A Model for Managing Data Integrity

Vikram Mallur

Thesis submitted to the

Faculty of Graduate and Postdoctoral Studies

In partial fulfillment of the requirements

For the M.Sc. degree in Computer Science (OCICS)

University of Ottawa

Ottawa, Ontario, Canada

August 2011

Abstract

Consistent, accurate and timely data is essential to the functioning of a modern organization. Managing the integrity of an organization's data assets in a systematic manner is a challenging task in the face of continuous update, transformation and processing to support business operations. Classic approaches to constraint-based integrity focus on logical consistency within a database and reject any transaction that violates consistency, but leave unresolved how to fix or manage violations. More ad hoc approaches focus on the accuracy of the data and attempt to clean data assets after the fact, using queries to flag records with potential violations and using manual efforts to repair. Neither approach satisfactorily addresses the problem from an organizational point of view.

In this thesis, we provide a conceptual model of constraint-based integrity management (CBIM) that flexibly combines both approaches in a systematic manner to provide improved integrity management. We perform a gap analysis that examines the criteria that are desirable for efficient management of data integrity. Our approach involves creating a Data Integrity Zone and an On Deck Zone in the database for separating the clean data from data that violates integrity constraints. We provide tool support for specifying constraints in a tabular form and generating triggers that flag violations of dependencies. We validate this by performing case studies on two systems used to manage healthcare data: PAL-IS and iMED-Learn. Our case studies show that using views to implement the zones does not cause any significant increase in the running time of a process.

Acknowledgements

I would like to first thank my supervisor Prof. Liam Peyton for his constant support and encouragement. Without his help, this thesis would not have seen the light of day. I have been able to learn a great deal working with him.

The PAL-IS system used in one of our case studies was built in consultation with Élisabeth Bruyère Hospital, Ottawa. I would like to thank the entire team that helped build the system, and with whom I have had the chance to work with. In alphabetic order they are Mana Azarm, Aladdin Baarah, Natalia Berezovskaia, Prof. Craig Kuziemsky, Dishant Langayan, Bradley McLeod and Alain Mouttham.

The iMED-Learn system used in our other case study is built in consultation with Dr. Gary Viner from The Ottawa Hospital Family Health Team. I would like to thank him for his guidance and insights as well as the chance to work with him.

Finally I would like to thank Mr. Randy Giffen, who has been collaborating with us on a project with Osler Hospital, for giving me the opportunity to do an internship at IBM.

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
3NF	Third Normal Form
AR	Association Rule
BCNF	Boyce-Codd Normal Form
BIDS	Business Intelligence Development Studio
CBIM	Constrant-Based Integrity Management
CFD	Conditional Functional Dependency
DBMS	DataBase Management System
DQ	Data Quality
ECA	Event, Condition, Action
EHR	Electronic Health Record
ETL	Extract, Transform and Load
FD	Functional Dependency
IC	Integrity Constraint
IND	INclusion Dependency
JD	Join Dependency
MDM	Master Data Management
MVD	Multi Valued Dependency
NP	Non-deterministic Polynomial time
OLAP	OnLine Analytical Processing
OLTP	OnLine Transaction Processing
PAL-IS	Palliative care Information System
SQL	Structured Query Language
SSIS	SQL Server Integration Services

Chapter 1. Introduction

1.1 Problem Statement

Consistent, accurate and timely data is essential to the functioning of a modern organization. Managing the integrity of an organization's data assets in a systematic manner is a challenging task in the face of continuous update, transformation and processing to support business operations. This is especially true in large organizations where data may be consolidated and reported on across many different departments and geographical regions or in areas like healthcare where, for example, a patient's electronic health record (EHR) may consist of data amassed from visits to several different organizations. Differences in data representation, errors in data entry, as well as processing of the data to support reporting can all potentially compromise the integrity of the data.

Most organizations have a variety of techniques and tools they employ to identify and correct for data quality issues. Unfortunately, these techniques assume that compromised data exists and wait to clean the data at the point where it is about to be used (Batini, Cappiello, et al. 2009). Reporting applications filter out suspect data to report on a "clean subset" of actual records. Similarly, extract, transform and load (ETL) processes "clean" data as they take disparate data from source systems into a common representation in a data warehouse. Source systems attempt to validate data on entry but often must live with compromised data.

There is a large body of work on the use of integrity constraints to ensure logically consistent data within a database (Fan, Geerts and Jia, A Revival of Integrity

Constraints for Data Cleaning 2008). This includes significant work on the use of dependency relationships within relational databases to flag inconsistencies. Implementing dependency relationships with triggers is an effective mechanism for rejecting any transaction that might result in a loss of integrity. Unfortunately, it is not always practical to simply reject transactions, nor is it possible to automatically fix transactions. Human interaction is often required to resolve issues of semantic accuracy that are the root cause of inconsistencies. The source system is usually the best place to resolve such issues but, even there, it is often necessary to first accept the record as it can take time to determine the correct resolution.

Integrity constraint approaches reject inconsistent data but may lose information, while cleaning approaches are ad hoc. The problem is how to provide a systematic approach to managing data integrity that can be used by organizations to ensure clean data. We would also like to minimize information loss by separating the clean data from the dirty data, while supporting integrated views.

1.2 Thesis Motivation and Contributions

Our motivation is to provide a systematic and sound approach to managing data integrity that can be used by organizations to both ensure that they have clean data that meets their requirements and that they minimize information loss that can result from rejecting flawed data. When an organization is handling large amounts of data, it can be difficult to keep track of changes that affect data integrity. New data coming in may not necessarily satisfy existing constraints. The idea behind our approach is that we do not want to reject changes to the database that compromise data integrity but rather manage data in the database, so that:

- compromised data is flagged and segmented from clean data for follow up;
- clean data can be viewed and interacted with for secure processing;
- a separate view of all data is also maintained for processing based on priority.

By using our approach, we envision that a database will never be allowed to enter an inconsistent state. In other words, once an area of clean data within the database is established, it stays clean. When new data enters the database, it must be possible to determine if any of the data is dirty and for what reason. Accordingly, our approach flags rows in the database that violate one or more constraints and shows the constraint that was violated by that row.

In this thesis, we provide a conceptual model of constraint-based integrity management (CBIM) that will support the resolution of logical inconsistencies in a systematic manner that flexibly combines integrity constraints for detecting inconsistencies with manual tools and techniques for resolving inconsistencies. Using case studies on PAL-IS (palliative care data) and iMED-Learn (family physician data), we demonstrate how the model can flexibly address and improve integrity management for a representative set of data processing tasks, and analyze how the model can be implemented in practice.

Our contributions are as follows:

- **Gap analysis:** A gap analysis of existing approaches to managing data integrity in industry and in the academic literature that clearly identifies where these approaches fall short. This gap is articulated in a set of criteria that can be used to evaluate approaches for managing data integrity.

- **Conceptual model of Data Integrity:** A systematic analysis of the types of data processes that occur in an organization through which integrity must be maintained and a systematic analysis of the types of integrity constraints that can be articulated to define integrity. This is coupled with the definition of a Data Integrity Zone and an On Deck Zone to segment compromised data which needs to be fixed and clean data for which data integrity is guaranteed. An example mechanism has been implemented and validated in two case studies based on this definition.
- **Tool support:** We also provide a table-based approach for defining constraints in a database that makes it easier for domain experts to articulate constraints. We provide a tool that generates database triggers to enforce dependency constraints based on our table-based format for defining constraints.

1.3 Thesis Methodology and Organization

Our methodology was based on design-oriented research (Hevner, et al. 2004), in which a gap analysis of existing approaches for managing data integrity is used to identify a problem, for which we iteratively attempt to design new possible solutions, and evaluating it based on a set of selected criteria. The intent is to demonstrate the potential validity of our new approach. The essential steps in our methodology were:

1. Identify the problem of managing data integrity based on a survey of literature and tools from industry and academia including:
    a. Review methods used for data cleaning.
    b. Examine different types of dependencies.
    c. Identify different data processes that occur in an organization.

    d. Identify criteria for evaluating approaches toward data integrity.
2. Define our approach as a potential solution to managing data integrity based on a model of partitioning a database into a Data Integrity Zone and an On Deck Zone.
3. Prototype an implementation of our approach
4. Perform case studies to test the feasibility of implementing our approach in a DBMS.
5. Evaluate our proposed solution using the criteria from 1(d) and then repeat steps 2, 3, 4, and 5 iteratively to improve on our approach
6. Compare our prototype to existing tools for managing data integrity.
7. Identify areas for future work.

The thesis is organized as follows. Chapter 2 gives a background on dependencies and explains the concept of data integrity. We outline some existing approaches for data cleaning and the concept of triggers in databases. We also show how conditional functional dependencies (CFDs), that are increasingly being used for cleaning, are related to association rules used in data mining, and look at an existing algorithm for discovering CFDs.

Chapter 3 explains our problem using a representative set of processes in a healthcare scenario. We perform a gap analysis in existing cleaning approaches, and identify criteria that are essential for data integrity. Then, we describe our conceptual model for data integrity using the concept of "data integrity" and On Deck Zones. We discuss ways of implementing this model in a DBMS and how it can be used for different processing tasks. We also show rules that can be used for flagging constraint violations

and show an example with a trigger. Lastly, we look at how our tool provides support for specifying constraints and for generating triggers to enforce dependencies.

In Chapter 4, we perform case studies based on the PAL-IS and iMED-Learn systems. PAL-IS is an electronic medical records application for patients receiving palliative care. iMED-Learn analyzes and reports on patient visits to help family physicians and residents assess their background. Chapter 5 contains an evaluation of our approach in comparison to other tools and techniques used for managing data integrity. Finally, in Chapter 6, we give a summary of our contributions, along with some limitations of our approach and some ideas for future work.

Chapter 2. Background

In this chapter, we will look at the concept of data integrity, and examine the role of integrity constraints and data cleaning. We will study the different types of dependencies that are a subset of integrity constraints. We then briefly touch upon association rules that have similar semantics to some dependencies and explore the role of database triggers. Finally, we look at some data cleaning tools some of which will be studied in more detail in Chapter 4.

2.1 Data Integrity

Integrity constraints are properties that must be satisfied by every instance of a database (Abiteboul, Hull and Vianu 1995). Data that violates these constraints is referred to as dirty data, and requires cleaning to satisfy the constraints. Since there is usually more than one way of making these changes, methods have been proposed to clean data using criteria such as cost and distance functions (Fan, Geerts and Jia, A Revival of Integrity Constraints for Data Cleaning 2008), (Yakout, et al. 2010). The goal of such approaches is to make the cleaned database as close as possible to the original.

Data integrity refers to the state in which data within a database is consistent with the integrity constraints defined for that data. In modern organizations, data is frequently exchanged between multiple databases and various processes are performed that can compromise integrity. In addition to transactions, these include ETL operations (Rahm and Do 2000) and aggregating data for reporting. It is also during these processes that constraints can be enforced to manage data integrity.

Figure 2.1 shows a hierarchy of integrity constraints that can exist on data. Dependencies, which are the most common type, are divided into functional dependencies (FD) and inclusion dependencies (IND). There is a third type known as join dependency that is not shown in the figure (Abiteboul, Hull and Vianu 1995). These are extended further by adding conditions to form conditional functional dependencies (CFD) (Bohannon, et al. 2007) and conditional inclusion dependencies (CIND) (Bravo, Fan and Ma, Extending Dependencies with Conditions 2007).

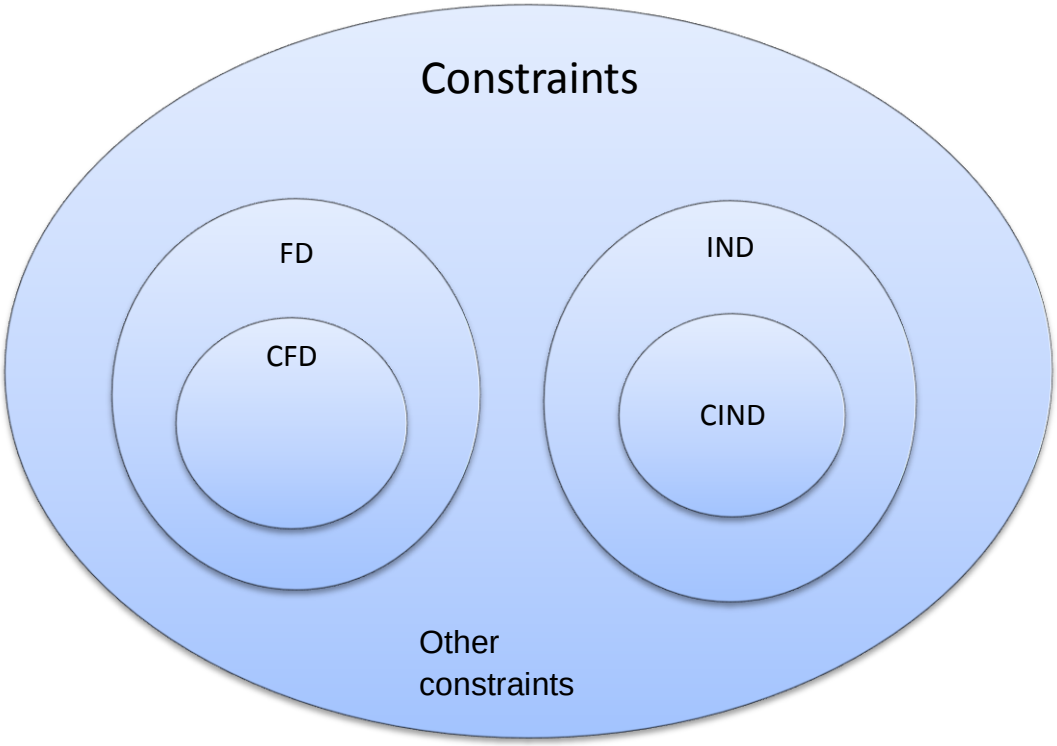


Figure 2.1 Constraint Hierarchy

There are also constraints like language consistency and data formatting that cannot be modeled with the usual semantics (see "Other constraints" in Figure 2.1).

These do not fit the form of dependencies but are nonetheless important. In an application with multi-language support, for instance, changing the English text will necessitate a change in the French text to keep both of them consistent with the intended meaning. The constraint here is not so much that the translation is accurate as the need to change the French text when the English text changes.

It has been shown that the satisfiability problem for CFDs is NP-complete (Bohannon, et al. 2007). This means that if we are given a set of CFDs, it is not possible to check within a reasonable amount of time whether the database can be made to satisfy all the CFDs at the same time. However, we can quickly determine if any of the CFDs are violated in the database.

Another example of other constraints is the different types of normal forms. Normalization is performed in databases with the goal of reducing data redundancy. It has been observed that the most common normal form in practice is the third normal form or 3NF (Kolahi and Libkin 2006). As a result, some redundancies may be present in the data that would have been eliminated in the Boyce-Codd normal form (BCNF). Since normalization increases the number of tables, resulting in faster updates but slower retrieval of data, some data may be intentionally kept in a lower normal form (Fehily 2005). In such cases, the support for a constraint may be inadequate. For example, it is not possible to enforce the FD $X \rightarrow Y$ in a table when X is not the primary key. Also, a foreign key constraint for $X \subseteq Y$ is possible only when Y is a primary key. In cases like these, the constraints will have to be enforced by other means like triggers.

The usual dimensions associated with data are accuracy, completeness, consistency and timeliness (Batini, Cappiello, et al. 2009). The purpose of data cleaning

(also known as data cleansing or scrubbing), which is described in the next section, is to improve data quality (DQ) in terms of these dimensions (Batini and Scannapieco, Data Quality: Concepts, Methodologies and Techniques 2006).

