8-2011

Test Data Extraction and Comparison with Test Data Generation

Ali Raza
Utah State University

TEST DATA EXTRACTION AND COMPARISON

WITH TEST DATA GENERATION

by

Ali Raza

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

_____	_____
Stephen W. Clyde Major Professor	Vicki Allan Committee Member
_____	_____
Renée Bryce Committee Member	Byron R. Burnham Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2011

ABSTRACT

Test Data Extraction and Comparison

with Test Data Generation

by

Ali Raza, Master of Science

Utah State University, 2011

Major Professor: Dr. Stephen W. Clyde
Department: Computer Science

Testing an integrated information system that relies on data from multiple sources can be a challenge, particularly when the data is confidential. This thesis describes a novel *test data extraction* approach, called *semantic-based test data extraction for integrated systems* (*iSTDE*) that solves many of the problems associated with creating realistic test data for integrated information systems containing confidential data. iSTDE reads a consistent cross-section of data from the production databases, manipulates that data to obscure individual identities while still preserving overall semantic data characteristics that are critical to thorough system testing, and then moves that test data to an external test environment.

This thesis also presents a theoretical study that compares *test-data extraction* with a competing technique, named *test-data generation*. Specifically, this thesis a) describes a comparison method that includes a comprehensive list of characteristics essential for testing the database applications organized into seven different areas, b)

presents an analysis of the relative strengths and weaknesses of the different test-data creation techniques, and c) reports a number of specific conclusions that will help testers make appropriate choices.

(122 pages)

ACKNOWLEDGMENTS

To my father, who filled my life with light, love, and hope. May Allah rest his soul in eternal peace.

To Dr. Clyde who guided my steps in this cumulative process. I would like to thank him for his invaluable assistance and support throughout my graduate career.

To Dr. Vicki Allan and Dr. Renée Bryce for all their help and suggestions on this thesis.

Ali Raza

CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGMENTS v

LIST OF TABLES ix

LIST OF FIGURES x

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND 4

2.1 Subject Application: CHARM 4
2.2 Problems Found in Testing of Integrated Systems 6
2.3 Personal Identifying Information and Anonymization 7

2.3.1 Personally Identifiable Information (PII) 7
2.3.2 Identification of PII throughTheir Impact Levels 8
2.3.3 Ways of Hiding PII 9

2.4 Related Work 11

3 SEMANTIC-BASED TEST DATA EXTRACTION \
FOR INTEGRATED SYSTEMS (ISTDE) 16

3.1 Overview 16
3.2 Description of iSTDE Environment 17
3.3 Approach 18

3.3.1 Specifying Extraction Parameters 20
3.3.2 Creation of Temporary Databases 20
3.3.3 Extraction and Loading of Real Data to Temporary Databases 24
3.3.4 PII Identification and Population 26
3.3.5 Data Mangling 27
3.3.6 Transferring Mangled Data 31
3.3.7 Destroying Mappings 32

3.4 Discussion of PII and Preserved Semantics in CHARM 32

3.4.1 Identification of PII in CHARM 32

3.4.2 Anonymizing PII in CHARM 33
3.4.3 List of Preserved Semantics in CHARM 33

3.5 Discussion of Extraction Set Size, Semantics Constraints and Mangling 34

3.5.1 Size Constraints 34
3.5.2 Exceptions 34

3.6 Experience with iSTDE and Discussion 35

4 COMPARISON OF TEST DATA EXTRACTORSWITH TEST DATA GENERATORS 37

4.1 Overview 37
4.2 Two General Approaches to Test Data Creation 39

4.2.1 Data Generation 39
4.2.2 Data Extraction 44

4.3 Comparison Method 48

4.3.1 Overview 48
4.3.2 Comparison Context 52

5 COMPARISON RESULTS 54

5.1 Comparisons in the Context of Standalone Databases 54
5.2 Comparisons in the Context of Federated Databases 69
5.3 Comparisons Related to Target Schemes 71
5.4 Comparisons Related to Set-up, Use, and Performance 72
5.5 Comparisons in the Context of Database Refactoring 74
5.6 Comparisons in the Context of Testing Techniques 78
5.7 Comparisons Related to Social and Time Factors 81

6 SUMMARY, FUTURE WORK, AND CONTRIBUTIONS 84

6.1 Summary 84
6.2 Contributions 85
6.3 Future Extensions of iSTDE as a Tool 87
6.4 Research Directions for Comparison ofTDGs and TDEs Approaches 88

REFERENCES 90

APPENDICES .....

Appendix A 94

Appendix B 99
Appendix C 103
Appendix D 106
Appendix E 107
Appendix F 109

LIST OF TABLES

Table Page

Table 2-1. Eight Software Packages Reviewed. 12

Table 2-2. Six Common Test-Data Generation Features. 12

Table 2-3. Test-Data Extraction Techniques. 14

Table 3-1. Noise Production Example in 1-1 Logical Dependent Domains. 29

Table 3-2. Example of Field-based Mangling. 30

Table 4-1. Seven Common Test-Data Generation Methods and Tools That Support Them. 43

Table 4-2. Five Test-Data Extraction Techniques and Tools That Support Them. 46

Table 4-3. Descriptions of Comparisons Criteria. 50

Table 5-1. Comparisons in Context of Standalone Database. 55

Table 5-2. Comparisons in Context of Federated Databases. 56

Table 5-3. Comparisons Related to Target Schemes. 56

Table 5-4. Comparisons Related to Set-up, Use, and Performance. 56

Table 5-5. Comparisons in Context of Database Refactoring. 56

Table 5-6. Comparisons in Context of Testing Techniques. 57

Table 5-7. Comparisons Related to Social and Time Factors. 57

Table 5-8. Matcher Results with Spelling Variations. 61

Table 5-9. Matcher Results with Incomplete Names. 61

Table 5-10. Mult-column Data Value Frequency Distribution: patient_id and usiis_patid Columns in Forecast_vw Table. 66

LIST OF FIGURES

Figure Page

Figure 2-1.CHARM architecture. 5

Figure 3-1. iSTDEoverview. 19

Figure 3-2. Domain creation. 21

Figure 3-3. Data extraction. 25

Figure 3-3. Data mangling. 31

Figure A-1. Old CHARM Test Data Creation Process 97

Figure A-2. New CHARM Test Data Creation Process 98

Figure A-3. iSTDE Process Centric Framework 98

Figure C-1. Actor's hierarchy for iSTDE. 103

Figure C-2. Initial use case modeling for iSTDE. 104

Figure C-3. Detailed use case modeling for iSTDE. 105

Figure D-1. Architecture diagram for iSTDE. 106

Figure F-1. Interface and communication protocols diagram for iSTDE. 109

CHAPTER 1

INTRODUCTION

Testing of database-intensive applications has unique challenges that stem from hidden dependencies, subtle differences in data semantics, target database schemes, and implicit business rules. These challenges become even more difficult when the application involves integrated and heterogeneous databases or confidential data. Proper test data that simulate real-world data problems are critical to achieving reasonable quality benchmarks for functional input-validation, load, performance, and stress testing.

In general, techniques for creating test data fall in two broad areas, namely, test-data generation and test-data extraction, that differ significantly in their basic approach, runtime performance, and the types of data they create. *Test-data generation* relies on generation rules, grammars, and pre-defined domains to create data from scratch. *Test data extraction* takes sample data from existing production databases and manipulates that data for testing purposes, while trying to maintain the natural characteristics of the data. This thesis describes a novel *test data extraction* approach, called *semantic-based test data extraction for integrated systems* (*iSTDE*) that solves many of the problems associated with creating realistic test data for integrated information systems containing confidential data. This thesis also presents a theoretical study that compares *test-data extraction* with a competing technique, named *test-data generation*.

The rest of this thesis is organized as follows. Chapter 2 discusses background work on test data generators and test data extractors, essential concepts regarding personal identifying information (PII's) and the CHARM (Child Health Advanced Record Management) environment in general. Chapter 3 describes the proposed test-data

extraction tool *(full name already given)* (*iSTDE*), which extracts test data for confidential integrated systems and effectively removes or hides all personal identifying information without altering important data characteristics. The chapter also discusses the use of iSTDE to test CHARM – a statewide integrated system for children's public health information [14].

Chapter 4 describes a theoretical study to help establish guidelines for software testers in making informed choices about various test-data creation techniques. The study starts with a detailed classification of different kinds of test-data creation techniques. It then illustrates the use of these techniques in existing test-data creation tools and discusses their usefulness in the context of standalone, integrated database systems, and confidential data. Next, it presents a method for comparing the relative strengths and weaknesses of the different test-data creation techniques. Finally, Chapter 5 presents the results of a comparison based on a study method and analyzes those results. At the most general level, we found that test-data extraction can produce more realistic test-data, whereas, test-data generators can be more efficient. However, we present a number of more specific conclusions that can help testers make appropriate choices.

In Chapter 6, we summarize our research work and contributions and discuss some exciting opportunities for future research.

Additional details about the iSTDE software and other work-products of this project are included in the appendices. Specifically, Appendix A describes an evolving iSTDE-centric software testing process in the CHARM environment. Appendix B lists the user-level goals and functional requirements for iSTDE. Appendix C presents these

requirements as a collection of use case diagrams. Appendix D contains the complete collection of architectural diagrams of iSTDE. Appendix E describes some important methods of major classes in iSTDE as way of quickly understanding the code. Lastly, in Appendix F, we provide some instructions for the deployment of iSTDE.

CHAPTER 2

BACKGROUND

2.1Subject Application: CHARM

Child Health Advanced Record Management (CHARM) is an integrated system that provides health care professionals with accurate and timely information about children in Utah, whose medical records are housed in various public healthcare databases, including Vital Records (VR), the Utah Statewide Immunizations Information System (USIIS), and Early Hearing Detection and Invention (HiTrack). These databases all reside on different host machines and use different database managers.

A collaborative team of software engineers from Utah State University (USU), Utah Department of Health (UDOH), and Multimedia Data Services Corporation (MDSC) started developing CHARM in November 2000. Its architecture, illustrated in Figure 2-1, is that of an arms-length information broker [29] with:

- A CHARM server, which is the information broker, and
- A CHARM agent for each connected database, also called a participating program or PP.

When a user of a PP requires CHARM-accessible data, the PP submits a request for that data to CHARM via its own agent. That agent is responsible for mapping PP-specific data types and identifiers to CHARM-specific data types and identifiers. It next passes the modified query onto the CHARM server. The CHARM server either looks up or computes an appropriate strategy for processing the query and then executes that strategy. This process may involve retrieving information from several other PPs via their CHARM agents and merging the results of those individual data retrievals into a final

query result. The CHARM server returns the final query result to the initiating agent, which then converts CHARM-specific data types and identifiers back to program-specific types and identifiers.

The first functional prototype was successfully demonstrated in March 2002. It was at this point that the developers began to see the real challenges of testing an integrated system that involves confidential data.

With the three original participating programs, the system made use of seven different databases: three from the participating programs, three used by the agents to

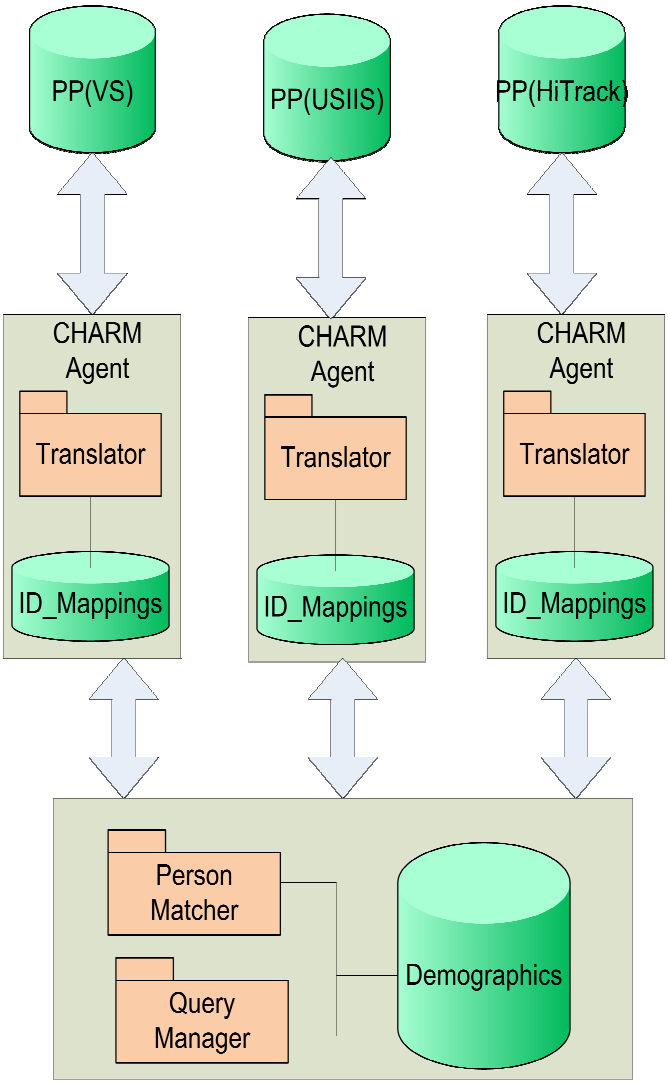

Figure 2-1.CHARM architecture.

map PP-specific IDs to internal CHARM IDs, and one used by the CHARM server to match and link persons based on their demographic information. (Today, there are six participating programs and 13 databases.) From an architectural design stand-point, separating ID-mapping data from the PP databases and from the demographic database used for matches allowed CHARM to preserve the data stewardship boundaries of the participating organizations, minimize its impact on legacy software, and avoid putting sensitive program-specific IDs into the central database [8].

2.2Problems Found in Testing of Integrated Systems

Testing an integrated, person-centric database, like CHARM, is challenging for several reasons. First, the real data is confidential since it contains sensitive personal-identifying information (PII) about the patients and their health care. Second, databases in an integrated system have syntactic, schematic, and semantic heterogeneities [35]. The syntactic heterogeneities are concerned with the differences in underlying data models of the DBMS, i.e., differences in data structure primitives, query language differences. They stem from the differences in the database managers used by the participating data sources. The schematic heterogeneities explain the differences related to presenting the equivalent or related data concepts in different database systems that have conflicting structural representations. This kind of heterogeneity is hard to test because different structural representations of identical information are hard to identify, and similar test cases cannot be used for other representations. The semantic heterogeneity arises when mismatches and conflicts arise because of differences in meaning, interpretation, and usage of data. Due to the subtle differences and difficulties in interpreting the identical or different instances of data values, testing it is hard. Third, the individual data sources in

an integrated information system were probably not initially designed for integrated testing, but the very existence of the integrated system implies that there are important dependencies between the databases. In this instance, we use a set of mapper databases that map these sets of databases. Fourth, the set of databases being used in CHARM contains sensitive demographic information about patients that is at great risk of being misused if confidentiality is compromised.

2.3Personal Identifying Information and Anonymization

2.3.1Personally Identifiable Information (PII)

Personally identifiable information (PII) as used in information security refers to information that can be used to uniquely identify, contact, or locate a single person or can be used with other sources to uniquely identify a single individual [24]. The abbreviation PII is widely accepted, but the phrase it abbreviates has four common variants, namely, personal, personally, identifiable, and identifying. Not all are equivalent, and for legal purposes, the effective definitions vary depending on the jurisdiction and the purposes for which the term is being used. Recently lawmakers have paid a great deal of attention to protecting a person's PII. One of the primary focuses of the Health Insurance Portability and Accountability Act (HIPAA) [27] is to protect a patient's PII.

The National Institute of Standards and Technology (NIST) provides a list of PII data [23]. The list includes the following information:

1. Name
2. Personal identification numbers such as SSN, driver license, passport number
3. Address information, including street address and email address

4. Asset information, which can include a person's computers and their corresponding addresses (e.g., IP addresses, MAC addresses)
5. Telephone numbers: cell, office, and home phone numbers
6. Personal characteristics, e.g., weight, height, eye color, X-rays, finger-prints, and other bio-metrics
7. Information identifying personally owned property, e.g., vehicle registration number
8. Information about an individual that is linked or linkable to one of the above (e.g., parents or child information, birth-date, race, religion, activities, employment information, financial information)

2.3.2 Identification of PII through Their Impact Levels

A systematic way to identify the PII would to evaluate the fields against their impact levels. Impact level is defined as "The potential level of harm caused from a breach of confidentiality when attempting to determine whose PII was the subject of a loss of confidentiality, as well as any adverse effects experienced by the [23]. There are three defined impact levels, low, moderate, and high [23].

1. Low: Loss of confidentiality, integrity or availability have limited adverse effect that can result in degradation of organization function, minor damage to organization assets, financial loss or minor harm to individual.
2. Moderate: Loss of confidentiality, integrity, or availability that has serious adverse effect and can result in significant degradation of organization function, significant damage to organization assets, financial loss or can involve loss of life or life threatening injuries.

3. High: Loss of confidentiality, integrity, or availability that can have severe or catastrophic adverse effect on individual or organization operations and can result in inability for an organization to perform the primary operations, major damage to organization assets, major financial loss, severe or catastrophic harm to individuals involving loss of life or serious life threatening injuries.

2.3.3Ways of Hiding PII

In general, there are two ways of hiding the *PII:* de-identification and anonymization. The following sections summarize these approaches.

2.3.3.1. De-identifying Information: The term de-identified information is used to describe records that have had enough PII removed or obscured, also referred to as masked or obfuscated, such that the remaining information does not identify an individual, and there is no reasonable basis to believe that the information can be used to identify an individual [23].

De-identified information can be re-identified (rendered distinguishable) by using a code, algorithm, or pseudonym that is assigned to individual records. The code, algorithm, or pseudonym should not be derived from other related information about the individual, and the means of re-identification should only be known by authorized parties and not disclosed to anyone without the authority to re-identify records.

De-identification could be accomplished by removing account numbers, names, SSNs, and any other identifiable information from a set of financial records. By de-identifying the information, a trend analysis team could perform an unbiased review on those records in the system without compromising the PII or providing the team with the ability to identify any individual.

Additionally, de-identified information can be aggregated for the purposes of statistical analysis, such as making comparisons, analyzing trends, or identifying patterns. An example is the aggregation and use of multiple sets of de-identified data for evaluating several types of education loan programs.

2.3.3.2. Anonymizing Information: Anonymized information is defined as previously identifiable information that has been de-identified and for which a code or other association for re-identification no longer exists [23]. Anonymizing information usually involves the application of statistical disclosure limitation techniques to ensure the data cannot be re-identified, such as:

1. Generalizing the Data—Making information less precise, such as grouping continuous values;
2. Suppressing the Data—Deleting an entire record or certain parts of records
3. Introducing Noise into the Data—Adding small amounts of variation into selected data;
4. Swapping the Data—Exchanging certain data fields of one record with the same data fields of another similar record (e.g., swapping the ZIP codes of two records);
5. Replacing Data with the Average Value— Replacing a selected value of data with the average value for the entire group of data.

Anonymized information is useful for system testing. Systems that are newly developed, newly purchased, or upgraded require testing before being introduced to their intended production (or live) environment. Testing generally should simulate real conditions as closely as possible to ensure the new or upgraded system runs correctly and

handles the projected system capacity effectively. If PII is used in the test environment, it is required to be protected at the same level that it is protected in the production environment, which can add significantly to the time and expense of testing the system.

Randomly generating fake data in place of PII to test systems is often ineffective because certain properties and statistical distributions of PII may need to be retained to effectively test a system.

2.4 Related Work

In general as discussed above, approaches for test-data creation fall into one of two general categories: one based on automatic generation of test-data and the other based on real data extraction. A review of eight automated test-data generation tools revealed six different common techniques for generating data at a field level, i.e., for a domain. See Table 2-1 for list of the tools review and Table 2-2 for the techniques each supports.

The first two techniques create random data, based on a field's data type along with some simple constraints. For example, an algorithm based on random generation could populate a salary field in a payroll table with values between $20,000 and $65,000. Similarly, a random-generation algorithm could populate a first-name field in a person table with a string between 1-10 characters long, containing characters A-Z. In general, random generation is more applicable to numeric fields than other types of domains. Six of the eight tools support random generation for numeric data, while only three support string generation.

The third technique constrains the random generation of data by percentages that represent value distributions in real data. For example, imagine a person table with 20%

Table 2-1. Eight Software Packages Reviewed.

Abr.	**Software Package**	**Author / Vendor**
DG	GenerateData.com [3]	GenerateData.com
SE	DTM Data Generator [4]	DTM Soft
FS	ForSQL Data Generator[5]	ForSQL
TS	Automated Test Data Generator[6]	Tethys Solutions
DN	DB Data Generator V2[9]	Datanamic
TB	TurboData[10]	Turbo Computer Systems, Inc.
TN	Tnsgen – Test Data Generator[7]	TNS Software Inc.
EM	EMS Data Generator for MySQL[11]	EMS Inc.

Table 2-2.Six Common Test-Data Generation Features.

