average of the employees salary should be maintained with a 10% error margin. To simplify, assume also that original average salary was 2500, which applying the defined error margin would gave us a value range from 2250 to 2750. If the values presented in Table 5.1 were final, this average salary restriction would not be fulfilled, given that the average employees salary in that table is 2812.5.

EMPLOYEE_ID	SALARY	DEPARTMENT_ID
125	3200	50
127	2400	100
130	2800	50
134	2900	80
135	2400	80
140	2500	80
141	3500	50
183	2800	50

Table 5.1: Example of a employees table

Therefore, as explained, if such a table arrived to the data modification step of ADS sampling engine, the system would subtract the difference between the actual and the original average salary ($2812.5 - 2500 = 312.5$) to all values where such operation would not modify the actual range of values, using a generated query as the one presented in Listing 5.4.

```sql
update employees
    set salary = salary-312.5
    where salary-312.5 > (select min(salary) from employees)
    and salary <> (select max(salary) from employees)
```

Listing 5.4: An example of a generated query for modifying tuples

This way, Table 5.1 would be modified into the Table 5.2. As one can observe by comparing both tables, the system did not modify any of the values that would fall behind the minimum salary value, nor the maximum salary value. Particularly, the actual average salary value for Table 5.2 became 2656, which is an acceptable value given that is within the defined value range.

EMPLOYEE_ID	SALARY	DEPARTMENT_ID
125	2888	50
127	2400	100
130	2488	50
134	2588	80
135	2400	80
140	2500	80
141	3500	50
183	2488	50

Table 5.2: Example of a employees table after tuple modification

Key modification, in turn, replaces any key containing mismatched data with one randomly selected from the sample. To exemplify this, consider the previous resulting Table 5.2, and also Table 5.3, where one can find the departments names for which the employees work. As one can observe, although in Table 5.2 we have an employee that works for the department that has an id equal to 100, no such department can be found in Table 5.3.

DEPARTMENT_ID	DEPARTMENT_NAME
130	Corporate Tax
160	Benefits
80	IT Helpdesk
50	Shipping

Table 5.3: Example of a departments table

This way, the system will randomly pick a department id from the ones he already have in the departments table, for example changing the previously mentioned id from 100 to 160. Therefore, table 5.2 would be transformed into table 5.4.

EMPLOYEE_ID	SALARY	DEPARTMENT_ID
125	2888	50
127	2400	160
130	2488	50
134	2588	80
135	2400	80
140	2500	80
141	3500	50
183	2488	50

Table 5.4: Example of a employees table after key modification

It is worth noting that both tuple and key modification can be performed only because the resulting samples are intended as user-defined representative datasets for use in test and development environments, and are not appropriate for any kind of statistical data analyses. This way, while it is crucial to ensure data consistency along with the user-defined restrictions, the truthfullness of the obtained data can be relaxed. It is not necessary to maintain all the original data statistical characteristics to obtain a database sample that can be used to accuratelly perform a specific application test.

6 Tests and result analysis

To correctly evaluate ADS's features, the test-phase should take into account three different databases: a small generic database, to be used also in the development environment; a mid-scale database having as many relations as possible between its tables (allowing us to perform some initial tests and also evaluate the system performance when confronted with many relations); and a large scale database, to measure the performance evaluation of our system.

However, due to time restrictions, we only used one database for testing ADS - the Oracle-XE default example database, which is the same database we used in Chapter 3 examples and also the database used in our development environment.

To this end, we performed four sets of tests, two extracting a simple random sample from the original database, and two more extracted by the ADS sampling engine algorithm presented in Chapter 5. The goal of these sets is to be able to compare ADS's sampling process with a simple random sample, and also study both methods facing different sample size restrictions. This way, in each two sets a twenty five and a fifty percent sample is requested. Given the amount of data contained even in a small database like this, and in order to be possible to compare each of these values within all the tests, we decided to only present the values that will be specified as statistical variable restrictions in ADS's execution: the maximum and average values for the salary attribute from the employees table, and the minimum and average value for the hire date attribute from that same table. In addition, we also present these same values obtained from the original database, presented as a range of values when appropriate, in order to be possible to compare the test results with the desired values. Given that all of these methods use some random operations, each set of tests comprises five independent runs of the sample extraction.