Accuracy is defined as the closeness between the actual value of data and the value it is supposed to represent. There are two kinds of accuracy namely syntactic and semantic. Syntactic accuracy can be determined by checking whether a value lies in the domain of acceptable values. Semantic accuracy, on the other hand, is harder to check and is verified by looking at the same data from different sources (Batini and Scannapieco, Data Quality: Concepts, Methodologies and Techniques 2006). A problem related to semantic accuracy is that of object identification. Known by different names in the literature such as entity resolution, merge/purge, object identification, record linkage and reference reconciliation, it refers to the process of identifying and merging records that refer to the same real-world entity (Benjelloun, et al. 2009)

Completeness refers to the presence of NULL or missing values in a database. During data cleaning, missing values can sometimes be computed by observing the relationship between attributes in other rows.

Consistency is used to capture semantic rules that are defined on the data. Integrity constraints are actually an example of such rules. A database is said to be consistent when none of its rows violates any of the constraints defined. Some problems focus on making a database consistent with a set of constraints by making the minimal number of changes (Bohannon, et al. 2007).

Finally, timeliness refers to the usefulness of the data with respect to its purpose. If a person's address has changed but not been updated in the database, then mail

addressed to him would be sent to the wrong location. These types of errors can financially impact an organization. It is estimated that dirty data costs businesses $600 billion annually (Fan, Geerts and Jia, A Revival of Integrity Constraints for Data Cleaning 2008).

2.2 Data Cleaning

Data cleaning solutions can be broadly classified along two lines. The first approach is based on the transformation of the data. This includes various activities that are performed with the goal of improving data. In addition to object identification, these include profiling and standardization. Profiling refers to the task of inferring intensional properties like database structure and fields containing similar values. Standardization involves formatting data values according to their attributes.

Popular tools include AJAX (Galhardas, Florescu, et al., AJAX: An Extensible Data Cleaning Tool 2000), Intelliclean (Lee, Ling and Low 2000), Potters Wheel (Raman and Hellerstein 2001) and Telcordia's tool (Caruso, et al. 2000). AJAX uses a framework for declarative data cleaning that has been proposed in (Galhardas, Florescu, et al., Declarative Data Cleaning: Language, Model, and Algorithms 2001) and uses the SQL99 syntax. IntelliClean is a tool that performs object identification based on rules that are extracted from domain knowledge. Potter's wheel is a tool for finding discrepancies in data and performing transformations. Telcordia's is another tool for performing object identification. Tools like these suffer either from lack of interactivity or require significant user effort. Other tools worth mentioning are Bellman, TAILOR and TRIO. Bellman is a DQ browser that helps users to understand the structure of databases by using profiling (Dasu, et al. 2002). TAILOR is a record linkage tool that can be tuned by

the user by adjusting parameters (Elfeky, Verykios and Elmagarmid 2002). TRIO is a system that can be used to manage both accuracy and the lineage of data (Widom 2005).

The second approach is constraint-based cleaning which will be the focus of this paper. This approach uses two main ideas. The first is consistent query answering (Arenas, Bertossi and Chomicki 1999) in which no changes are made to the data and only consistent data is returned in a query result. The second approach, also known as constraint-based repairing, will be the focus of this thesis and involves physically changing the data to satisfy the constraints. According to (Fan, Geerts and Jia, Semandaq: A Data Quality System Based on Conditional Functional Dependencies 2008), there are no commercial data cleaning systems based on this approach. The authors present a research prototype system called Semandaq for dealing with CFDs. It allows the user to specify CFDs on the data, detect violations, and handle data cleaning through user interaction. Unlike our work which deals with handling constraints in general, Semandaq has been designed specifically for CFDs.

CFDs are said to be contextual since they are relevant only to the given instance of data. They are also referred to as data quality rules (DQRs). But existing DQ tools do not focus on context-dependency rules (Chiang and Miler 2008), and most of them are limited to providing support for ETL operations (Yakout, et al. 2010). Manual cleaning of data using traditional constraints like functional and inclusion dependencies has been practiced in US statistical agencies for decades (Fan, Geerts and Jia, Semandaq: A Data Quality System Based on Conditional Functional Dependencies 2008). The use of conditional dependencies for cleaning helps to capture more errors that can commonly be found in real-life data. For example, the fact that the postal code determines the street in a

particular city cannot be expressed as an FD when there are data about other cities also present.

2.3 Dependencies

As described in section 2.1 a significant subset of integrity constraints can be expressed as dependencies in the relational model of data (Deutsch 2009). The different types of dependencies are functional, inclusion and join dependencies. Special cases of these are the key and multi-valued dependencies. Recently, new classes of dependencies have been proposed by adding conditions to traditional dependencies to form conditional functional dependencies (Bohannon, et al. 2007) and conditional inclusion dependencies (Bravo, Fan and Ma, Extending Dependencies with Conditions 2007).

2.3.1 Functional Dependency

A functional dependency (FD) is an expression of the form $R: X \rightarrow Y$ where R is a relation (table) and X, Y are attribute (column) sets in R (Abiteboul, Hull and Vianu 1995). This means that the value in column X functionally determines the value in column Y. In Table 2.1, the FD $\text{PatientID} \rightarrow \text{LastName}$ is violated because both Baker and Powell have the same patient id. This can be fixed by deleting one of the rows, making both last names the same, or by changing one of the patient ids.

Table 2.1 Functional dependency (Patient)

PatientID	LastName	OHIPNo
101618	Baker	0966-970-272-IN
101618	Powell	0962-889-606-MM
101731	Smith	0962-567-589-MM

Which cleaning operation is used depends on the context. For example, it could make sense to change the patient id rather than repeating patient information in the table.

2.3.2 Inclusion Dependency

An inclusion dependency (IND), also known as a referential constraint, is an expression of the form $R[X] \subseteq S[Y]$ denoting that the values in column X of table R are subsets of the values in column Y of table S (Abiteboul, Hull and Vianu 1995). Consider the data in Table 2.2 and the IND Encounter[PatientID] ⊆ Patient[PatientID]. Since the patient id 101935 is not in Table 1 this is a violation of the constraint. It can be fixed by including a row in the Patient table about the patient with id 101935. Another solution is to change the id to a value that is already present in the Patient table.

Table 2.2 Inclusion dependency (Encounter)

PatientID	EncounterType
101618	Phone
101731	InPerson
101935	InPerson

2.3.3 Conditional Dependencies

A conditional functional dependency (CFD) is an expression of the form $(X \rightarrow Y, t_p)$ where $X \rightarrow Y$ is an FD and t_p is a pattern tableau that assigns specific values to one or more columns mentioned in the expression. Note that an FD corresponds to a CFD with an empty pattern tableau. There are two types of CFDs. The first is the variable CFD which is of the form $[A = a_1, B] \rightarrow [C]$ which means that in all rows where column A takes the value a_1, column B functionally determines column C. So, unlike FDs, CFDs

apply only to a subset of rows in the table. It can also take the form $[A = a_1, B] \rightarrow [C, D = d_2]$ which specifies that in all rows where column A has the value a_1, in addition to the FD $B \rightarrow C$, column D must have the value d_2. There can also be CFDs of the type $[A] \rightarrow [B = b_1]$ which simply means that no matter what the value of A is, B must have the value b_1. The second type is the constant CFD which assigns values to every column mentioned in the CFD. It takes the form $[A = a_2] \rightarrow [C = c_2]$ indicating that, in all rows where the value of attribute A is a_2, the value of attribute C must be c_2. Constant CFDs are related to association rules that are used in data mining (Chiang and Miler 2008). Unlike an FD, a constant CFD can be violated by a single row in a table. There has also been a proposal to extend CFDs by including support for disjunction and inequality (Bravo, Fan and Geerts, et al. 2008).

A conditional inclusion dependency (CIND) is an expression of the form $(R[X] \subseteq S[Y], t_p)$ and is defined in a similar manner to the CFD. An example of a CIND could be

$$\text{Patient}[\text{PatientID}] \subseteq \text{Encounter}[\text{PatientID}, \text{EncounterType} = \text{"InPerson"}]$$

i.e., the fact that when a new patient is entered into the Patient table shown in Table 2.1, they must have an "InPerson" meeting that is recorded in the Encounter table shown in Table 2.2 (subsequent meetings could be through other means like phone or videoconference).

Table 2.3 Conditional Dependencies

	Constraint
(1)	$\text{Patient: [City} = \text{"Wahgoshig"}] \rightarrow [\text{Address} = \text{"PO Box 689"}]$
(2)	$\text{Patient: [City} = \text{"Ottawa"}, \text{PostalCode}] \rightarrow [\text{Address}]$
(3)	$\text{Patient}[\text{PatientID}] \subseteq \text{Encounter}[\text{PatientID}, \text{EncounterType} = \text{"InPerson"}]$

The constraints shown in Table 2.3 above are defined on data in Table 2.2 (Encounter) and Table 2.4 (Patient). All of them are violated by one or more rows in Table 2.4.

Table 2.4 Constraint Violations (Patient)

PatientID	City	PostalCode	Address
101618	Ottawa	K2G 1V8	1385 Woodroffe Ave
101731	Ottawa	K2G 1V8	100 Bank St
101935	Wahgoshig	P0K 1N0	First Nation

Constraint (1) which is violated by the third row can be fixed by changing the city or by setting the address to "PO Box 689". In general, a violation of a CFD can be fixed by changing the value of any column that appears in the antecedent ($X \rightarrow Y$ cannot be false when X is false). For constraint (2), which is violated by the first two rows, we can change the city in one of the rows, change one of the postal codes, or makes both addresses the same. Finally constraint (3), which is violated by the first row, can be fixed by changing the patient id to someone who has had an in-person encounter recorded, or inserting a new row into Encounters with the same patient id and encounter type being "InPerson".

2.3.4 Join Dependency

A join dependency (JD) is a constraint that exists when a database table can be recreated by performing a natural join on sets of attributes contained in the table (Abiteboul, Hull and Vianu 1995). When there are only two attribute sets, it is known as

a multivalued dependency (MVD). A JD $X \rightarrow\rightarrow Y$ holds on a table if, in any two rows where column X has the same value, the corresponding values in column Y can be switched without producing any new rows.

In the subset of the Encounter table shown in Table 2.5, adding a new row with the values [10168, 200119, 417] will satisfy the MVD PatientID $\rightarrow\rightarrow$ PhysicianID. Because an MVD can be enforced by adding rows to a table, it is also known as tuple generating dependency. By contrast, an FD can be enforced by deleting rows from the table. In this thesis, we focus strictly on functional and inclusion dependencies along with their conditional counterparts. Join dependencies are not normally mentioned in the context of data cleaning possibly because they occur in the fourth and fifth normal forms that are rare in practice. However, checking for JD violations in a database is not difficult.

Table 2.5 Join dependency (Encounter)

PatientID	PhysicianID	OHIPDiagnosisID
101618	100026	417
101618	200119	658
101618	100026	658

2.4 Association Rules

We now explain the concept of association rules used in data mining, and the relationship they have with dependencies. This section (including 2.4.1) can be skipped as it is not connected to the rest of the thesis.

A dependency of the form $X \rightarrow Y$ is said to be left-reduced if there is no attribute subset $Z \subseteq X$ s.t. $Z \rightarrow Y$. It is non-trivial if $Y \notin X$. A dependency that is both non-trivial and left-reduced is said to be minimal. Data quality mining (DQM) refers to the use of data mining methods to improve data quality in large databases (Hipp, Guntzer and Grimmer 2001). There is a close connection between left-reduced, constant CFDs and minimal association rules (ARs) (Fan, Geerts and Lakshmanan, et al. 2009). The difference is that a CFD $[A = a_1] \rightarrow [B = b_1]$ means that the value a_1 in attribute A functionally determines the value b_1 in attribute B, but in an AR there could be other tuples where $A = a_1$ but $B \neq b_1$ (Chiang and Miler 2008). Well-known algorithms for finding association rules include the apriori algorithm (Agrawal and Srikant 1994) and the 1R classifier (Nevill-Manning, Holmes and Witten 1995). A hierarchy by which FDs can be considered as the union of CFDs, and CFDs the union of ARs, has been explained in (Medina and Nourine 2008).

The original measures that were proposed for association rules are support and confidence (Agrawal and Srikant 1994). Various other measures have been proposed since, a survey of which has been provided by (Geng and Hamilton 2006). It has been shown that ranking using other measures results in ARs that have high support-confidence values (Bayardo and Agrawal 1999).

2.4.1 Finding Dependencies

The problem of discovering FDs from a relation R has been studied for many years and shown to be exponential in the arity (number of attributes) of R (Mannila and Raiha 1987). By Armstrong's decomposition rule a dependency $X \rightarrow YZ$ can be split into $X \rightarrow Y$ and $X \rightarrow Z$. So in this section, we consider only CFDs of the form $X \rightarrow A$

containing a single column on the RHS. Since discovering a CFD $(X \rightarrow A, t_p)$ requires finding the pattern tableau t_p in addition to the FD $X \rightarrow A$, it becomes inherently exponential. Given an FD $X \rightarrow A$, the problem of finding associated pattern tableaux t_p has been shown to be NP-complete (Golab, et al. 2008). An algorithm called CFDMiner (Fan, Geerts and Lakshmanan, et al. 2009) has recently been proposed for finding constant CFDs that hold on a given database instance. It cannot be used for finding approximate CFDs i.e., CFDs that are violated due to the presence of a few dirty tuples. There is also an algorithm for discovering CFDs (Chiang and Miler 2008) that extends the well-known TANE algorithm [Huh99] for finding FDs. It can be used for finding both constant and variable CFDs as well as traditional FDs. In this thesis, we assume that the dependencies are already known and do not focus on finding them in the data.

2.5 Database Triggers

Database triggers are one mechanism for implementing integrity constraints. In a database, a trigger or active rule consists of an ECA (event-condition-action) rule that specifies the action to be taken when an (optional) event occurs if the condition is satisfied (Bertossi and Pinto 1999). The generic form of a trigger is as follows:

`ON` *event*
`IF` *condition*
`THEN` *action*

We show here two types of triggers that can be written in the SQL Server DBMS. The first type checks the condition after the event. The event can be one or more of insert, update and delete operations. The name of the trigger and the table on which the event occurs must also be specified.

```
CREATE TRIGGER name
ON table AFTER
INSERT, UPDATE, DELETE
BEGIN
      condition → action
END
```

The second type of trigger checks the condition before the event occurs:

```
CREATE TRIGGER name
ON table INSTEAD OF
INSERT, UPDATE, DELETE
BEGIN
      condition → action
END
```

When the condition needs to be checked on the table without any event occurring, the rule must be written in other ways like using dynamic SQL. This will be useful while cleaning a database with known errors. Assuming all constraints are satisfied in a given database state, triggers can help ensure that the constraints are not violated during subsequent operations.

2.6 Tools

In this section, we describe some of the tools that are available for data cleaning and managing integrity. The data cleaning features of these tools can be divided into deduplication, profiling, schema validation and standardization. Deduplicatiom refers to the process of finding duplicate records and another name for object identification. Profiling is a technique used to map data from an unstructured source (like a text file) to a database schema. Schema validation is the process of checking for discrepancies and inconsistencies in a database schema. Table 2.6 summarizes the capabilities of each of these tools. These tools do not directly address the problem of constraint-based cleaning,

but are focused on ETL operations. Most of these tools are expensive (costing thousands of dollars) and meant to be used by large organizations.

DataFlux (DataFlux Data Quality n.d.) is a subsidiary of SAS Institute that provides software for data management. Its data quality solution provides support for profiling, standardization and transformation of data. DBE software (DBE Software n.d.) supports data quality through validation of database schema. IBM's InfoSphere QualityStage (InfoSphere QualityStage n.d.) allows the user to specify rules for standardization, cleaning and matching data. Trillium Software (Trillium Software n.d.) focuses on data quality for enterprise systems through profiling of customer and sales data. Winpure (WinPure Data Cleaning n.d.) is another tool for performing deduplication and standardization of data.