	Test Data Generation Features	**DG**	**SE**	**FS**	**TS**	**DN**	**TB**	**TN**	**EM**
1	Random numeric data generation		●	●	●	●	●	●	
2	Random string data generation				●		●	●	
3	Percentage-based data generation		●						
4	Generate data from user-defined grammars		●			●			
5	Generate data from predefined domains	●		●		●	●		
6	Generate data for database, with master child relations						●		

of the records having birth dates in 2008 and the remaining 80% in 2007. A tool that supports this type of data creation could preserve such distributions. Only one of the tools supports this type of random data generation.

The fourth technique generates data according to user-defined grammars. For example, the grammar Aa-9999 could generate data that has one capital letter, followed by one small letter, a dash and four numeric digits. This technique is most applicable for string domains with an implicit language that can be easily defined with a pattern or simple grammar. Interestingly, it is common for database schemes to have fields with simple hidden languages, but only two of the eight tools support this technique.

The fifth technique pulls randomly selected data from a pre-defined domain. For example, this technique could be used to populate a last-name field from a pre-defined domain of common Spanish names. Four of the eight tools support this technique, and several of them even had some built-in domains for female names, male names, countries, etc.

The sixth technique identifies an algorithm that links child records to parent records in hierarchical structures. For example, an algorithm that uses this technique could be used to generate data for a purchasing system consisting of customer, order, line item tables that relate to each other via referential integrity constraints.

Although automated test-data generation techniques can save time compared to collecting and loading meaningful test-data by hand, they fall short of producing test-data that possess many of characteristics found in real data, such as

- The presence or frequency of missing values;
- The presence or frequency of incomplete information;

- The presence of garbage data;
- Duplicates, wherein the duplicates were caused by or allowed to exist because of other field values; and
- Other characteristics caused by inter-field dependencies.

The second approach, test-data extraction, attempts to create test beds from real data sources. A review of the eight tools revealed extraction techniques at three different levels, namely, extracting data from a single file, extracting data from multiple tables in a single database, and extracting data from multiple unrelated databases. See the first three rows in Table 2-3.

Table 2-3. Test-Data Extraction Techniques.

	Test Data Extraction	**DG**	**SE**	**FS**	**TS**	**DN**	**TB**	**TN**	**EM**
1	From real data from files		●	●				●	
2	From real data from one database		●	●					●
3	From uncorrelated real data from multiple databases								●
4	Correlated real data from multiple related databases								
5	De-identified data from confidential databases								

Three of the tools support the first technique, which has some similarities to test-data generation from predefined domains. However, a key difference is that test-data extraction can produce test-data with realistic characteristics without explicitly having to state those characteristics.

The second technique deals with extracting test-data from multiple tables in a real database. This type of test-data extraction does everything supported by the first technique, but it also maintains inter-record dependencies across tables. However, these

dependencies can go beyond the referential integrity constraints mentioned above. Specifically, they can include frequency constraints involving fields from multiple tables. Three tools support this type of data extraction, at least to some degree.

The third technique, which only one of the eight reviewed tools supports, goes a step further by allowing users to extract data from multiple databases. However, without any cross-correlation of data between the databases, this technique can be viewed as simply a convenience for performing multiple, separate extractions.

Clearly, being able to create test-data for multiple databases is necessary for testing integrated systems, but it is not enough. To test an integrated system, its constituent components (i.e., participating information systems) need realistic and correlated slices of data that contain the same inter-relationships and hidden dependencies from the production databases. For example, it would be meaningless to extract one set of person records from one database and a non-overlapping set of records from another database. Testing would not be able to verify the results of any actual data integration.

Also, to test integrated systems that contain confidential data, it is important to remove or hide all identified personal information so that testing can be conduct in unsecured environments.

To address the need for correlating data across databases and for anonymized test-data, we added two additional test-data extraction techniques to Table 2-3, namely, correlated real data from multiple related databases and de-identified data from confidential databases. The iSTDE tools presented in the next section support these additional techniques.

CHAPTER 3

SEMANTIC-BASED TEST DATA EXTRACTION FOR INTEGRATED SYSTEMS (iSTDE)

3.1 Overview

In CHARM, developers attempted various techniques to generate realistic test-data. At first, the developers tried to create test-data by hand. This quickly proved to be time consuming and error prone. Next, the developers built an automated test-data generator that created test-data for each database using that database's scheme and codified knowledge about field domains, constraints, and overall data characteristics [13]. Such an approach allowed the developers to create large amounts of test-data, but correlating the information between different databases and creating patterns similar to those in the real data proved difficult.

In the latest version of CHARM, the developers have taken a new approach for creating test-data. Specifically, they created a distributed tool, called Semantic-based Test Data Extraction for Integrated Systems (iSTDE), which first extracts a consistent cross-section of data from the production databases. It next manipulates that data in a way that obscures individual identities, while preserving other important aggregate data characteristics, such as the frequency of name occurrences, the percentage of multiple births (i.e., twins), and the presence of bad data. Preserving these characteristics is critical to effective system testing of components like a person matcher. After de-identifying the test-data, iSTDE moves that test-data from the production environment to a test environment.

The rest of this chapter is organized as follows. Section 3.2 provides some additional background on the production and test environment of testing CHARM. Section 3.3 describes the process iSTDE uses to extract data, de-identify that data, and move it to a test environment. Experience and observations in using iSTDE are presented in Section 3.4.

3.2Description of iSTDE Environment

In general, multiple CHARM execution environments exist, including one for production, one for staging and user-acceptance testing, several for system testing, and at least six for development. From a data-security perspective, these environments can be grouped into two categories: confidential and unprotected. The confidential environments, which include the production environment and staging environment, are protected by firewalls in UDOH. Only authorized users can access these environments containing sensitive demographic and health care data. The unprotected environments, which include all the system testing and development environments, run on machines and networks outside of UDOH firewalls, and may be used by individuals not authorized to see real data.

Besides the access restrictions, the confidential and unprotected environments differ in terms of the database managers they use for the various data sources. The data sources in the confidential environments are either the actual production databases or staging databases for the production system. In either case, these databases are tied to legacy software and, therefore, rely on a number of different database managers, including Oracle, Microsoft SQLServer, PostgreSQL, and Pervasive.

All the unprotected environments, on the other hand, use PostgreSQL as the database manager to eliminate extra licensing fees that might otherwise be necessary. However, using a different database manager for testing introduces two new challenges.

First, the types of database that the integration system accesses will depend on the environment it is running in. So, testing iSTDE in one of the unprotected environments may not verify the correctness of the database drivers, connection strings, or SQL-statement syntax. For CHARM, we solved this problem by doing a final system test in the staging environment, which does use all of the same types of databases as the production environment.

Second, converting all the data to PostgreSQL for the unprotected environment introduces certain data-type mapping problems. Some data types in the original database do not have compatible data types in PostgreSQL. For example, SQLServer supports a global unique identifier (GUID) data type that PostgreSQL does not support. So, iSTDE has to map SQLServer GUIDs to an alternative data type, like VARCHAR. Section 3.3 describes iSTDE's solution to this problem in more detail.

3.3Approach

The iSTDE software itself is installed in a confidential environment, thereby ensuring that no unauthorized person can execute it. When iSTDE executes, it goes through seven steps to create a consistent set of de-identified test-data from the confidential data and then moves it to an unprotected environment. See Figure 3-1. Each of these steps is described in more detail below.

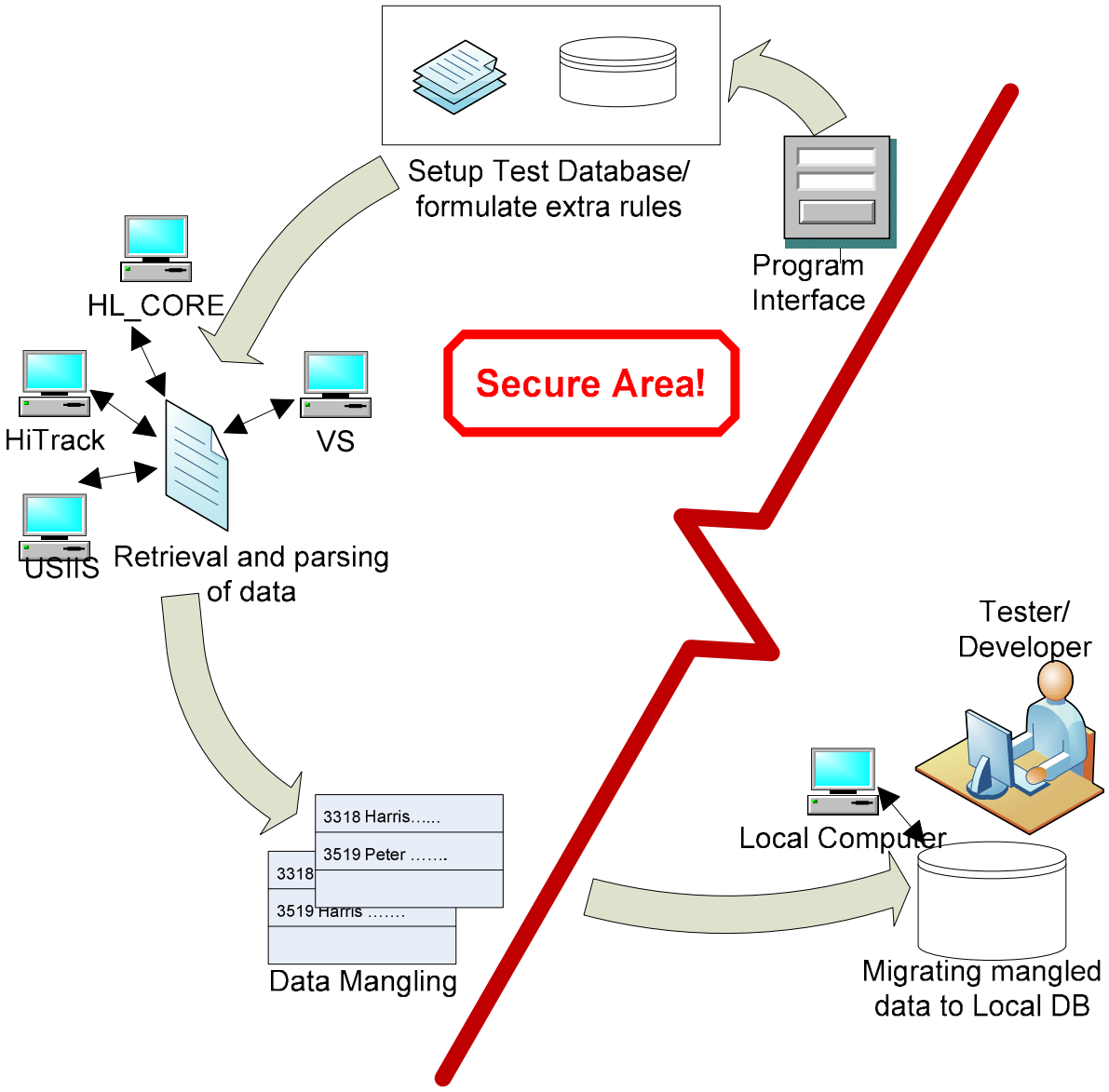

Figure 3-1. iSTDEoverview.

3.3.1 Specifying Extraction Parameters

In the first step, a user specifies what data to extract (e.g., all children born from 7/1/2008 to 9/30/2008) and the target environment wherein test-data should ultimately be sent, along with a username and password for accessing that environment. In addition, the user can specify the location of a temporary database within the confidential environment that iSTDE will use to collect and manipulate the test-data before sending it over to the target environment.

iSTDE also supports a number of other configuration parameters that the user typically does not change, such as connection strings for the various data sources in the confidential environment. These parameters are kept in a properties file and only need to be changed if the data sources in the confidential environment change.

3.3.2 Creation of Temporary Databases

The second step in the iSTDE execution involves creating the temporary database in a confidential environment to hold the extracted data from multiple source databases, while they are being collected and manipulated. See Figure 3-2. In this step, iSTDE first makes sure that there are no existing temporary databases in the confidential environment. It then retrieves schema metadata for all the source databases and transforms them into PostgreSQL creation scripts. Next, it executes those scripts to create the temporary database, with all of the necessary tables, indices, views, stored procedures, and triggers. Further into the process in Step 7, iSTDE destroys the temporary databases, so unnecessary copies of the extracted data are not left lying around.

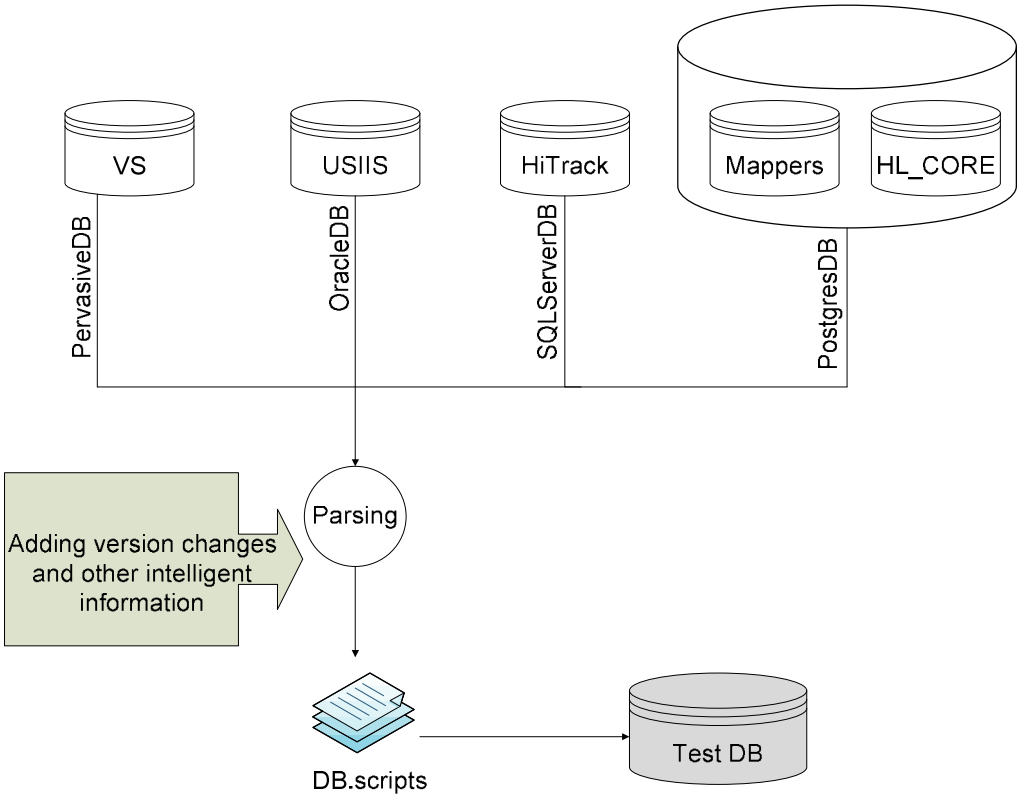

Figure 3-2. Domain creation.

iSTDE uses PostgreSQL for the temporary database because it is open source and it supports a broad range of features and data types. Nevertheless, it does not support everything such as complex data types etc.; nor did we find an open source database manager that did.

One challenge for Step 2 was accessing metadata. Some source databases do not allow external processes to read the database's metadata, neither do they support reflection. So, iSTDE could not automatically retrieve and analyze their structures directly. For such databases, iSTDE reads the metadata from an externally managed meta-data repository. This repository has to be updated manually when the real database's structure changes.

Another challenge was unsupported or incompatible data types, as mentioned in Section 3.3. For each unsupported data type, iSTDE designers selected an alternative PostgreSQL data type and wrote a mapping function that converts data from the original type to the alternate type. So, when iSTDE comes upon a field with an unsupported data-type, it simply looks up its alternative data type and uses that type in the table creation scripts for the temporary database. Then, later in Step 3, iSTDE uses the corresponding mapping function to convert the values from that field before placing them into the temporary database.

The third challenge was handling views in iSTDE. The test databases need to support or simulate any views in the real database such that the legacy programs and integrated system will function correctly. There are two approaches for supporting a view. The first is to include all the tables and their data that makes up the view in the test set. However, this can be problematic because some views are very complex and may end up requiring far more data to be extracted into the test database than necessary. A second approach is to implement the views as tables populated with a snapshot (or a portion of a snapshot) of the view. This approach can reduce both the amount of space required for the test-data and the extraction time. However, this approach is only appropriate if the integrated system does not need to update any of the data involved in the view.

Like views, database procedures also need to be either implemented directly in the test database or simulated through tables, since different database managers use different procedural languages and it is very difficult to automate their extraction and direct implementation. However, as with views, when the integrated system does need to

modify the underlying data, simulating a store procedure using a table of stored results is relatively straightforward. Historically, when this has not been possible for CHARM, a programmer manually creates versions of the store procedure in PostgreSQL's procedural language (Pl/SQL). Manual conversions of stored procedures need only be done once. After that, iSTDE can re-use them whenever needed.

A final challenge stems from version conflicts. The systems that comprise an integrated system, as well as the integration framework itself, evolve independent of each other. Changes do not occur in a lock-step chronology. One system will upgrade its database, while other systems are still using older structures. For example, over the past few years, there have been two major versions of the CHARM integration framework, and at least one significant database change to each of the participating programs. The database schemas for these versions have slightly different meta-data. Such was the case when the CHARM developers were testing Version 2, yet the production environment was still using Version 1 data structures. Mapping data across versions of integrated systems is a significant problem. iSTDE handles this by adding some additional metadata to the external metadata repository so that it can track the version and then re-map data if necessary.

Some challenges are still unresolved. For example, in cases wherein iSTDE has to store the meta-data for a system in an external repository, changes to the original database's structure can create an inconsistency. Organization procedures have to be put in place and followed to ensure that changes to a participating information system are reflected in iSTDE meta-data for that system. It would be better if more of this process could be automated or at least monitored by iSTDE.

3.3.3 Extraction and Loading of Real Data to Temporary Databases

The previous steps create empty temporary databases for holding the extracted data, with all the necessary tables, constraints, views (or their simulations), and stored procedures (or their simulations). Now in this the third step, iSTDE extracts a consistent slice of real data from the participating data sources in the confidential environment and loads that data into these temporary databases.

The process of extracting a data slice starts when iSTDE generates SQL queries through parsing and analyzing user-specified test-data selection criteria, e.g., child birth date range. One SQL query is generated for each of the relevant production databases. A challenge in data extraction was to ensure that the slice contains records for the same sample population across all of the participating programs. To ensure the slice's internal consistency, iSTDE uses cross-database links created and maintained by the integrated system. In CHARM, each agent maintains a mapping of its participating program's IDs to a common, internal CHARM ID. Together, these maps link the records for a person across all of the participating information systems. iSTDE uses and preserves these inter-database links to guarantee that the overall test-data are internally consistent.

When these SQL queries return result sets, iSTDE uses the data in these result sets to construct SQL insert statements. While constructing these SQL insert statements, individual data fields in a result set are parsed according to the temporary databases' metadata. Later these insert statements are written to data files located in a confidential environment. The purpose of generating data files is to effectively utilize the connection time on production databases. Figure 3-3 shows the data extraction process.

