THESIS

AN APPROACH FOR TESTING THE EXTRACT-TRANSFORM-LOAD PROCESS IN DATA WAREHOUSE SYSTEMS

Submitted by

Hajar Homayouni

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2018

Master's Committee:

Advisor: Sudipto Ghosh
Co-Advisor: Indrakshi Ray

James M. Bieman
Leo R. Vijayasarathy

ABSTRACT

AN APPROACH FOR TESTING THE EXTRACT-TRANSFORM-LOAD PROCESS IN DATA WAREHOUSE SYSTEMS

Enterprises use data warehouses to accumulate data from multiple sources for data analysis and research. Since organizational decisions are often made based on the data stored in a data warehouse, all its components must be rigorously tested. In this thesis, we first present a comprehensive survey of data warehouse testing approaches, and then develop and evaluate an automated testing approach for validating the Extract-Transform-Load (ETL) process, which is a common activity in data warehousing.

In the survey we present a classification framework that categorizes the testing and evaluation activities applied to the different components of data warehouses. These approaches include both dynamic analysis as well as static evaluation and manual inspections. The classification framework uses information related to *what* is tested in terms of the data warehouse component that is validated, and *how* it is tested in terms of various types of testing and evaluation approaches. We discuss the specific challenges and open problems for each component and propose research directions.

The ETL process involves extracting data from source databases, transforming it into a form suitable for research and analysis, and loading it into a data warehouse. ETL processes can use complex one-to-one, many-to-one, and many-to-many transformations involving sources and targets that use different schemas, databases, and technologies. Since faulty implementations in any of the ETL steps can result in incorrect information in the target data warehouse, ETL processes must be thoroughly validated. In this thesis, we propose automated balancing tests that check for discrepancies between the data in the source databases and that in the target warehouse. Balancing tests ensure that the data obtained from the source databases is not lost or incorrectly modified by

the ETL process. First, we categorize and define a set of properties to be checked in balancing tests. We identify various types of discrepancies that may exist between the source and the target data, and formalize three categories of properties, namely, completeness, consistency, and syntactic validity that must be checked during testing. Next, we automatically identify source-to-target mappings from ETL transformation rules provided in the specifications. We identify one-to-one, many-to-one, and many-to-many mappings for tables, records, and attributes involved in the ETL transformations. We automatically generate test assertions to verify the properties for balancing tests. We use the source-to-target mappings to automatically generate assertions corresponding to each property. The assertions compare the data in the target data warehouse with the corresponding data in the sources to verify the properties.

We evaluate our approach on a health data warehouse that uses data sources with different data models running on different platforms. We demonstrate that our approach can find previously undetected real faults in the ETL implementation. We also provide an automatic mutation testing approach to evaluate the fault finding ability of our balancing tests. Using mutation analysis, we demonstrated that our auto-generated assertions can detect faults in the data inside the target data warehouse when faulty ETL scripts execute on mock source data.

ACKNOWLEDGEMENTS

I would like to thank my advisors, Prof. Sudipto Ghosh and Prof. Indrakshi Ray, for their guidance in accomplishing this project. I would like to thank Prof. Michael Kahn, Dr. Toan Ong, and the Health Data Compass team at Anschutz Medical Campus at University of Colorado Denver for supporting this project. I also wish to thank the members of my M.S. thesis committee, Prof. James M. Bieman and Prof. Leo R. Vijayasarathy for generously offering their time and guidance. I would like to thank the Software Engineering group for their constructive comments in my presentations. Finally, I wish to thank the Computer Science staff for their help throughout my study at Colorado State University.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

A data warehouse system gathers heterogeneous data from several sources and integrates them into a single data store [1]. Data warehouses help researchers and data analyzers make accurate analysis and decisions in an efficient manner [2]. While each source focuses on transactions for current data, the data warehouses use large scale (petabyte) stores to maintain past records along with the new updates to allow analyzers to find precise patterns and trends in the data.

Researchers and organizations make decisions based on the data stored in a data warehouse [3]. As a result, the quality of data in a data warehouse is extremely important. For example, many critical studies are investigated using a health data warehouse, such as the impacts of a specific medicine are performed using patient, treatment, and medication data stored in the data warehouse. Thus, the data stored in a warehouse must be accurate.

An important building block in a data warehouse is the Extract, Transform, and Load (ETL) process that (1) extracts data from various source systems, (2) integrates, cleans, and transforms it into a common form, and (3) loads it into a target data warehouse. The sources and the target can use different *schemas*, such as proprietary and open source models, different *databases*, such as relational [4] and non-relational [5], and *technologies*, such as Database Management Systems (DBMSs) or Extensible Markup Language (XML) or Comma Separated Values (CSV) flat files. The transformations can involve various types of mappings such as one-to-one, many-to-one, and many-to-many. The steps for extraction, transformation, and loading are performed using multiple components and intermediate storage files. The process is executed using jobs that run in different modes such as *full* mode, which transforms all the data in the sources, or in *incremental* mode, which updates newly added or modified data to the data warehouse based on logs, triggers, or timestamps.

1.1 Problem Description

The complexity of the transformations can make ETL implementations prone to faults, which can compromise the information stored in the data warehouse that, in turn, leads to incorrect analysis results. Faulty ETL scripts can lead to incorrect data in the data warehouse [2]. Thus, functional testing of ETL processes is critical [6]. This testing activity ensures that any changes in the source systems are correctly captured and completely propagated into the target data warehouse [2]. The manner in which ETL processes are implemented and executed can also result in incorrect data in the target data warehouse. There is a need for systematic, automated approaches for ETL testing in order to reduce the effort and cost involved in the data warehouse life cycle. While most aspects of data warehouse design, including ETL, have received considerable attention in the literature, not much work has been done for data warehouse testing [7].

Factors that affect the design of ETL tests, such as platforms, operating systems, networks, DBMS, and other technologies used to implement data warehousing make it difficult to use a generic testing approach applicable to all data warehouse projects. The huge volume of data extracted, transformed, and loaded to a data warehouse makes exhaustive manual comparison of data for testing ETL impractical [1]. Furthermore, testing the ETL process is not a one-time task because data warehouses evolve, and data get incrementally added and also periodically removed [7]. Consequently, tests need to be designed and implemented in a manner that they are repeatable.

Faults in any of the ETL components can result in incorrect data in the target data warehouse that cannot be detected through evaluating the target data warehouse in isolation. Executing the components multiple times because of erroneous settings selected by the users can result in duplication of data. System failures or connection loss in any component may result in data loss or data duplication in the target data warehouse. Manual involvement in running the ETL process may cause the erroneous setting of ETL parameters that result in incorrect modes and truncation or duplication of data, or executing ETL jobs in the wrong order. Using duplicate names for the intermediate storage files may result in the overwriting of important information. Malicious programs may remove or modify data in a data warehouse. Such problems can be addressed by balancing

tests that check for discrepancies between the data in the source databases and that in the target warehouse. Balancing tests ensure that the data obtained from the source databases is not lost or incorrectly modified by the ETL process. These tests analyze the data in the source databases and target data warehouse and report differences.

The balancing approach called *Sampling* [8] uses the *Stare and Compare* technique to manually verify data and determine differences through viewing or *eyeballing* the data. Since data warehouses contain billions of records, most of the time, less than 1% of the entire set of records are verified through this approach. QuerySurge [8] also supports balancing tests but it only compares data that is not modified during the ETL transformation, whereas the goal of ETL testing should also be to validate data that has been reformatted and modified through the ETL process. Another method is *Minus Queries* [8] in which the difference between the source and target is determined by subtracting the target data from the source data to show existence of unbalanced data. This method has the potential for generating false positives because it may report duplications that are actually allowed in the target data warehouse.

1.2 Approach

In this work, we first provide a comprehensive survey of data warehouse testing techniques. We present a classification framework that can categorize the existing testing approaches as well as the new one that we propose in the context of a real world health data warehousing project. We also discuss open problems and propose research directions.

Next, we present an approach for validating ETL processes using automated balancing tests that check for discrepancies between the data in the source databases and that in the target warehouse. We present an approach to create balancing tests to ensure that data obtained from the source databases is not lost or incorrectly modified by the ETL process. We identify the types of discrepancies that may arise between the source and the target data due to an incorrect ETL process on the basis of which we define a set of generic properties that can be applied to all data warehouses, namely, *completeness*, *consistency*, and *syntactic validity*. Completeness ensures that

all the relevant source records get transformed to the target records. Consistency and syntactic validity ensure correctness of the transformation of the attributes. Consistency ensures that the semantics of the various attributes are preserved during transformation. Syntactic validity ensures that no problems occur due to the differences in the syntax between the source and the target data.

We systematically identify the different types of source-to-target mappings from the ETL transformation rules. These mappings include one-to-one, many-to-one, and many-to-many mappings of tables, records, and attributes involved in the ETL transformations. We use these mappings to automate the generation of test assertions corresponding to each property. We provide an assertion generation tool to reduce the manual effort involved in generating balancing tests of ETL and enhance test repeatability. Our approach is applicable to data warehouses that use sources with different data models running on different platforms.

We evaluate our approach using a real-world data warehouse for electronic health records to assess whether our balancing tests can find real faults in the ETL implementation. We also provide an automatic mutation testing approach to evaluate the fault finding ability of the balancing tests and demonstrate that the generated assertions can detect faults present in mock data.

The rest of the thesis is organized as follows. Chapter 2 presents a comprehensive survey of existing testing and evaluation activities applied to the different components of data warehouses and discusses the specific challenges and open problems for each component. Chapter 3 describes a motivating example. Chapter 4 defines a set of generic properties to be verified through balancing tests. Chapter 5 describes an approach to automatically generate balancing tests. Chapter 6 presents a demonstration and evaluation of our approach. Finally, Chapter 7 concludes the thesis and discusses directions for future work.

Chapter 2

Literature Survey

In this chapter, we present a comprehensive survey [9] of existing testing and evaluation activities applied to the different components of data warehouses and discuss the specific challenges and open problems for each component. These approaches include both dynamic analysis as well as static evaluation and manual inspections. We provide a classification framework based on *what* is tested in terms of the data warehouse component to be verified, and *how* it is tested through categorizing the different testing and evaluation approaches. The survey is based on our direct experience with a health data warehouse, as well as from existing commercial and research attempts in developing data warehouse testing approaches. The rest of the chapter is organized as follows. Section 2.1 describes the components of a data warehouse. Section 2.2 presents a classification framework for testing data warehouse components. Sections 2.3 through 2.5 discuss existing approaches and their limitations for each testing activity.

2.1 Data Warehouse Components

In this section, we describe the four components of a data warehousing system, which are (1) sources, (2) target data warehouse, (3) Extract-Transform-Load (ETL) process, and (4) front-end applications.

We use an enterprise health data warehouse shown in Figure 2.1 as a running example. This data warehouse integrates patient clinical data from hospitals into a single destination to support medical research on diseases, drugs, and treatments. While each hospital focuses on transactions for current patients, the health data warehouse maintains historical data from multiple hospitals. This history often includes old patient records. The past records along with the new updates help medical researchers perform long-term data analysis. The inputs of the data warehouse use different models, such as star or relational data model. The ETL process selects data from individual databases, converts it into a common model called Observational Medical Outcomes Partnership

Common Data Model (OMOP CDM) [10], which is appropriate for medical research and analysis, and writes it to the target data warehouse on Google BigQuery [11]. Each of the ETL phases is executed as a set of ETL jobs. The ETL jobs can run in the full or incremental modes that are selected using job configuration parameters.

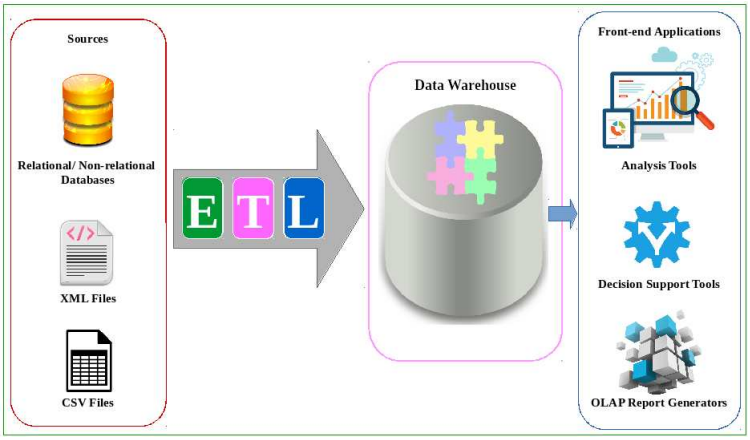

Figure 2.1: Health Data Warehouse Architecture

2.1.1 Sources and Target Data Warehouse

Sources in a data warehousing system store data belonging to one or more organizations for daily transactions or business purposes. The target data warehouse, on the other hand, stores large volumes of data for long-term analysis and mining purposes. Sources and target data warehouses can be designed and implemented using a variety of technologies including data models and data management systems.

A data model describes business terms and their relationships, often in a pictorial manner [12]. The following data models are typically used to design the source and target schemas:

- **Relational data model**: Such a model organizes data as collections of two-dimensional tables [4] with all the data represented in terms of tuples. The tables are *relations* of rows and columns, with a unique key for each row. Entity Relationship Diagram (ER) diagrams [13] are generally used to design the relational data models.

- **Non-relational data model**: Such a model organizes data without a structured mechanism to link data of different buckets (segments) [5]. These models use means other than the tables used in relational models. Instead, different data structures are used, such as graphs or documents. These models are typically used to organize extremely large data sets used for data mining because unlike the relational models, the non-relational models do not have complex dependencies between their buckets.

- **Dimensional data model**: Such a model uses structures optimized for end-user queries and data warehousing tools. These structures include *fact* tables that keep measurements of a business process, and *dimension* tables that contain descriptive attributes [14]. The information is grouped into relevant tables called dimensions, making it easier to use and interpret. Unlike relational models that minimize data redundancies and improve transaction processing, the dimensional model is intended to support and optimize queries. The dimensional models are more scalable than relational models because they eliminate the complex dependencies that exist between relational tables [15].

  The dimensional model can be represented by star or snowflake schemas [16], and is often used in designing data warehouses. The schemas are as follows:

  - *Star*: This schema has a fact table at the center. The table contains the keys to dimension tables. Each dimension includes a set of attributes and is represented via a one dimension table. For example, the sources in health data warehouse use a star data model called Caboodle from the Epic community [17].
  - *Snowflake*: Unlike the star schema, the snowflake schema has normalized dimensions that are split into more than one dimension tables. The star schema is a special case of the snowflake schema with a single level hierarchy.

The sources and data warehouses use various data management systems to collect and organize their data. The following is a list of data management systems generally used to implement the source and target data stores.

- **Relational Database Management System (RDBMS)**: An RDBMS is based on the relational data model that allows linking of information from different *tables*. A table must contain what is called a key or index, and other tables may refer to that key to create a link between their data [5]. RDBMSs typically use Structured Query Language (SQL) [18], and are appropriate to manage structured data. RDBMSs are able to handle queries and transactions that ensure efficient, correct, and robust data processing even in the presence of failures.

- **Non-relational Database Management System**: A non-relational DBMS is based on a non-relational data model. The most popular non-relational database is Not Only SQL (NoSQL) [5], which has many forms, such as document-based, graph-based, and object-based. A non-relational DBMS is typically used to store and manage large volumes of unstructured data.

- **Big Data Management System**: Management systems for big data need to store and process large volumes of both structured and unstructured data. They incorporate technologies that are suited to managing non-transactional forms of data. A big data management system seamlessly incorporates relational and non-relational database management systems.

- **Data Warehouse Appliance (DWA)**: DWA was first proposed by Hinshaw [19] as an architecture suitable for data warehousing. DWAs are designed for high-speed analysis of large volumes of data. A DWA integrates database, server, storage, and analytics into an easy-to-manage system.

- **Cloud Data Warehouse Appliance**: Cloud DWA is a data warehouse appliance that runs on a cloud computing platform. This appliance benefits from all the features provided by cloud computing, such as collecting and organizing all the data online, obtaining infinite computing resources on demand, and multiplexing workloads from different organizations [20].

Table 2.1 presents some of the available products used in managing the data in the sources and target data warehouses. The design and implementation of the databases in the sources are

Table 2.1: Available Products for Managing Data in the Sources and Data Warehouses

Product Category	Examples
DBMS	Relational: MySQL [21], MS-SQL Server [22], PostgreSQL [23]
	Non-relational: Accumulo [24], ArangoDB [25], MongoDB [26]
Big data management system	Apache Hadoop [27], Oracle [28]
Data warehouse appliance	IBM PureData System [29]
Cloud data warehouse	Google BigQuery [30], Amazon Redshift [31]

typically based on the organizational requirements, while those of the data warehouses are based on the requirements of data analyzers and researchers. For example, the sources for a health data warehouse are databases in hospitals and clinic centers that keep patient, medication, and treatment information in several formats. Figure 2.2 shows an example of possible sources in the health data warehouse. Hospital A uses a flat spreadsheet to keep records of patient data. Hospital B uses an RDBMS for its data. Hospital C also uses an RDBMS but has a different schema than Hospital B. The data from different hospitals must be converted to a common model in the data warehouse.

The target data warehouse for health data may need to conform to a standard data model designed for electronic health records such as the OMOP CDM.

2.1.2 Extract, Transform, Load (ETL)

The ETL process extracts data from sources, transforms it to a common model, and loads it to the target data warehouse. Figure 2.3 shows the components involved in the ETL process, namely Extract, Transform, and Load.