It is noteworthy that it is harder to obtain a representative sample from a small population. In fact, this difficulty is also the reason why we chose such sample sizes for our tests. Given the small size of the original database, requesting a ten or fifteen percent sample of such database resulted in a twenty or thirty tuple sample database, which resulted in a two or three tuples-per-table database, where very little could be performed in order to achieve a representative sample, and many times resulted in a impossible problem, i.e. there was no possible tuple combination that could be selected in order to achieve such a sample.

6.1 Simple random sample

As discussed in Chapter 5, if no statistical values restrictions are defined for a table in the sample specification, ADS extracts a simple random sample from that table. This way, if no statistical restrictions are defined for any table, ADS extracts a simple random sample from the whole database. As the sampling engine has no statistical values to maintain, it only guarantees database consistency, ensuring that a sound database sample is returned.

The results presented in Table 6.1 are the values obtained in five independent extractions of a random sample containing twenty five percent of the original database, with a ten percent error margin, and performing the anonymisation operations. This was achieved by using the following sample specification:

```
sample size 25% with error=10 and anonymisation = 1;
```

As can be observed in 6.1, very little can be guaranteed from such sample. For each run, there is a considerable variation in all of the observed values, and only by chance some of the resulting values are close to the ones found in the original database. Thus, not much can be discussed based on these results given that, as expected, only a few match the desired values. The only exceptions are the ones concerning the maximum salary and the minimum hire date, which match the expected value in three of the five test runs. Since all their matches occur in the same runs, both variables seem to be related.

Table 6.2 shows the values obtained in another five independent extractions, and this time a fifty percent simple random sample was requested, with a five percent error margin and performing the anonymisation operations. Such sample can be obtained using a specification similar to the previous:

```
sample size 50% with error=5 and anonymisation = 1;
```

The results from these runs are completely different from the previous ones, and mostly all results comply with the desired values. As in the previous test, these results are also close to what we expected. This is a consequence of extracting a greater sample, which increases the probability of selecting a more representative sample. And this is true for almost any sample whose size is equal to or greater than fifty percent of the original population sample, simply because we are selecting a higher number of elements.

	Desired Value	1st Run	2nd Run	3rd Run	4th Run	5th Run
Database Size	[48;59]	57	55	57	50	58
Average Salary	6462	8309	5357	8541	11185	10563
Max Salary	24000	13000	24000	24000	17000	24000
Min Salary	2100	4200	2200	2400	4400	4800
Median Hire Date	15-12-97	17-08-94	10-10-97	18-07-96	13-01-93	24-03-97
Max Hire Date	21-04-00	07-02-99	06-02-00	12-12-99	07-12-99	29-01-00
Min Hire Date	17-06-87	17-09-87	17-06-87	17-06-87	17-09-87	17-06-87

Table 6.1: Simple random sample specifying a sample size of 25%

	Desired Value	1st Run	2nd Run	3rd Run	4th Run	5th Run
Database Size	[102;112]	108	109	109	110	112
Average Salary	6462	6634	6102	5947	6932	7225
Max Salary	24000	24000	24000	24000	24000	24000
Min Salary	2100	2200	2200	2200	2100	2100
Median Hire Date	15-12-97	28-09-97	09-07-97	28-08-97	30-10-97	02-12-97
Max Hire Date	21-04-00	24-03-00	08-03-00	08-03-00	21-04-00	08-03-00
Min Hire Date	17-06-87	17-06-87	17-06-87	17-06-87	17-06-87	17-06-87

Table 6.2: Simple random sample specifying a sample size of 50%

However, as it is easy to see from the average salary value for the fifth run values, all values are still conditioned by the probabilities of the tuple selection. In this way, we cannot guarantee to hold any statistical variable value, nor ensure that the original data will present a normal distribution. Furthermore, we cannot even choose a particular data distribution, given that ADS should be able to extract a sample from any relational database.