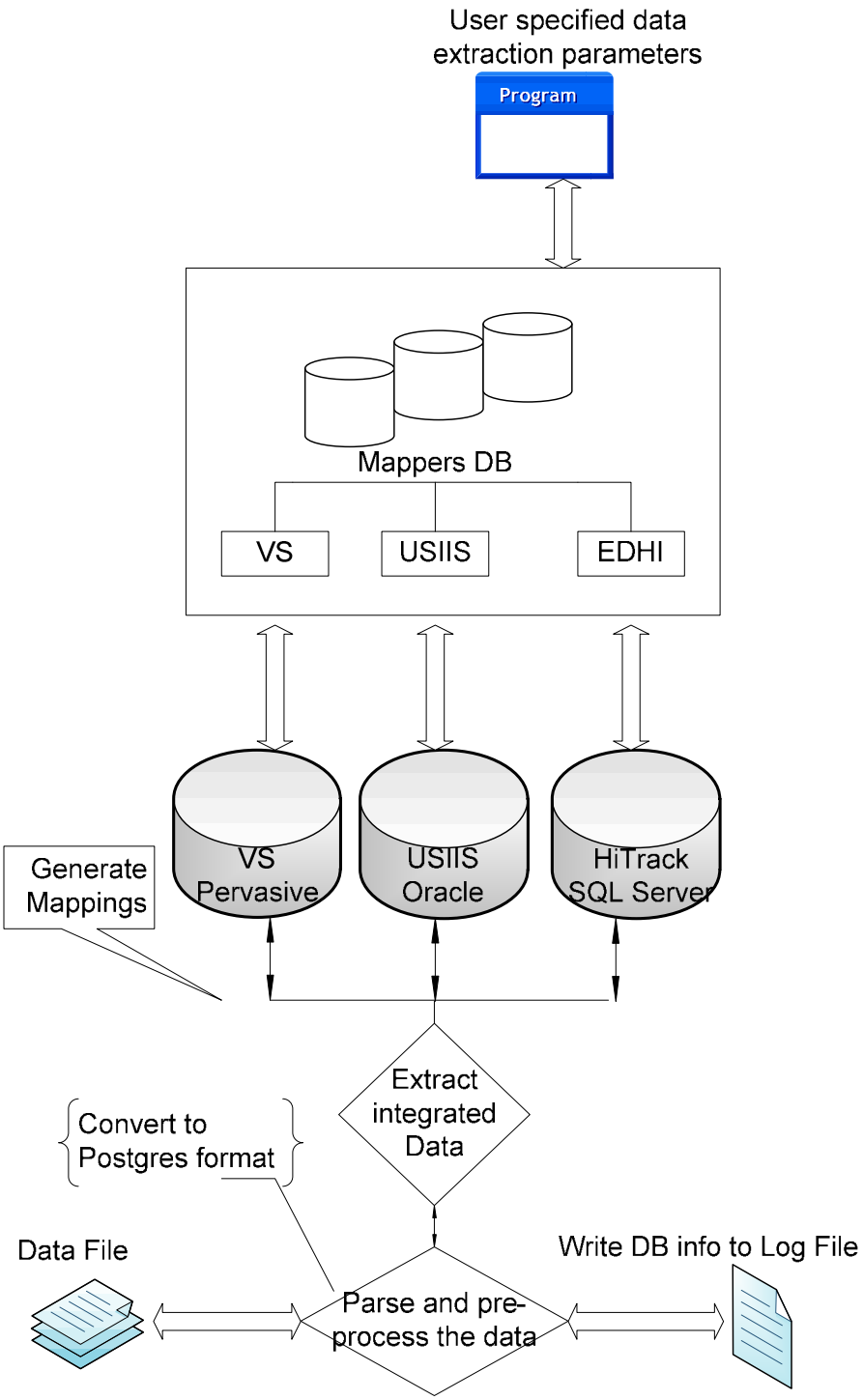

Figure 3-3. Data extraction.

Once the process of extracting data into data files is complete, iSTDE loads these files to the temporary databases. A challenge was to load data in such a way as to not violate referential integrity constraints. iSTDE deals with this challenge by loading data files for parent tables before child tables. However, the cyclic nature of relational interdependencies among tables makes this solution unfeasible in the long term. A better approach would be to first load the data into tables and later implement referential integrity constraints [34].

3.3.4PII identification and Population

In *iSTDE,* we selected the PII domains in accordance with guidelines provided by NIST [23]. The PII domains are populated from all the databases included in CHARM , and its size is not limited to the extracted test bed size. In addition, building of PII domains is a continuous incremental process that involves fresh re-extraction of identity domains once a month and is totally independent of iSTDE execution for creating a test bed. After PII domains selection, we build dictionaries for these domains. These domain dictionaries are data structures that consist of real domains and test domains. Real domains are data slices that are built from similar PII domains from across all databases included in CHARM, not just one database. For example, in the case of first male names, the dictionary real domain contains all first male name entries that exist in all tables of temporary databases. Test domains are populated by random shuffling of real domain entries. Even though it is random, still are there chances of an entry in a real domain mapping to the same entry in a test domain. Entries in test domains would be the newly assigned values for the real data. Populating the domain dictionaries is a separate,

ongoing process; thus, these domains are getting richer with the passage of time. Hence, it is less likely that a real domain value gets assigned to the same test domain value.

PII domains can be simple or composite. A simple domain consists of only one identifying element, and composite domains may involve multiple elements. For example, male first name and phone number are simple domains, whereas a full address domain (street, city, state, zip code) may be a composite domain. To preserve consistency, composite domains have to be mangled as a whole unit. For example, one instance of an address may be swapped with another random but complete address. iSTDE also sometimes subdivides a domain wherein swapping needs to be constrained by the value of some other element. For example, it partitions gender-dependent domains into female and male subsets, i.e., the first name domain is partitioned into male first names and female first names.

3.3.5Data Mangling

Once real data has been extracted and loaded into the temporary databases, iSTDE obfuscates that data by applying data mangling to each domain that contains personal indentifying information (PII). In this the fourth step, data mangling randomly swaps data values in the domain so the PII's of any given record are unrecognizable and untraceable, but without changing the overall characteristics of the data set (Figure 3.3). As mentioned early, preserving the overall characteristics of the data set is critical for thorough testing in integrated systems.

In the next step of the mangling process, *iSTDE* swaps all the values of PII domains in real data with the newly assigned test values, using domain dictionaries that provide mapping from real values to test values. Once it has mangled all the data, it then

deletes these dictionaries so that no one can perform reverse mapping to real data. Essentially, iSTDE deals with four different types of data mangling.

The first type is 1-1 logical domain dependency. Two domains are said to be logically dependent when they are semantically related to each other, a change in one domain requires a similar change in the other. Consider two domains, D1 and D2, which have a 1-to-1 logical dependency between them but have different data representations. When we swap a value in one of the domains, a corresponding swap must also be made in the second. More specifically, if x, x'$\in$ D1 and y, y'$\in$ D2 such that x $\leftrightarrow$ y, and x' $\leftrightarrow$ y', then if x is swapped with x', y must also be swapped with y' and vice versa. For example, consider two tables containing identical demographic information about patients. One table uses just one column to store birth dates, say for example, 05/11/2009 for patient A, while another table uses three columns to store the same birth date of patient A, say, 05 as MM, 11 as DD, and 2009 as YYYY. iSTDE ensures that the two tables maintain the same logical dependency after mangling, that is, if the birth date 05/11/2009 is swapped with some other date 07/10/2007 in one table, iSTDE also makes the same logical swap in the other table that uses three columns to represent the birth dates.

These 1-1 logical domain dependencies can also produce noise. For example as shown in Table 3-1, if value 6/27/2006 in Domain 1 is swapped with another value, a corresponding swap cannot be made in Domain 2 because no entry exists for the same birth date. In this case, iSTDE will not swap the value in Domain 2. Ideally, preserving this type of noise in the real data is a good idea as it enhances the quality of the test-data.

Table 3-1. Noise Production Example in 1-1 Logical Dependent Domains.

Domain 1	**Domain 2**		
mm/dd/yyyy	Year	Month	Day
5/1/2006	2006	May	1
5/6/2006	2006	May	6
5/10/2006	2006	May	10
6/26/2006	2006	June	26
6/27/2006			
	2007	Jan	1

The second type of dependency in the iSTDE mangling process is called data value dependency. Two domains D1 and D2 are said to have a data value dependency when for any single record that uses values from both domains, there is a constraint involving those values in these domains. Then, if values in D1 are swapped, a random swap must also be made in D2, but the original constrain must still hold (if the original record satisfies that constraint). More specifically, if x, x'$\in$ D1and y, y'$\in$ D2 such that x$\otimes$y where $\otimes$ represent some constraint, then if x is swapped with x', y can also be swapped with y' as long as x'$\otimes$y'. Stated another way, we can say that a child birth date in any of the databases cannot be greater than a parent birth date.

The third type of data mangling relates to the mangling of computed fields and partially computed fields. These two types of fields are considered dependent and are derived from other fields. For example, a full name field can be a computed field as it is

derived from a first name and last name. When iSTDE mangles the first name and last name, it also re-computes the full name to maintain name consistency. Partially computed fields are those fields that have partial independent values and partial computed values. For example, a contact name field can contain a brother name. It might be possible that two brothers have the same last name, so if we mangle the last name, we also need to re-compute the partial value in the contact name field.

The fourth type of mangling is field-based mangling. In this type of mangling, records are shuffled vertically rather than horizontally, i.e., one change in a record requires lots of field changes. See Table 3-2. As shown in Table 3-2, the ColumnName field contains other field names as values and the ColumnValue field contains the actual field values. iSTDE checks the individual ColumnName values and performs corresponding domain mangling in the ColumnValue field.

Table 3-2. Example of Field-based Mangling

Column Name	**Column Value**
firstName	Joe
lastName	Edward
After Field-based Managling	
middleName	Walker
firstName	James

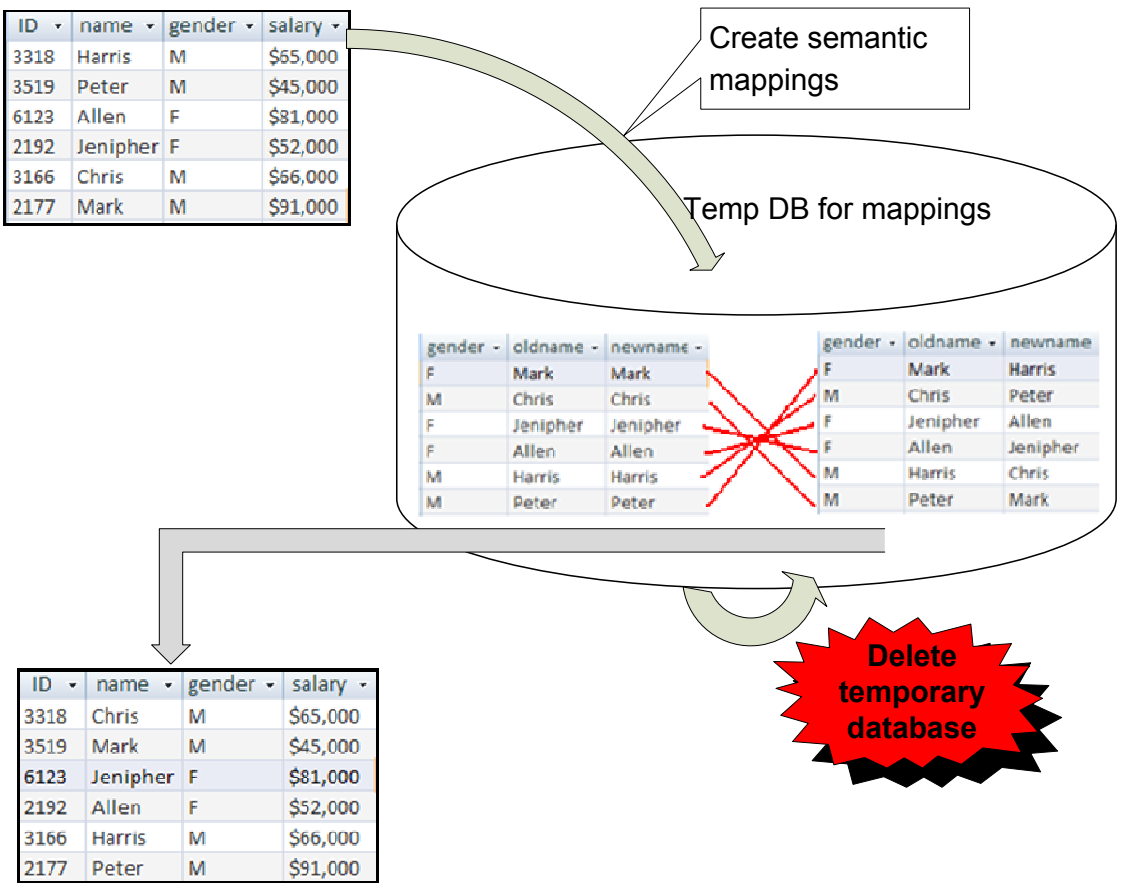

Figure 3-3. Data mangling.

3.3.6Transferring Mangled Data

Once mangling of data is complete, the fifth step in the entire process is the automatic transfer of the de-identified test-data to the user-specified, unprotected environment. To do this, iSTDE creates a dump of all the temporary databases, transfers them via a secure copy to the unprotected environment, and executes remote commands to restore those dumps in databases in the unprotected environment. A significant challenge while transferring the test-data was to manage the access controls and firewalls. iSTDE uses a number of built-in scripts in a confidential environment to manage these network transfer obstacles.

3.3.7 Destroying Mappings

In this semantic-based extraction process, iSTDE produces and uses data. The data files are created in the third step of iSTDE execution and contain extracted records from different database managers. Ideally, this step destroys all traces, i.e., data files that could indentify or even hint at any sensitive information about patients. Thus, iSTDE ensures the sensitivity of patients records by deleting the temporary databases as well as the data files mentioned above. Note, we do not delete domain dictionaries while destroying other mappings as they are part of a separate process and also because they exist in the secure environment so are not at risk of getting unwanted access.

3.4Discussion on PII and Preserved Semantics in CHARM

3.4.1Identification of PII in CHARM

In CHARM, there are hundreds of tables that include thousands of data fields. We only mangle identity domains or PII, which are the domains that have an impact level (e.g., low, moderate, or high as defined in Section 2.3.2). There is no benefit in mangling fields that have a "*not applicable*" impact level and would not increase confidentiality. Below is a list of PII domains in CHARM that have a low, medium, or high impact level:

1. Names : First names, last names, middle names
2. Birthdates
3. Addresses
4. Phone numbers
5. Personal identification numbers : social security numbers
6. Information about an individual that is linked to one of the above (parents names, parents addresses, titles, race)

3.4.2Anonymizing PII in CHARM

The motivation of mangling the identity domains comes from the above mentioned information about PII (Section 2.3). We assume that our set of identity domains are complete according to the PII standards mentioned by NIST [23] and HIPAA [25]. In CHARM, we used the data anonymization approach rather than data de-identification approach as there must not be any way to re-identify demographic information in the CHARM test environment.

3.4.3List of Preserved Semantics in CHARM

Ideally, capturing a complete list of preserved semantics for a data set would involve a huge amount of data, and compiling them in a list is not a feasible idea due to their different impacts on different organizations. However, we can restrict these semantics by constraining them to a specific application we are testing. The number of preserved semantic rules and their descriptions are subjective measures and are tied to the context of the application. For example, the kinds of semantics to be preserved for the CHARM matcher and address cleaner components are as follows. First Name frequencies – A shape of a histogram of name count over names is roughly the same, although it is not necessary for the names to be the same. More formally, "there exists" fn_1 : (|P(firstname==fn_1)| such that |P| = 5%) => there exists fn_2 : (|T(firstname==fn_1)| such that |P| = 5%).

1. Last Name frequencies – Same as for First Name frequencies.
2. Name / genders dependency – A person who is a male child or father would retain his gender characteristics after mangling.

3. Twins – Everybody born on a given day has the same birth date after mangling Complete address – An address can be characterized as a complete address when all the individual elements like street1, street2, state, city and zip code are the defining properties of one another.
4. Parent Last Name / Child Last Name –Relationship between the parents and their children.
5. Degree of Variations: Complete/Incomplete names, spelling variations, case variations, dummy names and garbage data.

If we add the *Sync Engine* to the context, we would add the following additional semantic-preserving constraint.

6. BD <= DD or DD=null or BD=null

3.5Discussion on Extraction Set Size, Semantics Constraints, and Mangling

3.5.1Size Constraints

There is no need to impose a lower-bound size constraint on the extracted test bed to guarantee the sensitivity of demographic information. The PII domains or identity domains are formed and populated from the whole set of databases in CHARM, which contain millions of records. Thus, even if the extracted test bed retrieves just one child record, that record would be mangled with any random instance of PII domains, built from millions of children born in Utah.

3.5.2 Exceptions

A non-PII domain can identify records based on prior information: Suppose a person has a rare disease, and somehow this case of the rare disease becomes known publicly. In such a case, even in the mangled data, the person can be identified from prior

disease information. For example, if Joe has some rare disease X, and in the mangled data Joe is represented as Ken, based upon prior information, one can assume that Ken's information is actually Joe's information. Hence, all Ken instances are Joe and vice versa. However, in the limited context of CHARM, such a scenario is very unlikely to occur. Also, a future version of iSTDE will safeguard against this problem by generating enough noise that it observes even rare conditions.

3.6Experience with iSTDE and Discussion

iSTDE provides an environment wherein CHARM applications can access all PPs' data in one database server. Developers can test their applications with full integration without external connections. The test beds produced by iSTDE are stored in a repository from which they can be quickly loaded and restored for testing purposes. Developers write unit tests in which they check complex queries using these test beds before running them on real data.

SyncEngine is a CHARM application that has been extensively used in iSTDE-based test beds. It checks for new updates in participating programs (PPs) and adds their references in Core and Pampers databases. It also copies the newly found PPs' records in CHARM's internal PP databases. Hence, SyncEngine populates these three sets of databases. PPs are independent databases that are owned by different health-related organizations. CHARM contains replicate database instances of each PP along with Core and Mappers databases (Figure 2-1). The Core database is a part of CHARM and contains reference IDs as well as some demographic information for all the records in PPs, whereas Mappers contains linking information between Core and CHARM's internal PP databases. The developer of SyncEngine used iSTDE and found that mangled

data provided the proper simulation of a real data environment. SyncEngine found consistency in all three sets of databases. The set of records that exist in the Core database also existed in CHARM PPs' databases. Additionally, the correctness of this consistency was determined by checking the person birthdates and their associated demographic information, such as names, addresses, etc. During execution, SyncEngine successfully used this data to detect new updates in PPs.

Other CHARM applications need test-data that are not only meaningful but also contain missing values, incomplete information, garbage data, and duplicates. For example, Matcher is an application that detects duplicate child records in the PPs' databases. Testing of Matcher requires records in which patients have more than one entry. Similarly, Query Manager checks for metadata formats as well as the length of the fields. iSTDE produces test beds from real production databases; thus, it provides all of these real data characteristics.

CHAPTER 4

COMPARISON OF TEST DATA EXTRACTORS WITH TEST DATA GENERATORS

4.1Overview

The software testing process for database applications often involves different testing techniques at different stages in development. Each testing technique is purposefully unique and thus requires different types of test-data characteristics. For example, in unit testing, the challenge is to form a small yet sufficient test bed for checking individual methods, regression testing requires a way to repeatedly run the test beds created for unit testing, functional testing needs a dataset that can verify a system's compliance with the requirements, input-validation testing is required to satisfy certain validation rules, performance and load testing demands large, consistent test beds that can be processed repeatedly to ascertain appropriate benchmarks, and integration testing needs correlated chunks of data across multiple data sources.

Getting the right dataset for testing standalone database applications is hard enough, but doing so for integrated applications using heterogeneous databases is even harder. Database application testing requires considerations beyond those found in algorithm-intensive applications. For example, test engineers need to verify the correctness of database schemes, business rules, key and non-key constraints, validation checks, data-type conversions, and range constraints. They also need to validate the application behavior against valid and invalid values, null conditions, and garbage data. Even then, the test dataset may not be sufficient unless it checks for inter-table and inter-database dependencies, such as hidden and explicit data dependencies, inter-table patterns

and data correlations across different databases in a federated system. To meet all these testing challenges, test engineers need testbeds that satisfy all these data characteristics.

Manually constructing test-data to meet all these criteria is typically not practical, because it is a laborious work that requires considerable attention to detail which, in turn, is itself prone to unintentional errors. Automatic creation of test-data, on the other hand, can quickly create large amounts of test-data with predictable characteristics, but generating all the complex sort of data characteristics at once is a difficult job, as we have seen in CHARM. Finally, the only option left is to use the real data, but here again we have many challenges, such as restricted access, semantic and syntactic differences in different data sources, and sensitivity of data.

The prime motivation in writing this comparison study is to provide a framework that helps testers in identifying the test-data requirements for the system and then assists them in selecting an appropriate test-data creation approach. The right selection of a test-data creation technique can deliver rich, appropriate, and domain-specific test-data that can significantly simplify and improve the overall testing process. A well established testing process thus ultimately reduces testing time through availability of large and proper test beds in development environment. It also improves the quality of testing by providing quality test-data. These two critical testing parameters (i.e., time and quality) thus become the contributing factors in cutting overall development and maintenance costs.

This chapter describes a theoretical study that makes two contributions towards helping software testers make informed decisions about how they approach the verification of a complex system. First, it presents an innovative comparison scheme for

analyzing the two fundamental test-data creation approaches, namely, test-data generation (*TDG*) and test-data extraction (*TDE*). Section 4.2 provides some additional background information about these approaches, and section 4.3 describes the comparison scheme in more detail. Section 4.4 includes a theoretical comparison of *TDG* and *TDE* using this scheme. This comparison not only serves as an example of how the scheme can be ably used, it also contains the author's conclusions about the relative strengths and weakness of both approaches.

4.2Two general Approaches to Test-Data Creation

A review of related literature and existing off-the-shelf software for creating the test-data reveals a wide range of approaches that vary in scope, strategy, testing features, price, and platform. However, they all fit into one of two general categories based on whether they utilize existing real data or generate the data, i.e., test-date extraction or test-data generation, respectively. The following sub-sections analyze and compare these techniques and their tools. In general, TDGs can quickly generate large data sets with limited capabilities to produce real data characteristics such as personal identifying information (Section 2.3). On the other hand, TDEs have real data characteristics but have issues regarding extraction time, are less resilient to data changes, and can compromise data privacy.

4.2.1 Data Generation

Generation techniques create test-data by executing various construction methods within the constraint of the target database scheme. The study explored various research-based generation theories and eight tools that support *test-data generation*.