Table 2.6 Comparison of commercial cleaning tools

Tool	Deduplication	Profiling	Schema validation	Standardization
DataFlux		✓		✓
DBE			✓	
DQguru	✓			✓
InfoSphere				✓
SSIS	✓	✓		✓
Trillium		✓		
WinPure	✓			✓

In our case studies, we compare our approach to the data cleaning support provided by SQL Server Integration Services (SSIS) (SQL Server Data Quality Solutions

n.d.). SSIS can be installed as part of SQL Server and executed through the SQL Server Business Intelligence Development Studio. It is a complex tool and can perform extensive profiling through data flows, in addition to deduplication and standardization. We will also compare our approach to DQGuru (SQL Power DQguru n.d.), a tool for data cleaning and master data management (MDM). It is a platform independent, open-source tool and can be used with any DBMS.

Chapter 3. A Model for Managing Data Integrity

In this chapter, we present our approach to managing data integrity. The conceptual model of data integrity that our approach is based on is described in section 3.3, but first we give an in depth analysis of the problem we are trying to solve in Section 3.1 and define the criteria that will be used to assess any solution to the problem in Section 3.2. Details of the data-integrity and On Deck Zones which are central to our approach are provided in Section 3.4, and in Section 3.5 we step through the main types of data processes to explain how our approach works. In Section 3.6 we introduce a tabular notation to specifying integrity constraints and in Section 3.7 we discuss how to generate database triggers to flag constraint violations.

3.1 Problem Description

Integrity constraint approaches reject inconsistent data but may lose information, while cleaning approaches are ad hoc. The problem is how to provide a systematic approach to managing data integrity that can be used by organizations to ensure clean data. We would also like to minimize information loss by separating the clean data from the dirty data, while supporting integrated views.

To help understand the problems faced by organizations in managing data integrity we illustrate with examples based on our experiences working with electronic health records (EHR) in our two case studies (see Chapter 4). We use a simplified data model representative of the types of data that was seen in our case studies to illustrate the types of constraints and challenges that come in to play when attempting to manage data integrity in the face of typical data processing tasks.

Figure 3.1 shows a subset of the logical model for the EHR data. It shows five tables and a view called DiagnosisEncounters. The underlined columns denote a primary key and the arrows show foreign key relationships. The Patient table contains a record of all patients who have visited the hospital and includes their name, contact information and other personal details. OHIPDiagnosis is a coding table used by physicians when they make a diagnosis that contains a list of all possible diagnoses, grouped by class of diagnosis and system of diagnosis. These are arranged into a classification hierarchy of classes within a system, and diagnoses within a class.

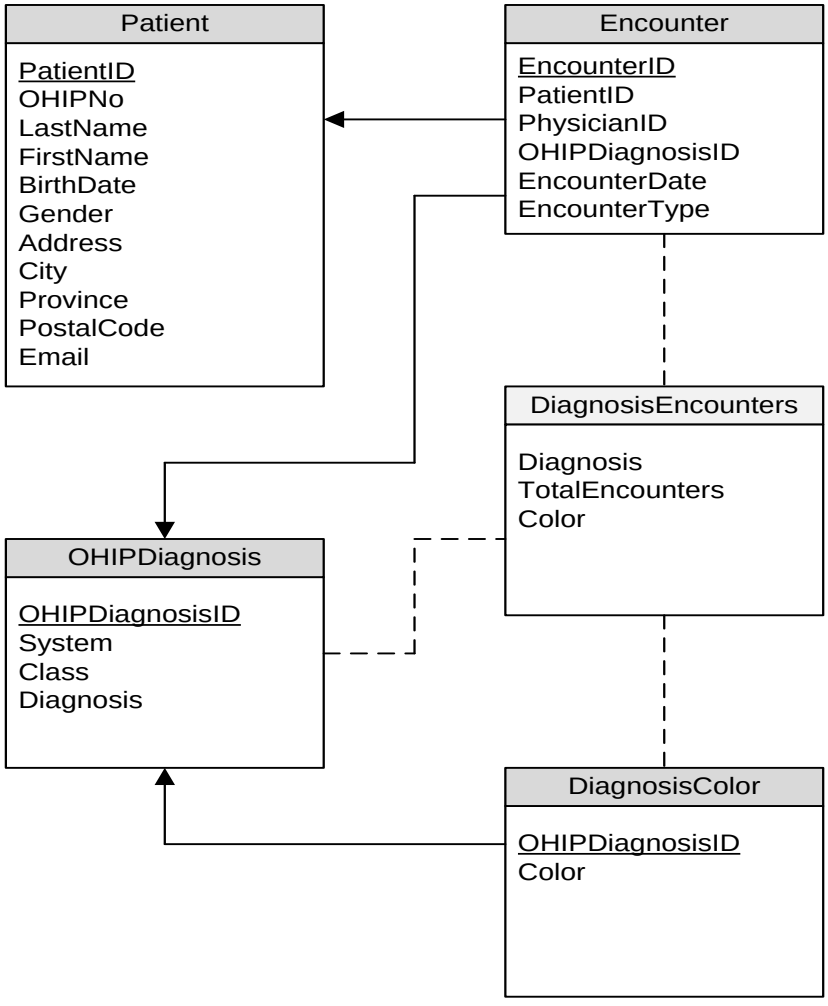


Figure 3.1 Logical Model of Example Data

An example row would be: [Diagnosis = “Prostate”, Class = “Malignant Neoplasm”, System = “Neoplasm”] to represent the condition better known as prostate cancer. The Encounter table keeps a record of all meetings between patients and physicians along with the diagnosis made. The DiagnosisColor table is used for reporting and allows the assignment of a unique color to each rubric. DiagnosisEncounters is a view that aggregates data to show the number of encounters for each diagnosis seen.

Figure 3.2 shows an overview of the processing tasks that are commonly performed in an organization which can affect data integrity. There are two kinds of actors who interact with the data. They are Admins and Users. There are two kinds of databases in the system: Online Transaction Processing databases (OLTP) and Online Analytical Processing databases (OLAP). Sometimes, the OLAP database can be a virtual database created by defining views over an existing OLTP database. However, if the reporting and transactional data have to reside on separate machines, the views will have to be materialized. There has been research on efficient ways to update materialized views (Gupta and Mumick 1995). The user enters transactional data through the App and views reports, while the Admin maintains the OLTP and OLAP databases using tools and SQL as well as running batch processes that load or process data.

Data is entered by the User into the OLTP database through the App (step 6). The OLTP database can be also be loaded directly from data sources through ETL (step 2). This task is performed by the Admin, whose other tasks include data cleaning (step 1) and performing schema changes (step 4) when required. Reports are generated (step 3) directly off the OLAP database or from cube(s) which are multi-dimensional views or extracts of data from the OLAP database. ETL is used to load the OLAP database from

the OLTP database and to create the cubes from the OLAP database (Step 2). The test data (step 5) is used during development of the App for testing purposes before real data gets loaded. Often the development team has no or limited access to real data.

In the following subsections, we analyze each of these six data processing tasks to understand the problems currently faced in managing data integrity.

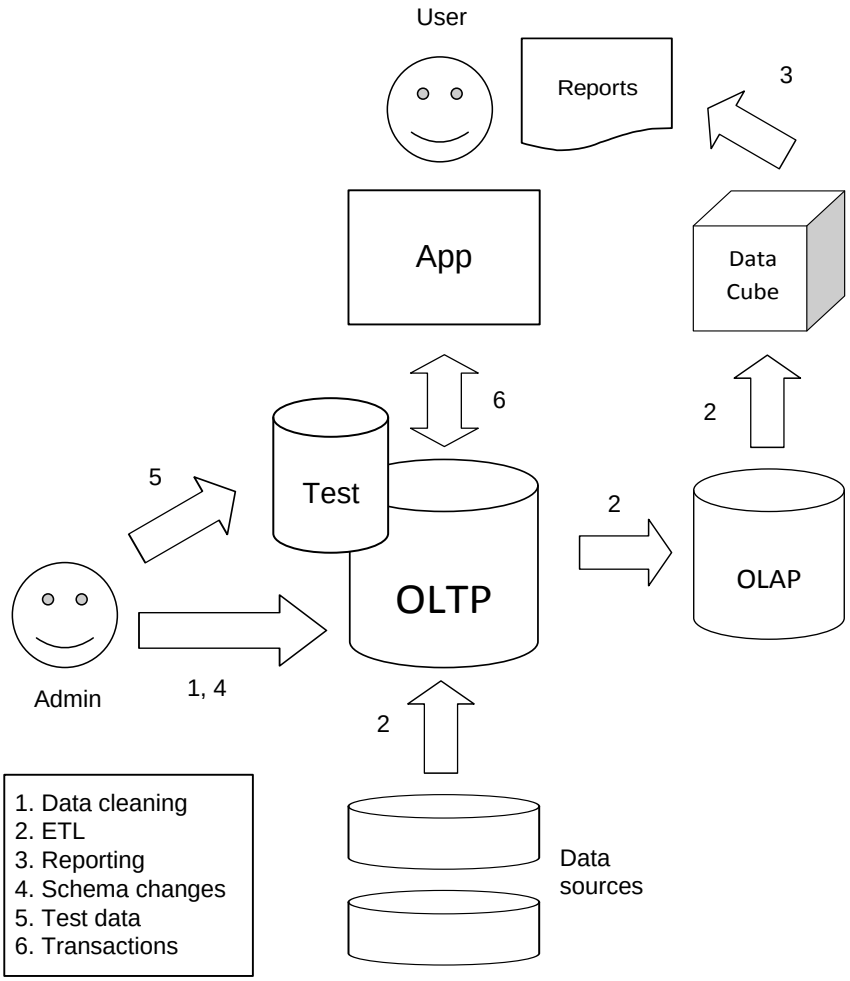


Figure 3.2 Data Processing (EHR)

3.1.1 Data Cleaning

Data cleaning is typically done to fix semantic errors in the data. We use the subset of a Patient table shown in Table 3.1 to illustrate examples of data cleaning. The

email address jw00.yahoo.ca has a syntactic error because it does not contain the @ symbol. If the name of White is required to be spelled Whyte, it becomes a semantic error. Assuming that the postal code uniquely identifies the city, we can detect an inconsistency between patients 3 and 4. The email for patient 3 is missing because it has either not been entered, or the patient might not have an email address. Also, patients 1 and 2 may refer to the same person and, if so, will have to be merged into a single row.

Table 3.1 Illustrating errors in data

Patient ID	Last Name	City	Postal Code	Email
1	Gray	Ottawa	K2E 1V8	agray@hotmail.com
2	Grey	Ottawa	K2E 1V8	agray@hotmail.com
3	Black	Toronto	K4D 3L9	
4	White	Ottawa	K4D 3L9	jw00.yahoo.ca
5	Dark	Ottawa	K6W 5J8	sidd@gmail.com

All of these errors can be expressed as constraint violations based on dependencies. For example, the dependency $\text{Email} \rightarrow \text{LastName}$ would notice the inconsistency between row 1 and row 2. In an ideal world, these errors would have been prevented from occurring (see Section 3.2 ETL and Section 3.6 Transactions for a discussion of this). But, typically such errors do exist in databases and manual methods (e.g. SQL statements) or tools are used to detect them.

In most situations, a domain expert would be required to investigate and manually correct the problem (e.g. how can one know whether Gray or Grey is the correct last name?). Even automatically correcting the email address to be jw00@yahoo.ca might

not suffice since it is possible the correct address was jw0@yahoo.ca but the "@" was misread as "0." The only possible automatic solution is to exclude the row (which is often done when reporting). Unfortunately, this results in information loss.

The problem with manual methods and tools is that there is typically no systematic approach to knowing what data in the database is clean, and what data is waiting to be cleaned.

Data cleaning refers to the process of correcting dirty data so that no constraints are violated. The cleaning operation to be performed is known as a repair, and depends on the type of constraint that is violated. For some constraints, there is more than one possible repair. Some repair problems focus on obtaining a new database that differs minimally from the original database and satisfied all the constraints (Fan, Geerts and Jia, A Revival of Integrity Constraints for Data Cleaning 2008).

3.1.2 ETL

Transforming data as one moves data from a source database to a central data warehouse can possibly introduce errors, but typically this is also an opportunity to flag and fix errors that might exist in the source. If the sources are not clean, then one can at least hope that the central data warehouse will be clean.

A typical use of ETL is for data integration in which data from different sources is combined to provide the user with a unified view of these data (Lenzerini 2002). The unified view can be either virtual or materialized. A virtual data integration system uses the mediator-wrapper architecture consisting of source schemas, a global schema, and a mapping between them (Batini and Scannapieco, Data Quality: Concepts, Methodologies and Techniques 2006). A data warehouse is an example of a place where materialized

data integration is used. The tasks involved in performing data integration are schema mapping, duplicate detection, and data fusion (Dong and Naumann 2009). Data inconsistency can occur when information comes from multiple sources (Rahm and Do 2000). In general, as is the case with data cleaning, the only automated fix to inconsistencies that can be made is to reject incoming rows with inconsistencies. This results in the loss of information. Typically, it requires domain experts to resolve.

3.1.3 Reporting

Data warehouses are often structured or modeled as OLAP data sources in order to support reporting. For example, a multidimensional model of data can be constructed on top of the warehouse by defining views. Common relational representations of these models are the star schema and the snowflake schema (Chaudhuri and Dayal 1997). Another approach is to structure the data as a cube, which can be used in commercial reporting tools like Analysis Studio. An OLAP schema will contain fact and dimension tables (Kimball and Ross 2002) and makes it easier to define data cubes that can be used in reporting applications.

In our eHealth example, we wanted to generate a report showing the number of visits based on our DiagnosisEncounters view. A column chart is a useful representation for this purpose. A constraint on this report could be that each diagnosis must be represented by a unique color. This can be achieved using the dependency [Color → OHIPDiagnosisID]. However, the implementation of this dependency is problematic, if we wish to allow users to dynamically select what colors represent what diagnoses when looking at a chart. There are hundreds if not thousands of possible diagnoses and it is difficult for a user to select unique colors. It would be awkward if this constraint blocked

users from picking colors that were already used and forced them to guess which colors had not already been used and only select from those. A flexible approach is needed that allows users to make inconsistent choices, while ensuring that they are aware of the inconsistencies.

3.1.4 Schema Changes

Changes to the database schema can occur due to changes in business requirements. For example, new columns may need to be added to a table for storing additional information. In Table 3.2, we show the impact of different types of changes on different types of dependencies. Adding a new column to a table and adding a new table to the database do not affect any of the existing dependencies since they depend only on the columns on which they were defined.

Table 3.2 Dependencies affected by schema changes

Change	**FD**	**CFD**	**IND**	**CIND**
Add column				
Add table				
Delete column			✓	✓
Delete table			✓	✓

However, deleting a column affects inclusion and conditional inclusion dependencies if the deleted column contained part of the data that was to be included. Similarly, deleting a table can affect existing INDs and CINDs. If the INDs have been defined as foreign key constraints, the DBMS will prevent the column or table from being deleted.

3.1.5 Test Data

Test data, also known as synthetic or artificial data, is used when there is not enough real data for testing, especially in terms of volume. There are commercial tools like RedGate (Redgate: Tools for SQL Developers n.d.) that can populate a database table with data corresponding to the column type. These tools have limited support for constraints, and the resulting data set may require manual cleaning. Adding constraints to test data is challenging due to the randomness of the data. Since it is often machine-generated, it bypasses any validation of data that may have been built into an application. This makes the validation of test data using constraints all the more essential. Test data is typically used to simulate data entry which is addressed in Section 3.6. It can often be specified as a simplified ETL task to feed data into a data warehouse from a "test" source facing some of the same problems discussed in Section 3.2. It can be used either in the OLAP or the OLTP databases.

3.1.6 Transactions

Consider the constraint: $[\text{Diagnosis} = \text{"Prostate"}] \rightarrow [\text{Gender} = \text{"Male"}]$ which can be used to specify that if a patient is diagnosed with prostate cancer then the gender must be male. The constraint will be violated if there are female patients with prostate cancer. This example might seem trivial but such errors are common in health care settings where a data entry person has to transfer data from hand-written forms and can accidentally select the wrong diagnosis or gender, or when test results on paper are placed in the wrong file (e.g., placing a male patient's prostate cancer test results in a female patient's file) and subsequently updated in the system by a clerk. The Encounter

table functions as a link between the gender column in the Patient table and the diagnosis column in the OHIPDiagnosis table by using the patient and OHIP diagnosis id attributes.