1. **Extract**: This component retrieves data from heterogeneous sources that have different formats and converts the source data into a single format suitable for the transformation phase. Different procedural languages such as Transact-SQL or COBOL are required to query the source data. Most extraction approaches use Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC) drivers to connect to sources that are in DBMS or flat file formats [32].

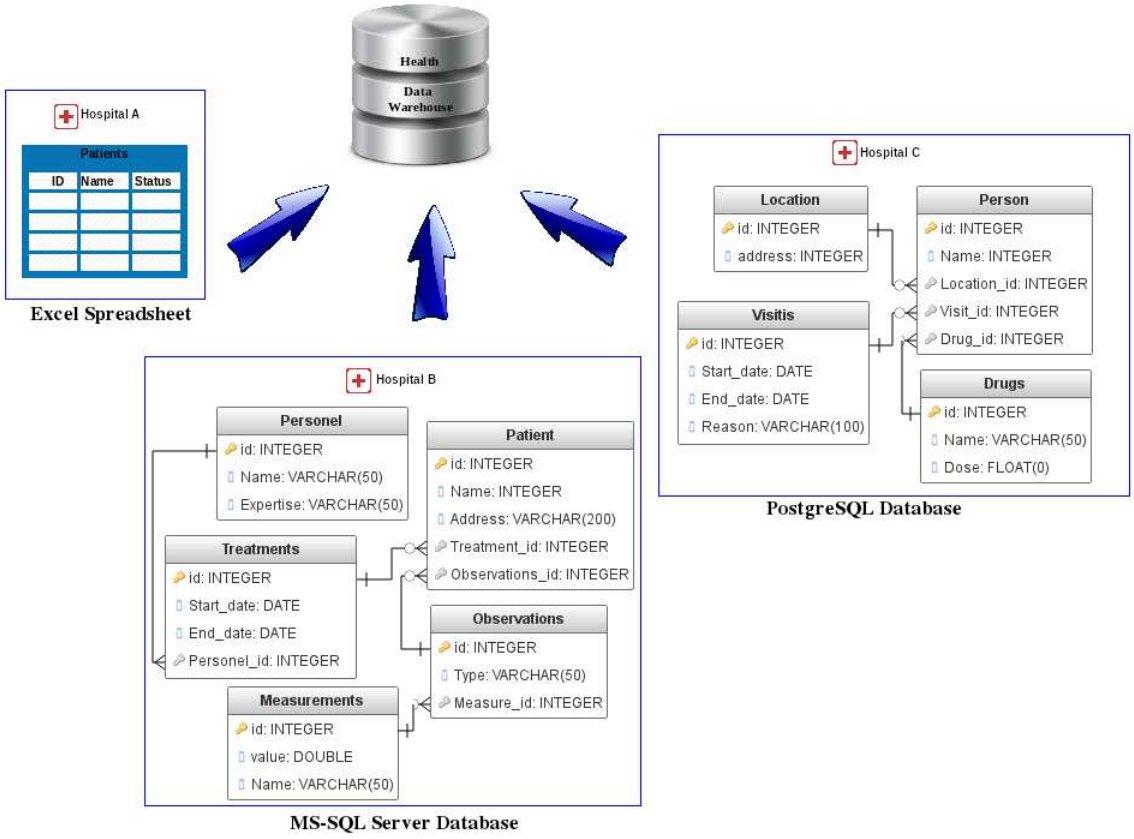

Figure 2.2: Sample Sources for a Health Data Warehouse

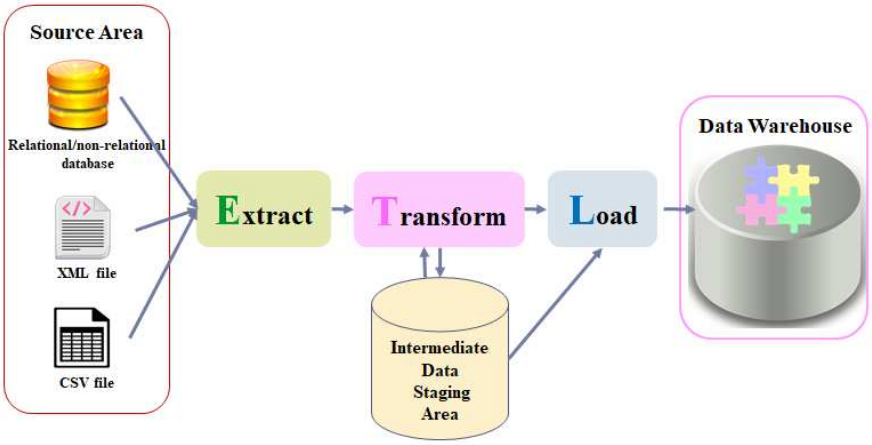

Figure 2.3: General Framework for ETL Processes

Data extraction is performed in two phases. Full extraction is performed when the entire data is extracted for the first time. Incremental extraction happens when new or modified data are retrieved from the sources. Incremental extraction employs strategies such as log-based, trigger-based, or timestamp-based techniques to detect the newly added or modified data. In

the log-based technique, the DBMS log files are used to find the newly added or modified data in the source databases. Trigger-based techniques create triggers on each source table to capture changed data. A trigger automatically executes when data is created or modified through a Data Manipulation Language (DML) event. Some database management systems use timestamp columns to specify the time and date that a given row was last modified. Using these columns, the timestamp-based technique can easily identify the latest data.

2. **Transform**: This component propagates data to an intermediate Data Staging Area (DSA) where it is cleansed, reformatted, and integrated to suit the format of the model of a target data warehouse [2]. This component has two objectives.

   First, the transformation process cleans the data by identifying and fixing (or removing) the existing problems in the data and prepares the data for integration. The goal is to prevent the transformation of so-called dirty data [33, 34]. The data extracted from the sources is validated both syntactically and semantically to ensure that it is correct based on the source constraints. Data quality validation and data auditing approaches can be utilized in this step to detect the problems in the data. Data quality validation approaches apply quality rules to detect syntactic and semantic violations in the data. Data auditing approaches use statistical and database methods to detect anomalies and contradictions in the data [35]. In Table 2.2 we present some examples of data quality validation applied to data cleansing of patients in our health data warehouse.

Table 2.2: Examples of Validation Applied To Data Cleansing

Validation	**Example of A Violation**
Incorrect value check	birth_date=70045 is not a legal date format
Uniqueness violation check	same SSN='123456789' presented for two people
Missing value check	gender is null for some records
Wrong reference check	referenced hospital=1002 does not exist
Value dependency violation check	country='Germany' does not match zip code='77'

Second, it makes the data conform to the target format through the application of a set of transformation rules described in the source-to-target mapping documents provided by the data warehouse designers [32].

3. **Load**: This component writes the extracted and transformed data from the staging area to the target data warehouse [1]. The loading process varies widely based on the organizational requirements. Some data warehouses may overwrite existing data with new data on a daily, weekly, or monthly basis, while other data warehouses may keep the history of data by adding new data at regular intervals. The load component is often implemented using loading jobs that fully or incrementally transform data from DSA to the data warehouse. The full load transforms the entire data from the DSA, while the incremental load updates newly added or modified data to the data warehouse based on logs, triggers, or timestamps defined in the DSA.

The ETL components, namely extract, transform, and load, are not independent tasks, and they need to be executed in the correct sequence for any given data. However, parallelization can be achieved if different components execute on distinct blocks of data. For example, in the incremental mode the different components can be executed simultaneously; the newly added data can be extracted from the sources while the previously extracted block of data is being transformed and loaded into the target data warehouse.

2.1.3 Front-end Applications

Front-end applications present the data to end-users who perform analysis for the purpose of reporting, discovering patterns, predicting, or making complex decisions. These applications can be any of the following tools:

- **Online Analytical Processing (OLAP) report generators**: These applications enable users and analysts to extract and access a wide variety of views of data for multidimensional analysis [36]. Unlike traditional relational reports that represent data in two-dimensional row

and column format, OLAP report generators represent their aggregated data in a multi-dimensional structure called cube to facilitate the analysis of data from multiple perspectives [37]. OLAP supports complicated queries involving facts to be measured across different dimensions. For example, as Figure 2.4 shows, an OLAP report can present a comparison of the number (fact) of cases reported for a disease (dimension) over years (dimension), in the same region (dimension).

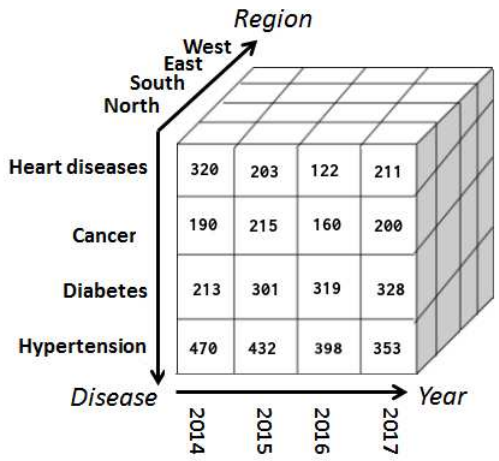

Figure 2.4: OLAP Cube Example of the Number of Cases Reported for Diseases over Time and Regions

- **Analysis and data mining**: These applications discover patterns in large datasets helping users and data analysts understand data to make better decisions [38]. These tools use various algorithms and techniques, such as classification and clustering, regression, neural networks, decision trees, nearest neighbor, and evolutionary algorithms for knowledge discovery from data warehouses. For example, clinical data mining techniques [39] are aimed at discovering knowledge from health data to extract valuable information, such as the probable causes of diseases, nature of progression, and drug effects.

- **Decision support**: These applications support the analysis involved in complex decision making and problem solving processes [40] that involve sorting, ranking, or choosing from options. These tools typically use Artificial Intelligence techniques, such as knowledge base

or machine learning to analyze the data. For example, a Clinical Decision Support [41] application provides alerts and reminders, clinical guidelines, patient data reports, and diagnostic support based on the clinical data.

2.2 Testing Data Warehouse Components

Systematic testing and evaluation techniques have been proposed by researchers and practitioners to verify each of the four components of a data warehouse to ensure that they perform as expected. We present a comprehensive survey by defining a classification framework for the testing and evaluation techniques applied to each of the four components.

Figure 2.5 shows the classification framework for the techniques applicable to the sources, target data warehouse, ETL process, and front-end applications. The framework presents *what* is tested in terms of data warehouse components, and *how* they are tested. The following are the data warehouse components presented in the framework:

- *The sources and the target data warehouse* store data. As a result, the same types of testing and evaluation techniques apply to them. We consider three different aspects to classify the approaches used to test these two components; these are (1) testing the underlying data, (2) testing the data model, and (3) testing the product used to manage the data.

- *The ETL process* requires the largest effort in the data warehouse development life cycle [2]. As a result, most existing data warehouse testing and evaluation approaches focus on this process. Various functional and non-functional testing methods have been applied to test the ETL process because it directly affects the quality of data inside the data warehousing systems.

- *The front-end applications* in data warehousing systems provide an interface for users to help them interact with the back-end data store.

We categorize the existing testing and evaluation approaches as functional, structural, usability, maintainability, security, performance and stress, scalability, reliability, regression, and recovery

testing. The shaded boxes represent the categories not covered by the existing testing approaches but that we feel are needed based on our experience with a real world health data warehouse project.

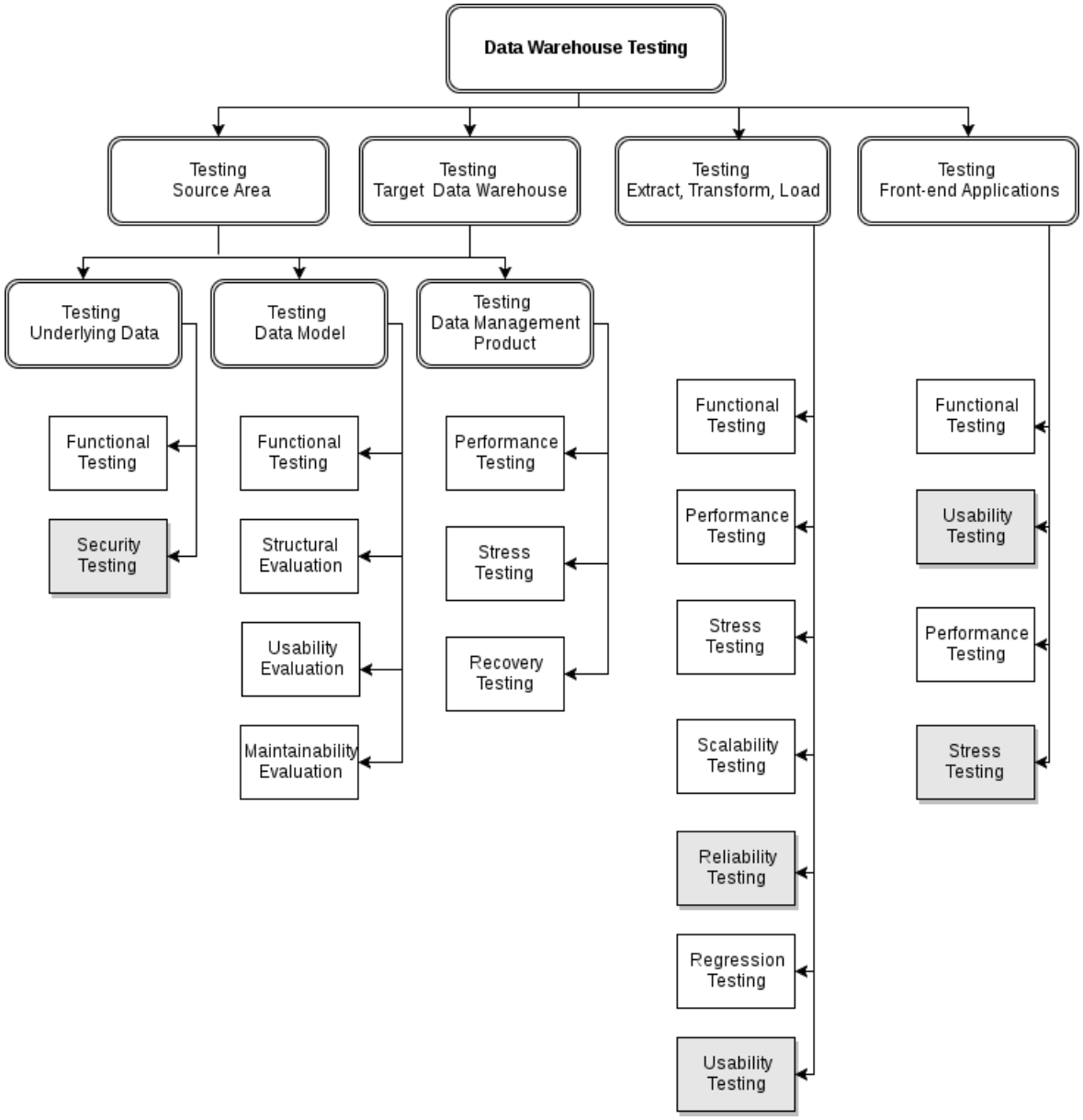

Figure 2.5: Classification Framework for Data Warehouse Testing

Other researchers have also defined frameworks for testing techniques that are applicable to the different components of a data warehouse. Golfarelli and Rizzi [7] proposed a framework to describe and test the components in a data warehouse. They defined the data warehouse compo-

nents as *schema*, *ETL*, *database*, and *front-end applications*. However, the *schema* and *database* are not exactly data warehouse components. Instead they are features of the sources and the target data warehouses. The framework uses seven different testing categories (functional, usability, performance, stress, recovery, security, and regression) applicable to each of the data warehouse components. Some non-functional testing techniques such as those for assessing scalability, reliability, and maintainability are not included.

Mathen [1] surveyed the process of data warehouse testing considering two testing aspects, which were (1) testing underlying data, and (2) testing the data warehouse components. The paper focused on two components in the data warehouse architecture, i.e., the ETL process and the client applications, and discussed testing strategies relevant to these components. Performance, scalability, and regression testing categories were introduced. Although testing the sources and the target data warehouse is critical to ensuring the quality of the entire data warehouse system, they were ignored in Mathen's testing framework. Moreover, other functional and non-functional aspects of testing data warehouse components, such as security, usability, reliability, recovery, and maintainability testing, and existing methods and tools for each testing type were not included.

In Sections 2.3 to 2.5, we describe the testing and evaluation activities necessary for each component in detail, and present the challenges and open problems.

2.3 Testing Source Area and Target Data Warehouse

In this section we target the locations that store the data in a data warehousing system, namely, the sources and the target data warehouse. If problems exist in the sources, they should be resolved before the data is extracted and loaded into a target where fault localization is much more expensive [3]. Fault localization is the process of finding the location of faults in a software system. Due to the fact that there are many components and processes involved in data warehousing systems, if the faulty data is propagated to the target data warehouse, finding the location of the original fault that caused subsequent error states will require a lot of effort. As a result, testing the source area is critical to ensuring the quality of data being propagated to the target data warehouse.

The quality of target storage area is also important [6] because this is the place where the data analyzers and researchers apply their queries either directly or through the front-end applications. Any problem in the target data warehouse results in incorrect information. Thus, testing must ensure that the target meets the specifications and constraints defined for the data warehouse.

We considered three different aspects to test the source area and the target data warehouse. These are (1) testing the underlying data, (2) testing the data model, and (3) testing the data management product.

2.3.1 Testing Underlying Data

In this testing activity, the data stored in the sources and the target data warehouse is validated against organizational requirements, which are provided by domain experts in the form of a set of rules and definitions for valid data. If the underlying data fails to meet the requirements, any knowledge derived from the data warehouse will be incorrect [42].