A survey of the literature uncovered twelve techniques that represent fundamentally different ways of generating test-data. Other techniques certainly exist, but they can be viewed as variations or compositions of these core ideas.

1. Riva D., Suarez and Tuya [32] generate the test-data through query constraints by using a declarative language named Alloy which defines elements, a set of relations between them, and a set of constraints that restricts the output space of coverage. This is an interesting test-data generation scheme, but its data-generation capability is strictly limited to queries input. In integrated systems creating such complex queries that involves dozen of tables and which also cover adequate semantics is not a practical idea.
2. Chan and Cheung [19] transform the embedded SQL statements into procedures in some general-purpose programming language, and thereby generate test cases using conventional white box testing techniques. But after language transformation, to make this method useful, we need to know more about the specific database testing characteristics.
3. Haller [31] models a generic software testing process, defines the goals and coverage conditions for different steps of the testing process, and finally provides a roadmap for the emerging domain of testing database-driven applications. It is an interesting framework; however, it does not provide an appropriate way of testing integrated databases.
4. Tsai, Volovik and Keefe [18] generate test cases for databases from relational algebra queries. Requiring the specification to be expressed in terms of relational algebra is in some ways difficult to implement. However data generation is based

upon the query which is broken down into predicates, which is having very limited applicability in integrated systems.

5. Haftman and Kossmann [33] describe an approach to reduce the regression testing time for database applications by running the tests in parallel. Though this approach can be an interesting extension to our current data extraction facility, it cannot become an alternative approach for our devised testing process.
6. Davies , Bevnon and Jones [17] propose generating the dataset from validation rules that include constraints and attributes from user input.
7. Dalal and Jain introduce model-based testing [20] to generate the test cases. In this approach, the authors build a data model to generate the test cases. A data model is essentially a specification of inputs to the software and can be developed early in the cycle from requirements information.
8. Zhang, Xu and Cheung [16] generate a set of constraints that collectively represents a property, against which the program is tested. However, governing a set of constraints for some unknown values and then binding them to domains is a difficult task.
9. Slutz [21] discusses stochastic testing of SQL. He developed a tool that can randomly create a very large number of SQL statements without human intervention. The SQL statements are generated randomly (or stochastically) which provides the speed as well as wider coverage of the input domains.
10. Gray's [22] approach can quickly generate billions of records for a table with dummy data, having certain statistical properties with a goal to test the performance and load of the database servers.

11. Chen et al. [15] have developed a framework that supports category-partition test case generation. Each valid combination of the input parameters, choices, or groups corresponds to a test frame or template for test cases. The tester guides the generation of test frames by specifying relationships between different input parameters.
12. DBTestGen [13] is another tool that generates the test-data after taking as input, the database schema- and semantics-based user-defined characterization rules in a hierarchical way.
13. Chays [30] developed a data generation tool AGENDA for testing database applications that takes as input the database schema, application source code, and sample values. The tool populates the database, generates inputs to the applications, executes the application on those inputs, and checks some aspects of correctness of the results database state with application output.

Among industry-supported data generation tools, we identified six common data generation methods. Table 4-1 lists these methods and shows which of the eight tools support them.

Table 4-1.Seven Common Test-Data Generation Methods and Tools That Support Them.

	Test-data Generation Features	***Data Generator [3]***	***DTM Data Generator [4]***	***ForSQL Data Generator [5]***	***Automated Test Data Generator [6]***	***DB Data Generator V2[9]***	***TurboData[10]***	***Tnsgen – Test Data Generator [7]***	***DBTESTGEN –Database Test Data Generator[13]***	***AGENDA[30]***
1	Random number generation		×	×	×	×	×	×	×	×
2	Random string generation				×		×	×	×	×
3	Percentage-based generation		×						×	×
4	Pattern/grammar-based generation		×			×			×	×
5	Pre-defined domains	×		×		×	×		×	×
6	Parent/child record generation						×		×	×

1. *Generating a random number*. This method typically allows the tester to specify some simple range limits or length constraints. Using a random-number generation method, for example, a tester can create numeric data for a salary field in a payroll table that range between $30,000 and $70,000. Six of the eight tools support random generation for numeric data.
2. *Generating random strings*. Like the above, a random-string generation method can create string data for names or address fields containing characters a-z or A-Z. Three tools provide support for this method.
3. *Percentage-based generation*. This method creates a percentage-based distribution of different data values for a field. For example, it can create data for a first name field in a person table, such that it contains 60% male names, 40% female names, and 10% null values. Only one of the tools supports this type of random data generation.

4. *Generation of data according to user-defined patterns or grammars*. For example, let '9' represent a digit, 'A' represent a capital letter, and 'a' a lower-case letter. A tester could use the pattern '9999-AaA-99' to create 11-character strings consisting of 4 digits, dash, an upper-case letter, a lower case letter, another dash, and two more digits. This technique is most applicable for string fields that require pattern-based data values, such as phone numbers, postal codes, and various kinds of identifiers. Two of the eight tools support this technique.
5. *Generating data from predefined domains or dictionaries*. A tester could use this method to populate a first name field with values from pre-defined domain of common person names. Five tools support this method and provide rich libraries of pre-defined domains.
6. *Generating records for tables that have a master-child dependency*. For example, testers can use this technique to create data for an inventory system consisting of order records, which in turn contain line item records.

For our comparison of the two test-data creation approaches, we needed to choose prototypical tools or techniques from each approach. For generation we selected the DBTestGen [13] and AGENDA [30] as prototypical tools because they appear more suitable and provide maximum coverage to the data generation techniques.

4.2.2Data Extraction

Test-data extraction, the second general approach, creates test-data from real data sources rather than through generation methods.

One of the most common data extraction problems is type of data access, i.e., random access versus sequential access. Data in text files only allow sequential access;

therefore, coordination between records to get a correlated data set is hard. Also, full structure of the text file is difficult to be captured in the file itself; thus, more external definition is needed. On the other hand, databases have support for random access, and by enforcing the intended structure, the actual dataset can easily be extracted and correlated.

Another problem is multiple data structures. The extractor needs to have flexible data structures that can support different types of data sources, e.g., multiple text files or multiple tables in a database. By having support for multiple data structures, an extractor can retrieve consistent (internally complete) data slices by joining the data structures in appropriate ways.

Last but not least, heterogeneity is one of the hardest challenges for extracting appropriate and synchronized data sets. While extracting similar data from multiple data sources, an extractor can come across different data integration problems, e.g., dealing with heterogeneity in the data sources, record matching and merging capabilities. These problems are considered among the hardest research problems in systems integration [35].

A review of nine tools that support test-data extraction (many of which also support test-data generation) uncovered three basic extraction methods, namely, extracting data from a file, extracting data from tables in a single database, and extracting data from multiple unrelated databases. Table 4-2 shows the correlation among these methods and the tools that support them. Table 4-2 also shows two additional extraction methods supported by iSTDE [14] and IBM Optim [27] for testing integrated systems and systems containing confidential data. A brief description of the methods is given below.

1. *Extraction of data from data files in a variety of storage formats*. This extraction method is actually similar to the pre-defined domain generation method, except that here, the data file contains "real" data instead of pre-defined sample data, and therefore should posses realistic qualities, such as some noise (bad data).

Table 4-2.Five Test-Data Extraction Techniques and Tools That Support Them.

	Test Data Extraction	***Data Generator [3]***	***DTM Data Generator [4]***	***ForSQL Data Generator [5]***	***Automated Test Data Generator [6]***	***DB Data Generator V2[9]***	***TurboData[10]***	***Tnsgen – Test Data Generator [7]***	***EMS Data Generator for MySQL[11]***	***Symentic-based Test Data Extraction for Integtrated Systems [14]***	***IBM Optim[27]***
1	From data from files		×	×				×		×	×
2	From data from multiple tables in one database		×	×					×	×	×
3	From uncorrelated data from multiple databases								×	×	×
4	Correlated real data from multiple related databases									×	×
5	De-identified data from confidential databases									×	×

2. *Extraction of correlated test-data from multiple tables in a single database, preserving inter-record dependencies within tables and across tables*. A common and relatively simple type of inter-record dependency is a referential integrity constraint. This method also supports implied referential integrity and row-count dependency.
3. *Extraction of data from multiple uncorrelated databases, running on different platforms, but without any cross-correlation of data between the databases*. This technique can be viewed as a convenient way to aggregate multiple-table extractions from different databases into one step.
4. *Extraction of data from multiple correlated databases*. Testing an integrated system and its constituent components requires correlated slices of data that preserve inter-database relationships and hidden dependencies. For example, while testing a record matching component in a person-centric application, it would be meaningless to extract one set of person records from one database and a non-overlapping set of records from another database. Such test beds would not be able to verify the results of any actual data integration.
5. *Extracting test-data from a system that contain confidential data*. This technique includes the special problem of preserving privacy. For example, if person data is used outside a secure environment, it must not include any recognizable personal identifying information. This fifth extraction method manipulates the extracted data to de-identify sensitive personal information without compromising the real data characteristics.

The above paragraphs discuss different data extraction techniques and related tools. Table 4-2 gives a comparative analysis of the tools and their supportive techniques. For our experiments, we selected iSTDE and IBM Optim as our prototypical tools because they provide support for the majority of our data extraction techniques. These tools can appropriately extract synchronized data sets from multiple data sources, can handle data transformation complexities from different types of database managers, and provides a good solution to deal with confidential data by applying data masking strategies.

4.3Comparison Method

4.3.1Overview

Usually, comparisons between two ideas or procedures can be done using empirical or theoretical methods [28]. In general, empirical methods involve collecting data on which the researchers base their conclusions. With theoretical methods, researchers rely on known facts, relations, properties, axioms, and existing theories to derive their conclusions. Unfortunately, doing a thorough comparison of test-data creation tools or techniques using empirical methods would be impractical because the cost of acquiring a representative set of tools is prohibitive. Even if we could obtain a sufficient number of tools, there would be so many extraneous variables in any experiment involving multiple tools from multiple venders that it would be difficult to formulate creditable conclusions from the experimental data.

Therefore, our comparison method uses a theoretical approach which consists of the following six-step process:

1. Establish comparison criteria that focus on the inherent difference between TDG and TDE.

2. Select representative tools that support the common TDG and TDE techniques.
3. Set a test environment for exercising the same tools.
4. Formulate theoretical statements about how TDG and TDE compare with respect to each of the criteria.
5. Validate those statements in a test environment with sample tools.
6. Draw conclusions.

The seven areas of comparison in Table 4.3 form a framework for comparing the two test-data creation approaches. Area A discusses a comparison of standalone databases restricted to inter-table dependencies in a single database. Area B compares inter-database related dependencies. The target-scheme (area C) is related to database objects that do not contain data. Comparisons on non-functional requirements (such as database access, usage and performance) is described in area D, database refactoring related comparison which involves continuous changes over time is discussed in area E. The preferences in testing techniques may have a significant impact on the choice of test-data creation approach; thus, this comparison is defined in area F. Similar to the testing

Table 4-3.Description of Comparison Criteria.

A. Stand-alone Database Context	
Comparison Criteria	**Description**
1. Support for Functional Column Dependencies	
a) One-way data dependency	Column A determines values for another column B or A→B.
b) Bi-directional data dependency	Columns A and B are dependent on each other, in other words, A→B and B→A.
c) Aggregate data dependency	Dependency among a group of columns say C_1, C_2, C_3, … C_n such that individually they have bits of information but as a unit, they provide a complete description.
2. Support for Functional Row Dependencies	
a) Row-wise dependency	A row filters possible sets of values in another row.
b) Nested fields	Fields are nestedly tied such that their values can be deduced in a hierarchical way.
3. Data represents various degree of variations	Incomplete and complete names, spelling or case variations, dummy values and garbage data.
4. Support for Derived Data	
a) Computed field values	A field that derives its values from two or more other fields.
b) Restricted value to computed set	Values that have partial independent values and partial computed values.
5. Grammars Support	
a) Multi-Grammars-based data generation with n-subsitutable components	Data generation with multiple segments, each one having its own grammar. A grammar can be semantics-based, regular-expression, context-free, etc.
6. Contains Data Frequency Patterns	
a) Unique values	These values are normally associated with rows for identification purpose.
b) Single columns data value frequency Distribution	Determines frequency distribution of data in a single column.
c) Multi-column data value frequency distribution	Two or more columns share a common frequency-based pattern of data values.
7. Exhibts Complex Data Structures	
a) Repeated groups	Repetition of data groups due according to database normalization rules.
8. Support for Inter-table Dependencies	
a) Explicit referential integrity constraint	References via {primary key– foreign key} relationship.
b) Indirect referential integrity constraint	An implied value-based referencewithout {primary key– foreign key} relationship.
c) Row-count constraint	For example, a column value with row count constraint in a parent table can provide summary information (or apply restrictions) about its children.
d) Inter-table patterns	Distribution of data values in a parent-child relationship pattern.

B. Federated Databases Context	
Comparison Criteria	**Description**
1. Support for Inter-database Dependency	Synchronized data across collections of federated databases.

C. Target Schemes Context	
Comparison Criteria	**Description**
1. Handling of variations in database managers	Different datbase managers have their own language formats.
2. Handling of views	How *TDGs* or *TDEs* handle view, i.e, process view as view or view as table.
3. Handling DB objects other than tables and views	For example, sequences, triggers and indexes.

D. Setup, Use, and Performance Context	
Comparison Criteria	**Description**
1. Need access to existing database	To create the test-data, *TDGs* do not need access to data whereas *TDEs* need that access.
2. Easy to deploy	Issues related to just deployment and not configuration.
3. Speed meets users expectations	Speed means execution time of *TDGs* and *TDEs* to create the test-data.
4. Easy to defining database characterization	Characterization determines what type of data would be generated for a column.

E. Database Refactoring	
Comparison Criteria	**Description**
1. Refactoring of database object that contains data	
2. Addition or deletion	DML operations on key columns, non-key columns, independent and dependent tables.
3. Replacement operations	Replacement issues related to *column, table, table-to-column vs. column-to-table and keys.*
4. Split operation	Splitting of tables, columns, and large objects.
5. Migration and reordering	Deals issues related to columns and tables.
6. Renaming	Deals renaming issues related to columns and tables.
2. Refactoring of database object that do not contain data	
1. Addition of Triggers	Handling of triggers and their effects on TDG and TDE.
2. Cascading deletion	Deleting a parent record will also delete the child records from dependent tables.
3. Addition of constraints	Key, non-key, and business rule constraints.
4. Encapsulate table with a view	Discusses how addition and deletion of views can affect TDG and TDE.
5. Introduce Index	Discussion on how indexes can affect performance of TDG and TDE.

F. Testing Techniques	
Comparison Criteria	**Description**
1. Unit Testing vs. Regression Testing	Method level testing and execution of unit test suits.
2. Functional Testing of Standalone modules	Checks completeness of functional requirements in individual components.
3. Integration Testing	Checks completeness of functional requirements in integrated set of components.
4. Performance Testing vs. Stress Testing	Evaulate and set performance benchmarks and checks systems behavior in heavy loaded environment.
5. Data Validation Testing	Application is tested for illegal, wild characters and many other types of validations.

G. Social and Time Factors	
Comparison Criteria	**Description**
1. Semantics changes due to domain evolution	Changes in meanings of data due to changes in database schemes.
2. Social factors that affect nature of data	Different cultural and social aspects have diffent types of data requirements.
3. Data Generation that does not expose personal Privacy	Real data may contains sensitve information that must be presereved and protected.

comparison, social and time factors may also affect application data requirements which is defined in area G.

4.3.2Comparison Context

Selecting an ideal testing environment to compare the two test-data creation approaches, i.e., TDG and TDE was a challenge. First of all, how to best define an ideal database testing environment was a difficult question. We found that an ideal environment should at a minimum exhibit the following characteristics. It should provide an integrated database environment, with varying database managers. These databases should be federated yet maintained independently. Data should have duplicate instances and contain both meaningful and bad data. Database schemes should exhibit well

designed as well as poorly designed characteristics, undergo continuous refactoring, have a rich set of data objects with a reasonable complexity level, allow varying access levels to their users, have large datasets, and contain sensitive information. A testing environment that can simulate these characteristics can be considered sufficient for the seven areas of comparison for the two approaches.

Utah Department of Health's Child Health Advance Record Management (CHARM) provides an ideal database testing environment and meets all the challenges mentioned above. It is a collection of integrated and federated databases with different database managers, such as Oracle, SQL Server, PostgreSQL, MySQL, etc., and are maintained by independent organizations. To keep the study within reasonable limits, our experiment only employed five CHARM participating databases: one that holds core demographics for record matching (CORE), the Vital Records database (VS), the Utah State-wide Immunizations Information System (USIIS), an Early Hearing Detection and Invention database (EDHI), and a New Born Screening database (NBS). These databases hold duplicate information about persons' demographics; some of the database schemes have poor designs, i.e., contain repeating groups; they all have large data sets both horizontally and vertically; different data sources give different levels of access to metadata and data; and the information they contain is person-sensitive.

CHAPTER 5

COMPARISON RESULTS

Using the comparison method described in Section 4.3, we evaluated the two test-data creation approaches in the testing environment on seven different areas. In our theoretical comparison approach, many test-data creation tools were used to prepare the test beds, and then the nature of the data was evaluated using sample data sets. Comparison results of this approach are described in this section. Tables 5-1 to 5-7 summarize this comparison.

5.1Comparison in Context of Standalone Databases

This section identifies eight different types of data characteristics we found in the CHARM integrated database environment, regarding standalone databases. The following paragraphs highlight each of these data-characterizations and evaluate the competitiveness of two test-bed creation approaches. Table 5-1 provides a summary of these comparisons.

5.1.1. Functional Column Dependencies. We found three types of functional column dependencies in CHARM: one-way data dependency, bi-directional data dependency, and aggregate data dependency.

One-way data dependency: In table T, column A determines the value for another column B or A→B. One-way data dependency exists between first name and gender columns in a person table, where first name determines gender. In CHARM, the matcher component uses a father name to search for his children. In the absence of one-way data

Table 5-1. Comparison in Context of Standalone Database.

	Check List	Test Data Extractor	Test Data Generator
1	Functional Column Dependencies a) One-way data dependency b) Bi-directional data dependency c) Aggregate field data dependency	• Default support, and promises quality data without complexity. • However, speed to re-extract data is inefficient.	• Support exists but does not promise quality data. • Defining characterization rules involves moderate to severe complexity. • Data generation speed is fast.
2	Functional Row Dependencies a) Row-wise dependency b) Nested fields	• Default support, and promises quality data without complexity. • However, speed to re-extract data is inefficient.	• Support exists but does not promise quality data. • Needs good schema and data understanding. • Defining characterization rules and conditional data structures that involves moderate to serve complexity. • Data generation speed is fast.
3	Degree of variations a) Complete and incomplete names b) Spellings variations c) Case variations d) Dummy names and Garbage data, such values containing unusual characters, representative of what might be found in real data.	• Sufficient support, and promises quality data with slow extraction speed.	• Support exits but don't promise quality data for complete and in complete names. • No support for spelling variations. • Support exits but don't promise quality data for case variations. • Insufficient supports also don't promise quality data and it is hard to create a collection of rich set of dummy and garbage data values.
4	Derived Data a) Computed field values b) Restricted value to computed set (RCS)	• Sufficient support with slow data; extraction with slow extraction speed.	• Sufficient support with fast generation speed for computed field values. • Insufficient support for RCS. Involves moderate to severe complexity
5	Grammars Support a) Multi-Grammars-based data generation with n-subsitutable components	• Default support, and promises quality data without complexity. • However, speed to re-extract data is inefficient	• Insufficient supports also don't promise quality data. • Defining characterizations for restricted set segments are not easy.
6	Data Frequency Patterns a) Unique values b) Single column data value frequency distribution c) Multi-columns data value frequency distribution	• Default support, and promises quality data without complexity. • However, speed to re-extract data is inefficient	• Sufficient support exists for unique value generation. • Support exists but don't promise quality data for b) and c). • Defining characterization rules for class-based distribution patterns or composition of more than one dependency involves moderate to serve complexity for b) and c). • Data generation speed is fast.
7	Complex Data Structures a) Repeated groups	• Default support without adding any complexity, but extraction speed is slow	• Extremely hard to generate repeating groups.
8	Inter-table Dependencies a) Explicit referential integrity constraint b) Indirect referential integrity constraint c) Row-count constraint d) Inter-table patterns	• Default support, and promises quality data in constant complexity • Defining characterizations is not an issue. • However, speed to re-extract data is inefficient.	• Support exits and promises quality synchronized data, but defining characterizations is a complex task. • Insufficient support. Needs good schema knowledge and involves complex characterizations. • Sufficient support. A scan of dependent tables can obtain row-count in parent tables. • Insufficient support Needs good schema.knowledge and involves complex characterizations.

Table 5-2. Comparisons in Context of Federated Databases.

	Check List	Test Data Extractor	Test Data Generator
1	Support for inter-database dependency	• Support exists with compromised speed. • Promises quality synchronized data sets across data sources.	• Hard to incorporate this characterization • Does not promise quality test data.