The violation can be detected during a transaction by checking if the resulting rows contain a female patient with prostate cancer. This violation could be resolved by rejecting the row; changing the patient's gender to male; or changing the diagnosis to something different from prostate. However, it could not be resolved automatically since it requires a domain expert (or at least someone directly familiar with the person) to determine the semantically accurate resolution. The different types of constraints that are affected during different types of transactions are marked in Table 3.3.

Table 3.3 Constraints affected by transactions

Operation	**FD**	**CFD**	**IND**	**CIND**
Insert	✓	✓	✓	✓
Update	✓	✓	✓	✓
Delete			✓	✓

3.2 Evaluation Criteria

We have identified evaluation criteria based on our analysis of gaps in existing tools for data cleaning, as well a review of the literature. These criteria are completeness, effort, scalability and skills required. We use these in chapter 5, to compare CBIM on a general level to methods based on manual cleaning or on logical consistency. For evaluating CBIM in detail, in addition to completeness, effort and skills required, we introduce another criterion called convenience that refers to the ease with which data that

satisfy or violate constraints can be viewed together or separately. We evaluate the scalability of CBIM separately in our experiments.

These criteria were chosen based on our interactions with customers and developers during development of the two projects with healthcare data mentioned in our case studies: PAL-IS and iMED-Learn, Completeness was an issue in PAL-IS where forms that were traditionally filled in by hand were being manually entered into the system. Also, constraints had to be handled on an ad hoc basis and some violations went unnoticed. We also observed that using manual tools to do the cleaning required some knowledge of SQL. So we decided to evaluate the effort needed to achieve completeness.

During development of PAL-IS, we found that the database was already dirty and it would help facilitate cleaning if we could have separate views of the clean and dirty data in the database. This is addressed by our convenience criterion. Also, handling integrity constraints required a domain expert who could understand the nature of the data. So we decided to evaluate our approach based on the skills required to use it. Completeness depends not only on whether all constraint violations are captured but also on whether all constraints have been specified. Effort refers to the amount of time a person would need to spend in specifying constraints and managing violations. Skills required refer to the understanding needed to handle constraints and their violations. The following list shows these criteria and the factors that influence them:

- Completeness is dependent on the following factors:
    - Specifying all constraints
    - Capturing all constraint violations
    - Fixing all constraint violations

    - Ensuring that fixes are semantically accurate
    - Minimizing information loss
- Effort: This refers to the time and monetary cost involved in
    - Specifying constraints
    - Flagging constraint violations
    - Fixing constraint violations
    - Ensuring semantic accuracy
    - Minimizing information loss
- Convenience with which clean and dirty can be separated:
    - Viewing only the clean data
    - Viewing only the dirty data
    - Viewing why data is dirty
    - Viewing all the data
- Skills required to do the following tasks:
    - Specifying constraints
    - Flagging constraint violations
    - Fixing constraint violations

3.3 Constraint-Based Integrity Management

In this subsection, we will explain our approach to constraint-based integrity management in terms of a conceptual model for data integrity. In subsequent sections, we provide more details on possible physical implementations of our conceptual model, as well as detailing our approach to how constraints can be specified and implemented within the context of this model.

3.3.1 Our Approach

In general, organizations must decide what data integrity means to them, and come up with a list of integrity constraints for their data. A careful analysis of the processes is required, and the constraints relevant to each database need to be defined. In our approach, called constraint-based integrity management (CBIM), we assume organizations start with an empty database and define the constraints that will be need to be enforced on the data.

We define a conceptual model for CBIM to show incoming rows from transactions against a database being dynamically sorted into either an On Deck zone or a Data Integrity zone within a database.

Our conceptual model is implemented using tables and views in a database and involves the use of triggers.Triggers are then created that validate data coming into the database, through inserts or updates. The constraint violations are flagged to separate dirty data from the clean data, while providing support for an integrated view of all data. In this way, no information is lost from the database.

We perform a systematic analysis of the types of data processing tasks in an organization that can affect data integrity to understand the issues involved and we perform a systematic analysis of the types of data integrity constraints that might be defined.

We also created a tabular notation for specifying constraints and a template-based process of generating triggers. Each of these is explained in a separate subsection.

3.3.2 Conceptual Model

Figure 3.3 shows our proposed conceptual model for data integrity. It ensures that data integrity is maintained at all times. Once we have "clean" data it stays clean. At the same time, it manages effectively the process of resolving or cleaning potentially "dirty" data when there is an attempt to add it to the system, rather than simply rejecting such data. This is achieved by a clear segmentation between a Data Integrity Zone where data is guaranteed to be "clean" and an On Deck Zone in which "dirty" data is flagged for cleaning.

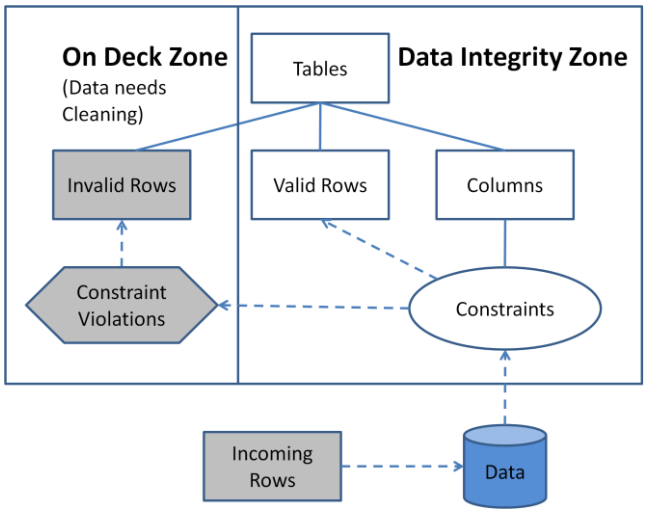


Figure 3.3 Conceptual Model of Data Integrity

Three views are defined. There is a complete "all data" view of both the "dirty" and "clean" data (a union of the On Deck and Data Integrity Zones). One can also choose a "data integrity" view of only the "clean" data (Data Integrity Zone), and one can choose an "on-deck" view of the flagged data and their associated constraint violations (On Deck Zone) in order to systematically clean the data.

The Data Integrity Zone is defined by constraints that are defined on columns within the database tables. When new data enters the database, it is checked whether the incoming row violates any of the existing constraints. If so, the data is flagged and moved to the On Deck Zone. The On Deck Zone contains all the invalid rows and the particular constraint violations that flagged them as invalid. If incoming data satisfies the constraints, it is added or updated into the valid rows in the tables.

Constraints could also be different from the dependencies we have seen so far. In an application with multi-language support we could have a constraint which says that when the content in a particular language is changed, all other languages must be modified to accurately translate the new content. For example, when a description in English is changed, the French description must also be changed. This could be achieved by implementing a "constraint" that moves French rows to the On Deck Zone whenever English rows are updated, flagging them for review by a domain expert.

3.4 Data Integrity Zone and On-Deck Zone

There are two different approaches that can be taken for implementing the On Deck Zone. The conceptual model could be implemented by storing the data in two databases with the same tables: one for valid rows and one for invalid rows. However, issues may arise if there are references between invalid rows and valid rows in different tables through foreign key references.

A second approach is to keep both valid and invalid rows in the same physical table but create a new column for each table in order to flag invalid rows. Views could be used to define the zones. This approach has some performance issues since the views will have to be recomputed after changes to the table.

A prototype implementation based on the first approach is explained in detail in chapter 4.

3.5 Data Integrity Processes

In this section, we take a look at each of the types of data processing that were identified in Section 3.2 to analyze how our conceptual model can be used to manage data integrity and improve upon issues faced with either ad hoc tools or logic-based integrity constraints that reject rows with constraint violations.

3.5.1 Data Cleaning

The idea behind our conceptual model is to catch dirty data before it can enter the Data Integrity zone. In that respect, the problem of data cleaning is transformed into one of ETL. All data in the Data Integrity Zone is clean. If one attempts to create dirty data via a transaction or ETL process then the incoming rows are automatically flagged and placed in the On Deck Zone waiting to be cleaned (transformed), and once cleaned, loaded into the Data Integrity Zone.

If one starts with a database that is dirty (which in our approach could only happen with a schema change or the addition of a constraint), then a process or tool would have to be run over the database to recheck all potentially affected rows for possible constraint violations and move all affected rows to the On Deck zone.

3.5.2 ETL

Data integrity issues are common during ETL processes due to the nature of the source data which is often in a different format to the OLAP data. However, since the nature of the source and target data can be different, designing an OnDeck Zone for

storing the unformatted source data can be challenging. There are many data cleaning tools specialized for handling this kind of operation and provide efficient mechanisms to structure and reformat the data. The operations occurring ETL are similar to the operations that happen during transactions through a user application. As shown in Figure 3.3, when a row is inserted or updated it is checked against the relevant constraints and, if there are no violations, allowed into the data integrity zone. If there is a violation of an existing constraint, then the row is flagged and placed in the on-deck zone.

3.5.3 Database Reporting

In some sense, one would expect reporting to have no issues in terms of data integrity since it is simply querying the data, not modifying it. However, there are additional integrity constraints specific to reports to consider. One may need to ensure that the extracted data is well formed and meets additional requirements, and there may be extra data defined and maintained which are used to configure the presentation for the report.

Consider again the constraint $[\text{Color} \rightarrow \text{OHIPDiagnosisID}]$ used for reporting. If the user assigns a color to a diagnosis X that has already been used for another diagnosis Y, then the the row in the DiagnosisColor table containing X is flagged as invalid since it is a duplicate color. However the report writer can choose to restrict the report to only diagnoses with valid color assignments (“clean” but “incomplete”), or it can allow a chart to be drawn with duplicate colors (“dirty” but “complete”) and the issue is flagged so that it will eventually be resolved.

3.5.4 Schema Changes

Schema changes are more complex. An analysis needs to be made to see if the schema change is independent of any constraint (e.g. adding a column). If it is, then there is no impact. However, if the schema does potentially impact a constraint, then in fact, we are creating a new version of the data. Potentially, this creates an ETL situation, in which all rows from the old version of the database have to be added to the new version of the database in order to check for constraint violations.

3.5.5 Test Data

Test data is handled similarly to the data used during transactions but, since it is typically loaded in batch, the time delays to review flagged rows are not problematic. Managing the integrity of test data can help simulate the issues that arise when real data is eventually used. As with ETL, our approach allows constraints to be specified with the data so that the same rules apply for Transactions, ETL and Test Data.

3.5.6 Transactions

Consider the example $[\text{Diagnosis} = \text{"Prostate"}] \rightarrow [\text{Gender} = \text{"Male"}]$ which we saw in Section 3.1.6. If this constraint is violated during a transaction by any row in the database, the row is flagged and placed in the On Deck Zone. However, depending on the application, it may result in complex application logic to maintain a view of both "clean" data and "dirty" data. A purely logical approach toward satisfying this constraint would have rejected the offending row or made an incorrect change to the data. Our mechanism encompasses the logical approach, and includes the ability to retain data that would have been logically rejected. This helps to minimize information loss. It allows the clean and dirty data to be viewed separately through views, and also provides an integrated view of

all the data. There are some constraints however that, when violated, force a rejection of the data. For example, if the primary key constraint is violated, the data will have to be rejected since it would not be allowed by the DBMS in any case.

3.6 Constraint Specification

A critical aspect of our approach is the specification of constraints. Our conceptual model requires a mechanism both for defining constraints and for implementing them in a manner that will flag and move rows with constraint violations into the OnDeck Zone. We have developed an easy to use tabular notation for specifying constraints as dependencies that are to be enforced dynamically on data. In this section we describe the tabular notation while in section 3.7 we discuss how to generate triggers from the notation that will implement our conceptual model.

In table 3.4, we show an example of constraint specification in our tabular notation using the example EHR data model we defined in section 3.1

Table 3.4 Specifying constraints

Constraint	Antecedent Table	Antecedent Columns	Antecedent Values	Consequent Table	Consequent Columns	Consequent Values	Description
Functional	Patient	OHIP			LastName		
Functional	Patient	Diagnosis Type	Prostate		Gender	Male	
Inclusion	Encounter	PatientID		Patient	PatientID		
Inclusion	Encounter	PatientID, Encounter Type	, Phone	Patient	PatientID		

Constraints of the form $X \rightarrow Y$ and $X \subseteq Y$ are written by referring to X and Y as the antecedent table and consequent table respectively. For an FD, the consequent table is blank since a functional dependency is defined on a single table. Conditional dependencies (both functional and inclusion) are written by assigning values to the

columns listed. In Table 3.4 the entry ", Phone" in antecedent values (in the last row of the table) means that no value is assigned to the first column (PatientID) in the antecedent table (Encounter), and a value of "Phone" is assigned to the second column (EncounterType). The last column is for a description that can be used to explain all these while listing the constraints. If the constraint is not a dependency then the first column may be left blank. The constraints table can be filled in directly or by using our tool (see Section 4.2.1). The tool uses the word "Other" in the constraint column for those constraints that are neither functional nor inclusion dependencies. These and all other constraints types we support are explained in more detail below.

We now look at how some of the common constraints encountered in databases can be specified using the notation in Table 3.4. The description column is optional but will be used to explain the constraint while flagging violations.

- Functional dependency (FD): $Table: Col1 \rightarrow Col2$

  Since a functional dependency is defined on a single table, the only table that needs to be specified is the antecedent table. In this case, the Col1 is the antecedent column, and Col2 is the consequent column. The columns for antecedent and consequent values are left empty for an FD.

- Key dependency: A key dependency can be specified by using the primary key constraint in a DBMS. It is equivalent to specifying FDs for every column in the table. So a primary key (not composite) on a table with five columns will require four FDs to be specified. This is because we expect in our implementation that the consequent for an FD has a single column (see Section 2.4.1 for an explanation). It could also be defines as a single constraint of type "Other" and explain the

constraint in the constraint in the Description column. The primary key column will be the Antecedent column and the four remaining columns will be the Consequent columns.

- Constant CFD: In a constant CFD every column has a value assigned to it. For example, $Table: [Col1 = "Val1", Col2 = "Val2"] \rightarrow [Col3 = "Val3"]$. Here the antecedent columns are Col1 and Col2 while the consequent column is Col3. The antecedent values are "Val1" and "Val2" while the consequent value is simply "Val3". Again, the consequent table is left empty since it is actually the same as the antecedent table.
- Variable CFD: In a variable CFD, there is a "_" in the antecedent and consequent value columns. The underscore value in a column is used in conditional (both functional and inclusion) dependencies to mean that the column can take any value from its domain. However, since we allow only one column in the consequent, the consequent table can have at most one "_". A CFD of the form $Tabl: [Col1 = Val1, Col2] \rightarrow [Col3]$ will have the antededent values Val1 and "_" respectively while the consequent value will be "_". For a variable CFD to be well-formed there must be at least one "_" in the consequent if there is a "_" in the antecedent, and vice-versa. This is to avoid a CFD of the type $[A = a_1] \rightarrow B$ which is of little use and simply means that when column A has the value a_1 column B can have any value.
- Inclusion dependency: Just like in an FD, an IND will have no entries for the antecedent and consequent values. However, there will be an entry for the consequent table column. For the IND $X[A] \subseteq Y[B]$, the values for antecedent

table, consequent table, antecedent column and consequent column are respectively X, Y, A and B. It must be noted that for an IND, the number of columns in the antecedent and consequent must be the same.