We describe existing functional and security testing approaches based on testing the underlying data in data warehouses as well as propose approaches based on our experience to achieve high quality data in a health data warehouse.

Functional Testing of Underlying Data

Functional testing of the underlying data is a type of data quality testing that validates the data based on quality rules extracted from business requirements documents. The data quality test cases are defined as a set of queries that verify whether the data follows the syntactic and semantic rules. This testing activity uses domain-specific rules, which are a set of business rules that are internal to an organization.

Examples of the data elements that are verified using data quality tests are as follows:

- **Data type**: A data type is a classification of the data that defines the operations that can be performed on the data and the way the values of the data can be stored [43]. The data type can be numeric, text, boolean, or date-time; these are defined in different ways in different languages.

- **Data constraint**: A constraint is a restriction that is placed on the data to define the values the data can take. Primary key, foreign key, and not-null constraints are typical examples.

Examples of semantic properties that we suggest are as follows:

- **Data plausibility**: A restriction that is placed on the data to limit the possible values it can take. For example, a US zip code can only take five digit values.

- **Logical constraint**: A restriction defined for the logical relations between data. For example, the *zip code=33293* does not match the *country=Germany*.

The data quality rules are not formally specified in the business requirements. The tester needs to bridge the gap between informal specifications and formal quality rules. Table 2.3 presents some examples of informally defined data quality rules for electronic health records [44]. Table 2.4 shows test cases defined as queries to verify the data quality rules presented in Table 2.3. Assume that after executing the test cases (queries), the test results are stored in a table called *tbl_test_results*. In this table, each record describes the failed assertion. The record includes the *test_id* that indicates the query number, *status* that takes as values *error* and *warning*, and *description* that contains a brief message about the failure. An empty table indicates that all the assertions passed.

Table 2.3: Data Quality Rules for Electronic Health Records

	Field	**Data quality rule**	**Property**
1	Weight	Should not be negative	Semantic (data plausibility)
2	Weight	Should be a numeric value	Syntactic (data type)
3	Sex	Should be *Male* or *Female* or *Other*	Semantic (data plausibility)
4	Sex	Should not be null	Syntactic (data constraint)
5	Start_date, End_date	Start_date of patient visit should be before End_date	Semantic (logical constraint)
6	Start_date, End_date	Should be a date value	Syntactic (data type)

Table 2.4: Test Cases to Assess Electronic Health Records

	Query
1	INSERT INTO tbl_test_results (test_id, status, description) values (SELECT 1 AS test_id, 'error' AS status, 'weight is negative' AS description FROM tbl_patients WHERE weight<0)
2	INSERT INTO tbl_test_results (test_id, status, description) values (SELECT 2 AS test_id, 'error' AS status, 'weight is non-numeric' AS description FROM tbl_patients WHERE weight.type<>DOUBLE OR weight.type<>INTEGER OR weight.type<>FLOAT)
3	INSERT INTO tbl_test_results (test_id, status, description) values (SELECT 3 AS test_id, 'error' AS status, 'Sex is invalid' AS description FROM tbl_patients WHERE !(Sex='Male' OR Sex='Female' OR Sex='Other'))
4	INSERT INTO tbl_test_results (test_id, status, description) values (SELECT 4 AS test_id, 'error' AS status, 'Sex is null' AS description FROM tbl_patients WHERE Sex=null)
5	INSERT INTO tbl_test_results (test_id, status, description) values (SELECT 5 AS test_id, 'error' AS status, 'start date is greater than end date' AS description FROM tbl_patients WHERE Start_date>End_date)
6	INSERT INTO tbl_test_results (test_id, status, description) values (SELECT 6 AS test_id, 'error' AS status, 'Invalid dates' AS description FROM tbl_patients WHERE Start_date.type<>Date OR End_date.type<>Date)

Data profiling [3] and data auditing [35] are statistical analysis tools that verify the data quality properties to assess the data and detect business rule violations, as well as anomalies and contradictions in the data. These tools are often used for testing the quality of data at the sources with the goal of rectifying data before it is loaded to the target data warehouse [32].

There exist data validation tools that perform data quality tests focusing on the target data. Data warehouse projects are typically designed for specific business domains and it is difficult to define a generalized data quality assurance model applicable to all data warehouse systems. As a result, the existing data quality testing tools are developed either for a specific domain or for applying basic data quality checks that are applicable to all domains. Other generalized tools let users define their desired data quality rules.

Achilles [45] proposed by the OHDSI community [46] is an example that generates specific data quality tests for the electronic health domain. This tool defines 172 data quality rules and verifies them using queries as test cases. The tool checks the data in health data warehouses to

ensure consistency with the OMOP common data model. It also uses rules that check the semantics of health data to be plausible based on its rule set. Table 2.4 shows some examples.

Loshin [47] provided a data validation engine called GuardianIQ that does not define specific data quality rules but allows users to define and manage their own expectations as business rules for data quality at a high level in an editor. As a result, this tool can be used in any data warehousing project. The tool transforms declarative data quality rules into queries that measures data quality conformance with their expectations. Each data is tested against the query set and scored across multiple dimensions. The scores are used for the measurement of levels of data quality, which calculates to what extent the data matches the user's expectations.

Informatica Powercenter Data Validation [48] is another example of a tool that generates data quality tests and is generalized for use in any data warehouse project. It allows users to develop their business rules rapidly without having any knowledge of programming. The test cases, which are a set of queries, are generated from the user's business rules to be executed against the data warehouse under test.

Gao et al. [49] compare the existing data quality validation tools for general use in terms of the operation environment, supported DBMSs or products, data validation checks, and case studies. All the tools discussed in Gao et al.'s paper provide basic data quality validations, such as null value, data constraint, and data type checks. However, they do not assure the completeness of their data quality checks through well-defined test adequacy criteria. In software testing, a test adequacy criterion is a predicate that determines what properties of a software application must be exercised to constitute a complete test. We can define the test adequacy criteria for data quality tests as the number of columns, tables or constraints exercised by the quality tests. The set of test cases (queries) must contain tests to verify the properties of all the columns in all the tables of the sources or the target data warehouse.

Furthermore, the fault finding ability of the data quality tests are not evaluated in any of the surveyed approaches. We suggest that new research approaches be developed using mutation analysis techniques [50] to evaluate the ability of data quality tests to detect possible faults in the data.

In these techniques, a number of faults are injected into the data to see how many of the faults are detected by the tests. Table 2.5 shows a number of sample faults to inject into the data to violate the data quality properties we defined in this section.

Table 2.5: Sample Faults Injected into Health Data for Mutation Analysis

Property	**Fault type**
Data type	Store a string value in a numeric field
Data constraint	Copy a record to have duplicate values for a primary key field
Data plausibility	Store a negative value in a weight field
Logical constraint	Set a pregnancy status to *true* for a male patient

Security Testing of Underlying Data

Security testing of underlying data is the process of revealing possible flaws in the security mechanisms that protect the data in a data storage area. The security mechanisms must be built into the data warehousing systems. Otherwise, if access control is only built into the front-end applications but not into the data warehouse, a user may bypass access control by directly using SQL queries or reporting tools on the data warehouse [51].

Every source database may have its access privileges defined for its data based on organizational requirements. Data loaded to the target data warehouse is supposed to maintain the same security for the corresponding data in the sources, while enforcing additional policies based on the data warehouse requirements. For example, if the personal information of the patients in a hospital is protected via specific techniques such as by defining user profiles or database access control [7], the same protection must be applied for the patient data transformed to the target health data warehouse. Additional access polices may be defined on the target health data warehouse to authenticate medical researchers who want to analyze the patient data.

Security testing of the underlying data in a data warehouse involves a comparison of the access privileges defined for the target data with the ones defined for the corresponding source data to determine whether all the required protections are correctly observed in the data warehouse. For this purpose, we can define security tests by formulating queries that return defined permissions associ-

ated with the data in both the sources and the target data warehouse, and compare the permissions for equivalent data using either manual or automatic techniques.

2.3.2 Testing the Data Model

As the data model is the foundation for any database, it is critical to get the model right because a flawed model directly affects the quality of information. Data model tests ensure that the design of the model follows its standards both conceptually and logically, and meets the organizational specifications. Documentation for the source and target model help equip testers with the required information for the systematic testing of data models.

Functional Evaluation of the Data Model

In this evaluation activity, the quality of the data model design is verified to be consistent with organizational requirements of the sources or the data warehouse. Some of the approaches are general enough to assess any data model (relational, non-relational, or dimensional), while there exist other approaches that evaluate a specific data model.

Hoberman [12] created a data model scorecard to determine the quality of any data model design that can be applied to both the source area and the target data warehouse. The scorecard is an inspection checklist that includes a number of questions and the score for each question. The number in front of each question represents the score of the question assigned by Hoberman. The organization places a value between 0 and the corresponding score on each question to determine to what extent the model meets the functional requirements. The following is a description of each question related to the functional evaluation of data models and the corresponding scores:

1. Does the model capture the requirements (15)? This ensures that the data model represents the organizational requirements.

2. Is the model complete (15)? This ensures that both the data model and its metadata (data model descriptive information) are complete with respect to the requirements.

3. Does the model match its schema (10)? This ensures that the detail (conceptual, logical, or physical) and the perspective (relational, dimensional, or NoSQL) of the model matches its definition.

4. Is the model structurally correct (15)? This validates the design practices (such as primary key constraints) employed for building the data model.

5. Are the definitions appropriate (10)? This ensures that the definitions in the data model are correct, clear and complete.

6. Is the model consistent with the enterprise (5)? This ensures that the set of terminology and rules in data model context can be comprehended by the organization.

7. Does the metadata match the data (10)? This ensures that the data model's description is consistent with the data model.

Golfarelli and Rizzi [7] proposed three types of tests on the conceptual and logical dimensional data model in a data warehouse:

- A *fact test* verifies whether or not the conceptual schema meets the preliminary workload requirements. The preliminary workload is a set of queries that business users intend to run against the target data warehouse. These queries help the data warehouse designers identify required facts, dimensions, and measurements in the dimensional data model [52]. For each workload, the fact test checks whether or not the required measures are included in the fact schema. This evaluation also measures the number of non-supported workloads.

- A *conformity test* assesses how well the conformed dimensions are designed in a dimensional data model. Such a model includes fact tables that keep metrics of a business process, and dimension tables that contain descriptive attributes. A fact table contains the keys to the dimension tables. A conformed dimension is one that relates to more than one fact. These dimensions support the ability to integrate data from multiple business processes. The

conformity test is carried out by measuring the sparseness of a bus matrix [53] that is a high-level abstraction of a dimensional data model. In this matrix, columns are the dimension tables, and rows are the fact tables (business processes). The matrix associates each fact with its dimensions. If there is a column in the matrix with more than one non-zero element, it shows the existence of a conformed dimension. If the bus matrix is dense (i.e., most of the elements are non-zero), it shows that there are dimensions that are associated with many facts, which indicates that the model includes overly generalized columns. For example, a *person* column refers to a wide variety of people, from employees to suppliers and customers while there is zero overlap between these populations. In this case, it is preferable to have a separate dimension for each population and associate them to the corresponding fact. On the other hand, if the bus matrix is sparse (i.e., most of the elements are zero), it shows that there is a few conformed dimension defined in the dimensional model, which indicates that the model includes overly detail columns. For example, each individual descriptive attribute is listed as a separate column. In this case, it is preferable to create a conformed dimension that is shared by multiple facts.

- A *star test* verifies whether or not a sample set of queries in the preliminary workload can be correctly formulated in SQL using the logical data model. The evaluation measures the number of non-supported workloads.

The above functional evaluation activities are manually performed via inspections. There is a lack of automated techniques.

Structural Evaluation of the Data Model

This type of testing ensures that the data model is correctly implemented using the database schema. The database schema is assessed for possible flaws. MySQL schema validation plug-in performs general validation for relational data models [54]. It evaluates the internal structure of the database schema and performs the following checks:

1. Validate whether content that is not supposed to be empty is actually empty. The tool reports an error if any of the following empty content exists in the relational database schema:

    - A table without columns
    - A view without SQL code
    - A table/view not being referenced by at least one role
    - A user without privileges
    - A table/object that does not appear in any ER diagrams

2. Validate whether a table is correctly defined by checking the primary key and foreign key constraints in that table. The tool reports an error if any of the following incorrect definition exists in the relational database schema:

    - A table without primary key
    - A foreign key with a reference to a column with a different type

3. Validate whether there are duplications in the relational database objects. The tool reports an error if any of the following duplications exist in the relational database schema:

    - Duplications in object names
    - Duplications in roles or user names
    - Duplications in indexes

4. Validate whether there are inconsistencies in the column names and their types. The tool reports an error if the following inconsistency exists in the relational database schema:

    - Using the same column name for columns of different data types

The above approach targets the structural validation of the relational data schema but it does not apply to non-relational and other data schema.

To assess the coverage of validation, we suggest using various structural metrics. These metrics are predicates that determine what properties of a schema must be exercised to constitute a thorough evaluation. The metrics are the number of views, routines, tables, columns and structural constraints that are validated during the structural evaluations.

Usability Evaluation of the Data Model

Usability evaluation of a data model tests whether the data model is easy to read, understand, and use by the database and data warehouse designers. A data model is usually designed in a way to cover the requirements of database and data warehouse designers. There are many common data models designed for specific domains, such as health, banking, or business. The Hoberman scorecard [12] discussed in the functional evaluation of the data model also includes a number of questions and their scores for usability evaluation of any data model. The data warehouse designer places a value between 0 and the corresponding score on each question to determine to what extent the model meets the usability requirements. The following is a description of each question related to the usability evaluation of data models and the corresponding scores:

1. Does the model use generic structures, such as data element, entity, and relationship (10)? This ensures that the data model uses appropriate abstractions to be transferable to more generic domains. For example, instead of using phone number, fax number, or mobile number elements, an abstract structure contains phone and phone type which accommodates all situations.

2. Does the model meet naming standards (5)? This ensures that the terms and naming conventions used in the model follow the naming standards for data models. For example, inconsistent use of uppercase and lowercase letters, and underscore, such as in Last Name, FIRST NAME and middle_name indicate that naming standards are not being followed.

3. Is the model readable (5)? This ensures that the data model is easy to read and understand. For example, it is more readable to group the data elements that are conceptually related into

one structure instead of scattering the elements over unrelated structures. For example, city, state, and postal code are grouped together.

The above approach involves human inspection, and there does not exist automated techniques for the usability testing of relational, non-relational, and dimensional data models.

Maintainability Evaluation of the Data Model

Due to the evolving nature of data warehouse systems, it is important to use a data model design that can be improved during the data warehouse life-cycle. Maintainability assessments evaluate the quality of a source or target data model with respect to its ability to support changes during an evolution process [55].

Calero et al. [56] listed metrics for measuring the complexity of a data warehouse star design that can be used to determine the level of effort required to maintain it. The defined complexity metrics are for the table, star, and schema levels. The higher the values, the more complex is the design of the star model, and the harder it is to maintain the model. The metrics are as follows:

- **Table Metrics**
  - Number of attributes of a table
  - Number of foreign keys of a table
- **Star Metrics**
  - Number of dimension tables of a star schema
  - Number of tables of a star schema that correspond to the number of dimension tables added to the fact table
  - Number of attributes of dimension tables of a star schema
  - Number of attributes plus the number of foreign keys of a fact table

- **Schema Metrics**
  - Number of fact tables of the star schema
  - Number of dimension tables of the star schema
  - Number of shared dimension tables that is the number of dimension tables shared for more than one star of the schema
  - Number of the fact tables plus the number of dimension tables of the star schema
  - Number of attributes of fact tables of the star schema
  - Number of attributes of dimension tables of the star schema

These metrics give an insight into the design complexity of the star data model, but there is no information in the Calero et al. paper on how to relate maintainability tests to these metrics. There is also a lack of work in developing metrics for other data models such as the snowflake model or relational data models.

2.3.3 Testing Data Management Product

Using the right product for data management is critical to the success of data warehouse systems. There are many categories of products used in data warehousing, such as DBMSs, big data management systems, data warehouse appliances, and cloud data warehouses that should be tested to ensure that it is the right technology for the organization. In the following paragraphs, we describe the existing approaches for performance, stress, and recovery testing of the data management products.

Performance and Stress Testing of Data Management Product

Performance testing determines how a product performs in terms of responsiveness under a typical workload [57]. The performance of a product is typically measured in terms of response time. This testing activity evaluates whether or not a product meets the efficiency specifications claimed by the organizations.

Stress testing evaluates the responsiveness of a data management product using an extraordinarily large volume of data by measuring the response time of the product. The goal is to assess whether or not the product performs without failures when dealing with a database with a size significantly larger than expected [7].

Because the demand for real-time data warehouses [2] and real-time analysis is increasing, performance and stress testing play a major role in data warehousing systems. Due to the growing nature of data warehousing systems, the data management product tolerance must be evaluated using unexpectedly large volumes of data. The product tolerance is the maximum volume of data the product can manage without failures and crashes. Comparing efficiency and tolerance characteristics of several data management products help data warehouse designers choose the appropriate technology for their requirements.

Performance tests are carried out on both real data or mock (fake) datasets with a size comparable with the average expected data volume [7]. However, stress tests are carried out on mock databases with a size significantly larger than the expected data volume. These testing activities are performed by applying different types of requests on the real or mock datasets. A number of queries are executed, and the responsiveness of the data management product is measured using standard database metrics. An important metric is the maximum query response time because query execution plays an important role in data warehouse performance measures. Both simple as well as multiple join queries are executed to validate the performance of queries on databases with different data volumes. Business users develop sample queries for performance testing with specified acceptable response times for each query [1].