Table 5-3. Comparisons Related to Target Schemes.

	Check List	Test Data Extractor	Test Data Generator
1	Handling of views	• Automated support.	• Automated support but less efficient • Hard to generate the view when source table is not included in testing domain.
2	Handling of database objects other than tables and views	• Manual intervention is required.	• Manual intervention is required.
3	Handling of variations in database managers	• Sufficient support by maintaining a repository of transformation rules.	• Insufficient support.

Table 5-4. Comparisons Related to Set-up, Use, and Performance.

	Check List	Test Data Extractor	Test Data Generator
1	Need access to existing database	• Heavily database dependent.	• Slightly database dependent
2	Easy to deploy	• Easier to deploy.	• Have deployment issues
3	Meeting users' expectations for speed	• Meets user expectations within certain constraints.	• Does not meet user expectations
4	Defining data characterization	• Required.	• Not required

Table 5-5. Comparisons in Context of Database Refactoring.

	Check List	Test Data Extractor	Test Data Generator
Refactoring of Database objects that contains data			
1	Addition, update or deletion • Key columns • Non-Key columns • Independent tables • Dependent tables	• Support without adding complexity, requires data re-extraction. • Overall less complex but an inefficient process.	• Support exists but needs to redefine scheme and regenerate data for all related tables. • Overall complex process but efficient.
2	Replacement [table-to-column vs. column-to-table, keys-replacement], split or merge operations on database objects	• Support without adding complexity, requires data re-extraction. Overall, less complex but an inefficient process.	• Support exists but needs to redefine scheme and regenerate data for all related tables. Overall complex process but efficient.
3	Migration and reordering	• Slow process, data re-extraction is required.	• Comparatively fast process and not complex either.
4	Renaming of database objects	• Slow process, data re-extraction is required.	• Comparatively fast process and not complex either.
Refactoring of Database objects that do not contain data			
1	Addition, update or deletion of triggers, cascading deletes, constraints and indexes	• Automatic support but inefficient approach. • Needs data re-extraction.	• Efficient approach without the need of data regeneration. • Includes manageable complexity to incorporate this characteristics.
2	Encapsulate table with a view	• Adaptable, needs schema update.	• Can be supported through manual intervention.

Table 5-6. Comparisons in Context of Testing Techniques.

	Check List	Test Data Extractor	Test Data Generator
1	Unit testing versus regression testing	• Not very useful.	• No very useful
2	Functional testing of standalone modules	• Sufficient support that promises quality data, but data extraction speed is inefficient.	• Insufficient support, not quality data either but generation speed is fast.
3	Integration testing	• Sufficient support that promises quality data, but data extraction speed is inefficient.	• Insufficient support, not quality data either but generation speed is fast.
4	Performance testing versus stress testing	• Least preferred.	• Preferable due to capability of quickly generating large amounts of data.
5	Data validation testing	• Preferred choice due to automatic support for rich data rules.	• Less preferred choice, involves a lot of complexity to inject data validation rules.

Table 5-7. Comparisons Related to Social and Time Factors.

	Check List	Test Data Extractor	Test Data Generator
1	Semantics changes due to domain evolution	• Provides support without complexity, needs data regeneration. • Promises quality data. Speed is an issue.	• Insufficient support. Needs to redefine complex characterization rules. • Does not promise quality data.
2	Social factors that affect nature of data	• Default support that promises quality data.	• Insufficient support.
3	Data generation does not expose personal privacy	• Support exists but is less efficient.	• Efficient support.

dependency which distinguishes a father from a mother, our component could not find any children, and thus would be unable to run many matching-related unit tests.

It is possible to incorporate this data characteristic both in TDG and TDE. TDE has default support for this characteristic and also delivers quality test-data. The complexity to include this characteristic in TDE remains constant while increasing its scalability; however, data extraction speed is inefficient. On the other hand, in TDG we need to write a set of characterization rules having moderate to severe complexity that generate values for the dependent column using the lookup table of names. Generation gets even more complicated for dependencies that do not use lookup tables, i.e., generating data for two date type columns where *child birth date < parent birth date*.

Overall, *TDG* promises fast data generation, but test-data quality is not at par with TDE-generated test-data.

Bi-directional data dependency. For table T, columns A and B are dependent on each other, or A→B and B→A. We can see bi-directional data dependency between state code and state name columns of the US States table, where both columns determine values for each other. In a *CHARM* component, there is a requirement to define unique value patterns for two columns, i.e., generate a unique value in the second column for every unique value of the first column. TDG's data generation technique for this characteristic is similar to that of one-way data dependency, with the one difference being that here we cannot distinguish between independent and dependent column(s). In comparison, *TDE* has an edge over *TDG* because of reduced complexity and quality test-data.

Aggregate data dependency. A dependency exists among a group of columns, say C1, C2, C3 … Cn, for Table T such that individually they have bits of incomplete information but as a unit they provide a complete description. For example, *address* table can be composed from five columns that have aggregate data dependency among them, i.e., {$street\_address_1$, $street\_address_2$, $state_1$, zip_1, $city_1$}. Individual fields contain pieces of address information about a person and are partially dependent on one another.

AddressCleaner. This component in *CHARM* needs its aggregate data dependency tested. This component takes the unclean address (partial, incomplete, or wrong address) as input and returns a cleaned address (a complete address). The addresses are searched and matched against a huge address database, maintained by an external system through an edit-distance approach, i.e., the more complete the address is, the more likely it exists

in the database. *AddressCleaner* cannot be tested if test-data do not have aggregate data dependency among the address fields.

The complexity to generate this characteristic in TDG involves a composition of bi-directional or one-way data dependency wherein the composition factor is the number of columns used in the aggregation. With multiple lookup tables, we can generate this aggregate data dependency. However, two problems can arise: the first is to get a relevant set of all lookup tables that are inter-related, such as lookup tables for zip codes, states, cities etc.; the second and perhaps harder problem is to search for a complete address using these lookups, which can greatly affect generation time and complexity. On the other hand, TDE supports this characteristic with constant time and complexity, and it does so with as little fuss as for one-way dependency. Further, it generates rich quality test-data.

5.*1.2. Functional Row Dependencies*. We came across two types of functional row dependencies in CHARM, namely, row-wise dependency, and nested fields.

Row-wise dependency. A row filters possible sets of values in another row. Row-wise designed tables add the flexibility in database schemes to escape refactoring efforts for possible schematic changes in the table's attributes. For example, we can see this dependency in the history_changes table of the VS database that maintains logs related to changes in its person table. Here, for one person, there exist multiple rows in the table that describe some information about that person, i.e., one row for gender and one row for first name, etc. To implement row-wise dependency, TDG not only requires knowledge of the metadata and characterization rules that exist on a table with row-wise dependency, i.e., history_changes table, but also the metadata and data values of the source table, i.e.,

person table. However, TDE does not need to put any extra effort into incorporating this dependency. Similarly, test-data quality is better in TDE but requires more time for extracting the test-data.

Nested Fields. Multiple fields in a table are tied in a nested way, and values for these fields are calculated in a hierarchical way. An example of nested field exists in the history_changes table for the VS database. Three of its fields {changeaction, changefield, changebefore} are dependent in a nested if-else structure as shown below.

```
If changeaction is A
    changefield is null
else
    ifchangefield == DAD_FIRST
        changebefore = 'Dad Name'
    ifchangefield == CHILD_SSN
        changebefore = 'SSN NUMBER
```

This dependency is a composition of functional row dependency and functional column dependency, wherein values are generated in a hierarchical way while taking information from both rows and columns. This kind of data generation for TDG is extremely hard, as conditional structures are affected by additions, and updates in both rows and columns require a good understanding of database schemes. However, we get default support for the conditional structure in *TDEs* with complexity depending on the query and produces quality test-data.

5.1.3. Degree of Variations. We identified four different kinds of name variations that can provide a rich set of test cases to perform input validation testing.

Table 5-8. Matcher Results with Spelling Variations.

FirstName	Matcher Output
Gaspar	Gaspar Valdez [30.0, 1.5]
Gasper	Gaspar Valdez [26.66, 1.334]

Table 5-9. Matcher Results with Incomplete Names.

First Name	Last Name	Matcher Output
Gas		Gaspar Valdez [20.0, 1.0]
Gas	Val	Gaspar Valdez [10.0, 0.5]

Spelling variations. Sometimes a name can be spelled in many different ways. For example, 'jimmy' as a first name in person table can be spelled as 'jimmy', 'jiimy', 'Jimmy', 'JIMMY', 'JIMI', or 'jimi', etc. Table 5-8 shows the variations in matching confidence results of the matcher with three variations of first names in the person table. Checking spelling variations is an important test case for examining the accuracy of matching algorithms. TDGs cannot generate different spelling variations because it is extremely hard to generate the same sound with different combinations of letters. However, TDEs have default support for this data characteristic due to its extraction ability.

Complete and incomplete names. We also need a combination of complete and incomplete names to test matcher. An incomplete name like 'Jo' can be a substring for many complete names like 'John', 'Johnson', 'Joddy', etc. Another example is when many persons have the same first name, for example, 'Byron', but have different last names, such as 'Byron Douglas', 'Byron John' or 'Byron Robert'. Table 5-9 describes the

results obtained from matcher with incomplete first names and last names. TDG can generate complete names from name domains. Incomplete names can also be generated easily by adding an additional function that can randomly generate substrings or search for names in a lookup table. As mentioned above, TDEs have a default support for this situation. Overall, in the presence of quality name domains, the performance of TDG and TDE is comparable.

Case variations. In actual data, we also find some case variations in first name and last name combinations like 'Abdullah Hassan', 'ABDULLAL HASSAN', 'Abdullah HASSAN'. Identification of similar person records among name variations in different databases can be a very powerful set of test cases for the matcher component. Both TDG and TDE support this. The complexity to generate data for this data characteristic in TDG is almost similar to the complexity for generating complete and incomplete names with an additional function that randomly changes the cases. TDE has default support for this characteristic at the expense of slow data extraction speed.

Dummy names and garbage data. Not only should the test bed contains variations in actual names but also different types of dummy values, say for example, 'DJ', 'Jim321','Girl', 'Boy', etc., names with special characters like 'To'e', 'Wall-J', 'Olivas-Parez' and 'Pulefa'alii' or garbage data such as 'A', 'aaaa', '@','^', etc. A common testing technique to find validation bugs is by processing a rich set of dummy names and garbage data. TDGs typically do not include rules for these kinds of variations because of the additional complexity they would add to the system. However, a possibility is to manually create a collection of a rich set of dummy and garbage data values. TDEs, on the other hand, get such variations for "free" because they exist in the real data.

5.1.4. Derived Data. The CHARM dataset defines two types of derived data: computed field values and restricted values to a computed set.

Computed field values. This field derives its values from two or more other fields or A=f (B1…Bn). For example, an employee's salary is computed from a set of fields, such as gross salary, tax, deductibles, leave taken, etc. Although computations can be done at the application level, sometimes it becomes mandatory to perform such computations at the database level behind some procedure or trigger. A reason for having computed fields in the database schemes is to reduce the query response time. Much important query processing time can be saved by providing already computed values as separate database columns that can be populated or computed from some trigger in a regular pattern. TDE has inherited support for that technique, whereas in TDG, generation is a simple and efficient function that needs source columns or constants for computations. However, small computational problems may arise, for example compute data value from three independent fields such as year, month, and day when one or more fields are null.

Restricted value to computed set. This category consists of values that have partial independent values and partial computed values. In a mathematical expression, this characterization can be represented as A=Randon Selection (f (B1…Bn)). For example, a contact name field can contain a brother name or some relative name, which is some sort of restricted value with respect to the person name. It might be possible that two brothers or relatives have the same last name. TDE has default support for it. However, to generate such dependency through TDG is similarly difficult to characterization rules in that it is not easy to precisely describe the restricted set segments that will be computed and generated.

5.1.5. Grammars Support. Ideally, a test bed can be generated with a variety of grammar rules. In general, they can be categorized as multi-grammars-based data generation with n-substitutable components which is described below.

Multi-grammars-based data generation with n-substitutable components is data generation with multiple segments, each one having its own grammar. Generation rules for each segment grammar can be semantic-based, regular-expression, context free grammar, sequence based generation, as well as others. For example, hospital invoices numbers can be represented with four grammar segments, i.e., UOU-07-387-1445. The first segment identifies the hospital name, the second describes a specialty code like urology, the third the patient account, and the fourth the individual bill number. To generate such an invoice number, we need to have a separate grammar function for each segment, using semantics, lookup tables, computed value sets and sequence numbers, etc. Though data generation for just one grammar is not hard, in a composition it becomes a problem because generation is defined to convey embedded meaning for records. TDGs are capable enough to generate data with multiple grammars; however, each individual grammar can have its own varying complexity and implementation and may not promise the required quality test-data. TDEs on the other hand support this characterization and promise quality data with constant complexity.

5.1.6. Data Frequency Patterns. In a database, we can expect three different kinds of frequency data patterns. Below, each one is evaluated in the context of generators and extractors.

Unique values. Unique values are normally associated with rows for identification purposes. There exist a number of ways to generate unique values, for example, through

sequence numbers, unique random numbers, semantics-related unique values, global unique identifiers (GUID), etc., each having its own varying time and complexity. In some generators, data generation rules are automatically defined if the generator is a primary key column. TDEs get this characteristic free of cost, but extraction itself can take a reasonable amount of time.

Single-column data value frequency distribution. This characteristic determines the frequency distribution of data values or a pattern of different values in a single column. For example, the first_name column in the person table of the CORE database can have 30% female names, 50% male names, and the remaining 20% as null values. In CHARM, data frequency patterns can raise bugs related to matching. For example, to find the number of twins, matcher identifies multiple-birthflags for possible twin matches. A frequency-based distribution pattern for twins that is close to reality can identify potential bugs in the matcher. TDG can incorporate this characteristic by defining distribution classes and their frequencies in generation rules. However, defining distribution classes as well as their frequencies can be hard when these classes are too numerous or the designer did not have enough knowledge about the real data distribution patterns. TDEs, on the other hand, have default support for this characteristic.

Multi-column data value frequency distribution. Sometimes two or more columns share a common frequency-based pattern of data values. For example, patient_id and usiis_patid columns in forecast_vw table of the USIIS database (Table 5-10) have a multi-column data value frequency distribution. These columns have different values, but the percentage of distribution is the same. For example, if patient_id is repeated ten times, the corresponding values for the usiis_patid follow the same sequence of occurrences. TDGs

Table 5-10.Mult-column Data Value Frequency Distribution: patient_id and usiis_patidColumns in Forecast_vw Table.

Patient_id	Usiis_patid
18839506	15707944
18839506	15707944
425102.1	15729992
425102.1	15729992

can simulate this data characteristic using a combination of two data characteristics, i.e., bi-directional data dependency and single-column data value frequency distribution. TDE do not need to do anything to achieve this characteristic.

5.1.7. Complex Data Structures. We identified one kind of complex data structure in CHARM, namely, repeating groups.

Repeating Groups. These are repetitions of certain sets of data values in a table due to improper normalization. In CHARM, we have different independent databases. Sometimes database designers do not follow proper normalization rules while designing their schemas, or domain evolutions can introduce repeating groups in some tables. We found such an example in a table named charm_nbs_mailer_results in the NBS database. This table does not fulfill the criteria of 1st-normal form. As per the standard definition, a table is in its 1st- normal form if it does not contain repeating groups of data. We identified three levels of repeating groups in the above table. Firstly, baby demographic information is the outermost repeating group, mother information is a second level repeating group, and sample tests information is the innermost repeating group. Sometimes, database designers intentionally introduce the repeating groups in the data for achieving better speed, because if records are retrieved and stored in a set format to a

single table, speed can be enhanced proportionally. Generating repeating groups in a TDG is extremely hard. Classifying the repeating groups requires a deep knowledge of the data as well as the repetition patterns of the repeating group. The problem gets even more complicated when one or more repeating groups have an aggregate-level data dependency. On the other hand, TDEs always can generate the repeating groups in a constant time with minimal complexity factors.

5.1.8. Inter-table Dependency. We identified four different kinds of inter-table dependencies. These are highlighted below.

Explicit referential integrity constraint. Also known as *primary-foreign key* relationship, this is considered an important data characteristic for testing database applications, as many times we need to test the applications for cascading inserts, deletes, and updates to ensure that data values are synchronized among tables. In the literature survey of test-data generators, we found [18 ,31] both support this characteristic. Overall, defining a characterization scheme and maintaining is a little bit tricky. But once it is defined, its performance in generating the test-data is reasonable. On the other hand, we are not worried about defining and maintaing this relationship in aTDE, but we do need to extract the test bed every time there is a change in the database scheme.

Indirect referential integrity constraint.This is an implied value-based inter-table reference without a {primary– foreign key} relationship.This constraint is a kind of *hidden data dependency*. Hidden dependencies can sometimes be a result of missing an explicit referential integrity constraint or incomplate database refactoring when data in tables already exist. For example, in the COREdatabase, *phone_number* and *address tables* have an implied reference with the *person* table because a phone number value of a

person hints at his address. A *phone number* value *'435-797-3786'* can suggest that the owner has a *Utah-based address* entry in the address table. If table A has to extract mutually exclusive data from three other tables, say B, C, and D, we have two possibilities:1) either introduce three foreign keys in table A; or 2) use indirect referential integrity constraint in table A, where just one column can contain reference column values from any of three tables. We cannot enforce an explicit referential integrity constraint here as a foreign key because it cannot be composed from a composition of reference keys. It is difficult to implement this characterisitc in TDGs because identifying this sort of dependency needs a sound knowledge of database schema and data. However, we get this characterization for free in TDE.

Row-count constraint. This constraint limits the number of records through row-count entry. In other words, the count column value in each record of the parent table provides summary information (or apply restrictions) about its children. The purpose of row-count constraint dependency is to reduce the query response time and promote simplicity for data extraction in a parent child relationship. Defining this generation technique is easier than defining characterizations for explicit referential integrity or indirect referential integrity mentioned above, so a linking effort is not needed. The number of child records can be calculated by quickly scanning the dependent tables. The resultant test-data quality should also be satisfactory. A TDE just requires a fresh data re-extraction every time without defining complex generation rules, though slow speed is an issue.

Inter-table patterns. This pattern shows the distribution of values in a parent-child relation. For example, a vaccine table in the EDHI database contains vaccine shots for

children stored in a person table. On average, a child can have up to five different shots, but there are some anomalies wherein a child has more than twenty vaccine shots. Inter-table patterns are important for testing the data-centric applications. Data generation rules, time, complexity, and quality of test-data for TDG and TDE would be similar to the generation techniques mentioned in explicit and implied referential integritydata characteristics**.**

5.2Comparisons in the Context of Federated Databases

This section compares two test-data creation approaches against inter-database dependencies or correlations among federated databases. (See Table 5-2 for a comparison summary).

So far, we have discussed test bed creation approaches and database testing issues in the context of standalone databases. But database testing does not end here. In fact these days, organizations are stressing building applications on integrated databases (by creating mappings among different standalone databases). When we talk about data integration for different databases, we are in fact talking about resolving the issues related to matching, linking, merging, and resolution of records among these data sources.

The challenge for test bed creation is to provide a testing environment wherein developers can have opportunity to test their applications for the challenges mentioned above. These challenges can be met when we have synchronized datasets across data sources. Practically speaking, it is easier to generate unsynchronized datasets in multiple databases, but it is harder to generate synchronized data sets among different data sources, though it is not impossible.

In TDG, incorporating these characterizations into databases is very hard, especially given that we already have the bulk of the characterizations related to other database testing areas mentioned in this study. So far in our survey of the research literature, we came across no data generation scheme that claims to generate the synchronized datasets for multiple databases.

However, TDEs especially have the flexibility to extract the synchronized datasets from federated databases. In the CHARM environment, iSTDE is able to extract the synchronized datasets among seven data sources. These data sources are synchronized with the use of a set of Mappers' databases that creates linking information among these data sources. Additionally, data sets from these data sources are also automatically transformed to PostgreSQL database semantics so that integrated databases can be simulated in a local development environment.

Though TDE gives us a solution to create synchronized test beds for integrated databases, we cannot ignore the time factor and complexity drawbacks. Adding a new data source to get a synchronized dataset requires adding a complexity factor say '*d*' for inter-database correlation, as well as another complexity factor for extracting data just related to its standalone extraction. Once we have a test bed with synchronized datasets, developers can use them for testing their applications for integrated testing problems, such as matching, merging, linking, and resolutions. SyncEngine, a CHARM application, identifies new updates in PPs and adds those references to the CORE and ID_Mappers (internal linking databases). An important test case for SyncEngine is to check the consistency of person records in this federated database environment. For example, a person record that exists in CORE must also have its reference in Mappers databases and

similarly in some other internal PP databases. We tested SyncEngine with iSTDE generated test-data and found good coverage for test cases dealing with consistency and other inter-database dependencies.

5.3Comparisons Related to Target Schemes

This section provides a comparison of handling of conversion of different types of target schemes, such as database objects, data types, and database managers. Table 5-3 shows comparison of the two approaches.