- Foreign key constraint: This is an inclusion dependency in which the consequent column is the primary key of the consequent table. This constraint can be enforced directly in a DBMS and so, does not need a trigger generated.
- Conditional inclusion dependency: Unlike an inclusion dependency, a CIND can have different number of columns in the antecedent and consequent. The antecedent and consequent values can be a constant or a "_". In a CIND, the number of "_" must be the same in both the antecedent and consequent columns, and the ordering of columns is important. In an IND $X[A, B = b] \subseteq Y[C, D = d, E = e]$ the antecedent and consequent tables are X and Y respectively. In this notation the columns are separated by commas, and the values assigned to individual columns are shown. When no value is assigned to a column, it is equivalent to assigning the value "_" to that column, and is not shown in the dependency. The antecedent columns in the example are A and B, and the consequent columns are C, D and E. The antecedent and consequent values are (_,b) and (_, d_, e) respectively.
- Join dependency: We currently do not have an implementation for generating triggers for join dependencies. However, they can be easily specified in our tabular notation. For example, a JD of the type $X \rightarrow\rightarrow Y$ on a table R will have antecedent table R, antecedent column X and consequent column Y.

- Other constraints: Other types of constraints can include semantic rules like the format for a telephone number, presence of @ in an email address, etc. These will have to be handled separately by the Admin since the generation of triggers will not be automatic. The description column in the constraints table is particularly useful here, since the person generating the triggers will be able to understand what the constraint is supposed to do, and the person fixing a constraint violation can look up the description of the constraint violated. For other constraints, columns like the antecedent table and antecedent columns can still be used to specify the table and columns on which the constraint is being defined. This brings a degree of organization to the way in which the constraints are specified.

3.7 Generating Triggers for Managing Data Integrity

One approach to constraint-based maintenance of data integrity is to implement the constraints as triggers. We can generate triggers for constraints specified in our tabular notation that explicitly manage data integrity in terms of the OnDeck and Data Integrity Zones mentioned in our conceptual model (Section 3.3). As well we have an algorithm for generating triggers directly from the compact table-based notation we defined in section 3.4.

In table 3.5 we list guidelines as to what sort of triggers need to be implemented for the different types of dependencies. Once the data is clean, the triggers can be generated according to the guidelines in Table 3.5 to check if new inserts or updates violate any dependencies. For conditional dependencies, t_p denotes the pattern tableau. An expression of the form $X = t_p[X]$ means that every column specified in X has the

value specified in t_p for that column. The trigger description shows the condition to check for depending on the type of constraint.

Table 3.5 Constraint implementation guidelines

Dependency type	**Trigger description**
Functional dependency $R: X \rightarrow Y$	Check if there is more than one row in table R with the same X value but different Y values.
Conditional functional dependency $R: X \rightarrow Y, t_p$	(i) Constant CFD: Check if there is a row in R where $X = t_p[X]$ and $Y \neq t_p[Y]$. (ii) Variable CFD: Check in rows where $X = t_p[X]$ is true if the FD $X \rightarrow Y$ is violated or $Y \neq t_p[Y]$.
Inclusion dependency $R[X] \subseteq S[Y]$	Check if there are rows in table R where the value in column X is not in column Y of table S.
Conditional inclusion dependency $R[X] \subseteq S[Y], t_p$	Check if there are rows in table R where $X = t_p[X]$ and there is no row in table S where the value in column X is in column Y of table S, and $Y \neq t_p[Y]$.

These triggers will be in SQL with conditions being checked during an insert, update or delete transaction. In all cases, if the condition is true, the trigger's action will be to flag the incoming row or the row that is being updated and place it in the On Deck Zone. The following is an example of a trigger created in SQL Server for enforcing the CFD $[\text{City} = \text{"Ottawa"}, \text{PostalCode}] \rightarrow [\text{Address}]$. It checks all inserted or updated rows that contain the city Ottawa whether there is a postal code with more than one address.

```
CREATE TRIGGER TRIG1 ON Patient
INSTEAD OF INSERT, UPDATE AS
BEGIN
```

```
DECLARE @zip VARCHAR(100);
DECLARE @newAddress VARCHAR(100);
DECLARE @oldAddress VARCHAR(100);
SELECT @zip = (SELECT PostalCode
FROM INSERTED WHERE City = 'Ottawa');
SELECT @newAddress = (SELECT Address
FROM INSERTED WHERE City = 'Ottawa')
SET @oldAddress = (SELECT DISTINCT Address FROM
Patient WHERE City = 'Ottawa'
AND PostalCode = @zip)
IF @newAddress = @oldAddress
OR @oldAddress IS NULL
INSERT INTO Patient SELECT * FROM INSERTED
ELSE
     //place affected rows in On Deck Zone
END
```

The above trigger is an example of a variable CFD. We now look at the generic forms of different triggers based on the type of dependency. The following statements are common among all the triggers:

```
CREATE TRIGGER TRIG_(1)_
ON _(2)_ AFTER INSERT, UPDATE AS
BEGIN
  _(3)_
END
```

In the above SQL, (1) is the name of the trigger, (2) refers to the table on which the trigger is being generated, and (3) is the body of the trigger that depends on the type of dependency. We look at five types in total. The following is a list of what needs to go into (3) above for each type of constraint.

Constant CFD: In this type of functional dependency, all columns have been assigned specific values. (4) refers to the consequent column, (5) to the antecedent column, (6) to

the antecedent value, and (7) to the consequent value. (8) refers to the on-deck table in which the dirty row will be placed. The antecedent and consequent columns must come from the same table, and the only operator allowed in the expressions is equality. This is because we are not using any expression engine in our code for generating the triggers. However, this was sufficient to model the constraints we encountered in our case studies.

```
DECLARE @count INT;
DECLARE @rhs VARCHAR(1000);
SET @rhs = (SELECT _(4)_ FROM INSERTED);
SET @count = (SELECT COUNT(_(4)_)
              FROM INSERTED
              GROUP BY _(5)_
              HAVING _(5)=(6)_);"
IF @count > 0 AND @rhs <> _(7)_
  INSERT INTO _(8)_ SELECT * FROM INSERTED;
```

Variable CFD: In this constraint, there is at least one underscore value defined in the antecedent and consequent values. (4) refers to the consequent table and (5) lists the antecedent columns (comma separated). (6) is a bit more complex and involves comparing the antecedent columns to their respective antecedent values. If the value is an underscore then it is obtained from the INSERTED table. For example for a variable CFD $[A = a, B] \rightarrow [C]$, the expression in (6) would be `A='a' AND B=(SELECT B FROM INSERTED)`. (8) refers to the on-deck table in which the dirty row will be placed. Here, again, the only operator allowed in the expression is equality.

```
DECLARE @count INT;
DECLARE @rhs VARCHAR(1000);
SET @rhs = (SELECT _(4)_ FROM INSERTED);
SET @count = (SELECT COUNT(DISTINCT _(4)_)
              FROM INSERTED
              GROUP BY _(5)_
              HAVING _(6)_);
IF @count > 1
```

```
INSERT INTO __(8)__ SELECT * FROM INSERTED;
```

Functional dependency: In this constraint, the columns for antecedent and consequent values will be empty. (4) refers to the consequent column, (5) to the antecedent table and (6) to the antecedent column. (7) is an expression comparing the antecedent columns to their respective antecedent values. (8) refers to the on-deck table in which the dirty row will be placed. In an FD, the antecedent and consequent columns must belong to the same table.

```
DECLARE @rhs INT;
SET @rhs = (SELECT COUNT(DISTINCT __(4)__)
            FROM __(5)__ GROUP BY __(6)__ HAVING __(7)__);
IF @rhs > 1
  INSERT INTO __(8)__ SELECT * FROM INSERTED
```

Inclusion dependency: In this dependency, the antecedent and consequent columns are empty (like in an FD). (4) refers to the consequent column, (5) to the consequent table, and (6) is an expression equating the consequent column to the antecedent column from the INSERTED table. For example, for the IND $A[X] \subseteq B[Y]$ the expression would be Y=(SELECT X FROM INSERTED). (8) refers to the on-deck table in which the dirty row will be placed. The number of antecedent and consequent columns must be the same.

```
DECLARE @rhs INT;
SET @rhs = (SELECT COUNT(__(4)__)
            FROM __(5)__ WHERE __(6)__);
IF @rhs < 1
  INSERT INTO __(8)__ SELECT * FROM INSERTED
```

Conditional inclusion dependency: This constraint is an inclusion dependency containing values in the columns for antecedent and consequent values. (4) is an expression that compares the antecedent columns to their respective antecedent values

(when the value is not an underscore), (5) refers to the consequent table, and (6) is an expression comparing all consequent columns to their respective consequent values or their values in the INSERTED table. (8) refers to the on-deck table in which the dirty row will be placed. Just like in CFDs, the conditions specified in the CIND may only consist of equalities. Inequalities and other operators are not permitted.

```
DECLARE @lhs INT;
DECLARE @rhs INT;
SET @lhs = (SELECT COUNT(*) FROM INSERTED
            WHERE _(4)_);
IF @lhs > 0
BEGIN
  SET @rhs = (SELECT COUNT(*) FROM _(5)_
              WHERE _(6)_);
  IF @rhs < 1
    INSERT INTO _(8)_ SELECT * FROM INSERTED
END
```

In the above examples, we looked at how triggers can be generated for capturing violations of some common types of dependencies. They do not cover all possible dependencies and, as we have seen, there are limitations on the types of data and expressions that can be used. For example, the only relational operator allowed in the conditions is equality, and regular expressions are not supported. Also, disjunction and negation are not permitted. Nonetheless, they provide support for a significant number of dependencies, and were sufficient for our case studies.

Chapter 4. Case Studies

4.1 Overview

We now look at case studies on two systems we helped to develop, PAL-IS and iMED-Learn, that are used in healthcare settings. PAL-IS has been developed in consultation with Elisabeth-Bruyère Hospital, Ottawa, and is being used to manage data about palliative care given to patients. iMED-Learn has been built in consultation with a physician from the Ottawa Hospital Family Health Team. It contains data about patient visits to family physicians, and its purpose is to provide feedback to physicians on the breadth and depth of their clinical experience.

In this chapter we first discuss how we prototyped an implementation of our conceptual model and created supporting tools for defining and generating constraints. Then, for each case study we look at the data processes that are relevant and describe how our conceptual model and supporting tools were used to help manage data integrity. This is contrasted with how data integrity was managed before our approach was introduced. We also look at how other tools could be used to manage data integrity. Finally, we performed experiments to test the scalability, performance and usability of our approach for these systems. Chapter 5 provides details about the results of these experiments and an evaluation of our approach in comparison with other approaches

4.2 Prototype Implementation

We built a prototype implementation of our approach that was common to both of our case studies. Both case studies used SQL Server to store data and ASP.NET to develop applications that either updated or reported on the data. We describe how we

specified constraints, implemented our conceptual model and generated triggers that would flag constraint violations.

4.2.1 Specifying Constraints

We built a web-based interface (see Figure 4.1) for specifying constraints as dependency relations in a table format and prototyped a tool to generate SQL triggers for each constraint that would check each transaction against the database and flag any rows that violated the constraint. The interface is a simple data entry table for entering constraints.

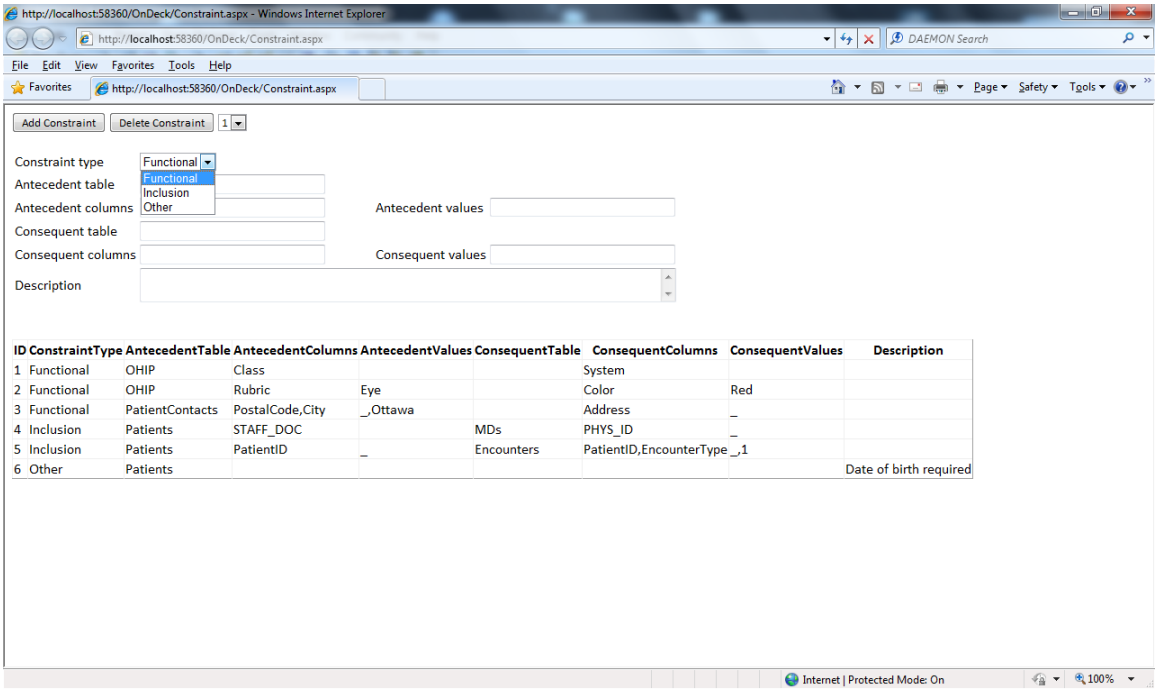


Figure 4.1 Tool for specifying constraints

To store the constraints defined in our interface, we created a table in SQL Server called "Constraints". Figure 4.2 shows a screenshot of the table containing all eight fields from the interface of Figure 4.1 (same as the model shown in Table 3.4 from

chapter 3) as well as an ID column that is used as a key. The ID can be referenced when there is a constraint violation to indicate the cause.

ID	ConstraintType	AntecedentTable	AntecedentColumns	AntecedentValues	ConsequentTable	ConsequentColumns	ConsequentValues	Description
1	Functional	OHIP	Class	*NULL*	*NULL*	System	*NULL*	*NULL*
2	Functional	OHIP	Rubric	Eye	*NULL*	Color	Red	*NULL*
3	Inclusion	Patients	STAFF_DOC	*NULL*	MDs	PHYS_ID		*NULL*
4	Inclusion	Patients	PatientID	_	Encounters	PatientID,EncounterType	_,1	*NULL*
5	Other	Patients	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	Date of birth required

Figure 4.2 Constraints table

The columns defined in the Constraints table in Figure 4.1 are respectively ID, ConstraintType, AntecedentTable, AntecedentColumns, AntecedentValues, ConsequentTable, ConsequentColumns, ConsequentValues and Description. The ConstraintType indicates the type of dependency and will be used by our tool to generate the appropriate trigger. Currently, the two possibilities are “Functional” and “Inclusion”. Conditional dependencies are indicated by the presence of an underscore “_” in the AntecedentValues and ConsequentValues columns. The Description column allows the person entering the constraints to write a short note explaining the constraint that can then be displayed to explain what the problem is if the constraint is not satisfied. It is also used to explain constraints that are not dependencies. For example, constraint 5 in Figure 4.1 is not a dependency and is explained in the “Description” column. It states that the date of birth must be a required field. Some of the other constraints, when written in logic notation as a dependency relationship, are shown in Table 4.1. A description of these dependencies in English is also shown.

Table 4.1 Different types of constraints from Figure 4.2

1	$\text{OHIP: [Class]} \rightarrow \text{[System]}$ In the OHIP table, a class must not belong to more than one system.
2	$\text{OHIP: [Rubric = "Eye"]} \rightarrow \text{[Color = "Red"]}$ In the OHIP table, if the Rubric column has the value "Eye" its corresponding color when displayed must be red.
3	$\text{Patients[STAFF\_DOC]} \subseteq \text{MDs[PHYS\_ID]}$ All values in the STAFF_DOC column of the Patients table must also be in the PHY_ID column of the MDs table.
4	$\text{Patients[PatientID]} \subseteq \text{Encounters[PatientID, EncounterType = 1]}$ Every value in the PatientID column of the Patients table must have at least one row the Encounters table with the same PatientID value and EncounterType column value of 1 (which means new patients can only be entered if they have an Initial Assessment form filled out).