Slutz [58] developed an automatic tool called Random Generation of SQL (RAGS) that stochastically generates a large number of SQL Data Manipulation Language (DML) queries that can be used to measure how efficiently a data management system responds to those queries. RAGS generates the SQL queries by parsing a stochastic tree and printing the query out. The parser stochastically generates the tree as it traverses the tree using database information (table names,

column names, and column types). RAGS generates 833 SQL queries per second that are useful for performance and stress testing purposes.

Most performance and stress testing approaches in the literature focus on DBMSs [59], but there is a lack of work in performance testing of data warehouse appliances or cloud data warehouses.

Recovery Testing of Data Management Product

This testing activity verifies the degree to which a data management product recovers after critical events, such as power disconnection during an update, network fault, or hard disk failures [7].

As data management products are the key components of any data warehouse systems, they need to recover from abnormal terminations to ensure that they present correct data and that there are no data loss or duplications.

Gunawi et al. [60] proposed a testing framework to test the recovery of cloud-based data storage systems. The framework systematically pushes cloud storage systems into 40,000 unique failures instead of randomly pushing systems into multiple failures. They also extended the framework to evaluate the expected recovery behavior of cloud storage systems. They developed a logic language to help developers precisely specify recovery behavior.

Most data warehousing systems that use DBMSs or other transaction systems rely on the Atomicity, Consistency, Isolation, Durability (ACID) properties [61] of database transactions to meet reliability requirements. Database transactions allow correct recovery from failures and keep a database consistent even after abnormal termination. Smith and Klingman [62] proposed a method for recovery testing of transaction systems that use ACID properties. Their method implements a recovery scenario to test the recovery of databases affected by the scenario. The scenario uses a two-phase transaction process that includes a number of service requests, and is initiated by a client application. The scenario returns to the client application without completing the processing of transaction and verifies whether or not the database has correctly recovered. The database status is compared to the expected status identified by the scenario.

2.3.4 Summary

Table 2.6 summarizes the testing approaches that have been applied to the sources and the target data warehouse that we discussed in this section. There are no methods proposed for the security testing of underlying data in data warehouse systems (as indicated by the shaded row in the table).

Table 2.6: Testing the Sources and the Target Data Warehouse

Test Category	Component	GuardianIQ [47]	Infromatica [48]	Hoberman [12]	Golfarelli and Rizzi [7]	MySQL plug-in [54]	Calero et. al, [56]	Slutz [58]	Gunawi et al. [60]	Smith and Klingman [62]
Functional	Underlying data Data model Product	✓	✓	✓	✓					
Structural	Data model					✓				
Usability	Data model			✓						
Maintainability	Data model							✓		
Performance	Product							✓		
Stress	Product							✓		
Recovery	Product								✓	
Recovery	Product									✓
Security	Underlying data									

We have identified the following open problems in testing the sources and the target data warehouse.

- In the area of *functional testing of underlying data*, there is no systematic way to assure the completeness of the test cases written/generated by different data quality assurance tools. We suggest that new research approaches be developed using a test adequacy criterion, such as number of fields, tables, or constraints as properties that must be exercised to constitute a thorough test.

- The data quality rules are not formally specified in the business requirements for *the functional testing of the underlying data*. The tester needs to bridge the gap between informal specifications and formal quality rules.

- It is difficult to design a generalized *data quality test* applicable to all data warehouse systems because data warehouse projects are typically designed for specific business domains. There are a number of generalized tools that let users define their desired data quality rules.

- The fault finding ability of the *data quality tests* are not evaluated in the literature. One can use mutation analysis techniques to perform this evaluation.

- No approach has been proposed for *the security testing of underlying data*. One can compare the access privileges defined for the target data with the ones defined for the corresponding source data to ensure that all the required protections are correctly observed in the target data warehouse.

- There is a lack of automatic *functional evaluation* techniques for data models. The existing functional evaluation activities are manually performed through human inspections.

- There is a lack of *structural evaluation* techniques for non-relational and dimensional schema. The existing approaches focus on the relational data schema.

- No formal technique has been proposed for *the usability testing of data models*. The proposed approaches are typically human inspections.

- In the area of *maintainability testing of data models*, a number of design complexity metrics have been proposed to get an insight into the capability of the data model to sustain changes. However, there is no information on how to design maintainability tests based on the metrics.

- The heterogeneous data involved in the data warehousing systems make *the performance and stress testing of data management products* difficult. Testers must use large datasets in order to perform performance and stress tests. Generating this voluminous data that reflect the real characteristics of the data is an open problem in these testing activities.

- There is a lack of work in *performance and stress testing of data warehouse appliance and cloud data warehouses*. The proposed approaches in the literature typically focus on testing DBMSs.

2.4 Testing ETL Process

This testing activity verifies whether or not the ETL process extracts data from sources, transforms it into an appropriate form, and loads it to a target data warehouse in a correct and efficient way. As the ETL process directly affects the quality of data transformed to a data warehouse [63], it has been the main focus of most data warehouse testing techniques [2]. In this section, we describe existing functional, performance, scalability, reliability, regression, and usability testing approaches as well as propose a new approach based on our experience in testing the ETL process in a health data warehouse [64].

2.4.1 Functional Testing of ETL Process

Functional testing of ETL process ensures that any changes in the source systems are captured correctly and propagated completely into the target data warehouse [2]. Two types of testing have been used for evaluating the functionality of ETL process, namely data quality and balancing tests.

Data Quality Tests

This testing activity verifies whether or not the data loaded into a data warehouse through the ETL process is consistent with the target data model and the organizational requirements. Data quality testing focuses on the quality assessment of the data stored in a target data warehouse. Data quality tests are defined based on a set of quality rules provided by domain experts. These rules are based on both domain and target data model specifications to validate the syntax and semantics of data stored in a data warehouse. For example, in our health data warehouse project, we use data quality rules from six clinical research networks, such as Achilles [45] and PEDSnet [65] to write test cases as queries to test the data quality. Achilles and PEDSnet define a number of rules to assess the quality of electronic health records, and report errors and warnings based on the data.

These quality rules are defined and periodically updated in a manner to fit the use and need of the health data users. Achilles defines its data quality rules as SQL queries while PEDSnet uses R. Table 2.7 shows examples of two data quality rules that are validated in Achilles.

Table 2.7: Examples of Achilles Data Quality Rules

Rule_id	Data Quality Rule	Status	Description
19	Year of birth should not be prior to 1800	warning	Checks whether or not year of birth is less than 1800
32	Percentage of patients with no visits should not exceed a threshold value	notification	Checks whether or not the percentage of patients that have no visit records is greater than 5

Note that the tools described in Section 4.1.1 to test the quality of the underlying data in a data warehousing system can also be used to execute data quality tests for the ETL process. The difference is that in the context of ETL testing, the tools have a different purpose, which is to test any time data is added or modified through the ETL process.

Balancing Tests

Balancing tests ensure that the data obtained from the source databases is not lost or incorrectly modified by the ETL process. In this testing activity, data in the source and target data warehouse are analyzed and differences are reported.

The balancing approach called *Sampling* [8] uses source-to-target mapping documents to extract data from both the source and target tables and store them in two spreadsheets. Then it uses the *Stare and Compare* technique to manually verify data and determine differences through viewing or *eyeballing* the data. Since this task can involve the comparison of billions of records, most of the time, a few number of the entire set of records are verified through this approach.

IBM QuerySurge [8] is a commercial tool that was built specifically to automate the balancing tests through query wizards. The tool implements a method for fast comparison of validation query results written by testers [66]. The query wizards implement an interface to make sure that minimal effort and no programming skills are required for developing balancing tests and obtaining

results. The tool compares data based on column, table, and record count properties. Testers select the tables and columns to be compared in the wizard. The problem with this tool is that it only compares data that is not modified during the ETL transformation, which is claimed to be 80% of data. However, the goal of ETL testing should also be to validate data that has been reformatted and modified through the ETL process.

Another method is *Minus Queries* [8] in which the difference between the source and target is determined by subtracting the target data from the source data to show existence of unbalanced data. The problem with this method is the potential for false positives. For example, as many data warehouses keep historical data, there may be duplicate records in the target data warehouse corresponding to the same entity and the result might report an error based on the differences in number of records in the source and the target data warehouse. However, these duplications are actually allowed in the target data warehouse.

In Chapter 5 we describe an approach for generating balancing tests from source-to-target mappings that are derived from the ETL scripts under test.

2.4.2 Performance, Stress, and Scalability Testing of ETL Process

Performance tests assess whether or not the entire ETL process is performed within the agreed time frames by the organizations [67]. The goal of performance testing of ETL is to assess ETL processing time under typical workloads that are comparable with the average expected data volume [6].

Stress tests also assess the ETL processing time but under a workload which is significantly larger than the expected data volume. The goal of stress testing of ETL is to assess ETL tolerance by verifying whether or not it crashes or fails when dealing with an extraordinarily large volume of data.

Scalability testing of ETL is performed to assess the process in terms of its capability to sustain further growth in data warehouse workload and organizational requirements [1, 67]. The goal of scalability testing is to ensure that the ETL process meets future needs of the organization.

Mathen [1] stated that this growth mostly includes an increase in the volume of data to be processed through the ETL. As the data warehouse workload grows, the organizations expect ETL to sustain extract, transform, and load times. Mathen [1] proposed an approach to test the scalability of ETL by executing ETL loads with different volumes of data and comparing the times used to complete those loads.

In all of these testing activities, the processing time of ETL is evaluated when a specific amount of data is extracted, transformed, and loaded into the data warehouse [7]. The goal is to determine any potential weaknesses in ETL design and implementation, such as reading some files multiple times or using unnecessary intermediate files or storage [1]. The initial extract and load process, along with the incremental update process must be evaluated through these testing activities.

Wyatt et al. [68] introduced two primary ways to measure the performance of ETL, i.e. time-based and workload-based. In the time-based method, they check if the ETL process was completed in a specific time frame. In the workload-based approach they test the ETL process using a known size of data as test data, and measure the time to execute the workload. Higher performing ETL processes will transfer the same volume of data faster. The two approaches can be blended to check if the ETL process was completed in a specific time frame using a specific size of data.

The above testing approaches test the entire ETL process under different workload conditions. However, the tests should also focus on the extraction, transformation, and load components separately and validate each component under specific workloads. For example, the performance testing of the extraction component verifies whether or not a typical sized data can be extracted from the sources in an expected time frame. Applying the tests separately on the constituent ETL components and procedures helps localize existing issues in the ETL design and implementation, and determine the areas of weaknesses. The weaknesses can be addressed by using an alternative technology, language, algorithm, or intermediate files. For example, consider the performance issue in the *Load* component of ETL, which incorrectly uses the full mode and loads the entire data every time instead of loading only the new added or modified data. In this case, the execution time of the *Load* component is considerably longer than the *Extract* and *Transform* components. This

problem can be localized if we apply the tests on the individual components instead of on the entire ETL process.

2.4.3 Reliability Testing of ETL Process

This type of testing ensures the correctness of the ETL process under both normal and failure conditions [68]. Normal conditions represent situations in which there are no external disturbances or unexpected terminations in the ETL process. To validate the reliability of the ETL process under normal conditions, we want to make sure that given the same set of inputs and initial states, two runs of ETL will produce the same results. We can compare the two result sets using properties such as completeness, consistency, and validity.

Failure conditions represent abnormal termination of ETL as a result of loss of connection to a database or network, power failure, or a user terminating the ETL process. In such cases the process should be able to either complete the task later or restore the process to its starting point. To test the reliability of the ETL process under abnormal conditions, we can simulate the failure conditions and compare the results from a failure run with the results of a successful run to check if the results are correct and complete. For example, we should check that no records were loaded twice to the target data warehouse as a result of the failure condition followed by re-running the ETL process.

Most ETL implementations indirectly demonstrate reliability features by relying on the ACID properties of DBMSs [68]. If DBMSs are used in a data warehousing system to implement the sources or the target data warehouse components, these component recover from problems in the Extract, Transform, or Load processes. However, there are data management products other than DBMSs used in the data warehousing systems that do not support ACID properties (e.g. Google Cloud Bigtable [69] that is a NoSQL data management product). In such cases, reliability needs to be addressed separately and appropriate test cases must be designed.

Note that balancing tests may be performed to compare the two result datasets in both normal and abnormal conditions to verify the proposed properties in addition to other reliability tests.

2.4.4 Regression Testing of ETL Process

Regression tests check if the system still functions correctly after a modification. This testing phase is important for ETL because of its evolving nature [7]. With every new data warehouse release, the ETL process needs to evolve to enable the extraction of data from new sources for the new applications. The goal of regression testing of ETL is to ensure that the enhancements and modifications to the ETL modules do not introduce new faults [70]. If a new program is added to the ETL, interactions between the new and old programs should be tested.

Manjunath et al. [71] automated regression testing of ETL to save effort and resources with a reduction of 84% in regression test time. They used Informatica [48] to automatically generate test cases in SQL format, execute test cases, and compare the results of the source and target data. However, their approach uses the *retest all* [72] strategy, which re-runs the entire set of test cases for regression testing of the ETL process. Instead, they could use *regression test selection* [72] techniques to run a subset of test cases to test only the parts of ETL that are affected by project changes. These techniques classify the set of test cases into retestable and reusable tests for regression testing purposes in order to save testing cost and time. A retestable test case tests the modified parts of the ETL process and needs to be re-run for the safety of regression testing. A reusable test case tests the unmodified parts of the ETL process and needs not to be re-run, while it is still valid [73].

Mathen [1] proposed to perform regression testing by storing test inputs and their results as expected outputs from successful runs of ETL. One can use the same test inputs to compare the regression test results with the previous results instead of generating a new set of test inputs for every regression test [1].

2.4.5 Usability Testing of ETL Process

The ETL process consists of various components, modules, databases, and intermediate files with different technologies, DBMSs, and languages that require many prerequisites and settings to be executed on different platforms. ETL is not a one-time process; it needs to be executed

frequently or any time data is added or modified in the sources. As a result, it is important to execute the entire process with configurations that are easy to set up and modify.

Usability testing of ETL process assesses whether or not the ETL process is easy to use by the data warehouse implementer. This testing activity determines how easy it is to configure and execute ETL in a data warehouse project.

We suggest to assess the usability of the ETL process by measuring the manual effort involved in configuring ETL in terms of time. The configuration effort includes (1) providing connection information to the sources, data staging area, or target data warehouse, (2) installing prerequisite packages, (3) pre-processing of data before starting the ETL process, and (4) human interference to execute jobs that run each of the Extract, Transform, and Load components. We can also do a survey with different users that gives us more information about the difficulty level.

2.4.6 Summary

Table 2.8: Testing Extract, Transform, Load (ETL)

Testing Category	GuardianIQ [47]	Informatica [48]	QuerySurge [8]	Wyatt et al. [68]	Mathen [1]	Manjunath et al. [71]
Functional	✓	✓	✓			
Performance				✓		
Stress						
Scalability					✓	
Reliability						
Regression						✓
Usability						

Table 2.8 summarizes the existing approaches to test different aspects of the ETL process. As can be seen from the table, scalability, reliability, and usability testing were not reported in the

literature even though they are critical for a comprehensive testing of the process. We identified the following open problems in ETL testing. We summarize areas and ideas for future investigation.

- In *the functional testing of ETL*, there is not a systematic way to assure the completeness of the test cases written/generated as a set of queries. As with the functional testing of underlying data, we can use appropriate test adequacy criteria to evaluate and create a thorough test.

- There is a lack of systematic techniques to generate mock test inputs for *the functional testing of the ETL process*. We can use input space partitioning techniques to generate test data for all the equivalent classes of data. Current tools generate random test data with not much similarity with the characteristics of real data.

- The fault finding ability of *the balancing tests* is not evaluated in the surveyed approaches. We can use mutation analysis for this evaluation.

- In *the performance, stress, and scalability testing of ETL*, the existing approaches test the entire ETL process under different workloads. We can apply tests to the individual components of ETL to determine the areas of weaknesses.

- The heterogeneous data involved in the data warehousing systems make *the performance, stress, and scalability testing of the ETL process* difficult. Testers must use large heterogeneous datasets in order to perform tests.

- The existing ETL implementations rely on ACID properties of transaction systems, and ignore *the reliability testing of the ETL process*. We can perform the balancing tests proposed in Chapter 5 to compare the results of the ETL process under normal conditions with the ones under abnormal conditions to verify the properties, namely, completeness, consistency, and syntactic validity.

- No approach has been proposed to *test the usability of the ETL process*. We define this testing activity as the process of determining whether or not the ETL process is easy to use by the

data warehouse implementer. One can test the usability of the ETL process by assessing the manual effort involved in configuring ETL that is measured in terms of time.

2.5 Testing Front-end Applications

Front-end applications in data warehousing are used by data analyzers and researchers to perform various types of analysis on data and generate reports. Thus, it is important to test these applications to make sure the data is correctly, effectively, and efficiently presented to the users.

2.5.1 Functional Testing of Front-end Applications

This testing activity ensures that the data is correctly selected and displayed by the applications to the end-users. The goal of testing the functionality of the front-end applications is to recognize whether the analysis or end result in a report is incorrect, and whether the cause of the problem is the front-end application rather than the other components or processes in the data warehouse.