Thereby, a simple random sampling mechanism would not be sufficient to achieve our goals, as a high probability of complying with a particular restriction also means that there is some probability of not complying with such restriction.

6.2 ADS sampling engine sample

The results presented in this section are obtained using ADS's complete sampling process. As discussed in Chapter 5, this process begins by extracting a stratified random sample from the original database, using the specified statistical variables values to determine its *strata*. Afterwards, a *First-choice hill-climbing* optimisation is performed, adding or removing tuples towards the desired statical values while ensuring consistency in the sample database. Finally, if after this second step some restriction remains unfulfilled, a data and/or key modification step is performed, ensuring that all restrictions are met. These final modifications are only possible due to the goal of the extracted samples: being used as development and test environments databases. As discussed in Chapter 1, while having a representative and sound database can be a great advantage to developers, we assume that this representativeness does not need to comply with truly statistical representativity, which would be much more expensive to obtain. Nevertheless, given that these data modifications are only performed after ADS tried to optimise the extracted sample, we also ensure that the extracted sample data distribution is as close as possible to the original. And if all the restrictions are already fulfilled, no data modification is performed.

The results presented in table 6.3 are obtained from five independent ADS executions, where we specified that it should maintain the maximum and medium values for the salary attribute from the employees table, and also the median and maximum hire date for that same table. This specification was performed as bellow:

```
sample size 25% with error=10 and anonymisation = 1;
preserve max, avg for SALARY from EMPLOYEES with error=10;
preserve min, median for HIRE_DATE from EMPLOYEES;
```

As it can be observed, not only all the specified restrictions are met, but also all the resulting values a generally much closer to the original values than the ones presented in table 6.1, from the twenty five percent simple random sample. We think that this is a very good result, mainly related with better tuple collection distribution provided by the initial stratification, followed by the performed sample optimisation. In addition, it should be worth mentioning that while a data modification was performed to the medium salary value in all runs, no data modification was applied to the median hire date at the fourth and fifth runs. On the other hand, some key modifications also occurred, which made possible to maintain the median hire date unchanged in those runs.

Table 6.4 presents the values from another five independent extractions, this time specifying a fifty percent sample. As before, such sample can be obtained using a specification similar to the previous:

```
sample size 50% with error=5 and anonymisation = 1;
```

	Desired Value	1st Run	2nd Run	3rd Run	4th Run	5th Run
Database Size	[48;59]	59	57	49	55	55
Average Salary	[5815;7108]	6631	6461	6338	6674	7014
Max Salary	24000	24000	24000	24000	24000	24000
Min Salary	2100	2100	2500	2500	2400	2400
Median Hire Date	15-12-97	11-03-97	17-08-97	10-07-97	10-03-97	10-03-97
Max Hire Date	21-04-00	03-02-00	10-08-99	29-01-00	21-04-00	24-03-00
Min Hire Date	17-06-87	17-06-87	17-06-87	17-06-87	17-06-87	17-06-87

Table 6.3: ADS results specifying a sample size of 25%

```
preserve max, avg for SALARY from EMPLOYEES with error=10;

preserve min, median for HIRE_DATE from EMPLOYEES;
```

Although at first sight the results from these test runs may appear to be worst than the ones from the previous test presented in table 6.3, that is not the case. ADS only performed data modification operations in the third and fifth runs, being able to find a suitable tuple combination in all the other runs. Additionally, the results obtained from a fifty percent simple random sample, i.e. the ones presented in table 6.2, can also look generally better than the ones obtained with this test. But that also is not true. As discussed before, ADS's ultimate goal is to obtain a database sample accordingly with a sample specification, and in this test we achieved that goal without performing any data modification in three of the five runs. And all the statistical value restrictions were always fulfilled in all the test runs, creating a uniform and much more reliable result-set.

As said before, these tests should be considered as a proof of concept validation, and should be complemented by sampling some other databases in future work. To that end, we identified two candidate systems for evaluation: the Mondial database [May99], which is a mid-scale freely available database containing world-wide geographical information, featuring hundreds of relations between its tables; and the complete Oracle example database package, which is a large-scale database that mirrors a large company database, featuring a few hundred gigabytes of information. For more information about the Oracle examples databases one can refer to [gR11].