4.2.2 Implementing the Conceptual Model

There are several possible ways to implement our conceptual model for the databases in each case study. For our prototype implementation, our On Deck Zone where constraint violations were flagged was implemented as "shadow tables", while our Data Integrity Zone was a view. We created a shadow table for each table in the database in order to define our On Deck Zone. Each shadow table had all the columns of the original table it was shadowing plus a constraint column that stores the ID of the constraint that was violated, and a column for showing the description of the constraint so that it becomes immediately clear what caused the row to be "flagged" into the On Deck Zone. The Data

Integrity Zone was defined as a set of views, one for each original table, that contains rows from the original table, that have NOT been flagged in the shadow On Deck table.

Figure 4.3 shows a table called OHIP for which a table called OHIP_OnDeck has been created to flag rows from the OHIP table that violate constraints. It has a composite primary key made up of the key from the OHIP table and the ID of the constraint. This allows a row to be flagged more than once in the table if it violates multiple constraints. The Data Integrity Zone view of the OHIP table is defined by a view containing all rows that are in OHIP but not in OHIP_OnDeck. The view definition is as follows:

```
CREATE VIEW OHIP_Integrity AS
SELECT * FROM OHIP
WHERE DxCode NOT IN (SELECT DxCode FROM OHIP_OnDeck)
```

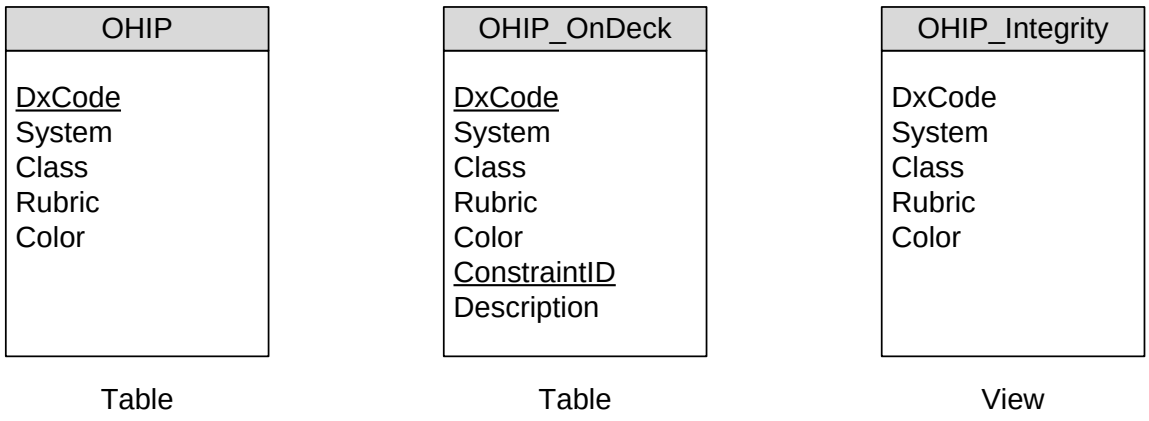


Figure 4.3 Data integrity and On Deck Zones for OHIP table

4.2.3 Generating Triggers

Finally, we prototyped a tool for generating the database triggers that flagged rows into the On Deck Zone whenever a transaction took place against the database that violated a constraint. Figure 4.4 shows a screenshot of our interface used for generating

triggers. It shows the data in the constraints table and allows the user to select a constraint from the drop-down by using its ID (in this case Constraint 4 is selected). When the "Trigger" button is pressed, the SQL code is generated and shown in the text box below the constraints. The SQL statement can be copied and used in a DBMS to create a trigger.

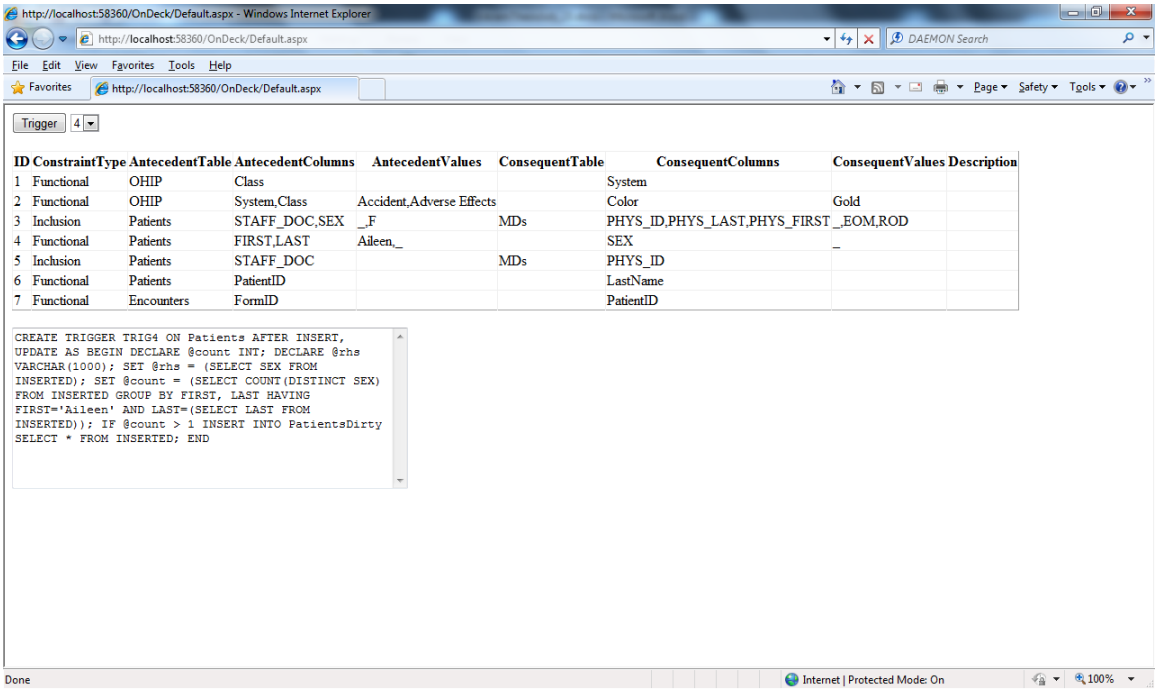


Figure 4.5 Trigger generated for selected constraint

In Table 4.2 the SQL generated for constraint 4 is shown. In line 1, the name of the trigger is set to TRIG4 since the ID of the selected trigger is 4, and the trigger is created on the antecedent table (Patients). The trigger is set to fire after an insert or update statement. The selected constraint is the following variable CFD on the Patients table:

$$[\text{FIRST} = \text{Aileen}, \text{LAST}] \rightarrow [\text{SEX}].$$

Table 4.2 Generated SQL statement

```
1   CREATE TRIGGER TRIG4 ON Patients AFTER INSERT, UPDATE AS
2   BEGIN
3     DECLARE @count INT;
4     DECLARE @rhs VARCHAR(1000);
5     SET @rhs = (SELECT SEX FROM INSERTED);
6     SET @count = (
7       SELECT COUNT(DISTINCT SEX) FROM INSERTED
8       GROUP BY FIRST, LAST
9       HAVING FIRST='Aileen' AND LAST=
10      (SELECT LAST FROM INSERTED));
11    IF @count > 1
12      INSERT INTO Patients_OnDeck
        SELECT *,4,''  FROM INSERTED;
13  END
```

Lines 6-10 check if there are rows in the table that have the first name Aileen, the last name as in the inserted or updated row, and more than one distinct sex. If so, line 12 moves the new row to the Patients_OnDeck table, along with the constraint id and description.

The SQL generated for other types of dependencies in Figure 4.4 are as follows. Note that columns must be ordered as shown in Figure 4.2 i.e., constraint id and description must be the last two columns.

- Functional dependency (1)

$$\text{OHIP: [Class]} \rightarrow \text{[System]}$$

```
CREATE TRIGGER TRIG1 ON OHIP AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @rhs INT;
  SET @rhs = (SELECT COUNT(DISTINCT System) FROM OHIP
  GROUP BY Class
  HAVING Class= (SELECT Class FROM INSERTED));
  IF @rhs > 1
    INSERT INTO OHIP_OnDeck
    SELECT *,1,'' FROM INSERTED
END
```

- Constant CFD (2)

$$\text{OHIP: } [\text{System} = \text{Accident}, \text{Class} = \text{Adverse Effects}] \rightarrow [\text{Color} = \text{Gold}]$$

```
CREATE TRIGGER TRIG2 ON OHIP AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @count INT;
  DECLARE @rhs VARCHAR(1000);
  SET @rhs = (SELECT Color FROM INSERTED);
  SET @count = (
    SELECT COUNT(Color) FROM INSERTED
    GROUP BY System, Class
    HAVING System='Accident' AND
           Class='Adverse Effects');
  IF @count > 0 AND @rhs <> 'Gold'
    INSERT INTO OHIP_OnDeck
    SELECT *,2,'' FROM INSERTED;
END
```

- Inclusion dependency (5)

$$\text{Patients}[\text{STAFF\_DOC}] \subset \text{MDs}[\text{PHYS\_ID}]$$

```
CREATE TRIGGER TRIG5 ON Patients AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @rhs INT;
  SET @rhs = (
    SELECT COUNT(PHYS_ID) FROM MDs
    WHERE PHYS_ID=(SELECT STAFF_DOC FROM INSERTED));
  IF @rhs < 1
    INSERT INTO Patients_OnDeck
    SELECT *,5,'' FROM INSERTED
END
```

- Conditional inclusion dependency (3)

$$\text{Patients}[\text{STAFF\_DOC}, \text{SEX} = \text{F}] \subseteq$$
$$\text{MDs}[\text{PHYS\_ID}, \text{PHYS\_LAST} = \text{EOM}, \text{PHYS\_FIRST} = \text{ROD}]$$

```
CREATE TRIGGER TRIG3 ON Patients AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @lhs INT;
  DECLARE @rhs INT;
  SET @lhs = (SELECT COUNT(*) FROM INSERTED
              WHERE SEX='F');
  IF @lhs > 0
  BEGIN
    SET @rhs = (SELECT COUNT(*) FROM MDs
      WHERE PHYS_ID=(SELECT STAFF_DOC FROM INSERTED)
      AND PHYS_LAST='EOM' AND PHYS_FIRST='ROD');
    IF @rhs < 1
      INSERT INTO Patients_OnDeck
      SELECT *,3,'' FROM INSERTED
  END
END
```

4.3 PAL-IS

The first case study we performed was on the PAL-IS system. We were part of a team of developers that built an EMR application for Bruyère hospital to manage health records for patients receiving palliative care in the community. Figure 4.5 shows a walkthrough of the data processes taking place in and around the PAL-IS application and database. Patient data is entered into the OLTP database by the User through the App (step 6 below). Data from other clinics can also be loaded directly into the OLTP database through batch ETL processes (step 2 below). The App also generates reports that can then be exported to Microsoft Excel (Step 3 below). An Admin may perform schema changes as was required as we released different versions of the PAL-IS (Step 4). Finally, an important aspect of the development and deployment process for PAL-IS was to use Test data (built by the Admin) while developing and testing the application, since the development team was not allowed to see or use real patient data (Step 5).

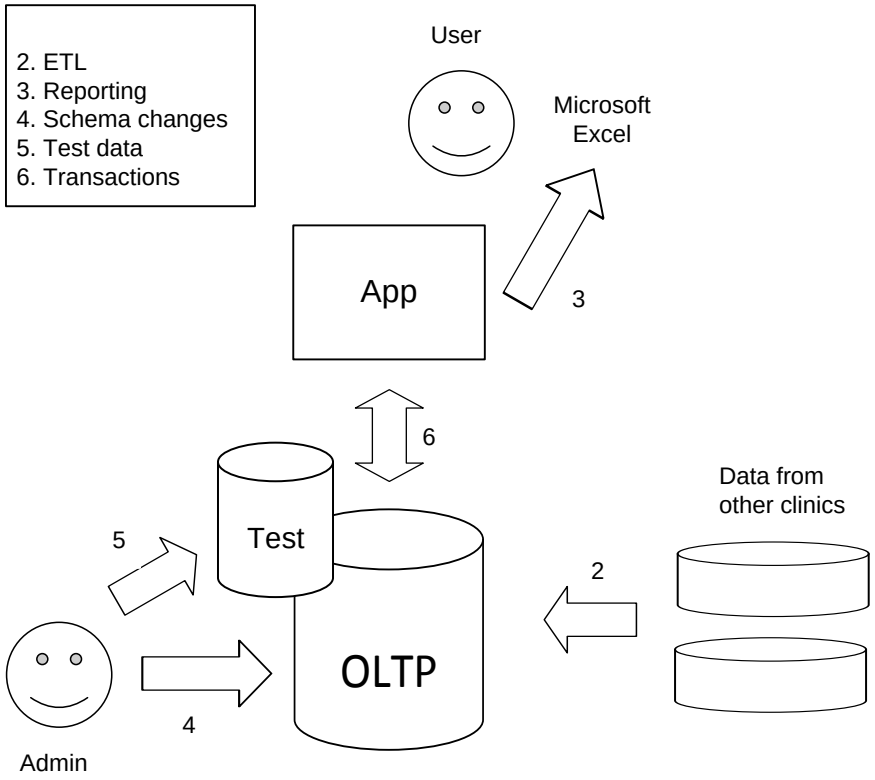


Figure 4.5 Processes relevant to PAL-IS

4.3.1 Data Integrity Problems

To understand the data integrity problems that we needed to address for PAL-IS, it is important to understand the data schema. Figure 4.6 shows a simplified subset of the database schema for PAL-IS. A patient's personal information is stored in the Patients table and their contact information in PatientContacts. Every time a patient is seen, a new form is created and an entry added to the Encounters table. If it is the patient's first visit, it is also added to the InitialConsultationForms table. Other related information is stored in the Allergies, Drugs and Symptoms tables. The ESASThresholds table contains a value (Edmonton Symptom Assessment Score) assigned to different symptoms like nausea and tiredness for each patient. The type of care required for each patient is stored in the CareTeams table.

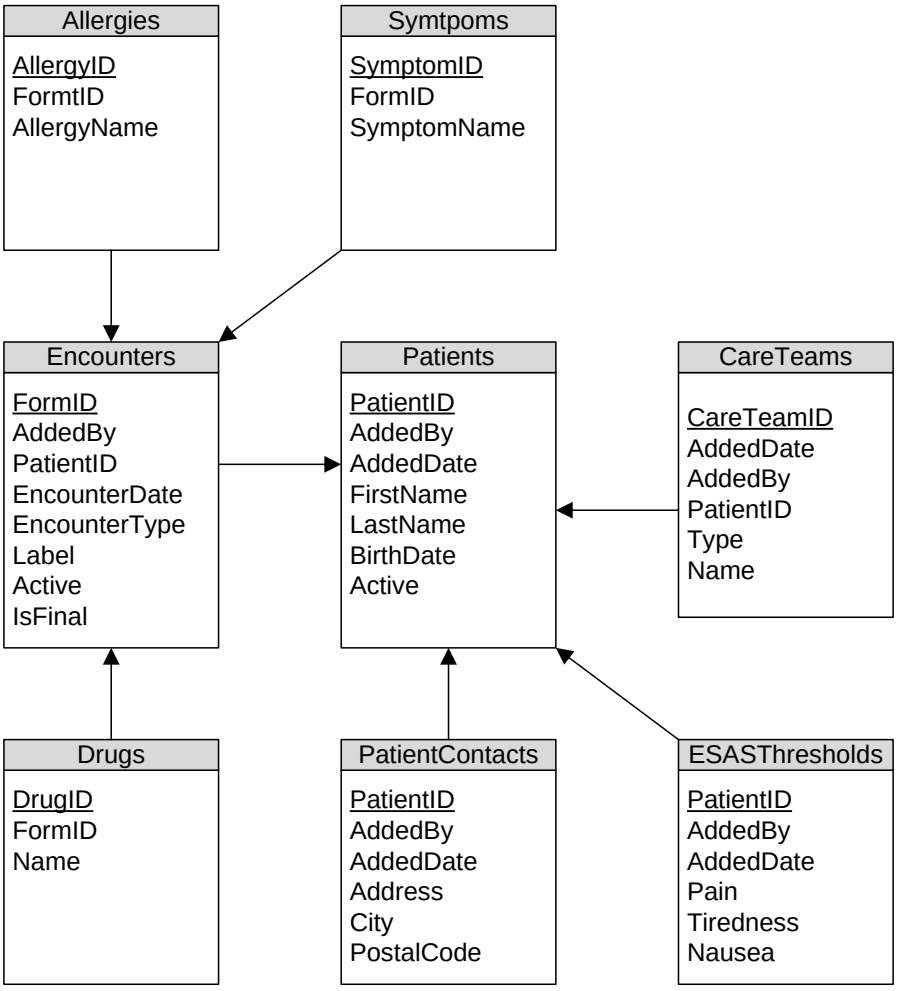


Figure 4.6 Partial database schema for PAL-IS

PAL-IS defines different types of patients forms that need to be filled in (see figure 4.7 for an example). Filling out these forms can be time-consuming and the application prevents the forms from being saved until certain required fields are filled in. This can result in integrity problems if the user enters "fake" values so as to be able to save the forms, but the fake values later remain unnoticed in the database.