Golfarelli and Rizzi [7] compared the results of analyses queries displayed by the application with the results obtained by executing the same queries directly (i.e., without using the application as an interface) on the target data warehouse. They suggested two different ways to create test cases as queries for functionality testing. In a black-box approach, test cases are a set of queries based on user requirements. In a white-box approach, the test cases are determined by defining appropriate coverage criteria for the dimensional data. For example, test cases are created to test all the facts, dimensions, and attributes of the dimensional data.

The approaches proposed by Golfarelli and Rizzi are promising. In our project, we have used Achilles, which is a front-end application that performs quality assurance and analysis checks on health data warehouses in the OMOP [10] data model. The queries in Achilles are executable on OMOP.

The functional testing of the front-end applications must consider all possible test inputs for adequate testing. As it is impossible to test the functionality of the front-end applications with all

possible test inputs, we can use systematic input space partitioning [74] techniques to generate test data.

2.5.2 Usability Testing of Front-end Applications

Two different aspects of usability of front-end applications need to be evaluated during usability testing, namely, ease of configuring and understandability.

First, we must ensure that the front-end application can be easily configured to be connected to the data warehouse. The technologies used in the front-end application should be compatible with the ones used in the data warehouse; otherwise, it will require several intermediate tools and configurators to use the data warehouse as the application's back-end. For example, if an application uses JDBC drivers to connect to a data warehouse, and the technology used to implement the data warehouse does not support JDBC drivers, it will be difficult to connect the front-end apps to the back-end data warehouse. We may need to re-implement parts of the application that set up connections to the data warehouse or change the query languages that are used. We suggest evaluating this usability characteristic by measuring the time and effort required to configure the front-end application and connect it to the target data warehouse.

Second, we must ensure that the front-end applications are understandable by the end-users [67], and the reports are represented and described in a way that avoids ambiguities about the meaning of the data. Existing approaches to evaluate the usability of generic software systems [75] can be used to test this aspect of front-end applications. These evaluations involve a number of end-users to verify the application. Several instruments are utilized to gather feedback from users on the application being tested, such as paper prototypes [76], and pre-test and post-test questionnaires.

2.5.3 Performance and Stress Testing of Front-end Applications

Performance testing evaluates the response time of front-end applications under typical workloads, while stress testing evaluates whether the application performs without failures under significantly heavy workloads. The workloads are identified in terms of number of concurrent users,

data volumes, and number of queries. The tests provide various types of workloads to the front-end applications to evaluate the application response time.

Filho et al. [77] introduced the OLAP Benchmark for Analysis Services (OBAS) that assesses the performance of OLAP analysis services responsible for the analytical process of queries. The benchmark uses a workload-based evaluation that processes a variable number of concurrent executions using variable-sized dimensional datasets. It uses the Multidimensional Expressions (MDX) [78] query language to perform queries over multidimensional data cubes.

Bai [79] presented a performance testing approach that assesses the performance of reporting systems built using the SQL Server Analysis Services (SSAS) technology. The tool uses the MDX query language to simulate user requests under various cube loads. Metrics such as average query response time and number of queries answered are defined to measure the performance of these types of reporting services.

The above two tools compare the performance of analysis services such as SSAS or Pentaho Mondrian. However, the tools do not compare analysis services that support communication interfaces other than XML for Analysis (XMLA), such as OLAP4J.

2.5.4 Summary

Table 2.9: Testing Front-end Applications

Test Category	Golfarelli and Rizzi [7]	Filho et al. [77]	Bai [79]
Functional	✓		
Usability			
Performance		✓	✓
Stress			

Table 2.9 summarizes existing approaches that test front-end applications in data warehousing systems. Although it is important to assess usability, none of the approaches addressed usability testing. Stress testing of the front-end applications is not reported in the surveyed approaches.

Below is a summary of the open problems in testing front-end applications, and ideas for future investigation.

- Existing approaches proposed for *functionality testing of the front-end applications* compare the results of queries in the application with the ones obtained by directly executing the same queries on the target data warehouse.

- *Functional testing of the front-end applications* must consider all types of test inputs. We can use input space partitioning techniques to generate test data for this testing activity.

- To support *usability testing of front-end applications*, we define a new aspect of testing to assess how easy it is to configure the application. We can test this aspect of usability by measuring the manual effort involved in configuring the front-end application in terms of time.

- The voluminous data involved in the data warehousing systems makes *performance and stress testing of the front-end applications* difficult. Testers must use large datasets in order to perform realistic tests.

Chapter 3

Motivating Example

We use the same enterprise health data warehouse to motivate our approach. The ETL process uses transformation rules that are described in the documents provided by the data warehouse designers. Tables 3.1 and 3.2 show some of the transformation rules that are used to generate the *Location*, *Person*, and *Visits* tables in the target health data warehouse. The rules include the names of the corresponding source and target tables, the source and target columns, and selection condition. The selection condition determines the portion of the source table(s) that is transformed to the target data warehouse. The rules use table-level, record-level, and attribute-level mappings that are one-to-one, many-to-one, or many-to-many.

3.1 One-to-one mappings

In one-to-one mappings, each table, record, or attribute in the target data warehouse maps to a single table, record, or attribute in the source. This mapping can be of three types: one-to-one table mapping, one-to-one record mapping, and one-to-one attribute mapping.

In a one-to-one table mapping, the table in the target data warehouse corresponds to a single table in the source. Table 3.1 shows how the *Location* table is obtained from a single table called *Address* in the source.

In a one-to-one record mapping, a record in the target table corresponds to a single record in the source. Table 3.1 shows how each record in the *Location* table is obtained from a single record in the *Address* table in the source.

In a one-to-one attribute mapping, the attribute in the target corresponds to a single attribute in the source. Table 3.2 shows the *person_id* attribute is obtained from a single attribute called *Patient_Key* in the source table.

Table 3.1: Transforming Single Source Table to Single Target Table

Source table:: column	Target table:: column	Selection Condition
Address:: *Address_Key*	*Location*:: *location_id*	transform all the new addresses (*Year*>2000)
Address:: *Address_string*	*Location*:: *address_1*	transform all the new addresses (*Year*>2000)
Address:: *City*	*Location*:: *city*	transform all the new addresses (*Year*>2000)
Address:: *Postal_code*	*Location*:: *zip*	transform all the new addresses (*Year*>2000)

Table 3.2: Transforming Multiple Source Tables to Single Target Table

Source table:: column	Target table:: column	Selection Condition
Patient:: *Patient_Key*	*Person*:: *person_id*	transform all the current patients
Patient:: *Name*	*Person*:: *person_name*	transform all the current patients
Patient:: *Day_of_birth*, *Month_of_birth*, *Year_of_birth*,	*Person*:: *date_of_birth*	transform all the current patients
Patient:: *Address_Key*, *Address*:: *Address_Key*	*Person*:: *location_id*	transform all the current patients with their new addresses (*Year*>2000)
Patient:: *Sex*, *Concept*:: *Concept_id*, *Concept_code*, *Domain_id*	*Person*:: *gender_value*, *Person*:: *gender_concept_id*	transform all the patients with their sex using *Concept* values *Female*, *Male*, or *Other*

Table 3.3: Transforming Single Source Table to Single Target Table by Many-to-one Record Aggregation

Source table:: column	Target table:: column	Selection Condition
Patient:: *Patient_Key*	*Number_of_Patients*:: *Record_count*	transform number of patients

Table 3.4: Transforming Single Source Table to Single Target Table by Many-to-many Record Aggregation

Source table:: column	Target table:: column	Selection Condition
Visit:: *Visit_Key*	*Visits*:: *Visits_per_year*	transform number of visits per year
Visit:: *Visit_date*	*Visits*:: *Year*	transform the year from the *visit_date*

3.2 Many-to-one mappings

In many-to-one mappings, a table, record, or attribute in the target data warehouse is obtained from multiple tables, records, or attributes in the source. This mapping can be of three types: many-to-one table mapping, many-to-one record mapping, and many-to-one attribute mapping.

In many-to-one table mappings, a table in the target data warehouse is obtained from different types of operations applied to multiple tables in the source, such as join and union. Table 3.2 shows how the *Person* table is obtained from a join of three tables called *Patient*, *Address*, and *Concept*.

In many-to-one record mappings, a record in the target table is obtained from different types of aggregation operations applied to multiple records in the source. In Table 3.3, there is a single record in a target table with a *Record_count* attribute that reflects the number of records in the *Patient* table in the source.

In many-to-one attribute mappings, an attribute in the target data warehouse is obtained from a combination of attributes in the source table(s). Different operators may be used to combine the source attributes. These operators include arithmetic (e.g., addition and subtraction), string (e.g., concatenation), date (e.g., date_diff and date_add), statistical (e.g., mean and standard deviation), and other aggregation operators (e.g., min, max, and sum). Table 3.2 shows how the *date_of_birth* attribute is a concatenation of three attributes in the source table (*Year_of_birth*, *Month_of_birth*, and *Day_of_birth*).

3.3 Many-to-many mappings

In many-to-many mappings, multiple tables, records, or attributes in the source are combined to create multiple tables, records, or attributes in the target data warehouse. This mapping can be of three types: many-to-many table mapping, many-to-many record mapping, and many-to-many attribute mapping.

In many-to-many table mappings, multiple tables in the source are used to create multiple tables in the target data warehouse. As shown in Table 3.2, the *Patient*, *Address*, and *Concept* tables in the source are used to create the *Person* and *Location* tables in the target.

In many-to-many record mappings, multiple records in the source are aggregated to create multiple records in the target table. Table 3.4 shows how the records in the *Visits* table are the aggregation of records in the source table, which are grouped by the *Year* attribute.

In many-to-many attribute mappings, multiple attributes in the source are used to create multiple attributes in the target data warehouse. Table 3.2 shows how the *Sex*, *Concept_code*, *Concept_id*, and *Domain_id* attributes in the source create the *gender_value* and *gender_concept_id* attributes in the target data warehouse.

3.4 Need for balancing tests

When the ETL process is executed, faults in any of the ETL phases as well as incorrect configuration settings can lead to incorrect or incomplete data in the target data warehouse. For example, considering the first row of an intermediate CSV file as the header by setting a parameter `skip_first_row parameter` to TRUE, when the first row includes a real record leads to loss of data. Incorrectly setting the `from_date` and `to_date` parameters for an incremental ETL job can result in a mismatch of these parameters in the three phases. This can lead to data loss because the portion of the data that is extracted will not be transformed and loaded to the target data warehouse. An unintentional deletion of intermediate files on Google Cloud Storage by data warehouse users can lead to loss of data. Using incremental mode in the extraction and transformation phases but full mode in the load phase will replace the entire target table with a portion of the data.

Therefore, various checks must be performed to prevent any data loss or modifications. Validation needs to support the checking of transformations for all the types of mappings. The following are examples of checks that must be performed:

1. The target table has the correct name (*Location*), and attributes (*location_id*, *address_1*, *city*, and *zip*).

2. The target table has the correct name (*Person*), and attributes (*person_id*, *person_name*, and *location_id*, *gender_value*, *gender_concept_id*).

3. The *Person* table must not lose any current patients.

4. The *Location* table must not lose any addresses of patients admitted after the year 2000.

5. The target table must include the new addresses of patients, i.e., addresses of patients admitted after the year 2000.

6. Only valid values should be transformed to the *gender* attribute in the target table.

7. The target table must not contain any patient that does not satisfy the selection criteria.

8. No attribute value of transformed patient and address records should be corrupted.

Chapter 4

Balancing Properties

In this chapter we identify the types of discrepancies that may arise between the source and the target data due to an incorrect ETL process on the basis of which we define a set of generic properties that can be applied to all data warehouses, namely, *completeness*, *consistency*, and *syntactic validity*. Below we describe each of the properties along with the corresponding assertions and examples of how the properties are verified for different mapping types.

4.1 Completeness

This property ensures that all the records in the sources that must be transformed and loaded into the warehouse are actually present in the warehouse. A data warehouse may encounter errors during the loading process and some data may get uploaded multiple times or not at all. The completeness property requires a comparison based on record counts on the source table(s) and target data warehouse tables. To verify the completeness property, we match the record counts and the counts of distinct records.

4.1.1 Record count match

In this check, we compare record counts in the target table with the record counts of the corresponding table(s) in the source. If there is a *one-to-one table mapping* and *one-to-one record mapping* between a source and a target table, the record counts are identical provided there is no selection condition. However, the test assertion must take into account the selection condition. For example, according to the transformation rule in Table 3.2, only current patients are transformed to the target table. So, the test assertion should compare the number of records in the target table with the number of persons in the source table whose *current* attribute value is TRUE.

If there is a *one-to-one table mapping* and *many-to-one record mapping* between a source table and a target table, then the number of records in the target table should be equal to 1. For exam-

ple, the *Number_of_Patients* table in Table 3.3 has only one record, which indicates the number of records in the source table. If there is a *one-to-one table mapping* and *many-to-many record mapping* between a source table and a target table, then the number of records in the target table should be equal to the distinct number of the attribute values that is grouped by in the aggregation operation in the source table subject to the selection conditions. For example, in Table 3.4 the number of records in the *Visits* table must be equal to the distinct number of *Year* values.

$Target\_record\_count = Distinct\ Count(source\_attribute)$

where the $source\_attribute$ is the one that is grouped by in the aggregation operation.

If there is a *many-to-one table mapping* and *one-to-one record mapping* between the source tables and a target table, the record counts in the target must match the number of records resulting from the operations applied to the source tables. Calculating the number of records in database operations is non-trivial. For example, join operations may include a combination of different types of joins (i.e., left join, right join, and inner join) with join conditions and other constraints. We need to perform the entire join operation to calculate the corresponding number of records in the source tables to the number of records in the target table. We can also use upper and lower bounds for the numbers of records resulting from join operations:

$Upper\ bound = N * M$

$Lower\ bound = 0$

$Lower\ bound\ in\ left\ join = N$

$Lower\ bound\ in\ right\ join = M$

where *N* and *M* are the number of records in the left and right table of the join respectively.

If there is a *many-to-one table mapping* and *many-to-one record mapping* between the source tables and a target table, the record counts in the target table must be equal to 1. If there is a *many-to-one table mapping* and *many-to-many record mapping* between multiple source tables and a target table, then the number of records in the target table should be equal to the number of distinct attribute values that are grouped by the aggregation operation in the source tables subject to the selection and join conditions.

4.1.2 Distinct record count match

The number of distinct records in the target table must match the corresponding ones in the source table(s). This assertion is useful in data warehouses that keep the history of the data. For example, as the old patient records are kept in the target data warehouse and there are duplicate records maintained for each patient, the number of distinct patient records in the target *Person* table should be compared with the number of patient records in the source *Patient* table. This assertion also should take into account the different types of mappings. For a *one-to-one table mapping* and *one-to-one record mapping*, the number of distinct records in the target table is compared with the number of distinct records in a single source table subject to the selection conditions. For a *many-to-one table mapping* and *one-to-one record mapping*, the number of distinct records in the target is compared with the number of distinct records resulting from a join of multiple source tables. In *many-to-many record mapping*, the number of distinct records in the target table should be equal to the number of distinct attribute values that are grouped by the aggregation operation in the source tables subject to the selection and join conditions.

4.2 Consistency

This property ensures that the attribute contents in the target table conform to the corresponding ones in the source table(s) based on the transformation specifications.

The consistency property compares the attribute contents of the source table(s) with those of the target data warehouse for the specifications stated in the transformation rules. It ensures semantic correctness during the transformation process. Since there may not be a simple one-to-one correspondence between a source and a target attribute, verifying this property is often challenging. Below we give examples to show how we check the consistency property.

4.2.1 Attribute value match

The values stored in a target attribute must be consistent with the values of the corresponding attribute(s) in the source schema. This type of check is applicable for all data types. The source

attributes may be in one or more source tables. In a *one-to-one attribute mapping*, the target attribute value is compared with a single attribute value in the source for equality, where equality for floating point values is checked by assessing whether the difference between the attribute values is within some error bounds. For example, the value of the *gender_value* attribute is compared with the value of a single *Concept_code* attribute for equality. This attribute is obtained by joining the *Patient* and *Concept* tables. In a *many-to-one attribute mapping*, an attribute in the target is a function of attribute(s) of the source schemas. In this case, the target attribute is compared with the source attributes for the operations, such as multiplication, concatenation, and summation defined in the transformation rules. For example, we should compare the value in the *date_of_birth* attribute with the values of the three attributes in the source tables to see if the target attribute is a correct concatenation of the source attributes.

4.2.2 Attribute constraint match

Some constraints in a source attribute must hold in its corresponding target attribute. In a *one-to-one attribute mapping* the constraint of a single source attribute should hold for its corresponding target attribute. For example, the *Patient_Key* attribute is a primary key in the *Patient* table with a *NOT NULL* constraint. This constraint should also hold for the corresponding *person_id* attribute in the target table based on the transformation specification. In a *many-to-one attribute mapping*, the constraints on all or some of the source attributes that are involved in the transformation should also hold for the target attribute depending on the transformation specifications. For example, the *Year_of_birth* attribute in the source table may have an additional constraint, "$Year\_of\_birth < date.now()$", which is identified in the source schema. We need to derive corresponding constraints that can be checked on the target attribute.