5.3.1. Handling of Views. Database views do not contain the data. They are merely snapshots of database tables. Normally, both TDE and TDG (that rely on metadata schemes) support automatic handling of views. Thus, views can be automatically extracted along with database tables through metadata generation process. However, in some situations, pointing views to tables is difficult, particularly when the actual table is not included in the test bed generation scheme. Some TDEs, especially iSTDE, deal with this challenge by implementing a view as table in the test bed, thus eliminating the need for a view source table. However, it is difficult for a TDG to deal with this challenge unless it defines the generation based characterization rules for view source table. In the CHARM environment, forecast_vw exists as a view in the USIIS database. In a testing environment, we need the view but not its source table. Thus iSTDE converts a view to table metadata and populates it through view-based queries from source tables.

5.3.2. Handling of Database Objects Other Than Tables and Views. Providing automatic syntactic-conversion support for handling database objects, such as procedures, functions, triggers and sequences, is hard for both TDGs and TDEs as these database objects are somehow dependent on the underlying semantics languages of their database

managers. It is extremely hard to define a subroutine that can automatically convert procedural logic defined in one database manager to another one. Thus, both TDGs and TDEs require a manual intervention to redefine the procedural languages transformation.

5.3.3. Handling of Variations in Database Managers. In an integrated database environment, we can have data sources with a variety of database managers, i.e., Oracle, SQL Server, PostgreSQL, Pervasive, and many more. This variety may raise a certain degree of syntactic heterogeneity, such as differences in data types, internal data representations, and certain queries support. While creating an integrated-data environment for testing purposes, our goal is to simulate all these data sources in a homogeneous data-environment. Conversion of this syntactic heterogeneity to a syntactic homogeneity is a big challenge.

For both TDGs and TDEs, even solving data type differences is not an easy task. In our literature survey, we came across hardly any techniques for TDGs that can sufficiently overcome handling of various data types in database managers. However, in the case of TDEs, we found that at least in iSTDE, this data type conversion challenge of rule-based transformation is being dealt with to some extent by maintaining a repository of transformation rules.

5.4Comparisons Related to Set-up, Use, and Performance

Non-functional based comparison of the two approaches is presented in this section. Table 5-4 summarizes this comparison.

5.4.1. Need for Access to Existing Databases. Normally TDGs depend on a database for generating metadata schema only. Once they have schema, only field characterization rules would be enough to generate the test bed. However, TDEs heavily

depend on database access for both metadata and data generation. This continuous database access requirement makes TDE susceptible to possible data interruptions due to network or database server problems, whereas TDGs are very resilient to these errors as well as add a proportionally smaller load on the database servers.

Additionally, there are situations wherein data sources do not provide metadata access due to their internal policies. For example, in the CHARM environment, within the*Vital Statistics* data source, we have access to the data but not its metadata. In this situation, manual intervention is needed to design the database schema for TDGs and TDEs. However, some TDEs such as iSTDE take a smart approach and can automatically describe the tables' metadata reversely from the extracted data.

5.4.2. Ease of Deployment. TDGs are comparatively easier to deploy than TDEs. Once the data tester is installed on the developer's machine, it is supposed to automatically create the test-data and populate it in the test database. However, in the case of TDEs, the task is not so trivial, especially in a data-sensitive environment, wherein it is also a requirement to restrict real data exposure. Additionally, the developer needs to specify the target and source database connection information, the volume of data to be extracted, and other instructions related to data migration to unsecure environment.

5.4.3. Meeting Users' Expectations for Speed. The amount of time it takes to generate the test bed is another important factor in the choice of test-data creation approach. Normally, it can be assumed that TDG takes significantly less time to generate the data as compared to TDE. However, a TDG's speed is affected by the complexity of the characterization rules. For example, a TDG can quickly generate a set of unique

values as compared to another TDG that is generating unique values with two additional characterization rules: semantic-number generation and partitioned class-based generation scheme. On the other hand, TDE data extraction time is not dependent on characterization rules, but it is dependent on the volume of data, federated data sources, load on the data managers, query joins, and network speed. In CHARM, iSTDE takes almost two days to extract the data from seven data sources for just two weeks of data. However, this performance can be enhanced by concurrent extraction from multiple data sources and data loading strategies (i.e., post-enabling of data constraints after loading and batch-loading strategies).

5.4.4. Defining Data Characterization. In almost every TDG, there is a need to specify the characterization for each field before data generation. The characterization defines generation rules such as types of data, value templates, and constraints. These TDGs thus lose their powers if the table is very large, the underlying database scheme is very complex, or data-generation rules are very complex. For example, the birthmaster table in the VS database has approximately four hundred columns. In such a case, defining the data characterization for every column along with constraints can seriously affect the motivation for using a TDG. A TDE on the other hand, does not need to define any sort of characterization rules for tables' metadata, nor can a complex set of constraints affect the speed of extraction.

5.5Comparisons in Context of Database Refactoring

Database refactoring is a continually changing process. From the testing point of view, this dynamic nature of refactoring can have a deep effect on the choice of test bed creation approaches, e.g., TDG and TDE. In this section, we discuss a few of the

databases'refactoring processes and their effects in considering the selection of test bed creation approaches (Table 5-5 provides a comparison summary).

5.5.1. Refactoring of Database Objects That Contain Data

Addition or deletion of key columns. A TDE needs to re-extract data if key columns are added or deleted. A TDG on the other hand needs to redefine the database scheme first and then re-generate the data for whole referenced data entities.

Addition or deletion of non-key columns. For addition, partial re-generation for non-key columns would work in the case of a TDG because existing data in other columns would remain intact. However, in the case of deletion, even re-generation is not required, just schematic change (non-key columns deleted) would be enough. However, a TDE only scheme would require updating for both addition and deletion, and re-extraction would only be needed if data were present. Overall, TDG and TDE performance in the context of complexity and generation time is comparable for refactoring of non-key columns.

Independent tables. Independent tables can be either mirror tables, log tables, lookup tables, or parent tables that have not yet been referenced. Both TDE and TDG are adaptable to these tables without any severe complexity.

Dependent tables. In child-tables or tables referencing key-columns from other tables, a TDE needs to re-extract the data just for the dependent tables. For a TDG, we need to define the target scheme for the dependent tables and re-generate the data for just those parts of the schemes involving the dependent tables. Overall, for this type of refactoring, there is not much of a difference in terms of cost, but in terms of time, a TDG would be preferable to TDE, even though the TDG process is more complex.

Column replacement. In the context of complexity, a TDE is better as one does not need to take care of replacement issues and their trickle effect on other tables and columns. Re-extraction alone will solve the problem. However, in a TDG, simple replacement might not be an issue of concern. But complexity for redefining database schemes can significantly increase if it is a key column, given that it can affect many other columns and tables. Once characterization is defined, a TDG will supersede TDE in terms of the time factor.

Split operation. During database refactoring, we come across three types of split operations.

- Splitting tables : Address tables can be split into address and state tables;
- Splitting columns: Customer name columns can be split into first name, last name;
- Splitting large objects (LOB) to table: Mailing address columns in customer tables can be split as a separate table address (id, street, city, state, zip).

Causes of these splits can be to enhance design, performance, sharing, or privacy related issues. As TDGs are not dependent on the data sources for generation, data generation is very fast once schemes and characterization rules are defined. TDEs, on the other hand, have to re-extract the chunks of data sets, which can consume a significant amount of time. However, this process does not require redefining the database schemes and characterization rules. Merging-based refactoring problems also have a similar type of comparison for extractors and generators.

Migration and reordering. TDGs are adaptable to reordering of columns without undergoing any major changes in database schemes. TDEs, on the other hand, need to re-

extract the data (which involves a time factor) from the perspective of any reordering of database schemes. Migration of database columns, however, may require major changes in database schemes for TDGs, as these migrated columns can affect relationships among tables. Although TDEs will automatically cope with database column migration problems, extraction of data can take a considerable amount of time.

Renaming. Renaming a column, view, or table does not add any extra complexity to a TDG in terms or time, complexity, or cost. Ideally, data regeneration should not be required from TDGs when columns of data sources are renamed. TDEs however are more sensitive to the column renaming as extraction can be effected. However, existing test beds of TDEs can be made invulnerable to such changes if the TDEs are well designed with an independent layer between the metadata and data.

5.5.2. Refactoring of DatabaseObjects That Do Not Contain Data.

Triggers. A trigger was added to a column that calculates the data values for a column. Both TDGs and TDEs are adaptable to this change without the need for any generation or extraction.

Cascading deletion. Deleting a record in a parent table will also delete the dependent records from child tables. This refactoring does not affect either TDGsorTDEs, as they would remain adaptable.

Constraints. Constraints can be of three main types, namely, primary-key constraints, referential integrity constraints, and business-rule constraints. If we are adding constraints to an existing database scheme in a TDG, we may need to clean the existing data and re-generate it, given that the addition of constraints can conflict with the existing data. However, such may not be the case with TDEs, as the addition of new

constraints would be invalidated by the data sources if they have conflicts with the existing data. Thus, TDE data sources will always ensure compatibility with new constraints.

Encapsulate table with a view. Views are considered snapshots of data; they do not contain any data nor affect the data in existing tables. Thus, introducing a view on a table will not require re-extraction for a TDE. A TDG can support these characteristics through manual intervention.

Indexes. For similar reasons described for views, data generation from both TDGs and TDEs will not be affected by introducing indices on the tables.

5.6Comparisons in Context of Testing Techniques

In this section, we discuss the different testing methodologies in the context of the two competing test-data creation approaches (Table 5-6).

5.6.1. Unit Testing versus Regression Testing. Neither TDGs nor TDEs proved very useful for unit testing, nor did they prove useful for regressing testing. A developer needs small data sets of known values with specified inputs and outputs that can be used to test methods at the class level. Conversely, both TDEs and TDGs generate large datasets that are comparatively difficult to test at the unit level. Also these datasets may not contain the specific set of values that can be used as expected inputs. A widely acceptable method for unit testing and regression testing is manual data generation. This way, unit test level values can be easily injected. Similarly, in regression testing, we need to clean the database state and populate the fresh database state after every test. This situation creates problems with the data generated through TDGs and TDEs. However, TDGs and TDEs can be useful in certain situations wherein testing of a unit level

functionality needs hundreds of rows to check the comparison result. For example, in matcher sometimes we may need many records as inputs to get a match.

5.6.2. Functional Testing of Standalone Modules. One challenge in functional testing is to achieve maximum feature coverage. In order to achieve good feature coverage, the data should be rich enough in terms of integration, business rules, good, and bad data. A TDE's performance is much better than a TDG's for functional testing. It is very difficult to simulate or generate all data characteristics in TDG. Doing so requires a good understanding of the application domain and database schemes along with other data characteristics. In the CHARM testing environment, developers use the test beds generated by iSTDE (a type of TDE) for functional testing of their modules, and this proves very useful. We tried to use some generators for functional testing, but the results were not promising as generating functional characteristics through generators is hard.

5.6.3. Integration Testing. Very few familiar types of integration testing techniques are big bang, top-down, bottom-up, and sandwich-based integration testing. Normally, developers first develop and test their standalone modules in an individual testing environment. After individual testing, these modules are integrated. Integration can be either at the application level or the data-level. But in both cases, the data somehow needs to be integrated. In the CHARM environment, many components use seven different data-sources. When components are integrated, we need correlated datasets from all seven data sources being used in CHARM. iSTDE proved very useful as a data input source in this environment for all four types of integration testing techniques. One additional benefit of iSTDE is that its complexity does not affect the addition of new data sources for inter-tables or inter-database relations. Developers working in the

CHARM environment have a rich repository of test beds extracted from iSTDE that provide a sufficient level of integration coverage with rich datasets. On the other hand, a TDG does not prove a useful resource for integration testing. First, it is very hard to generate correlated datasets for different data sources, and second, doing so does not promise a quality synchronized dataset. Additionally, complexity increases multiple times for a TDG when additional data sources are integrated.

5.6.4. Performance Testing versus Stress Testing. Performance testing for databases is used to evaluate whether existing datasets meet certain thresholds or benchmarks for response time, etc. This testing technique requires large datasets and concurrent users. Normally, we may not be very concerned about the semantics-based data characteristics of the database for performance testing, but we do need a large number of records. A TDG is a better choice than a TDE for performance testing, as we can quickly generate large number of data rows very, though in some cases defining database schema might be a non-trivial task. A TDE, on the other hand, can be a competitive choice as far as it provides large number of records, even though it takes a reasonable amount of time for extracting large data sets.

Stress testing is slightly different from performance testing. Its purpose is to analyze the data source behaviors or find bugs by running the data sources in unpredictable environments (measuring performance with abrupt changes in data loads). Stress testing is considered a kind of negative testing whereas performance testing is considered a positive testing. A key requirement for stress testing is a number of datasets that can be added to increase load on database managers. Here again, a TDG would be a preferred data generation technique, as it more easily and quickly generates a number of

datasets as compared to a TDE. However, in the case of a rich repository that has a reasonable number of test beds, the performance of both a TDG and a TDE would be comparable.

5.6.5. Data Validation Testing. Input validation testing is a common testing technique. Software applications are tested for illegal and wild characters, mismatched data types, field length validations, and many other checks. Input for data validation testing can either be from the user end (top-down approach) or database end (bottom-up approach). From a database testing point of view, we are concerned with bottom-up data validation testing. Here, the challenge is to have a rich dataset that contains sufficient data validation testing characteristics (some of them are mentioned above). A TDE is preferable to a TDG because it inherits all the real data characteristics from actual data sources, thus providing the perfect environment for data validation testing. In a TDG, however, we need to inject all the real data characteristics, which is a difficult task.

5.7Comparisons Related to Social and Time Factors

This section deals with changes in data and data schemes related to social and time factors, as well as how these changes can have an impact on the choice of test-data creation approaches. Table 5-7 provides a summary of the comparison.

5.7.1. Semantics changes due to domain evolution. Mergers and expansions are part of everyday activities for any organization. As a result, database schemes (syntactic) and data definitions (semantics) evolve over time. Changes in database schemes come under the category of database refactoring (discussed above). Dealing with semantic changes can be a challenge for both TDEs and TDGs. For example, in a hospital inventory, we can expect an alternate code scheme for surgery equipment. Initially, say

‘C01’ code was used to represent equipment ‘A’; later it was assigned code ‘C011’. This change does not affect the database scheme, but it does affect the information context. This change may not affect the TDG data scheme. However, it can affect characterization rules for data generation. Both a TDG and TDE should take care of these domain evaluations. As per experience, we can say that a TDE would be a better performer for handling these domain evolution changes because actual data sources would always be harmonious with domain evolution. A TDG, on the other hand, needs a redefinition of characterization rules because of domain evolutions. Redefining these rules require a good understanding of domain evolution changes as well as database schemes.

5.7.2. Social factors that affect the nature of the data. Many times, data definitions are somehow dependent on social or cultural aspects. For example, patient names in a hospital located in the United States have a higher proportion of English names whereas a hospital in India has more Hindi names. Apparently, this is not a big issue as far as testing is concerned, but when we talk about integration testing it can be a problem. Consider for example a situation wherein we need to scan the names or clean the names via matching from some independent data source that have real addresses or names. A TDE is a better choice than a TDG because its data source is real production data that would always be consistent with social or cultural aspects. However, in the case of a TDG, we might not have access to domains that are consistent with the cultural aspects of the data. A possible solution would be to use the TDE to build the domains that can be used by TDG.

5.7.3. Data generation that does not expose personal privacy. Organizations often implement security policies to cope with external data threats, but they often neglect

internal security loopholes. One critical internal data threat is the access software development-related people have to real data. Software developers and testers need access to the real data, which in turn can have sensitive and private information about persons or organizations. Ideally, this sort of information should not be accessible to these people, but for development purposes they do need this data. From the perspective of the two data generation approaches, TDG would be a better choice as it does not provide any clue about the sensitive nature of the data. The TDE approach is certainly a risk to internal data security; however, with some data scrambling techniques, we can overcome this threat. Some TDEs such as iSTDE and IBM Optim provide a secure way to access and share the sensitive real data from integrated data sources both inside and outside the organization.

CHAPTER 6

SUMMARY, FUTURE WORK, AND CONTRIBUTIONS

6.1Summary

In Chapter 2, we describe an innovative approach for developing a tool that can create test-data for integrated databases and other applications containing sensitive information. Semantics-based Test Data Extraction for Integrated Systems (iSTDE) is a tool that generates a testing bed for a system of heterogeneous databases, specifically, for CHARM.

As described, the execution of iSTDE consisted of six steps. In the first step, the user specified selection criteria that includes a description of the target environment, the data sources from which to extract the data, and query parameters. The second step is the creation of domains in the secure environment for all the independent databases, including metadata transformation to Postgres format. In the third step, we extract the correlated data sets from real federated databases and transfer them in data domains created in the previous step. The fourth step relates to reshuffling or mangling the real data in order to de-identify the sensitive demographic information. In the fifth step, the program exports the mangled data from the secure environment to the specified unsecure testing environment. The sixth and final step removes all linking information created and used during program execution.

This study also provides an in-depth comparison of two test-data creation approaches: test-data generation (TDG) and test-data extraction (TDE), respectively. We first defined a comparison method that included seven different areas of comparisons, selection of representative tool from the two approaches, setting up a test environment

and formulation of theoretical conclusions for comparing TDE and TDG. We next validated those conclusions from anecdotal evidences in our CHARM project. In general, our comparison method concluded that TDEs had the potential to create more realistic test-data and were specifically suitable for testing federated database applications. However, they might compromise data confidentiality and require more system resources and time for test-data creation than TDGs. On the other hand, TDGs might be an apt choice for person-centric applications that do not have many characterization rules. TDGs would work well to quickly generate datasets but fall well short of ensuring adequate test coverage for complex integrated system.

6.2Contributions

Our research delivers the following contributions in the area of testing software applications for integrated databases.

6.2.1 An Approach That Provides Test Data for Integrated Systems

CHARM components deal with many independent databases running on different machines, all of which share different aspects of patient demographics. Testing of these integrated systems was a challenge. Our approach of providing test-data through iSTDE successfully delivers a test bed that preserves the integrity of these databases.

6.2.2A New Approach to Provide Real Test-data, Without Compromising Sensitive Information

Real data in the CHARM environment contains sensitive demographic information about patients. CHARM developers need to use this data during the application development phase. However at the same time, it is also important that

patients' sensitive information not be made public. The challenge was to devise a way to give developers access to the datasets containing plenty of testing characteristics without exposing sensitive data. Our iSTDE approach provides test-data that exhibits real data characteristics important for testing, and it also does not compromise sensitive information.

6.2.3ANovel Comparison Method for Comparing Test-data

In Chapter 4, we provide a theoretical comparison of the two test-data creation approaches. We did not find a comparison scheme in the literature survey that can guide the testers in creating an appropriate data set for testing their database applications. Our comparison method involved seven different comparison areas (Section 4.3).

6.2.4Theoretical Conclusions with Anecdotal Evidence Explaining the Effectiveness of the Two Approaches

From the comparison schemes, we identified important testing characteristics for database applications and provided theoretical conclusions about the effectiveness of two data creation approaches. We also validated these conclusions with anecdotal evidence from our CHARM project.

6.2.5In-depth Analysis of Available TDG and TDE Tools and Techniques

This thesis explores both research-oriented and commercially-supported tools and techniques for TDGs and TDEs and provides an in-depth analysis and comparison of these tools and techniques (Section 4.2). Software developers and testers can gain a good understanding, guidelines, and exposure to different ways of creating test-data sets.

6.2.6New Software Testing Process for Data-Centric Integrated Systems

See Appendix A for more details.

6.3 Future Extensions of iSTDE as a Tool

The current version of iSTDE is a desktop-based application that runs in a confidential environment. It was primarily developed to test the CHARM applications. However, we have now identified some future enhancements in our iSTDE program that can be of valuable not only for our project but also for various future research perspectives.

1. We can use generators on top of extractors to extend the quality of mangling. As discussed before, our PII domains are populated from the CHARM domain. The range of values in these PII domains can be further enriched by using a generator that can use some mutation algorithm to create many more variations of these values, thus ultimately contributing to better quality of data anonymization.
2. Speed of iSTDE can be optimized in order to reduce the overall time it takes to create a test-data set. These improvements would include optimization of SQL queries for extraction and data mangling components. From our experience in iSTDE, more than two-thirds of the test bed creation time is spent on data extraction. By employing a right balance of concurrency for data extraction, we can dramatically reduce the overall running time.
3. iSTDE is currently a desktop-based application that runs in a confidential environment. Only external authorized actors are allowed to run this program.

With some modifications, it can be converted into a web-service-based application. This would make iSTDE accessible to a large group of users.

4. The current design of iSTDE is not extensible, it is tightly coupled with the source databases, and addition of new data sources need extensive programming that can increase code duplication and derails performance. With some refactoring and trimming of the overall application architecture, this undesirable coupling can be reduced and design can be further generalized.