These errors can go undetected when there is a large amount of data in the system, and manually detecting them can be tedious. One example of a data integrity problem was that when the application creates a new patient their date of birth defaults to January 1, 1940 until the user changes it (see Figure 4.7). (This was done for usability reasons, at

the request of the hospital, so that when the calendar pops up it displays the year that corresponds to the typical age of a palliative care patient (~70 years old).

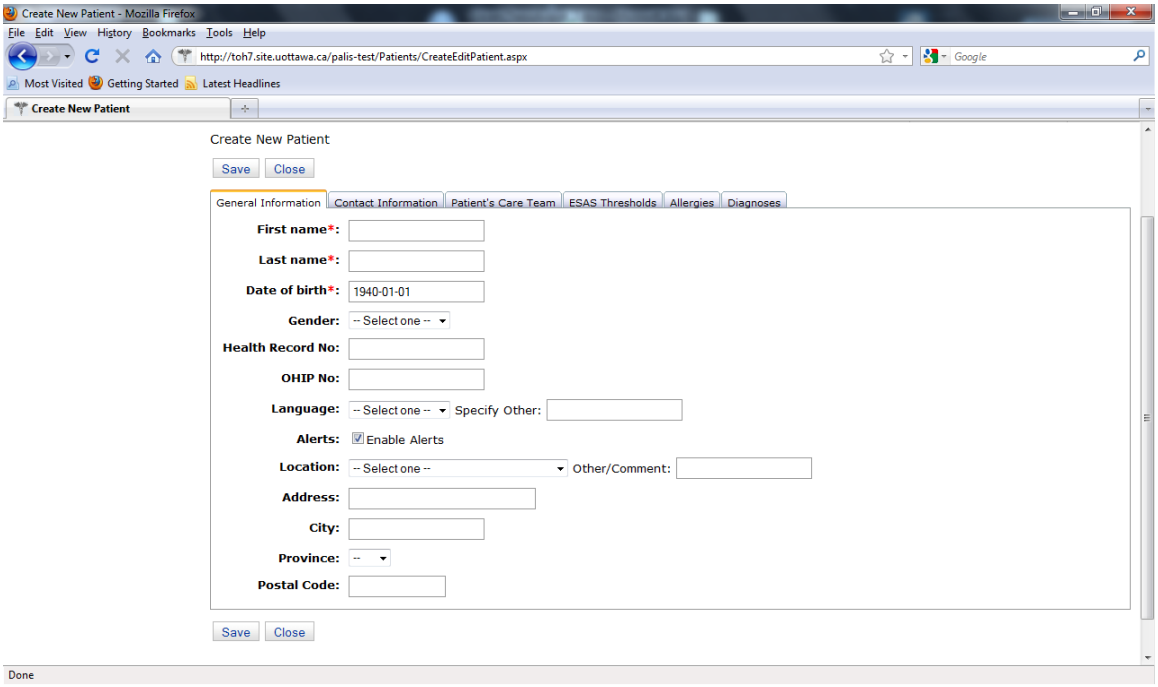


Figure 4.7 Screenshot of PAL-IS while creating new patient

However, the user may forget to enter a date of birth or may not be able to enter one at the time the record is created if the patient's birth date is not known. To manage this issue a report was created that lists all patients with the default date of birth, and creates a "Date of Birth Not Entered" report (we show how this problem is better managed using our conceptual model in section 4.3.4). The SQL for this query is as follows:

```
SELECT FirstName, LastName, AddedBy, AddedDate, PatientID
FROM Patients WHERE Active = 'True'
AND DATEPART (Year, BirthDate)='1940'
AND DATEPART(Month, BirthDate)= 'January'
AND DATEPART(Day, BirthDate)='1'
ORDER BY LastName
```

The reports in PAL-IS are shown in a grid view control from which the data can be copied, pasted and viewed in Microsft Excel. Figure 4.8 shows a screenshot of an example report generated in PAL-IS.

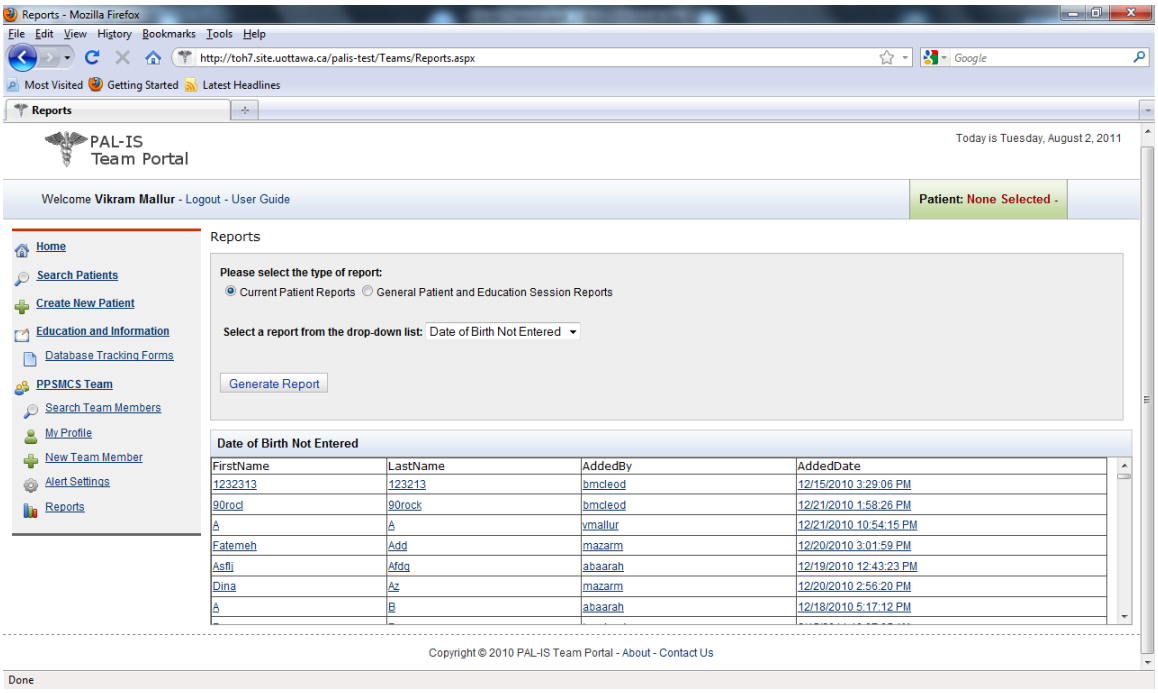


Figure 4.8 Screenshot of PAL-IS showing a report

4.3.2 Current Approaches to Managing Data Integrity

Before we implemented our conceptual model, data integrity was managed by a combination of reports to check for issues after the fact, and by hard-coded checking of constraints in the application code itself. For example, reports are generated that provide a list of patients who have not had their forms finalized or who have had their date of birth entered.

For hard-coded constraints, it helps to understand the architecture. PAL-IS is an ASP.NET application developed using a three-tier architecture. This includes presentation, business and data layers. The presentation layer contains the graphical user

interface (GUI) and the database resides in the data layer. The business layer acts as a mediator between the presentation and data layers and contains methods that map data from one layer to the other. Each layer had hard-coding of constraints.

The only constraints defined in the PAL-IS database were primary keys (no two rows with the same primary key allowed). In the presentation layer of the application, JavsScript is used to write client-side code that can be used for validation without having to reload (post back) the page. This speeds up the application and makes for a better browsing experience. An example of client-side validation is when a user is filling out a database tracking form and enters the date of request to be after the date of delivery. The application will then signal an error and prevent the form from being submitted. In cases like these, the user might again enter a fake value just to get rid of the annoying error messages and data errors might go undetected. However, most constraints were hard-coded in the business layer. For example, while creating a new patient a constraint that is checked is that no two patients have the same first name, last name and date of birth. Understanding such constraints once they are coded can be difficult and time-consuming. Also, such constraints apply only to transactions performed through the application, and not to data that has been brought into the database through ETL.

4.3.3 Other Tools/Approaches

When we started to work on improved methods for managing data integrity we first looked at some third party tools. In particular, we took a typical list of three constraints on PAL-IS data and examined how they would be handled in two data cleaning tools: SSIS and DQguru. The constraints are as follows:

1. The date of birth must be entered.

This is a constraint on the Patients table, and all rows containing the default date of birth need to be flagged.

2. All forms must be finalized.

   This is a constraint on the Encounters table and refers to the electronic signing of forms. All rows where the IsFinal column is set to False need to be flagged.

3. The postal code must uniquely determine the city.

   This constraint is a functional dependency on the PatientContacts table. If a postal code maps to two different cities in the table, then it is a violation of the constraint. (Note that this was an issue after amalgamation in Ontario, grouped the cities Nepean, Gloucester, Manotick and others into a single mega-city Ottawa. It became quite common to have databases where the same postal code sometimes mapped to Nepean and sometimes mapped to Ottawa, for example.)

SSIS allows three tasks to be performed on the data: profiling, cleansing and auditing. Profiling is required to be performed before cleansing. Profiling is performed by assigning indicators (metrics) that are conditions for assessing data, and then performing transformations on the data. One of the transformations is the conditional split. The list of profiling tasks available in SQL Server 2008 is as follows: column length distribution, column null ratio, column pattern, column statistics, column value distribution, functional dependency, candidate key, value inclusion, and parsing the output.

Since the date of birth in PAL-IS is defaulted to January 1, 1940, the condition can be set to check all rows in the Patients table that have this value in the BirthDate column. Similarly, for the second constraint, the condition must check for a “False” value

in the IsFinal columns of the Encounters table. Since SQL Server 2008, functional dependencies can be checked during profiling. Once the data has been profiled, the results can be viewed in the "Profile Viewer." It shows some results like the FD strength which is the percentage of rows that satisfy the dependency. Once profiling is complete, SSIS can be used for resolving any issues encountered. The available are to fix the affected row, discard the row, or stop the process. SSIS provides audit tables that keep track of errors found in the data and their details.

In conclusion, SSIS is a sophisticated tool that can be used to handle various kinds of constraints. But it seems to lack support for conditional and inclusion dependencies. There is also a steep learning curve while getting used to the environment of SSIS and being able to create the data flows. Also, profiling the data is time-consuming and may have to be performed after every ETL operation.

We also looked at DQguru. DQguru, like most other data cleaning tools in the market today, has been designed for performing ETL. However, it is Java-based and includes an open source tool for data cleaning and MDM. And it has the advantage of being able to work with any DBMS including SQL Server. The three main functions of DQguru are deduplication, cleansing and address correction. The cleaning in DQguru is done through transformations. The formatting of a telephone number is an ideal example for this kind if task. For the first and second constraints in our example in this section, deduplication tool in DQguru can be used to match rows that have the same date of birth, or the same value in the IsFinal column. DQguru then shows the rows that match the criteria. The type of errors encountered through dependencies (e.g., a functional dependency between two columns) cannot be handled through DQguru. Despite that,

DQguru is simple to use and is very well suited for integrity issues involving standardization of data.

4.3.4 Our Approach

In this section, we will look at how some of the constraints we have seen for PAL-IS can be handled using our prototype implementation. We use the same constraints that we looked at in Section 4.2.3 and see if using our approach makes it easier.

1. The date of birth must be entered.
2. All forms must be finalized (signed electronically).
3. The postal code must uniquely determine the city.

These constraints are defined on the Patients, Encounters and PatientConacts tables respectively. The first step is to create shadow tables for each of these to hold the on-deck data. The schema for these tables are shown below in Figure 4.9:

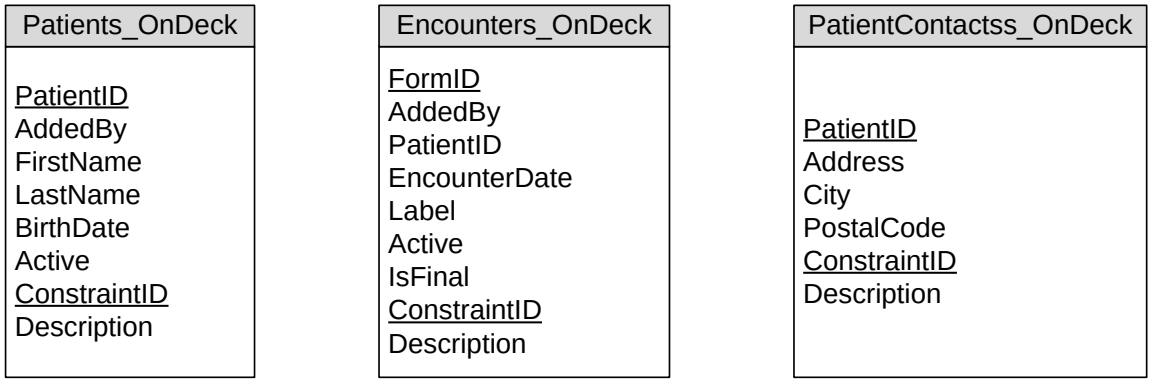


Figure 4.9 Schema for on-deck tables in PAL-IS

The views for the Data Integrity Zone corresponding to the above tables are named Patients_Integrity, Encounters_Integrity and PatientContacts_Integrity. The SQL for creating them are respectively as follows:

```
CREATE VIEW Patients_Integrity AS
SELECT * FROM Patients
WHERE PatientID NOT IN
(SELECT PatientID FROM Patients_OnDeck)

CREATE VIEW Encounters_Integrity AS
SELECT * FROM Encounters
WHERE PatientID NOT IN
(SELECT PatientID FROM Encounters_OnDeck)

CREATE VIEW PatientCintacts_Integrity AS
SELECT * FROM PatientContacts
WHERE PatientID NOT IN
(SELECT PatientID FROM PatientContacts_OnDeck)
```

We now show the SQL for generating triggers that use the zones we have just created. In the first constraint, we only need to flag rows in the Patients table that have the default date of birth entered.

```
CREATE TRIGGER TRIG1 ON Patients AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @dob DATETIME;
  SET @dob = (SELECT BirthDate FROM INSERTED);
  IF @dob = '1/1/1940'
  BEGIN
      INSERT INTO Patients_OnDeck
      SELECT *,1,'Date of birth not entered' FROM INSERTED
  END
END
```

In the second constraint, we want to flag rows in the Encounters table that have the IsFinal column set to false.

```
CREATE TRIGGER TRIG2 ON Encounters AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @isFinal VARCHAR(10);
  SET @isFinal = (SELECT IsFInal FROM INSERTED);
  IF @isFinal = 'False'
  BEGIN
      INSERT INTO Patients_OnDeck
      SELECT *,2,'Form not finalized' FROM INSERTED
  END
END
```

The third constraint is the CFD [PostalCode] → [City] in the PatientContacts table. The trigger for this can be generated using our tool.

```
CREATE TRIGGER TRIG3 ON PatientContacts
AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @rhs INT;
  SET @rhs =
    (SELECT COUNT(DISTINCT City) FROM PatientContacts
    GROUP BY PostalCode HAVING PostalCode =
    (SELECT PostalCode FROM INSERTED));
  IF @rhs > 1
    INSERT INTO PatientContacts_OnDeck
    SELECT *,3,'Multiple cities for same postal code'
    FROM INSERTED
END
```

Our prototype implementation works well in handling these constraints since any rows that violate them are immediately flagged for cleaning. And since the On Deck Zone tables are created in the same database, no changes are needed in the application to handle these violations. This allows the constraints to be seamlessly integrated into the application without the need to run an external tool like SSIS for checking consistency. It however requires the creation of Data Integrity Zone views and On Deck Zone tables for each table.