4.2.3 Outliers match

The min/max values of a numeric attribute in the source and target tables must match. In a *one-to-one attribute mapping*, the min/max value of the target attribute is compared with that of the corresponding attribute value in the source. For example, the *weight* attribute in the target table

has a corresponding *Weight* attribute in the source table. As only current patients are transformed, we compare the min/max *Weight* values with that of patients who have their *current* attribute equal to TRUE in the source table. In a *many-to-one attribute mapping*, the min/max of a target attribute value is compared with that of a combination of multiple attribute values in the source that use different operations. For example, the *BMI* attribute in the target table is calculated from the *Weight* and *Height* attributes in the source table. By this check we compare the min/max value for *BMI* with the min/max of ($Average\_Weight/(Average\_Height)^2$) value.

4.2.4 Average match

The average value of a numeric attribute in the target table must match the average value of the corresponding attribute in the source table. In a *one-to-one attribute mapping*, the average of the target attribute values is compared with the average of the corresponding attribute values in the source. For example, the *height* attribute in the target table has a corresponding *Height* attribute in the source table. We compare the average *Height* values with that of patients who have their *current* attribute equal to TRUE in the source table. In a *many-to-one attribute mapping*, the average of target attribute values is compared with that of a combination of multiple attribute values in the source that are created using different arithmetic, statistical, and other aggregation operations. For example, by this check we compare the average value for *BMI* with the average of ($Average\_Weight/(Average\_Height)^2$) value.

4.3 Syntactic validity

This property ensures that the syntax of attributes in the target table conforms to the syntax of their corresponding attributes in the source table(s). Below we provide examples of checks for syntactic validity.

4.3.1 Attribute data type match

The data type of an attribute with any data type in a target table must be consistent with the data type of its corresponding attribute/attribute(s) in the source. For example, in PostgreSQL, the *Name* attribute is of type `VARCHAR` but the corresponding attribute in the target table in Google BigQuery is *person_name* of type `STRING`. In a *one-to-one attribute mapping*, the type of the target attribute is compared to the type of the corresponding single attribute in the source. For example, we compare the data type of *person_name* in the target table with *Name* in the source table. In a *many-to-one mapping*, the type of target attribute depends on the operation applied to the source attributes to create the target attribute. For example, a target attribute may be defined as an average of three source attributes of type INTEGER should be of type FLOAT. Another example is that a target attribute created by concatenating a STRING source attribute with another attribute of type INTEGER should be of type STRING.

In certain data warehouses such as ours, the target data types are explicitly identified in the target data model specifications. In such cases, test assertions should compare the types of the target attributes with the ones defined in the target data model.

4.3.2 Attribute length match

The maximum allowed lengths for character-based attributes vary across DBMSs and data warehouse systems. For example, a `VARCHAR` attribute can take a maximum size of 8000 in MS SQL Server, as opposed to 4000 in Oracle. By matching the attribute length, we ensure that no information is lost in truncating data from the source to the target table. We compare the average length of attribute values in the target with the corresponding ones in the source for both *one-to-one* and *many-to-one* attribute mappings.

4.3.3 Attribute boundary match

The range allowed for numeric attributes differs across DBMSs and data warehouse systems. For example, an integer attribute's range is from -9,223,372,036,854,775,808 to

9,223,372,036,854,775,807 in Google BigQuery, while it can only take -2147483648 to +2147483647 in PostgreSQL. By matching the attribute's highest and lowest values, we ensure that information is not lost in truncating data from the source to the target system. Our target data warehouse is on Google BigQuery, which guarantees to cover the highest range of data for all its data types in comparison to the source dialects.

4.4 Completeness of the Properties

We define a set of properties to be checked in balancing tests. Satisfying these properties is a necessary but not sufficient condition for correctness of the ETL transformations. Due to the number of heterogeneous records involved in the real-world data warehouses, it is impossible to compare all the individual records/attributes in the target to the corresponding ones in the source. In order to make the testing process feasible, we defined properties on an aggregate basis. For example, we compare the average and outliers of the attribute values in the source with the corresponding ones in the target instead of analyzing all the individual values. As a result, there may be cases where the balancing properties are satisfied but there are faults remaining in the data.

Furthermore, we do not consider the relationships between multiple tables in defining the balancing properties. For example, we do not define properties to be checked to verify the foreign key constraints between multiple tables. Defining such properties for table integrity checks is the subject of future work.

Chapter 5

Approach

In this chapter we present an approach to create balancing tests to ensure that data obtained from the source databases is not lost or incorrectly modified by the ETL process. It is an approach that is independent of dialects used in the source and the target data warehouse. We automate the generation of test assertions to reduce the manual effort involved in generating balancing tests of ETL and enhance test repeatability. Section 5.1 presents our approach to extract the source-to-target mappings from the ETL transformation rules. Section 5.2 describes how we use these mappings to automatically generate balancing tests.

5.1 Identify Source-To-Target Mappings

The properties defined in Section 4 require the existence of source to target mappings [3] that describe how one or more attributes in the source tables are related to one or more attributes in the target data warehouse. These source-to-target mappings are derived from the ETL transformation rules [3] described in a specification document, comments inside the transformation scripts, spreadsheets, ER diagrams, or SQL scripts. In this work, we assume that the ETL transformation rules are documented in a structured manner. Identifying these mappings from unstructured documents requires Natural Language Processing [80] that is out of the scope of this work. In the data warehouse used in our evaluation, the ETL transformation rules are described as SQL view codes. This data warehouse includes only one-to-one record mappings. In our approach, we consider each table or attribute in the target and compare it with the corresponding table(s) or attribute(s) in the source. The original one-to-one and many-to-one transformations remain the same, but the many-to-many transformations become a set of many-to-one transformations. Thus, we are left with several one-to-one and many-to-one mappings. Below we describe how we identify different mapping types in our data warehouse.

Table 5.1: Mapping Table Structure along with the Assertions For The Mappings

Mappings	Source table	Condition	Source column	Target table	Target column	Source analysis query	Target analysis query	Assertion type	Assertion
1-1 table	Address	Year>2000		Location		RecordCount (Source table) WHERE Condition=TRUE	RecordCount (Target Table)	Record count match	If Source analysis value != Target analysis value THEN ERROR: Record count mismatched
*-1 table	[0]:Patient [1]:Concept Operation: LEFT JOIN	IsCurrent=1		Person		RecordCount (Source table[0]) WHERE Condition=TRUE	Distinct RecordCount (Target Table)	Distinct record count match	If Source analysis value != Target analysis value THEN ERROR: Distinct record count mismatched
*-1 attribute	Patient	IsCurrent=1	[0]:Weight [1]:Height operation: BMI	Person	BMI	operation(Source column[0::1]) From source table WHERE Condition=TRUE	target column from target table	attribute value match	If Target analysis value NOT IN Source analysis value THEN ERROR: attribute value mismatched
1-1 attribute	[0]:Patient [1]:Concept Operation: LEFT JOIN	IsCurrent=1	concept_code	Person	gender_value	Count(source column) From source table WHERE Condition=TRUE and ISNULL(source column)	Count(target column) From target table WHERE ISNULL(target column)	attribute constraint match	If Target analysis value != Source analysis value THEN ERROR: attribute NULL constraint mismatched
1-1 attribute	Address	Year>2000	Address_Key	Location	location_id	Min/Max(Source column) From source table WHERE Condition=TRUE	Min/Max(Target column) from target table	Outliers match lattribute boundaries match	If Source analysis value != Target analysis value THEN ERROR: Outliers mismatched
1-1 attribute	Address	Year>2000	City	Location	City	Avg (Source column) From source table WHERE Condition=TRUE	Avg (Target column) from target table	Average match	If Source analysis value != Target analysis value THEN ERROR: Average mismatched
*-1 attribute	Patient	IsCurrent=1	[0]:Year [1]:Month [2]:Day operation: Concat	Person	date_of_birth	Avg(length(Operation(Source_column[0::2]))) From source table WHERE Condition=TRUE	Avg(length(Target_column)) from target table	attribute length match	If Source analysis value != Target analysis value THEN ERROR: attribute length mismatched
*-1 attribute	Patient	IsCurrent=1	[0]:Year [1]:Month [2]:Day operation: Concat	Person	date_of_birth	Type(Operation (Source_column[0::2]))	Type(Target column)	attribute data type match	If Source analysis value NOT EQUIVALENT Target analysis value THEN ERROR: attribute data type mismatched

5.1.1 One-to-one table mapping

In this case, no SQL operations, such as join and union are applied to the source tables to create the target table. Considering the ETL transformation rules as SQL views, the name of the single source table corresponding to the target table is extracted using the *FROM* clause in the SQL statement. The selection condition in the *WHERE* clause determines the portion of the source table records that needs to be transformed to the target table. If there is no selection condition to create the target table, then all the records are transformed and both the source and the target tables are of the same cardinality. The source-to-target mapping row identified for this case has the following information:

[source_table, selection_condition, target_table].

As Table 5.1 shows, an example of a one-to-one table mapping row is:

[Address, (Year>2000), Location].

5.1.2 One-to-one attribute mapping

In this case, the target attribute maps to a single attribute in the source. In our ETL transformation rules, the single source attribute corresponding to the target attribute is extracted from the *AS* clause in a SQL statement. The word before *AS* is the source and the word after *AS* is the target attribute. The source-to-target mapping row identified for this case has the following information:

[source_table, selection_condition, source_attribute, target_table, target_attribute].

As Table 5.1 shows, an example of a one-to-one attribute mapping row is:

[Patient, (IsCurrent=1), Concept_code, Person, gender_value].

5.1.3 Many-to-one table mapping

In this case, some table-level SQL operations, such as join and union are applied to the source tables to create the target table. The names of the tables included in the operation clause of the SQL statement determine the corresponding source tables that are used to create a target table. The selection condition in the *WHERE* clause determines the portion of source tables that have contributed to the SQL operation. The source-to-target mapping row identified for this case has the following information:

[source_table[], selection_condition[], table_operation, target_table],

where *source_table[]* is an array of source table names that contributed to the *table_operation*, which is either join or union, to create *target_table*. The *selection_condition[]* array is a list of selection conditions for each of the source tables.
As Table 5.1 shows, an example of a many-to-one table mapping row is:

[[Patient, Concept], (IsCurrent=1), (Left Join), Person].

5.1.4 Many-to-one attribute mapping

In this case, arithmetic, string, and statistical operations are applied to the source attributes to create the target attribute. The multiple source attributes corresponding to the target attribute are extracted from the *AS* clause in a SQL statement. The names of the attributes included in the attribute-level operation clause, which appears before the *AS* clause of the SQL statement, determine the source attributes. The word after *AS* is the target attribute. The source-to-target mapping row identified for this case has the following information:

[source_table, selection_condition, source_attribute[], attribute_operation, target_table, target_attribute],

where *source_attribute[]* is an array of attributes that are combined through the *attribute_operations*, which can be arithmetic, string, and statistical, to create *target_attribute*.

As Table 5.1 shows, an example of a many-to-one attribute mapping row is:

[Patient, (IsCurrent=1), [Weight, Height], (BMI), Person, BMI],

5.2 Generate Balancing Tests

Test assertions compare the test outputs with the expected outputs to verify whether a test passed or failed [81]. Balancing test assertions compare the data in the source with the corresponding data in the target data warehouse for the defined properties.

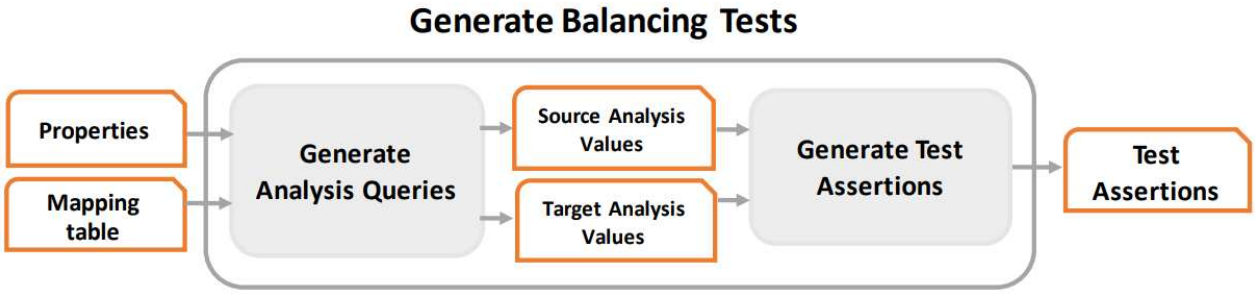

Figure 5.1: Balancing Test Generator Architecture

We use the mapping table generated in Section 5.1 to generate the test assertions. Figure 5.1 shows the architecture of our balancing test generator. The first component uses the mapping table to generate analysis queries for the source and target. Analysis queries are SQL queries that are executed on the source and target tables and attributes to generate the values used for equality checks in the test assertions. These values are called *analysis values* in this chapter. The second component produces test assertions that use the analysis values to determine whether or not the property is satisfied.

5.2.1 Generate Analysis Queries

Algorithm 1 shows how we generate the analysis queries for the source and target using mapping table as input.

Algorithm 1 Analysis Query Generator Function

```
1: function ANALYSISQUERYGENERATOR(mapping, property)
2:     switch property do
3:         case RecordCountMatch
4:             if mapping is one-to-one then
5:                 target_analysis_query = "SELECT Record_count(mapping.target_table)"
6:                 source_analysis_query= "SELECT Record_count(mapping.source_table)
7:                 WHERE mapping.selection_condition = TRUE"
8:         case DistinctRecordCountMatch
9:             if mapping is many-to-one and mapping.table_operation="LEFT JOIN" then
10:                target_analysis_query = "SELECT Distinct_record_count(mapping.target_table)"
11:                source_analysis_query= "SELECT Record_count(mapping.source_table[0])
12:                WHERE mapping.selection_condition[0] = TRUE"
13:        case AttributeValueMatch
14:            target_analysis_query = "SELECT ALL(mapping.target_attribute)"
15:            if mapping is one-to-one then
16:                source_analysis_query= "SELECT ALL(mapping.source_attribute)"
17:            else source_analysis_query= "SELECT ALL(mapping.attribute_operation(mapping.source_attributes))"
18:        case AttributeConstraintMatch
19:            target_analysis_query = "SELECT Count(ISNULL(mapping.target_attribute))"
20:            if mapping is one-to-one then
21:                source_analysis_query= "SELECT Count(ISNULL(mapping.source_attribute))"
22:            else source_analysis_query= "SELECT Count(ISNULL(mapping.attribute_operation(mapping.source_attributes)))"
23:        case OutliersMatch||AttributeBoundaryMatch
24:            if Type(target_attribute) is Numeric then
25:                target_analysis_query = "SELECT Min/Max(mapping.target_attribute)"
26:                if mapping is one-to-one then
27:                    source_analysis_query= "SELECT Min/Max(mapping.source_attribute)"
28:                else source_analysis_query= "SELECT Min/Max(mapping.attribute_operation(mapping.source_attributes))"
29:        case AverageMatch
30:            if Type(target_attribute) is Numeric then
31:                target_analysis_query = "SELECT Avg(mapping.target_attribute)"
32:                if mapping is one-to-one then
33:                    source_analysis_query= "SELECT Avg(mapping.source_attribute)"
34:                else source_analysis_query= "SELECT Avg(mapping.attribute_operation(mapping.source_attributes))"
35:        case AttributeLengthMatch
36:            if Type(target_attribute) is Textual then
37:                target_analysis_query = "SELECT Avg(length(mapping.target_attribute))"
38:                if mapping is one-to-one then
39:                    source_analysis_query= "SELECT Avg(length(mapping.source_attribute))"
40:                else source_analysis_query= "SELECT Avg(length(mapping.attribute_operation(mapping.source_attributes)))"
41:        case AttributeDataTypeMatch
42:            target_analysis_query = "SELECT DATA_TYPE(mapping.target_attribute)"
43:            if mapping is one-to-one then
44:                source_analysis_query= "SELECT DATA_TYPE(mapping.source_attribute)"
45:            else source_analysis_query= "SELECT DATA_TYPE(mapping.attribute_operation(mapping.source_attributes))"
46:    return source_analysis_query, target_analysis_query
```

One-to-one table mappings. For each of the one-to-one table mapping rows, we generate analysis queries for the record count match property. The analysis query for the target data warehouse returns the number of records in the target table. As the corresponding source table is a single table, the analysis query for the source returns the number of records in that source table subject to the selection condition.

Many-to-one table mappings. For each of the many-to-one table mapping rows, we generate analysis queries for the distinct record count match property. The analysis query for the target returns the distinct number of records in the target table.

The analysis query for the source returns the number of records in the leftmost table in a *LEFT JOIN*, and the number of records in the rightmost table in a *RIGHT JOIN* operation, which are subject to the selection condition.

For more complicated cases, such as *INNER JOIN* operation or a combination of different types of *JOIN* operations, it is not possible for the analysis query of the source to count the number of records obtained from the join unless it performs all the join operations. In our case study, we only have a chain of *LEFT JOIN* operations. In this case, the number of records in the leftmost table is the minimum number of records that will be returned. As a result, we compare the number of distinct records in the target table with the number of records in the leftmost table of the join operation subject to the selection condition.

Attribute mappings. For each of row in the mapping table, we generate source and target analysis queries for the following properties.

Attribute value match. This analysis is performed for the target attributes with any data type. If it is a one-to-one attribute mapping, then the query returns all the values of the source and target attributes. Otherwise, if it is a many-to-one attribute mapping, then the target analysis query returns all the values of target attribute and the source analysis query returns all the values of the corresponding source attributes by applying the operation: *ALL(operation(source columns))*.