6. 4 Research Directions for Comparison of TDGs and TDEs Approaches

So far, our comparison of two test-data creation approaches is limited to just the relational database domain. However, integrated systems include many other types of data sources, such as XML files, csv files, network, hierarchical databases, OLTP, and OLAP. Reassessing the comparison scheme for generators and extractors in the context of these other non-traditional data sources can be an interesting research direction.

In this study, we provide a comparative analysis of TDGs and TDEs. Dealing with data confidentiality is a big challenge for TDE. For this purpose, there exist a number of data scrambling and masking techniques that can de-identify sensitive data. These techniques can provide varying results in terms of their complexity level, the nature of the data, and target applications. Appropriate classifications and comparison of these data scrambling or masking techniques can provide specific conclusive arguments for effective masking techniques by application types. Similarly, categorizing different applications and data types and then identifying their respective appropriate masking techniques is another potential research area.

In software testing, we can find code review checklists for many programming languages. These checklists assist in finding and fixing overlooked mistakes in peer code reviews. A similar kind of peer-review practice can be initiated for database applications. In this study, we identified a similar kind of checklist that includes eight areas for testing database applications. These can help database applications peer reviews. Additionally, different data generators can use this checklist while designing their respective features and error coverage.

There exists a research opportunity for identifying the efficient use of creating and re-using the test beds generated in the context of software testing process. Database application test suites can use one or more test beds. Reusing these test beds triggers the need to manage their generation and usage in an organized way. A mature and well accepted software testing process that can define a protocol for effective utilization of a test bed can dramatically affect the time, cost, and quality parameters of software productivity.

REFERENCES

[1] Chays, D., Shahid, J., and Frankl, P.G. Query-based test generation for database applications. In *Proceedings of 1st Workshop on Testing Database Systems*, 2008.

[2] Oracle. Mapping SQL data types to Java. http://download.oracle.com/javase/1.3/docs/guide/jdbc/getstart/mapping.html. Last accessed on February 09, 2011.

[3] GenerateData.com. A tool for generating test data. www.generatedata.com. Last accessed on February 09, 2011.

[4] SqlEdit. DTM Database Tools. www.sqledit.com. Last accessed on February 09, 2011.

[5] forSQL. forSQL Data Generator: A tool for automatics generating test data for developers. www.forsql.com. Last accessed on February 09, 2011.

[6] Automation Anywhere, Inc. Automated Test Generator: A tool for generating meaningful randomized data for QA testing, load testing. www.tethyssolutions.com/T10.htm, Last accessed on February 09, 2011.

[7] TNS Software, Inc. A tool for generating meaningful randomized data for QA testing, load testing. www.tns-soft.com.Last accessed on February 09, 2011.

[8] Clyde, S. CHARM Architectural Overview. White Paper, http://charm.health.utah.gov, September 2002.

[9] Datanamic. Generating the test data from a variety of sources, including the database tables. www.datanamic.com. Last accessed on February 09, 2011.

[10] Turbo Computer Systems. TurboData: Generate the test data from a real database. www.turbodata.ca. Last accessed on February 09, 2011.

[11] SQL Manager.net. EMS Data Generator for PostgreSQL: Generate the test data by a variety of sources. www.sqlmanager.net/en/products/postgresql/datagenerator. Last accessed on February 09, 2011.

[12] Dymek, D. and Kotulski, L. Using UML (VR) for supporting the automated test data generation. In *Computational Intelligence and Security Workshops*, 2007.

[13] Maddy, N.R. *DBTESTGEN – Database Test Data Generator for CHARM*. Master's Thesis, Utah State University, 2006.

[14] Raza, A. and Clyde, S. Semantic-based test data extraction for integrated systems (iSTDE). In *Proceedings of International Conference on Health Informatics*, 2010.

[15] Chen, T.Y., Poon, P.-L., and Tse, T. H. A choice relation framework for supporting category partition test case generation.Technical Report TR-2003-01, Department of Computer Science and Information Systems, Hong Kong University, 2003.

[16] Zhang, J., Xu, C., and Cheung, S.C. Automatic generation of database instances for white-box testing. In *Proceedings of 25th Annual International Computer Software and Applications Conference*, 2001.

[17] Davies, R.A., Beynon, R.J.A., and Jones, B.F. Automating the testing of databases. In Proceedings of the 1st *International Workshop on Automated Program Analysis, Testing and Verification,* 2000.

[18] Tsai, W.T., Volovik, D., and Keefe, T.F. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. on Software Engineering 16*, 3 (March 1990), 316– 324.

[19] Chan, M.Y. and Cheung, S.C.Testing database applications with SQL semantics. In *Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications*, March 1999.

[20] Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., and Horowitz, B.M. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[21] Don Slutz. Massive stochastic testing of SQL. In *Proceedings of the 24th International Conference on Very Large Databases*, 1998.

[22] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P.J. Quickly generating billion record synthetic databases. *ACM Special Interest Group on Management of Data 23*, 2 (June 1994), 243–252.

[23] McCallister, E., Grance, T., and Scarfone, K. Guide to Protecting the Confidentiality of Personally Identifiable Information (PII). Special Publication 800-122, National Institute of Standards and Technology, U.S. Department of Commerce, April, 2010.

[24] Narayanan A., Shmatikov V., Myths and Fallacies of "Personally Identifiable Information", Communications of the ACM June 2010, Vol. 53, No.6

[25] *Health Insurance Portability and Accountability Act of 1996*. House Report 104-736.

[26] FIPS PUB 199. Standards for security categorization of federal information and Information Systems. Federal Information and Processing Standards, Feb. 2004.

[27] IBM. Optim TM Data Privacy Solution. www-01.ibm.com/software/data/optim/core/data-privacy-solution/ Last accessed on February 09, 2011.

[28] T. Trimpe, The Scientific Method, https://umdrive.memphis.edu/ggholson/public/scimethodwkst.pdf.,Last accessed on February 09, 2011.

[29] Clyde, S. and Salkowitz, S. *The Unique Records Portfolio*. .Public Health Informatics Institute, 2006.

[30] Chays, D. *Test Data Generation for Relational Database Applications*. Doctoral Dissertation, Polytechnic University 2004.

[31] Haller, K. White-box testing for database-driven applications: A requirements analysis. In *Proceedings of the 2nd International Workshop on Testing Datbase Systems*, 2009.

[32] de la Riva, C., Suarez-Cabal, M., and Tuya, J. Constraint-based test database generation for SQL queries. In *Proceedings of 5th International Workshop on Automation of Software Testing*, 2010.

[33] Haftman, F. and Kossmann, D. A framework for efficient regression tests on database applications. *International Journal on Very Large Data Bases 16*,1 (January 2007), 145-164.

[34] Ramakrishnan R., Gehrje J., Database Management Systems, McGraw-Hill Science; 3rd Edition ISBN 978-0072465631

[35] M. Dixon, J.Kohoutkova, Managing Heterogeneity in Inter-Operating Medical Information Systems, *10th ERCIM Database Research Group Workshop on Heterogeneous Information Management*

APPENDICES

Appendix A

iSTDE-CENTRICSOFTARE TESTING PROCESS FOR INTEGRATED SYSTEMS (AN IDEA FOR A FUTURE PUBLICATION)

We have three environments in our CHARM project: development environment, staging environment, and production environment. Actual databases run on the production environment. The staging environment is used by developers for final integration testing of their CHARM components before deploying them on the production environment. The development environment is used for implementation and testing individual components. We are more concerned with the development environment, wherein iSTDE has initiated a software testing process.

1. Description of Development Environment

A development environment consists of a large collection of PostgreSQL databases. Every CHARM developer has his/her own set of databases in this environment. This structure makes it easier for developers to test their individual components without affecting others'. Because developers need a separate set of database characteristics to test their CHARM components, it is possible that these developers have inculcated some special states in their databases that give them the best coverage according to their components. These states might be achieved by importing some data from one or more unique iSTDE generated test beds. Thus, the development environment provides a proper prototypical simulation of real production databases.

2. iSTDE-based Software Testing Process for generating and using test-data

1. Execution of the test-data extractor (iSTDE) in a secure environment

2. Migration of test databases and anonymized test-data to the test repository
3. Loading and reuse of one or more test databases from test repositories to the development environment

Whenever any developer or a set of developers need test-data to populate a given development environment, the developer logs in to the secure environment to run iSTDE. Only authorized users are allowed to enter this environment. The developer then specifies the parameters necessary for iSTDE execution. Some of these parameters might be a start-end date range and the databases from which s/he wants to generate the test bed. The execution time of the iSTDE tool is very long. It takes hours or days to get the test-data for few weeks of work. This slow time is mainly due to extracting the integrated set of data across all the databases that are running in different locations. After starting the iSTDE execution process, the developer can logoff the machine; the iSTDE process will then run as an independent operating system process. Once the execution of iSTDE is finished, the system holds data files that contain only the mangled data. The tool copies this mangled data in a specific repository maintained just for extracted data for iSTDE-based test beds. The next step is generating properly named batch scripts that will run the program in this repository. This repository is accessible to all the CHARM developers as it contains only the mangled data. The developer executes that script, which first asks for the access credentials. It then creates fresh instances of the databases along with data and related database objects to the developer-specific database schemas on the development environment. Once the developers think that their databases are in a dirty state, they can re-run these databases scripts to again populate their databases with the original data-states.

The test bed repository contains a collection of different test beds generated by iSTDE. One developer can use a test bed from a repository created by other developers, so the developers do not need to run the time consuming executions of iSTDE again and again. This reuse of test beds motivates developers to confidently use the test beds. With more experience, the repository is becoming richer and richer in terms of test beds and data characteristics. Many times, it is not sufficient for the developers to just use one test bed for testing their components. They need data from multiple test beds in their personal databases to test full coverage of their programs. Loading the data from a test bed repository to developer databases on the development environment is a very speedy process, because data is already in a mangled form and just need to be loaded into one database server.

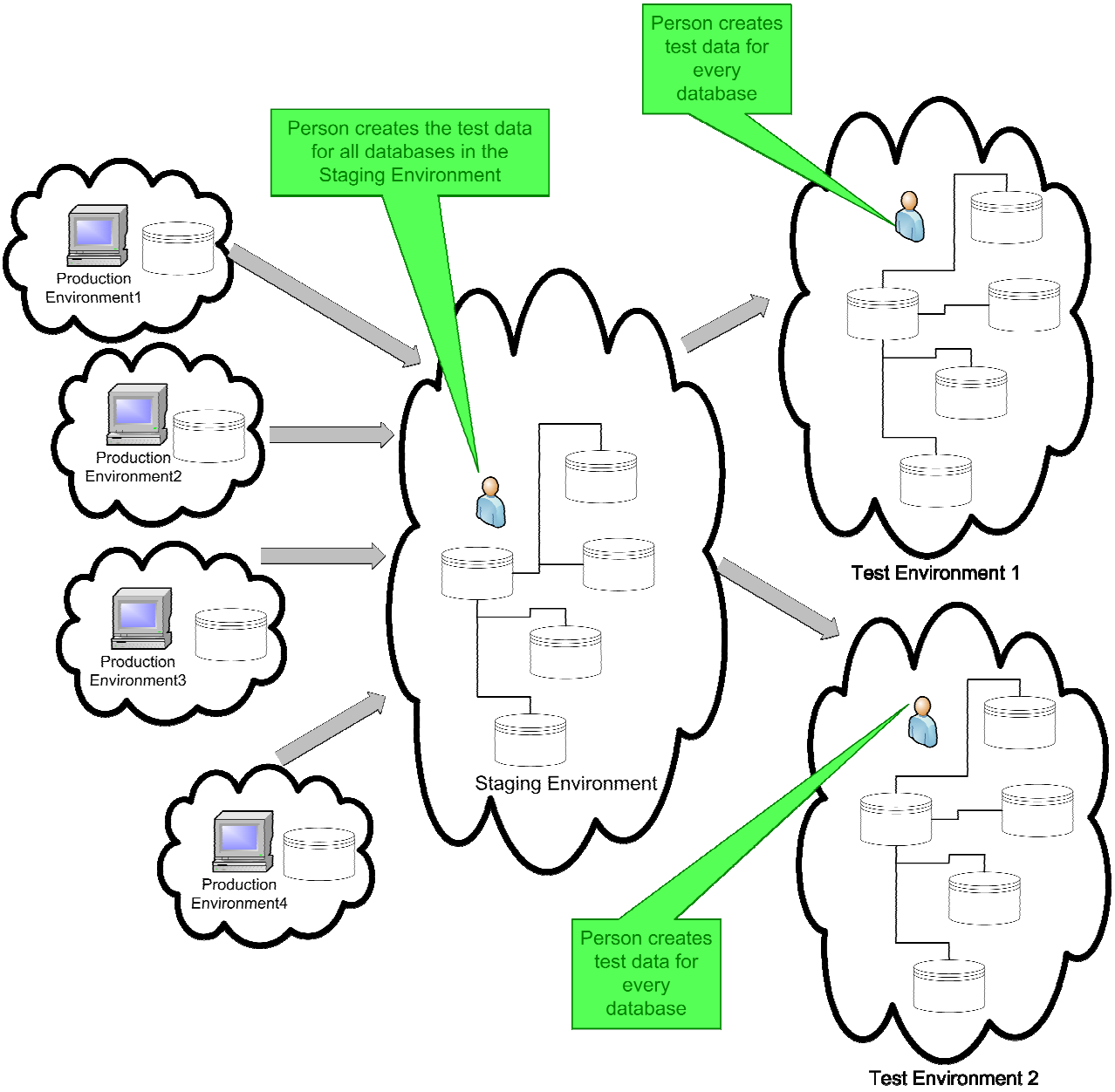

Figure A-1. Old CHARM Test Data Creation Process

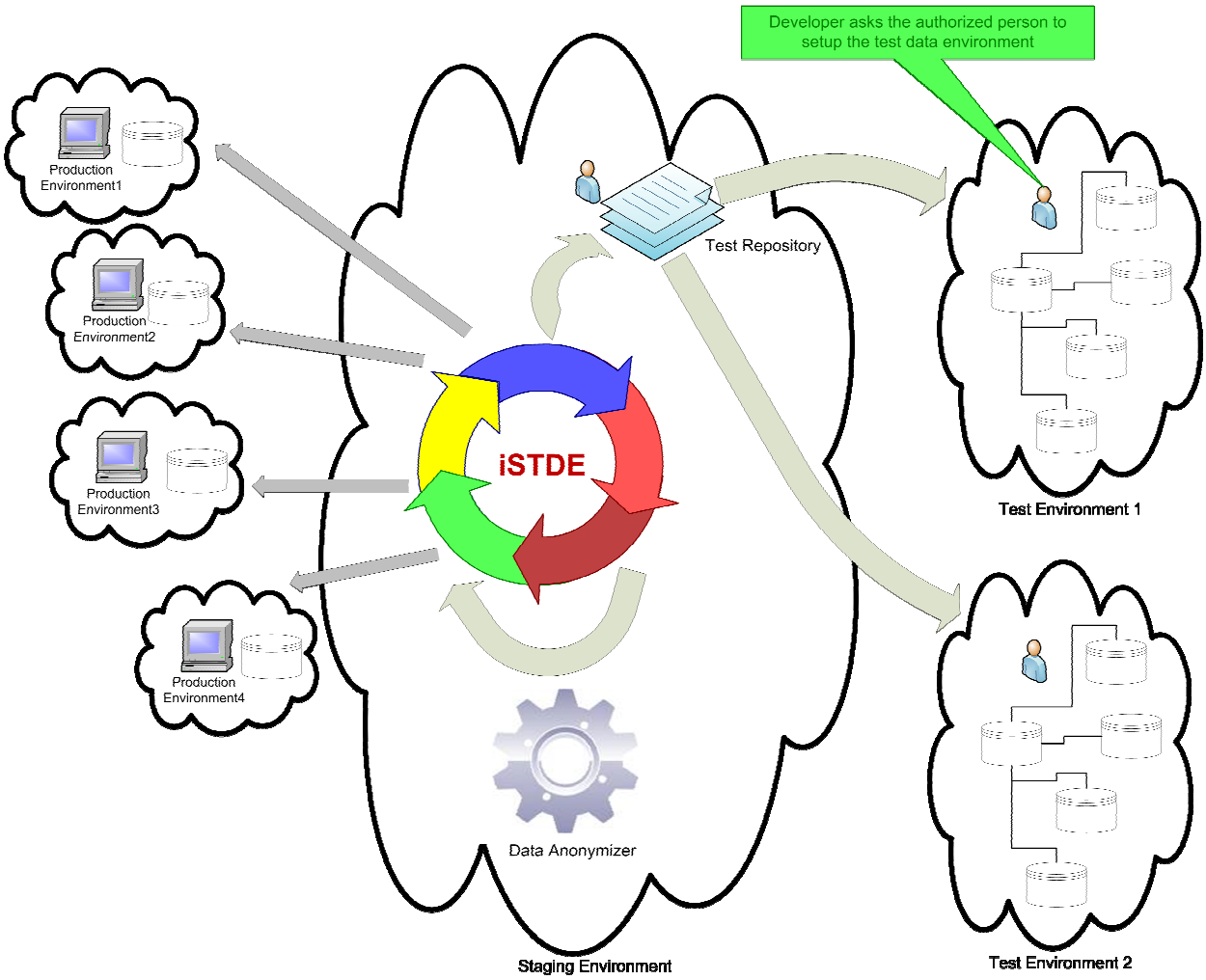

Figure A-2. New CHARM Test Data Creation Process

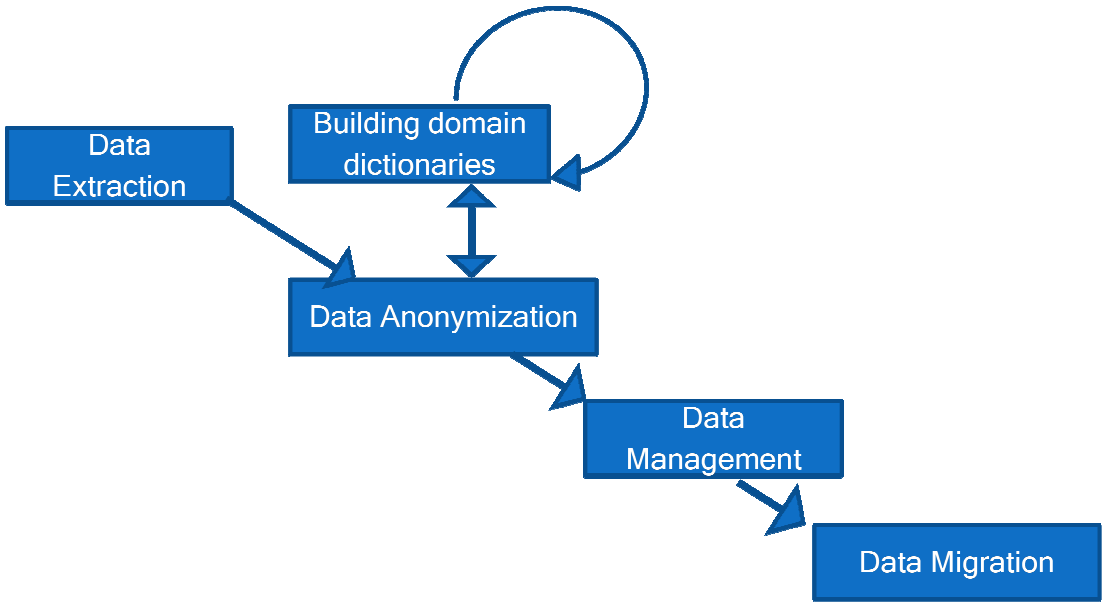

Figure A-3. iSTDE Process Centric Framework

3. **List of activities and tasks in iSTDE-Process Centric Framework**

- Activity 1 : Data Extraction
    - Selection of domains in integrated systems
    - Metadata/data extraction and simulation
    - Resolve data heterogeneities
- Activity 2 : Building Knowledge Dictionaries
    - Identifying and collecting knowledge about *PII*
- Activity 3: Data Anonymization
    - Apply techniques to anonymize data
- Activity 4: Data Management
    - Collection and management of test beds in repository
- Activity 5: Data Migration
    - Transfer of huge datasets from repository to test environment
        - Staging environment
        - Individual testing environment

Appendix B

USER GOALS AND FUNCTIONAL REQUIREMENTS

1. Goals and Design Considerations:

Goal: To provide homogeneous data at one place for testing purpose without compromising the sensitivity of the data. This dataset should also maintain the characteristics of the real data essential for testing.

2. Design Consideration:

2.1. iSTDE should run as a standalone application and provide test-data.