As we discovered that the Mondial database schema does not contain any foreign-key restriction, which ADS uses to ensure data consistency. Such database would be a great candidate for testing the data redefinitions, and their relevance in ADS capabilities of succefully fullfill the specified objectives. In addition, the full Oracle example database

	Desired Value	1st Run	2nd Run	3rd Run	4th Run	5th Run
Database Size	[102;112]	104	108	109	112	110
Average Salary	[5815;7108]	7101	7077	6393	7088	6762
Max Salary	24000	24000	24000	24000	24000	24000
Min Salary	2100	2400	2100	2200	2200	2200
Median Hire Date	15-12-97	20-10-97	18-10-97	16-11-97	28-11-97	28-09-97
Max Hire Date	21-04-00	23-02-00	06-02-00	08-03-00	21-04-00	08-03-00
Min Hire Date	17-06-87	17-06-87	17-06-17	17-06-87	17-06-87	17-06-87

Table 6.4: ADS results specifying a sample size of 50%

pack (which is a commercial product) constitutes a close-to-reality large database, featuring some more advanced database data organisation techniques such as partitioning. This way, it would be a great way to perform some large-scale database sampling test.

7 Conclusions and future work

In this work we present ADS, a flexible and modular system capable of extracting an anonymised, consistent and representative sample from a relational database, to be used in testing and development environments, given a specification of what needs to be kept in the sample.

To the best of our knowledge, not much has been done in obtaining representative and consistent samples from a database in order to be used in application development environments. Therefore, we believe that this is an important research, which not only paves the way for future methods to create development and test databases based in real data, but also helps to bridge the gap between academic research and the common organisation problems. The clear advantage of our work is that our solution relies on real data, which allows developers to get a deeper understanding of the said data characteristics. Consequently, we believe this approach to be a powerful tool that helps developers to create more accurate and therefore better applications.

Although the tests performed in Chapter 6 are insufficient to evaluate and make a final assumption about ADS's time and memory consumption, they show that the prototype is capable of providing an anonymised sample accordingly to the sample specification. Moreover, despite the size of the database used for testing was not as large as desired, the results indicate our method to be quite fast in finding a solution.

It is also worth noting that it is harder to obtain a representative sample from a small population. In fact, in a small population, it is more likely that each element has a higher relevance towards the population characterisation than in a larger one. As an extreme example, consider two people: a father and a son. To obtain a small sample from this population that maintains its age average, we can only select one of the elements of the population, and so we would never obtain such average, given that both would have the same relevance regarding that characteristic.

While the usefulness of a small sample database could be questioned (and in a real-life scenarios such database could be simply anonymised and directly delivered to developers), we consider that successfully obtaining a 25 percent sample from such a small database was an encouraging result. Applying our solution to a bigger database would probably achieve at least the same level of success, as our trials with a 50 percent sample seem to indicate.

Nevertheless this implementation should be considered as a working prototype for a proof of concept, that leaves room for further development and/or improvements. To this end, we consider that ADS's flexibility and modularity plays a major role since it eases the optimisation of the current features. Moreover, these characteristics also allow the creation of new features that were not considered during the development of this thesis.

For instance, one of the features that can be greatly improved is the data anonymisation. This anonymisation is currently implemented through a cypher which creates a consistent data mask that prevents data from being directly disclosed. However, as previously discussed in Chapter 2, this method can be easily reversed by using some data analysis techniques. This way, one could develop a different anonymisation module using *k-anonymity* [Swe02], which was proven to be a much more reliable data anonymisation technique, and is already supported by the current sampling specification language through the "anonymisation level" parameter. In fact, the *k-anonymity* concept lies in ensuring that an attacker can not match any of the anonimysed data into less than k original individuals. As a result, such implementation could be improved by merging k similar tuples into a single one, while maintaining the desired statistical values. This way, one could also use this technique to create a different sampling step, where similar tuples were merged instead of being randomly selected.