Attribute constraint match. We perform this analysis for the source attribute with *"NOT NULL"* constraint. If it is a one-to-one attribute mapping, then the query returns the number of NULL values of the source and target attributes. If it is a many-to-one attribute mapping, then the target analysis query returns the number of NULL values of the target attribute and the source analysis

query returns the number of NULL values of the corresponding source attributes by applying the operation: *Count(ISNULL(operation(source columns)))*.

Outlier match and attribute boundary match. If it is a one-to-one attribute mapping, then the query returns the maximum and minimum values of the source and target attributes. If it is a many-to-one attribute mapping, then the target analysis query returns the minimum and maximum of the target attribute and the source analysis query returns the minimum and maximum of corresponding source attributes by applying the operation: *min/max(operation(source columns))*.

Average match. If it is a one-to-one attribute mapping, then the query returns the average value of the source and target attributes. If it is a many-to-one attribute mapping, then the target analysis query returns the average of the target attribute and the source analysis query returns the average of corresponding source attributes by applying the operation *(Avg(operation(source columns)))*.

Attribute length match. If it is a one-to-one attribute mapping, then the query returns the average length of the source and target attributes. If it is a many-to-one attribute mapping, then the target analysis query returns the average length of target attribute and the source analysis query returns the average length of the corresponding source attributes by applying the operation: *Avg(length(operation(source columns)))*.

Attribute data type match. If it is a one-to-one attribute mapping, then the query returns the type of the source and target attributes. If it is a many-to-one attribute mapping, then the target analysis query returns the type of target attribute and the source analysis query returns the type of the corresponding source attributes by applying the operation: *DATA_TYPE(operation(source columns))*.

Table 5.1 shows examples of the analysis queries for each of the source-to-target mappings. Executing the scripts containing `source_analysis_queries` on the source and `target_analysis_queries` on the target of the data warehouse generates the source and target anal-

ysis values respectively. However, each of these datasets uses different dialects. As a result, we need to transform the analysis queries in SQL format to be executable with other dialects. We use an open source translator called SqlRender [82] from the OHDSI community, which transforms queries in SQL format into the desired dialect format. SqlRender did not originally support transformations into Google BigQuery, which is one of the dialects used in the health data warehouse. We extended SqlRender to add translation into Google BigQuery. We shared our extension [83] with OHDSI open source community and they added Google BigQuery to their supported dialects with our collaborations.

5.2.2 Generate Test Assertions

In this component, test assertions are generated to compare the analysis values. The algorithm takes the `source_analysis_values` and `target_analysis_values` as inputs and produces test assertions as output. The test assertions compare each of the values in the target analysis with the corresponding value in the source. If there is a mismatch, then the algorithm sets the status for that analysis as *violated*.

Chapter 6

Demonstration and Evaluation

We demonstrate our approach by validating the ETL scripts used in the health data warehouse. These ETL scripts were previously tested by the original developers using a *Stare and Compare* [84] approach that manually verifies a few randomly selected records and determines differences through viewing or *eyeballing* the records. We also perform an experiment where we evaluate the ability of the generated assertions to detect faults that appear in the target data as a result of faulty ETL scripts.

6.1 Validation of ETL Scripts

For each hospital, we tested nine tables in the source that were transformed into three tables in the target data warehouse. There were two different datasets on Google BigQuery for each hospital. Each target dataset included three tables, namely, *Location*, *Person*, and *Provider*. Tables 6.1 and 6.2 show the number of records in the source and target tables in both hospitals that we used to evaluate our approach. The analysis queries were in MSSQLServer format and the datasets were in different dialects (MSSQLServer and Google BigQuery).

Table 6.1: Number of Records under Test in the Source

Hospital	**Source Tables**								
	t1	t2	t3	t4	t5	t6	t7	t8	t9
$Source\ A$	19,184,566	4,245,491	2,695,800	13,801,322	677,407	119,572	5,378,994	56,032	28,493,258
$Source\ B$	3,263,722	1,326,619	2,691,829	13,794,480	325,172	69,810	0	20,814	11,253,273

Table 6.2: Number of Records under Test in the Target Data Warehouse

Hospital	**Target Tables**		
	Location	Person	Provider
$Target\ A$	4,220,269	5,378,994	119,572
$Target\ B$	1,312,192	1,613,488	69,810

It took two hours to generate and execute the balancing tests. The results of the test assertions are stored as a table in a result dataset on Google BigQuery. The presence of a row in the table in-

dicates a failure of some assertion. Our generated assertions revealed 11 errors in the ETL process and implementation which were not found by the *Sampling* approach previously used by the data warehouse testers. For the data warehouse of hospital A, there were six rows in the test results table indicating six assertion violations (three *Average Mismatches*, two *Record Count Mismatches*, and one *Outlier Mismatch*). For hospital B, there were five rows in the test results table indicating five assertion violations. These include four *Average Mismatches* and one *Outlier Mismatch*. We reported the errors to the data warehouse developers. We produced relevant meta-data along with the error messages to help developers localize the problem. This meta-data includes the time of error occurrence, the corresponding analysis query, and test assertion query that produces the error. However, automatically finding the root causes of the errors is one of the directions for future work.

6.2 Evaluation of Fault Finding Ability of Assertions

Faulty ETL scripts manifest themselves in the data stored in the target data warehouse. Instead of traditional mutation, which mutates the ETL scripts, we used a mutation analysis technique to mutate the randomly selected data stored in the target data warehouse. The goal was to measure the ability of the generated assertions to detect these faults. A fault is detected when at least one generated test assertion fails as a result of the fault. We observed the percentage of faults in the data detected by test assertions.

In many cases testers typically do not have access to real data. As copying and modifying patient data in the health data warehouse were not allowed, we used a mock dataset to conduct this evaluation. We used an off-the-shelf tool called Databene Benerator [85] to populate a Caboodle database on MSSQLServer with mock test data. Benerator is a random test data generation tool that supports scripting and makes test data generation fully automated. Benerator generates data based on data types and constraints defined for the dataset. Benerator can be customized by restricting each attribute value to generate more meaningful data (e.g., weight attribute can only take positive

values). Executing Benerator results in a database filled with mock data satisfying the defined attribute types, constraints, and customizations.

We used Benerator to generate 100 records for each of the three tables under test. For the mock source database, we executed the ETL process to generate the target data warehouse on Google BigQuery.

Table 6.3 shows different types of mutation operators used one-by-one to inject faults in the target data. We defined mutation operations with the goal of violating the *record count match*, *distinct record count match*, *attribute value match*, *attribute constraint match*, *outliers match*, *average match*, and *attribute length match* properties defined in Section 4. The mutation operators we used are such that each one violates at least one of these properties and all these properties are violated by at least one operator. An operator to force the *attribute boundary mismatch* problem to manifest itself is not needed because of Google BigQuery. This property is guaranteed by Google BigQuery, which supports the largest range of numeric values among all the dialects in the health data warehouse. The *attribute data type match* property was already tested by the data warehouse developers in the data model design phase. So in our mutation analysis approach we ignored defining an operator to evaluate this property.

Table 6.3: Mutation Operators Used To Inject Faults In Target Data

Operator	**Description**
AR	Add random record
DR	Delete random record
MNF	Modify numeric field
MSF	Modify string field
MNF_{min}	Modify min of numeric field
MNF_{max}	Modify max of numeric field
MSF_{length}	Modify string field length
MF_{null}	Modify field to null

AR and DR are table-level operators that apply to all the target tables. The rest are attribute-level operators that apply to all the numeric and string attribute in target tables.

Table 6.4: Injected Faults and Failure Data

Mutation Operator	**Table Name**			**Failures**					
	Location	Person	Provider	Record count mismatch	Average mismatch	Length mismatch	Outlier mismatch	Attribute value mismatch	Attribute constraint mismatch
AR	3	3	3	9	21	4	0	0	0
DR	3	3	3	9	20	4	2	0	0
MNF	2	15	6	0	20	0	3	32	0
MSF	6	3	5	0	2	2	0	11	0
MNF_{min}	2	15	6	0	20	0	6	32	0
MNF_{max}	2	15	6	0	22	0	3	31	0
MSF_{length}	6	3	5	0	0	3	0	15	0
MF_{null}	1	1	1	0	0	0	1	2	2
Total	**25**	**58**	**35**	**18**	**105**	**13**	**15**	**123**	**2**

- MNF: Change a numeric attribute value to a random value in a randomly selected record of the target table.
- MSF: Change a string attribute value to a random value in a randomly selected record of the target table.
- MNF_{min}: Change the minimum value of numeric attribute to a random value.
- MNF_{max}: Change the maximum value of numeric attribute to a random value.
- MSF_{length}: Change a string attribute value to a random value with different length in a randomly selected record.
- MF_{null}: Modify an attribute value with NOT-NULL constraint to NULL in a randomly selected record.

For each operator, we expect the following property violations:

- AR: record count and distinct record count match.
- DR: record count and distinct record count match.
- MNF: attribute value match and average match.
- MSF: attribute value match and attribute length match.

- MNF_{min}: outlier match and average match.
- MNF_{max}: outlier match and average match.
- MSF_{length}: attribute length match.
- MF_{null}: attribute constraint match.

Our mutation engine takes each operator, a target table, and a target attribute as input, and randomly selects a record in the target table to mutate the attribute value based on the operator. Then, the test assertions are executed to verify whether or not the injected fault is detected. The number and type of failures in test assertions caused by the fault is reported. As the operators were applied one-by-one, the target_table is restored before the next operator is applied.

Table 6.4 shows the number and types of faults injected one-by-one into the data of three OMOP tables under test (*Location*, *Person*, and *Provider*), and the numbers and types of assertion failures that were caused.

It took six hours for the entire mutation analysis to proceed. The test assertions detected 92% of 118 seeded faults in the target data. All the assertions that should have failed actually failed. Executing the test assertions on the source and faulty target mock datasets resulted in 276 failures in 44 assertions.

Table 6.4 shows the assertion failures as a result of injecting different faults. Most of the faults resulted in the failure of multiple assertions. For example, changing an attribute value resulted in an attribute value mismatch, average mismatch, and outlier mismatch failures.

All the undetected faults resulted from mutating attributes in the target table that did not map to any attribute in the source tables. These attributes are filled with some default values. For example, in the *Location* table, *address_2* is a new attribute defined for the target table and takes `NULL` as its default value. Thus, the generated assertions detected all the seeded faults in the target attributes that map to at least one attribute in the source tables.

6.3 Threats to Validity

This is a pilot study where we used one ETL transformation. However, the ETL scripts did involve complex transformation rules, such as many-to-one table and attribute mappings. There are a few other threats to validity of our results.

- **Threats to external validity**

  - *Single program*: We used only one ETL script in our work and the results may not generalize to other ETL scripts. However, this was a fairly complex script involving 1500 lines of code and complex transformation rules involving joins of tables.
  - *Three tables used*: We did not use all the tables in OMOP because at the time of conducting the experiments, we did not have access to the ETL scripts for those tables. This will be addressed in future work. However, we did consider both simple one-to-one and complex many-to-one transformation rules that involved multiple tables and joins of tables based on selection condition.
  - *No comparison with other approaches*: We did not compare our test generation approach with other tools. The only available automatic balancing test generation tool was QuerySurge [84]. This was an impractical approach that did not have assertions to validate the data that is modified through the ETL process. However, our objective was to generate balancing tests for validating the data that is also reformatted through the ETL process. We compared our approach with the *Sampling* approach used by the health data testers who used the "stare and compare" technique on a small number of records that were randomly selected from the heterogeneous number of patient records.
  - *Limited to full-load*: In data warehousing, data are loaded from a source database to the data warehouse incrementally. Our approach checks the whole source and target tables. Testing the incremental mode will be the subject of future work.

- **Threats to internal validity**

  - *Single test data generator*: We used random data generated using Benerator to conduct mutation analysis. It is possible that other test data generators or tools that use more sophisticated test generation techniques (e.g., input space partitioning or code coverage based) may produce other results. This will be the subject of future work.

Chapter 7

Conclusions and Future Work

We surveyed existing approaches to test and evaluate each of the data warehouse components. First, we described the components of a data warehouse using examples from a real-world health data warehouse project. We provided a classification framework that takes into account *what* component of a data warehouse was tested, and *how* the component was tested using various functional and non-functional testing and evaluation activities. Then, we surveyed existing approaches to test and evaluate each component. Most of the approaches that we surveyed adapted traditional testing and evaluation approaches to the area of data warehouse testing. We identified gaps in the literature and proposed directions for further research. We observed that the following testing categories are open research areas. Future research needs to focus on filling these gaps for comprehensively testing data warehouses.

- *Security testing* of the underlying data in the source and target components
- *Reliability testing* of the ETL process
- *Usability testing* of the ETL process
- *Usability testing* of the front-end applications
- *Stress testing* of the front-end applications

Next, we presented an approach to create balancing tests to validate the ETL process as a critical component in data warehouses and ensure that there is no data corruption or data loss. Our approach automatically generates test assertions for comparing the data in the source systems with the corresponding data in the target data warehouse and checks for properties related to completeness, consistency, and syntactic validity. We defined the different types of source-to-target mappings and described how to generate balancing tests for them. To summarize, the following lists our achievements in this research project:

- We defined different source-to-target mapping structures and identified how to generate balancing tests for different types of mappings.
- We automatically generated a set of balancing test assertions that could find previously undetected real-world faults in the health data that were caused by the ETL process.
- By isolating the analysis phase from the test assertion generation phase, we made our approach applicable to data warehouses that use sources running on different platforms.
- Our approach uses a translator, which makes the approach applicable to data warehouses that use sources and targets with different SQL dialects.
- We evaluated our approach on a health data warehouse that uses data sources with different data models running on different platforms. Our approach found previously undetected real-world faults in the ETL implementation.
- We demonstrated that our set of assertions can detect faults present in mock data when faulty ETL scripts execute. Our generated assertions detected all the faults injected into the target data that maps to some data in the source.

Our future plans include improving the efficiency and effectiveness of ETL testing in data warehouses as follows:

- Define new properties for table integrity checks and add test assertions to verify these properties.
- Evaluate our approach on bigger datasets, such as applying them to all the tables in the health data warehouse.
- Identify source-to-target mappings for more complicated cases, such as different types of join operations and selection conditions.

- Use input space partitioning techniques to generate input data in a more systematic manner. We can customize Benerator to generate the input data for all partitions. We are going to define equivalent classes of data with respect to the properties.

- Consider incremental-load in generating analysis values and executing test assertions for the newly added or modified data through the incremental-load.

- Develop an approach that aids developers in quickly and effectively localizing faults in the ETL process.

- Develop an approach that selects test cases to make regression testing more efficient than using a test-all process.

Moreover, the following techniques need to be developed or improved in all the testing activities in order to enhance the overall verification and validation of the data warehousing systems.

Test automation needs to improve to decrease the manual effort involved in data warehouse testing by providing effective test automation tools. The data involved in data warehouse testing is rapidly growing. This makes it impossible to efficiently test data warehouses while relying on manual activities. Existing testing approaches require a lot of human effort in writing test cases, executing tests, and reporting results. This makes it difficult to run tests repeatedly and consistently. Repeatability is a critical requirement of data warehouse testing because we need to execute the tests whenever data is added or modified in a data warehouse. The approaches previously discussed in this chapter are based on statistical analysis, manual inspections, or semi-automated testing tools that still need manual effort for generating test input values and assertions. Existing approaches to software test automation can be utilized to fully automate the tasks involved in data warehouse testing. However, automatic test assertion generation (oracle problem) is an open problem for software systems in general [86] because the expected test outputs for all possible test inputs are typically not formally specified. Testers often manually identify the expected outputs using informal specifications or their knowledge of the problem domain. The same problem exists for automatically generating test assertions for testing the data warehouse components.

If the expected outputs are not fully specified in the source-to-target transformation rules or in data warehouse documentation, it will be difficult to automatically generate test assertions. Future research needs to fill the gap between informal specifications and formally specified outputs to automatically generate test assertions.

Like other generic software systems, data warehouse projects need to implement agile development processes [87], which help produce results faster for end-users and adapt the data warehouse to ever-changing user requirements [88]. The biggest challenge for testing an agile data warehouse project is that the data warehouse components are always changing. As a result, testing needs to adapt as part of the development process. The design and execution of these tests often take time that agile projects typically do not have. The correct use of regression test selection techniques [72] can considerably reduce the time and costs involved in the iterative testing of agile data warehousing systems. These techniques help reduce costs by selecting and executing only a subset of tests to verify the parts of the data warehouse that are affected by the changes. At the same time, testers need to take into account trade-offs between the cost of selecting and executing tests, and the fault detection ability of the executed tests [72].

There will be a growing demand for real-time analysis and data requests [2]. The data warehouse testing speed needs to increase. Most of the existing functional and non-functional testing activities rely on testing the entire source, target, or intermediate datasets. As the data is incrementally extracted, transformed, and loaded into the target data warehouse, tests need to be applied to only the newly added or modified data in order to increase the speed of testing. Testing with the entire data should be applied only in the initial step where the entire data is extracted from the sources, transformed, and loaded to the target data warehouse for the first time.

Most of the existing tools rely on using real data as test inputs, while testers typically do not have access to the real data because of privacy and confidentiality concerns. Systematic test input generation techniques for software systems can be used in future studies to generate mock data with the characteristic of the real data and with the goal of adequately testing the data warehouse components. For example, we proposed to use random mock data generation tools that populate a

database with randomly generated data while obeying data types and data constraints. Genetic and other heuristic algorithms have been used in automatic test input generation for generic software systems [89] with the goal of maximizing test coverage. The same idea can be utilized in data warehouse testing to generate test data for testing different components with the goal of maximizing test coverage for the component under test.