2.2. It should generate data without compromising the sensitivity of data.

2.3. It should also generate correlated data having real data characteristics.

2.4. The data extraction time must be comparatively less with respect to the previous methods.

2.5. It should also extract data from heterogeneous databases.

2.6. Mangled data should remain consistent in all the databases.

3. Functional Requirements:

Functional requirements of the seven steps in iSTDE are listed below:

3.1. Specification of extraction parameters.

3.2. Creation of temporary databases.

3.3. Extraction and loading of real data to temporary databases.

3.4. Data mangling.

3.5. Identification and population of PII domains.

3.6. Transferring mangled data.

3.7. Destroying mappings.

Now we further elaborate on the above requirements:

3.1. Specify extraction parameters

Its purpose is to provide user-specified data selection criteria. It includes parameters like:

3.1.1. Target environment description

3.1.2. Data sources to extract data.

3.1.3. Selection criteria (e.g., birth data range)

3.1.4. Management of some parametric information using properties file.

3.2. Creation of temporary databases

3.2.1. Creating databases that hold extracted data for mangling process.

3.2.2. Populating metadata from the schemes of real production databases.

3.2.3. Building scripts for tables, indices, sequences, key constrains.

3.2.4. Adding additional schematic information to make it compatible with real databases.

3.3. Extraction and loading of real data to temporary databases

3.3.1. Extracting a consistent slice of real data from heterogeneous databases and loading them into temporary databases.

3.3.2. Parsing and analyzing data-selection criteria.

3.3.3. Run time generation of SQL queries.

3.3.4. Execution of queries to production databases.

3.3.5. Creation of SQL inserts statements and writing them to data files.

3.4. Data mangling

3.4.1. Mangling swaps that identify domains of data in such a way that identities in resulting dataset become untraceable, but the dataset as whole contains the real data semantics required for testing.

3.4.2. Selection of identity domains, i.e., male first names, female first names.

3.4.3. Building of dictionaries from identity domains.

3.4.4. Data shuffling in dictionaries.

3.4.5. Replacing all old entries of dictionaries in real data with new entries.

3.4.6. After mangling, deleting the dictionaries so that no one can perform reverse mappings.

3.5. Transfer mangled data

3.5.1. Automatically transferring the mangled data from secure to unsecure environment.

3.5.2. Dumping and restoring scripts are created automatically.

3.5.3. Executing these scripts that will perform the following tasks:

3.5.1.1. Taking the backup of temporary databases.

3.5.1.2. Restoring this backup to developer's machine on unsecure environment.

3.5.4. Considering the access controls and firewalls.

3.6. Destroy mappings

3.6.1. Removing all tracings on secure environment that can trace any valuable information about the original data. For example, it removes:

3.6.1.1. All the data files created during data extraction; and

3.6.1.2. Temporary databases.

Appendix C

USE CASE MODELING

1. Use Case Modeling

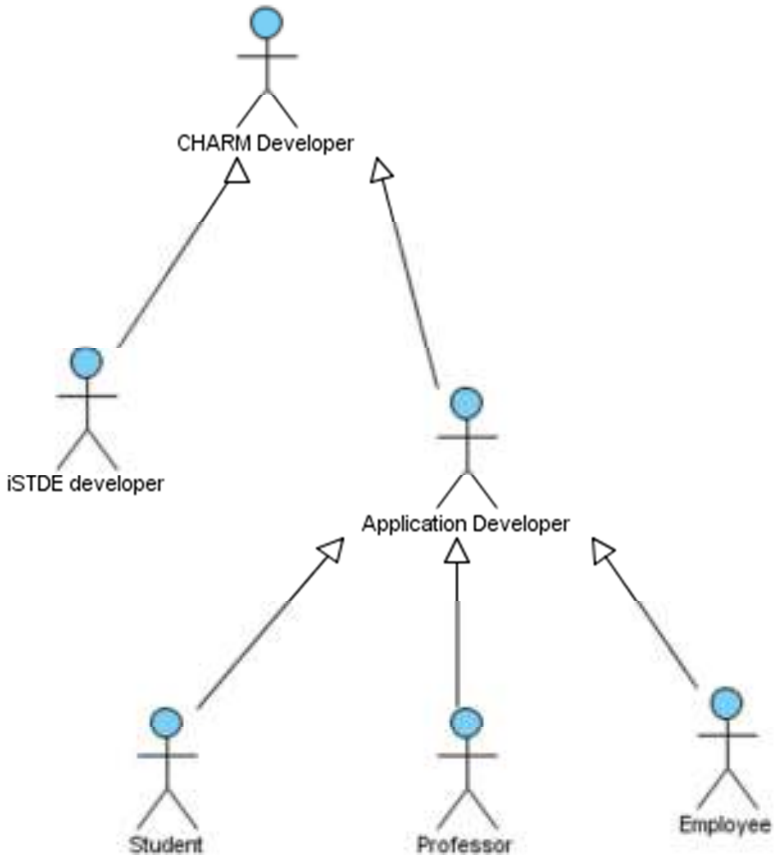

Figure C-1. Actor's hierarchy for iSTDE.

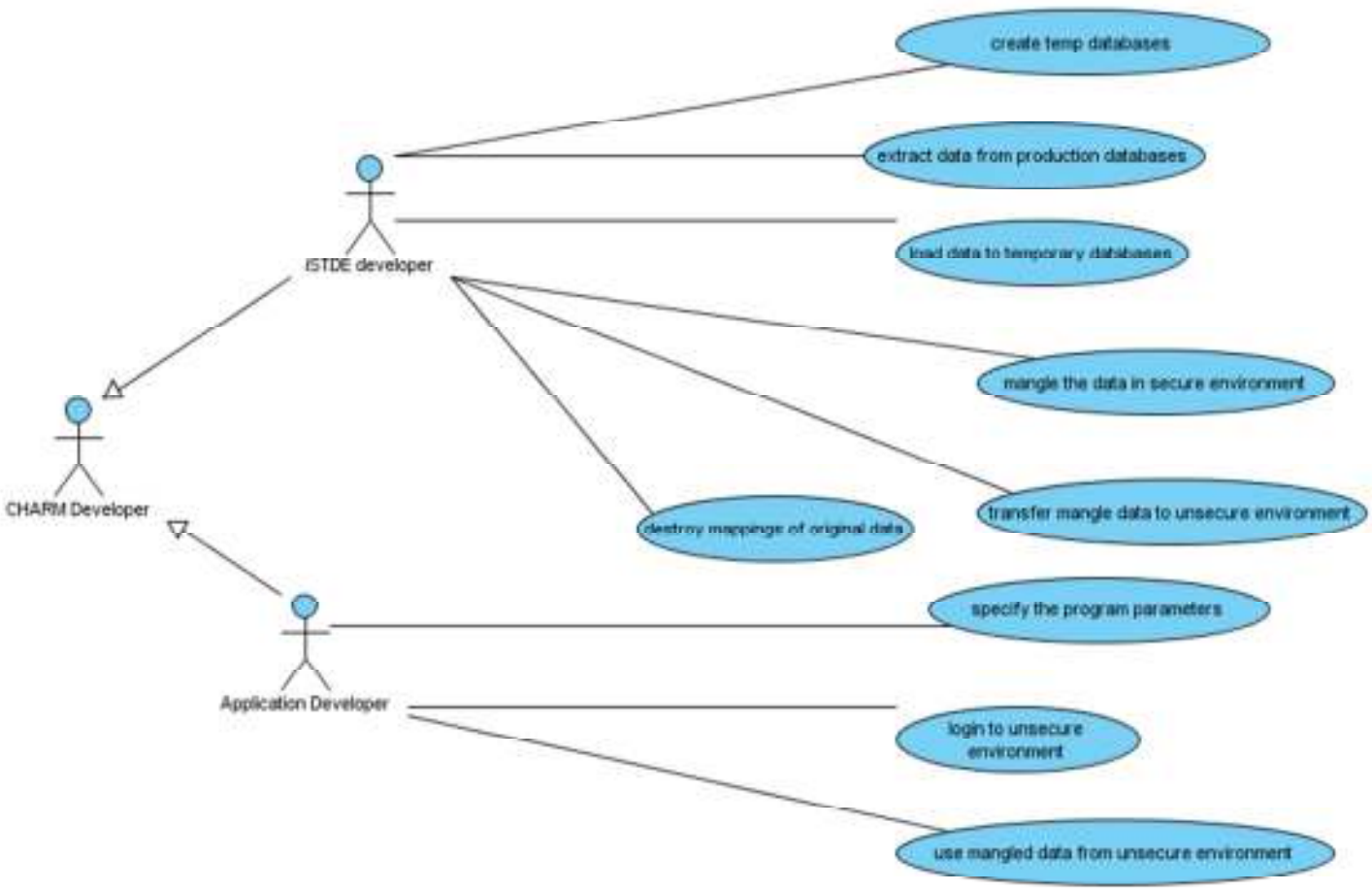

Figure C-2. Initial use case modeling for iSTDE.

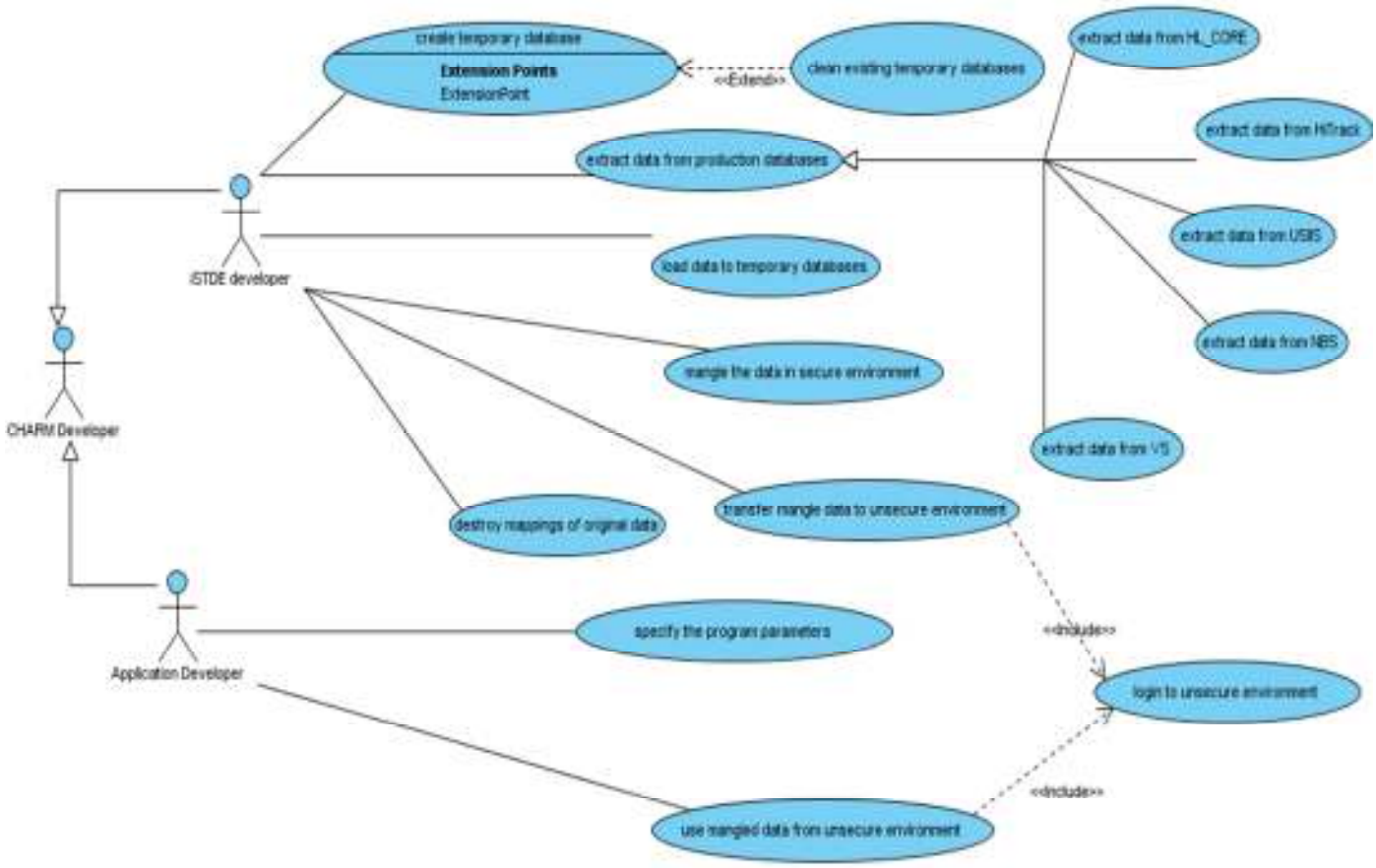

Figure C-3. Detailed use case modeling for iSTDE.

Appendix D

ARCHITECTURAL DIGRAM FOR iSTDE

1. Architectural Design

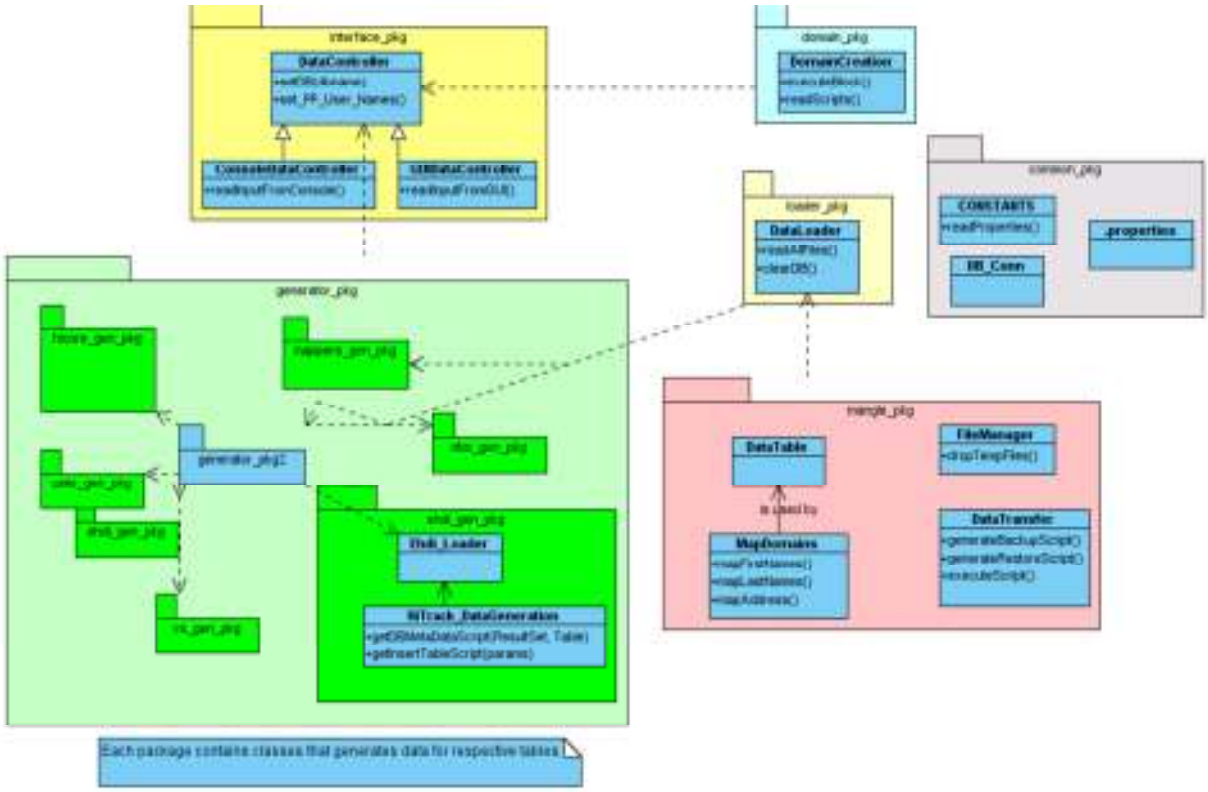

Figure D-1. Architecture diagram for iSTDE.

Appendix E

IMPLEMENTATION

1. ImportantMethods in Sample Set of Classes in iSTDE

1.1. Domain Creation.java

public static void executeBlock(BufferedReaderbr,Connectiondbcon) throws Exception: This method reads a block of statements regarding metadata and executes these statements on the development machine.

1.2. HL_CORE_DB.java

public static void geneate_hl_core_person(String from_date, String to_date,Stringtablename): Method reads the data from the person table of HL_CORE. After converting the data into insert statements, writes them to the data file.

1.3. CONSTANTS.java

public static void loadProperties(): Method loads the values in the properties file to the variables of constants in Java.

1.4. DataTransfer.java

Public static String generateRestoreScipt(String user_name): Method generates the restore script for the user passed as parameter.

1.5. FileManager.java

public void deleteFiles(File file): Deletes all the temporary files that are used for data loading purposes.

1.6. MapDomains.java

public void generateMappings(Table map, ArrayList<Table> ref, String domain): Method generates the mapping tables and loads the data.

public void mangleMappingTable(Table map_tab, String dom_name):Mangles the data in the mapping table.

Public boolean updateMapTable_RecordByIndex(Table map_tab, String map_col, String newVal, int index): Updates the domains in the original table based upon the mapping table.

1.7. Test bedLogger.java

public static void writeErr(Exception ex, String data):This method of the logger class is used to write the exception details to the log file.

Appendix F

DEPLOYMENT

1. Dependencies on External Packages or Components

iSTDE has dependencies on some jar files, such as

- msSQLServer.jar: Contains files related to SQL Server JDBC database driver.
- msutil.jar: Contains other common files that help in development.
- ojdbc14.zip: Contains files related to Oracle JDBC database driver.
- postgresql.jar: Contains files related to PostgreSQL JDBC database driver.
- pvjdbc2.jar: Contains files related to pervasive JDBC database driver.
- sqljdbc.jar: Contains JDBC driver information for SQL Server.

2. External Component Dependencies on iSTDE

SyncEngine, CoreAgent, Matcher, PP Agent, AlertEngine

3. Interfaces and Communication Protocols

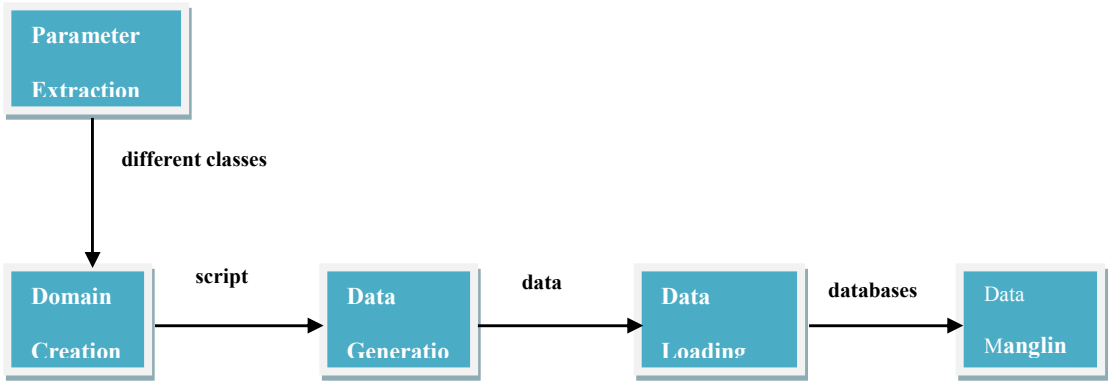

Figure F-1. Interface and communication protocols diagram for iSTDE.

- Parameter Extraction: Initializes different variables providing information about test beds. These parameter values are then used in different steps of iSTDE execution.
- Domain Creation: Creates metadata in PostgreSQL that matches the real database metadata.
- Data Generation: Uses script files generated by the domain creation module and further creates the datafiles containing un-mangled data from a real database in PostgreSQL format.
- Data Loading: Loads or dumps the datafiles into the temporary PostgreSQL databases created in the Domain Creation step.
- Identification and Population of PII domains: PII domains are identified from across all the databases involved in CHARM and then populated as a separate process.
- Data Mangling: De-identifies the real data to make it untraceable. Resulting de-identified dataset exhibits the real data characteristics and its semantics.

4. Build Instructions

Test bedGen.bat file is used for building the iSTDE. Makes a compile version of the program and runs it.

- Path: CHARM\Implementation\version2\code\src\iSTDE

5. Component Use and Testing

Setup process includes the following steps:

- Place the iSTDE folder present under the path mentioned below into any location of your system.
- Path: CHARM\Implementation\version2\code\src\iSTDE.
- Modify the property file “constant.properties” under the path: above.
- CHARM\Implementation\version2\code\src\iSTDE\Mapper.
- Make sure your VPN account is proper working.
- Double Click the file “Test bedGen.bat” present inside the iSTDE folder.
- Check the error log and log file for successful working module.
- Error log path: iSTDE\Extractor\
    - After double clicking the file, the GUI window will appear, showing where you need to input the credentials like “user” and “date range”. It will also show other parameters needing to be checked according to the requirement.
    - Once you have selected the parameters, click “start” to begin the extraction process.

6. Deployment Instructions:

Repeat the same process as described in the “Component Use and Testing “section for a specific location of the production system.

	Source	Library	Distribution
Locations:	CHARM\Implementation\version2\code\src\iSTDE	Undergoing some further extensions and testing on	Undergoing some further

		staging server.	extensions.
Resp. Person:	Ali Raza		