Also regarding further research, the sample optimisation algorithm described in section 5.2 can be extended not only to perform some new actions to find different neighbours for a given state, but also to apply alternative local search algorithms, such as simulated annealing [RN03] or genetic search [RN03]. Note that, the whole sampling engine was carefully designed as a separated component and where each possible operation is also defined in a separated sub-component. This design makes possible the re-usage of the current implementation and eliminates the overhead of developing all the internal components that were created in the production of this thesis.

If a different sampling engine is used, bearing in mind that no direct data modification was carried out, ADS could also be easily adapted to different areas of study, like data-mining or database cleaning.

Furthermore, another interesting further work is to create a graphical user interface. Such interface would allow developers to define the sample specification directly in the database entity-relationship diagram, thus providing an even more intuitive way to developers to express their needs. Moreover, since the language interpreter is also implemented as a separate component, one could easily replace the interpreter by such interface.

Bibliography

[BG00] Jesús Bisbal and Jane Grimson. Database prototyping through consistent sampling. In *In the International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR'2000), to appear. Scuola Superiore Guglielmo Reiss Romoli (SSGRR*, 2000.

[BH06] Paul G. Brown and Peter J. Haas. Techniques for warehousing of sample data. In *ICDE*, page 6, 2006.

[CHY96] Ming-Syan Chen, Jiawei Han, and Philip S. Yu. Data mining: An overview from a database perspective. *IEEE Trans. Knowl. Data Eng.*, 8(6):866–883, 1996.

[CMN98] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD Conference*, pages 436–447, 1998.

[Coc77] William G. Cochran. *Sampling Techniques, 3rd Edition*. John Wiley, 1977.

[Dal86] Tore Dalenius. Finding a needle in a haystack or identifying anonymous census records. *Journal of Official Statistics*, 2(3):329–336, 1986.

[DB211] IBM DB2. http://www-01.ibm.com/software/data/db2/, 2011.

[GHYZ04] Hong Guo, Wen-Chi Hou, Feng Yan, and Qiang Zhu. A monte carlo sampling method for drawing representative samples from large databases. In *SSDBM*, pages 419–420, 2004.

[gR11] Oracle Database Express Edition 11g Release 2. http://www.oracle.com/technetwork/database/express-edition/downloads/index.html, 2011.

[HK04] Peter J. Haas and Christian Koenig. A bi-level bernoulli scheme for database sampling. In *SIGMOD Conference*, pages 275–286, 2004.

[JL96] George H. John and Pat Langley. Static versus dynamic sampling for data mining. In *KDD*, pages 367–370, 1996.

[KKMS09] Vidyadhar G. Kulkarni, Subodha Kumar, Vijay S. Mookerjee, and Suresh P. Sethi. Optimal allocation of effort to software maintenance: A queuing theory approach. *Production and Operations Management*, 18(5):506–515, 2009.

[LNS90] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical selectivity estimation through adaptive sampling. In *SIGMOD Conference*, pages 1–11, 1990.

[Man11] IBM Optim Integrated Data Management. http://www-01.ibm.com/software/data/data-management/optim-solutions/, 2011.

[May99] Wolfgang May. Information extraction and integration with FLORID: The MONDIAL case study. Technical Report 131, Universität Freiburg, Institut für Informatik, 1999. Available from http://dbis.informatik.uni-goettingen.de/Mondial.

[MM44] William G. Madow and Lillian H. Madow. On the theory of systematic sampling. *Annals of Mathematical Statistics*, 15(1):1–24, 1944.

[Olk93] Frank Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[Ora11a] Oracle. http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions001.htm, 2011.

[Ora11b] Oracle. http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_stats.htm, 2011.

[Ora11c] Oracle. http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_10002.htm#i2065953, 2011.

[PF00] Christopher R. Palmer and Christos Faloutsos. Density biased sampling: An improved method for data mining and clustering. In *SIGMOD Conference*, pages 82–92, 2000.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[Swe00] Latanya Sweeney. Uniqueness of simple demographics in the u.s. population, 2000.

[Swe02] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[Toi96] Hannu Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.

[ZPLO97] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li, and Mitsunori Ogihara. Evaluation of sampling for data mining of association rules. In *RIDE*, 1997.