Identifying a test adequacy criterion helps testers evaluate their set of tests and improve the tests to cover uncovered parts of the component under test. Determining adequate test coverage is a limitation of current testing approaches. Further research needs to define test adequacy criteria to assess the completeness of test cases written or generated for different testing purposes. For example, test adequacy criteria for testing the underlying data can be defined as the number of tables, columns, and constraints that are covered during a test activity. The adequacy criteria can also be defined as the number of data quality rules that are verified by the tests. Test adequacy criteria for white-box testing of the ETL process can be defined as the number of statements or branches of the ETL code that are executed during the ETL tests.

The fault finding abilities of existing testing approaches need to be evaluated. Mutation analysis techniques [50] can be used in future studies to evaluate the number of injected faults that are detected using the written/generated test cases. These techniques systematically seed a number of faults that simulate real faults in the program under test. The techniques execute the tests and determine whether or not the tests can detect the injected faults. Faults can be injected into both the code and the data in a data warehousing system. Different functional and non-functional tests are supposed to fail as a result of the injected faults. For example, balancing tests should result in failures because of imbalances caused by the seeded data faults. Functional testing of front-end application must fail due to the incorrect data that is reported in the final reports and analysis. A fault in the ETL code may result in the creation of unnecessary intermediate files during the ETL process and cause the performance tests to fail. Using these techniques help testers evaluate test cases and improve them to detect undetected faults.

Due to the widespread use of confidential data in data warehousing systems, security is a major concern. Security testing of all the components of data warehouses must play an important role in data warehouse testing. There are different potential security challenges in data warehousing systems that need to be addressed in future studies:

First, there are many technologies involved in the data warehousing implementations. Different DBMSs, data warehouse products, and cloud systems are being used to store and manage data of the sources, the intermediate DSA, and the target data warehouse. Correctly transforming data access control and user privileges from one technology to the other is a significant challenge in the security of the data warehousing systems. Future research in security testing needs to develop techniques that compare the privileges defined in the sources and the ones defined in the target of a transformation to ensure that all the privileges are correctly transformed with losing any information.

Second, due to the large number of interactive processes and distributed components involved in data warehousing systems, especially those containing sensitive data, there are many potential security attacks [90], such as man-in-the-middle, data modification, eavesdropping, or denial-of-service. The goal of such attacks may be to read confidential data, modify the data that results in misleading or incorrect information in the final reports, or disrupting any service provided by the data warehousing system. Some of the consequences can be detected using the previously discussed functional and non-functional testing approaches. For example, if the data was modified through an attack, balancing tests (Section 5.1.2) can detect the faulty data in the target data warehouse. However, comprehensive security techniques can prevent these types of attacks before the problem is propagated to the target data warehouse where error detection and fault localization is much more expensive. Security testing should detect vulnerabilities in the code, hardware, protocol implementations, and database access controls in a data warehousing project and report them to the data warehouse developers to avoid the exploitation of those vulnerabilities.

An alternative to data warehouse testing is to develop the data warehouse in a way that proves that the data warehouse implements the specifications and it will not fail under any circumstances.

This approach is called *correct by construction* [91] in software engineering context. It guarantees the correct construction of software, and thus, does not require testing. Data warehousing systems include different distributed components, programs, and processes on various platforms that are implemented using different technologies. The large number of factors that affect data warehousing systems at run-time makes it practically impossible to prove that a data warehouse meets its specifications under all circumstances. Testing must be performed to validate the data warehousing systems in different situations. As a result, data warehouse testing is likely to be an active research field in the near future.

Bibliography

[1] Manoj Philip Mathen. Data warehouse testing. *Infosys DeveloperIQ Magazine*, pages 1–8, 2010.

[2] Vincent Rainardi. *Building a Data Warehouse with Examples in SQL Server*. Apress, 1st edition, 2008.

[3] Doug Vucevic and Wayne Yaddow. *Testing the Data Warehouse Practicum: Assuring Data Content, Data Structures and Quality*. Trafford Publishing, 2012.

[4] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science*. W. H. Freeman, C edition, 1994.

[5] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on NoSQL Database. In *6th International Conference on Pervasive Computing and Applications*, pages 363–366, Oct 2011.

[6] Matteo Golfarelli and Stefano Rizzi. Data Warehouse Testing: A Prototype-based Methodology. *Information and Software Technology*, 53(11):1183–1198, 2011.

[7] Matteo Golfarelli and Stefano Rizzi. A Comprehensive Approach to Data Warehouse Testing. In *12th ACM International Workshop on Data Warehousing and OLAP*, pages 17–24, New York, NY, USA, November 2009.

[8] QuerySurge: big Data Testing, ETL Testing & Data Warehouse Testing. http://www.querysurge.com/ (Accessed 2017-09-20).

[9] Hajar Homayouni, Sudipto Ghosh, and Indrakshi Ray. Data Warehouse Testing. *Accepted for Publication in Advances in Computers*, 2017.

[10] Observational Medical Outcomes Partnership Common Data Model (OMOP CDM). https://www.ohdsi.org/data-standardization/the-common-data-model (Accessed 2017-09-20).

[11] Google BigQuery. https://cloud.google.com/bigquery/ (Accessed 2017-09-20).

[12] Steve Hoberman. *Data Model Scorecard: Applying the Industry Standard on Data Model Quality*. Technics Publications, 1st edition, 2015.

[13] Qing Li and Yu-Liu Chen. *Entity-Relationship Diagram*, pages 125–139. Springer Berlin Heidelberg, 2009.

[14] Mladen Varga. On the Differences of Relational and Dimensional Data Model. In *12th International Conference on Information and Intelligent Systems*, pages 245–251, 2001.

[15] Ralph Kimball, Margy Ross, Warren Thornthwaite, Joy Mundy, and Bob Becker. *The Data Warehouse Lifecycle Toolkit*. Wiley, 2nd edition, 2008.

[16] Vivekanand Gopalkrishnan, Qing Li, and Kamalakar Karlapalem. Star/Snow-Flake Schema Driven Object-Relational Data Warehouse Design and Query Processing Strategies. In *DataWarehousing and Knowledge Discovery*, pages 11–22, Berlin, Heidelberg, August 1999.

[17] Caboodle Data Model. http://www.med.upenn.edu/dac/epic-clarity-data-warehousing.html (Accessed 2017-10-14).

[18] Ajay Askoolum. Structured Query Language. In *System Building with APL + Win*, pages 447–477. John Wiley & Sons, Ltd, 2006.

[19] Foster Hinshaw. Data Warehouse Appliances: Driving the Business Intelligence Revolution. *DM Review Magazine*, pages 30–34, September 2004.

[20] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[21] MySQL. https://www.mysql.com (Accessed 2017-05-02).

[22] Microsoft SQL Server. https://www.microsoft.com/sql-server (Accessed 2017-05-02).

[23] PostgreSQL. https://www.postgresql.org (Accessed 2017-05-02).

[24] Apache Accumulo. https://accumulo.apache.org (Accessed 2017-05-02).

[25] ArangoDB: Highly Available Multi-model NoSQL Database. https://www.arangodb.com (Accessed 2017-05-02).

[26] MongoDB. https://www.mongodb.com (Accessed 2017-05-02).

[27] Apache Hadoop. https://hadoop.apache.org (Accessed 2017-05-02).

[28] Oracle: Integrated Cloud Applications and Platform Services. https://www.oracle.com (Accessed 2017-05-02).

[29] IBM PureApplication. http://www-03.ibm.com/software/products/pureapplication (Accessed 2017-05-02).

[30] BigQuery: Analytics Data Warehouse. https://cloud.google.com/bigquery (Accessed 2017-05-02).

[31] Amazon Redshift | Data Warehouse Solution. //aws.amazon.com/redshift (Accessed 2017-05-02).

[32] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. Wiley, 1st edition, 2004.

[33] Jose Barateiro and Helena Galhardas. A survey of Data Quality tools. *Datenbank Spektrum*, 14:15–21, 2005.

[34] Mong Li Lee, Hongjun Lu, Tok Wang Ling, and Yee Teng Ko. Cleansing Data for Mining and Warehousing. In *10th International Conference on Database and Expert Systems Applications*, pages 751–760, Berlin, Heidelberg, August 1999.

[35] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26(1):65–74, 1997.

[36] George Colliat. OLAP, Relational, and Multidimensional Database Systems. *ACM SIGMOD Record*, 25(3):64–69, 1996.

[37] Crystal Reports: Formatting Multidimensional Reporting Against OLAP Data. http://www.informit.com/articles/article.aspx?p=1249227 (Accessed 2017-10-10).

[38] Michael J. Berry and Gordon Linoff. *Data Mining Techniques: For Marketing, Sales, and Customer Support*. John Wiley & Sons, Inc., 2nd edition, 1997.

[39] Jimison Iavindrasana, Gilles Cohen, Adrien Depeursinge, Henning Muller, R. Meyer, and Antoine Geissbuhler. Clinical Data Mining: a Review. *Yearbook of Medical Informatics*, pages 121–133, 2009.

[40] Jung P. Shim, Merrill Warkentin, James F. Courtney, Daniel J. Power, Ramesh Sharda, and Christer Carlsson. Past, Present, and Future of Decision Support Technology. *Decision Support Systems*, 33(2):111–126, 2002.

[41] Eta S. Berner, editor. *Clinical Decision Support Systems: Theory and Practice*. Springer, 2nd edition, 2006.

[42] M. Pamela Neely. *Data Quality Tools for Data Warehousing – A Small Sample Survey*. AD-a362 923. State University of New York at Albany, 1998.

[43] Mark A. Weiss. *Data Structures & Algorithm Analysis in C++*. Pearson, 4th edition, 2013.

[44] Michael G. Kahn, Tiffany J. Callahan, Juliana Barnard, Alan E. Bauck, Jeff Brown, Bruce N. Davidson, Hossein Estiri, Carsten Goerg, Erin Holve, Steven G. Johnson, Siaw-Teng Liaw, Marianne Hamilton-Lopez, Daniella Meeker, Toan C. Ong, Patrick Ryan, Ning Shang, Nicole G. Weiskopf, Chunhua Weng, Meredith N. Zozus, and Lisa Schilling. A Harmonized Data Quality Assessment Terminology and Framework for the Secondary Use of Electronic Health Record Data. *EGEMS (Washington, DC)*, 4(1):1244, 2016.

[45] OHDSI/Achilles. https://github.com/OHDSI/Achilles (Accessed 2017-05-11).

[46] Observational Health Data Sciences and Informatics (OHDSI): Opportunities for Observational Researchers. *Studies in Health Technology and Informatics*, 216:574–578, 2015.

[47] David Loshin. Rule-based Data Quality. In *11th ACM International Conference on Information and Knowledge Management*, pages 614–616, New York, NY, USA, November 2002.

[48] Informatica. https://www.informatica.com/ (Accessed 2017-04-14).

[49] Jerry Gao, Chunli Xie, and Chuanqi Tao. Big Data Validation and Quality Assurance – Issuses, Challenges, and Needs. In *IEEE Symposium on Service-Oriented System Engineering*, pages 433–441, May 2016.

[50] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[51] Kristy Browder Edwards and George Lumpkin. Security and the Data Warehouse. Technical report, Oracle, 2005.

[52] Mukesh K. Mohania and A. Min Tjoa, editors. *Data Warehousing and Knowledge Discovery: 12th International Conference, DaWaK*. Springer, 2010.

[53] Matteo Golfarelli and Stefano Rizzi. *Data Warehouse Design: Modern Principles and Methodologies*. McGraw-Hill, Inc., 1st edition, 2009.

[54] MySQL: MySQL Workbench Manual: 9.2.3 Schema Validation Plugins. https://dev.mysql.com/doc/workbench/en/wb-validation-plugins.html (Accessed 2017-04-14).

[55] George Papastefanatos, Panos Vassiliadis, Alkis Simitsis, and Yannis Vassiliou. Design Metrics for Data Warehouse Evolution. In *International Conference on Conceptual Modeling*, pages 440–454, Berlin, Heidelberg, October 2008.

[56] Coral Calero, Mario Piattini, Carolina Pascual, and Manuel A. Serrano. Towards Data Warehouse Quality Metrics. In *the International Workshop on Design and Management of Data Warehouses*, pages 1–10, Interlaken, Switzerland, June 2001.

[57] Bayo Erinle. *Performance testing with JMeter 2.9*. Packt Publishing Ltd, 1st edition, 2013.

[58] Donald R. Slutz. Massive Stochastic Testing of SQL. In *24th International Conference on Very Large Data Bases*, pages 618–622, San Francisco, CA, USA, August 1998.

[59] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An Agenda for Testing Relational Database Applications. *Software Testing, Verification & Reliability*, 14(1):17–44, 2004.

[60] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *8th USENIX Conference on Networked Systems Design and Implementation*, pages 238–252, 2011.

[61] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[62] Bernard Smith and Vance J. Klingman. Method and Apparatus for Testing Recovery Scenarios in Global Transaction Processing Systems, October 1997. US Patent 707/999.202.

[63] Neveen ElGamal, Ali ElBastawissy, and Galal Galal-Edeen. Data Warehouse Testing. In *the Joint EDBT/ICDT Workshops*, pages 1–8, New York, USA, 2013.

[64] Hajar Homayouni, Sudipto Ghosh, and Indrakshi Ray. On Generating Balancing Tests for Validating the Extract-Transform-Load Process for Data Warehouse Systems. 2018. Submitted to *11th IEEE Conference on Software Testing, Validation and Verification.* Available at http://www.cs.colostate.edu/etl/papers/icst2018.pdf.

[65] Pedsnet. https://pedsnet.org/ (Accessed 2017-05-11).

[66] Marius Marin. A Data-Agnostic Approach to Automatic Testing of Multi-dimensional Databases. In *7th International Conference on Software Testing, Verification and Validation*, pages 133–142, March 2014.

[67] Jeff Theobald. Strategies for Testing Data Warehouse Applications. *Information Management*, 17(6):20, 2007.

[68] Len Wyatt, Brian Caufield, and Daniel Pol. Principles for an ETL Benchmark. In *Performance Evaluation and Benchmarking*, pages 183–198, Berlin, Heidelberg, August 2009.

[69] Google Cloud Bigtable. https://cloud.google.com/bigtable/ (Accessed 2017-05-11).

[70] Larissa T. Moss and Shaku Atre. *Business Intelligence Roadmap: The Complete Project Lifecycle for Decision-Support Applications*. Addison-Wesley Professional, 2003.

[71] T. N. Manjunath, Ravindra S. Hegad, H. K. Yogish, R. A. Archana, and I. M. Umesh. A Case Study on Regression Test Automation for Data Warehouse Quality Assurance. *International Journal of Information Technology and Knowledge Management*, 5(2):239–243, 2012.

[72] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[73] Briand Lionel C., Labiche Yvan, and G. Soccar. Automating Impact Analysis and Regression Test Selection based on UML Designs. In *International Conference on Software Maintenance*, pages 252–261, October 2002.

[74] Ilene Burnstein. *Practical Software Testing: A Process-Oriented Approach*. Springer, 2003.

[75] Joseph S. Dumas and Janice C. Redish. *A Practical Guide to Usability Testing*. Intellect Ltd, Rev Sub edition, 1999.

[76] Carolyn Snyder. *Paper Prototyping: The Fast and Easy Way to Design and Refine User Interfaces*. Morgan Kaufmann, 1st edition, 1877.

[77] Bruno Edson Martins de Albuquerque Filho, Thiago Luis Lopes Siqueira, and Valeria Cesario Times. OBAS: An OLAP Benchmark for Analysis Services. *Journal of Information and Data Management*, 4(3):390, 2013.

[78] Byung-Kwon Park, Hyoil Han, and Il-Yeol Song. XML-OLAP: A Multidimensional Analysis Framework for XML Warehouses. In *Data Warehousing and Knowledge Discovery*, pages 32–42, Berlin, Heidelberg, August 2005.

[79] Xin Bai. Testing the Performance of an SSAS Cube Using VSTS. In *7th International Conference on Information Technology: New Generations*, pages 986–991, Las Vegas, NV, USA, April 2010.

[80] James F. Allen. Natural Language Processing. In *Encyclopedia of Computer Science*, pages 1218–1222. John Wiley and Sons Ltd., Chichester, UK, 2003.

[81] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[82] SqlRender. https://github.com/OHDSI/SqlRender (Accessed 2017-09-20).

[83] Sqlrender With BigQuery Supports. https://github.com/hajarhomayouni/SqlRender (Accessed 2017-09-20).

[84] Querysurge. http://www.querysurge.com/ (Accessed 2017-10-15).

[85] Databene benerator. http://databene.org/databene-benerator/ (Accessed 2017-05-15).

[86] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[87] Des Greer and Yann Hamon. Agile Software Development. *Software: Practice and Experience*, 41(9):943–944, 2011.

[88] Lawrence Corr and Jim Stagnitto. *Agile Data Warehouse Design: Collaborative Dimensional Modeling, from Whiteboard to Star Schema*. DecisionOne Press, 3rd edition, 2011.

[89] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-Data Generation Using Genetic Algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[90] Henk C. A. Van Tilborg and Sushil Jajodia, editors. *Encyclopedia of Cryptography and Security*. Springer, 2nd edition, 2011.

[91] L. P. Carloni, K. L. McMillan, A. Saldanha, and A. L. Sangiovanni-Vincentelli. A Methodology for Correct-by-construction Latency Insensitive Design. In *IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No.99CH37051)*, pages 309–315, November 1999.