4.3.5 Scalability, Performance and Usability

The implementation we have discussed here for our conceptual model requires the creation of an additional table and view for each table that has a constraint defined on it. The data that have been flagged are repeated in the On Deck Zone rather than being moved.

We tested our approach using two experiments. The first one measures the time needed to execute a query as the number of rows increases. It measures the time taken to retrieve data from the entire table, the On Deck Zone and the Data Integrity Zones. The purpose of this experiment is to determine whether there is significant overhead in accessing the Data Integrity Zone through a view.

The second experiment measures the time needed to complete a process as the number of constraints increases. Each constraint has an associated trigger that flags constraint violations. For this experiment, we only used data that would be in the Data Integrity Zone. We simulated the creation of a new patient and encounter respectively by running an SQL script containing some insert and update statements that would be typically executed by the PAL-IS application. The data created and updated by these scripts was all in the Data Integrity Zone.

As far as usability is concerned, the implementation of CBIM shown in this thesis requires the creation of an On Deck table and a Data Integrity view for each table in the database on which a constraint has been defined. In other words, using CBIM doubles the number of tables and creates as many views as the original number of tables. The number of triggers required will be the same as the number of constraints since each constraint will have its own trigger. Our tool can be used to specify constraints and, if they are

dependencies, create triggers in the database. So depending on the constraint, some work may be needed here to manually create triggers.

Another issue is how to handle complex queries across the database that involve two or more tables. While querying data from a single table, accessing the clean or dirty rows is straightforward. However, when the query involves a join on two or tables, one or more of the tables containing dirty rows can cause rows in the query result to be dirty. In general, a dirty view of the database is less interesting than a dirty view of a single table because violations are fixed on a row-by-row and table-by-table basis. To ensure that all rows in a query result are clean, all tables referenced in the query must clean. For example, in a query involving the Patients and Encounter tables, querying using the PatientsIntegrity and EncountersIntegrity is the only way to guarantee that the resulting rows will all be clean.

4.4 iMED-Learn

The second case study we performed was using the iMED-Learn system. We were part of a team of developers that built the system for a family physician at the Ottawa hospital to help doctors and residents track the breadth of their experience (how many different diseases had they had experience with) in comparison to others in order to help them manage their training.

4.4.1 Data Integrity Problem

Figure 4.10 shows a walkthrough of the relevant data processes for iMED-Learn. Data from various family Clinics are loaded into a Staging database via batch ETL processes (step 2 below). Data integrity issues can arise here if, for example, there is conflict between the IDs used by different physicians or even patients. The Admin

performs any data cleaning that may be required. The staging data is then transformed and loaded into an OLAP database that can used by a tool like SQL Server Analysis Services to create a data cube. The data cube is created by selecting data from the OLAP database that correspond to fact and dimension tables, and can be used to generate reports in Microsoft Excel. The reporting application is an ASP.NET program that accesses the OLAP data and generates its own reports. The user has access to both the application and the cube loaded in Excel, and can select the data and type of chart to show in the reports.

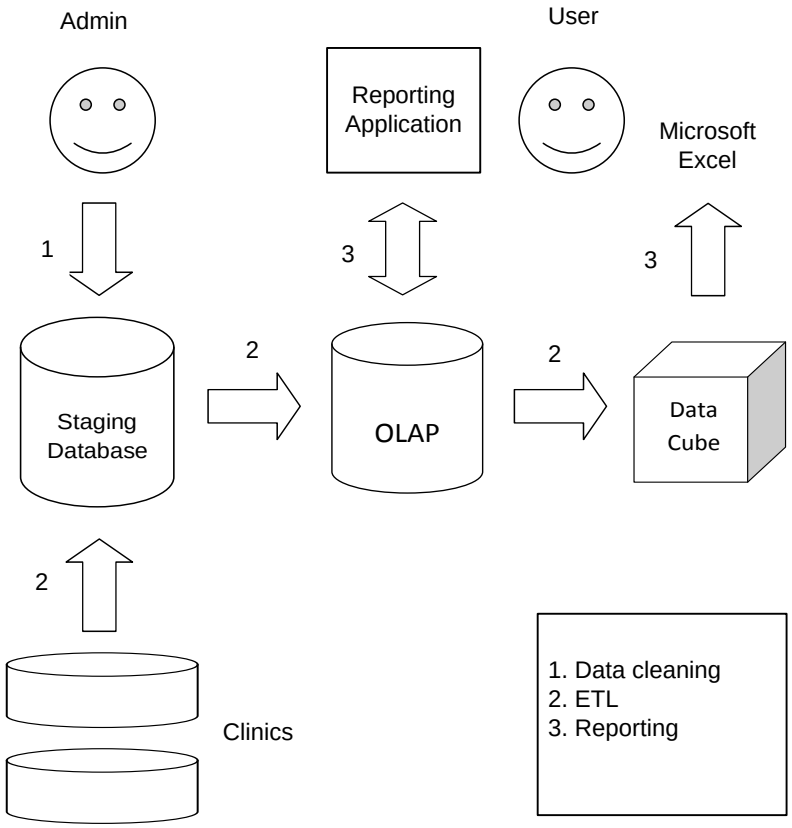


Figure 4.10 Processes relevant to iMED-Learn

The iMED-Learn OLAP Database contains data about diagnoses seen by family patients during patient visits. Figure 4.11 shows the database schema containing six tables. The Patients table contains information about patients including their first and last

names, date of birth and gender. MOH and Version refer to the number assigned to an individual by the Ministry of Health. MRN is the medical record number kept by the hospital for each patient. Staff_doc refers to the primary physician who sees the patient. The MDs table contains a list of people who see the patient, including their first and last names. Staff type indicates the type of physician and can be a dietician, MD, NP (nurse practitioner), resident, RN (registered nurse) or SW.

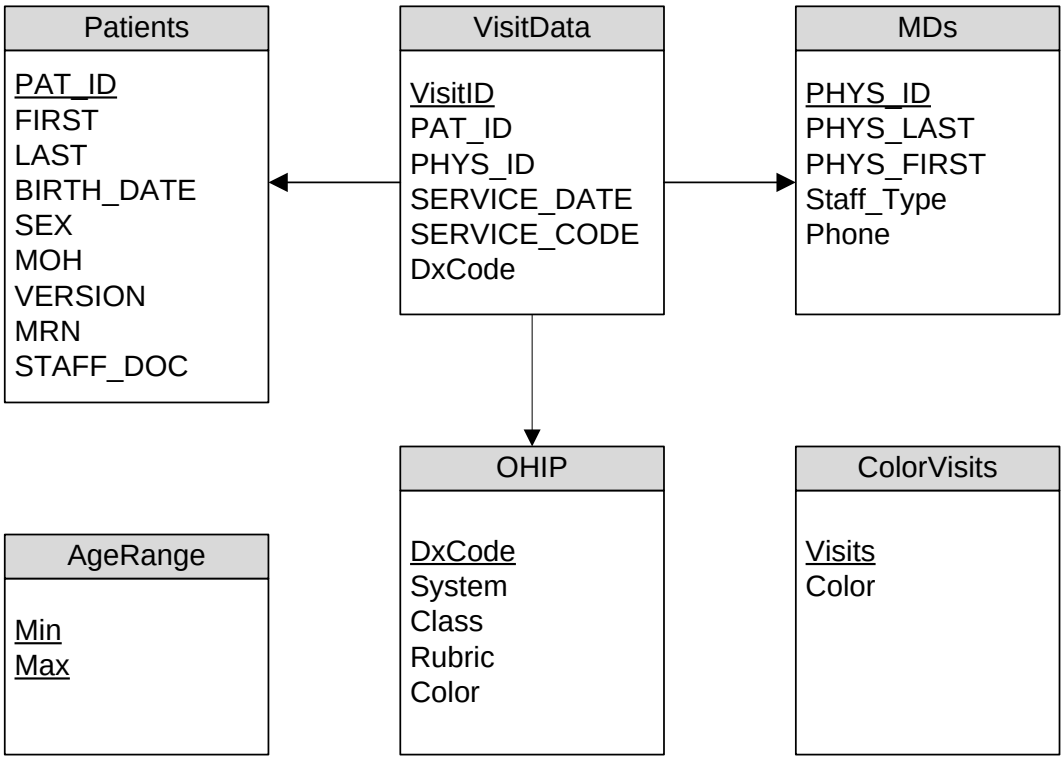


Figure 4.11 iMED-Learn staging database schema

The OHIP table contains a list of all the diagnoses arranged into a hierarchy of rubrics within a class, and classes within a system. The color column indicates the color to use for indicating the rubric while using charts for reporting. The fourth table VisitData keeps a record of all patient visits. It contains for each visit the patient id, the id of the physician who saw that patient, the date of visit, a code that denotes the type of

service rendered, and a reference to the diagnosis made. AgeRange and ColorVisits are tables used by the reporting application (see Figure 4.15). The former lists the age groups that need to be shown in one of the reports. ColorVisits specifies the color that indicates the number of visits (see the legend of Figure 4.15).

4.4.2 Current Approaches

Since iMED-Learn is used mainly for reporting, its schema is transformed into a star database containing fact and dimension tables as shown in Figure 4.13. The table FactEncounters is the fact tables and it contains foreign keys referencing each of the other dimension tables. The table "Date" is the time dimension that is ubiquitous in schemas used for reporting. It contains details about every date value in the calendar for the required duration.

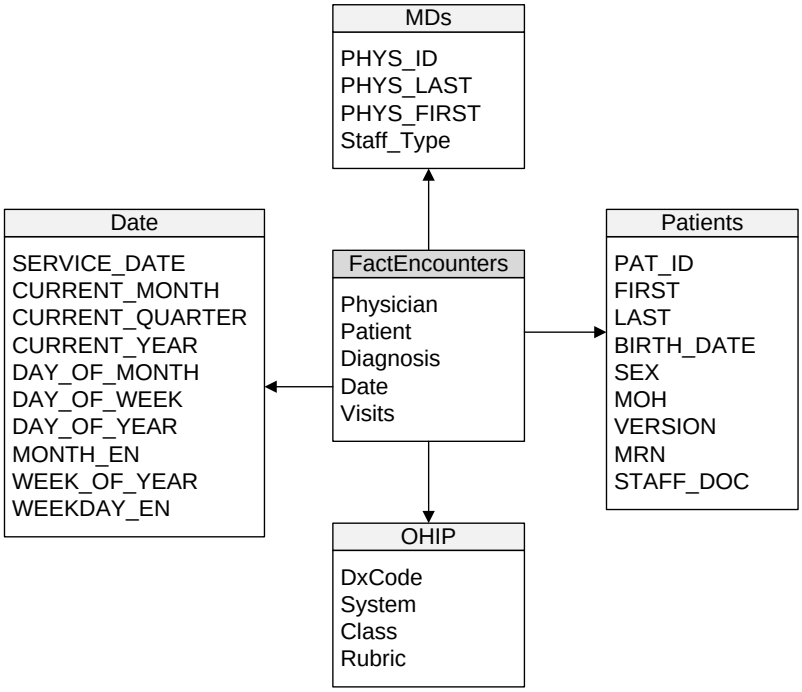


Figure 4.12 iMED-Learn OLAP/cube schema

The star schema shown in Figure 4.12 can be used in a reporting tool like Business Intelligence Development Studio (BIDS) to create a cube using Analysis Services. A cube is a data structure that can be "browsed" along different dimensions to perform operations like drill-down, roll-up and filtering. It has the same structure as shown in Figure 4.12. Figure 4.13 shows a part of the cube that has been browsed. It shows the number of visits on different days of the week for each system diagnosed. For example, it can be seen from the cube that the number of patients with accidents seen on Wednesdays is 39.

DAY OF WEEK	System	Visits
⊞ Sunday		1433
⊞ Monday		1215
⊞ Tuesday		579
⊟ Wednesday	Accident	39
	Cardiovascular	43
	Digestive	47
	Diseases of Blood And Blood-Forming Organs	13
	Endocrine, Nutritional and Metabolic Diseases and ImmunityDisorders	134
	Genito-urinary	18
	Infections and Parasitic Diseases	8
	Mental Disorders	20
	MSK	27
	Neoplasms	7
	NeuroSensory	1
	Non-specific	10
	Perinatal	1
	Pregnancy	9
	Prevention-Social	83
	Respiratory	387
	Skin	19
	Total	866
⊞ Thursday		92
⊞ Friday		115
⊞ Saturday		1744
Grand Total		6044

Figure 4.13 Browsing the cube in BIDS

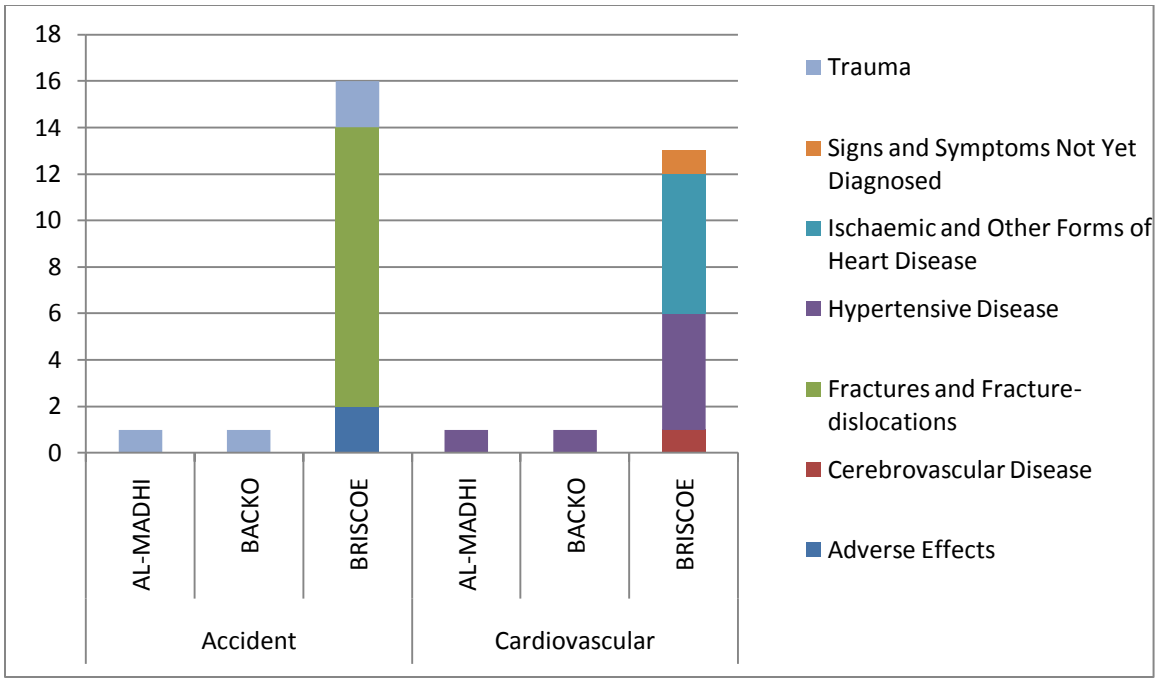


Figure 4.14 Excel chart showing classes

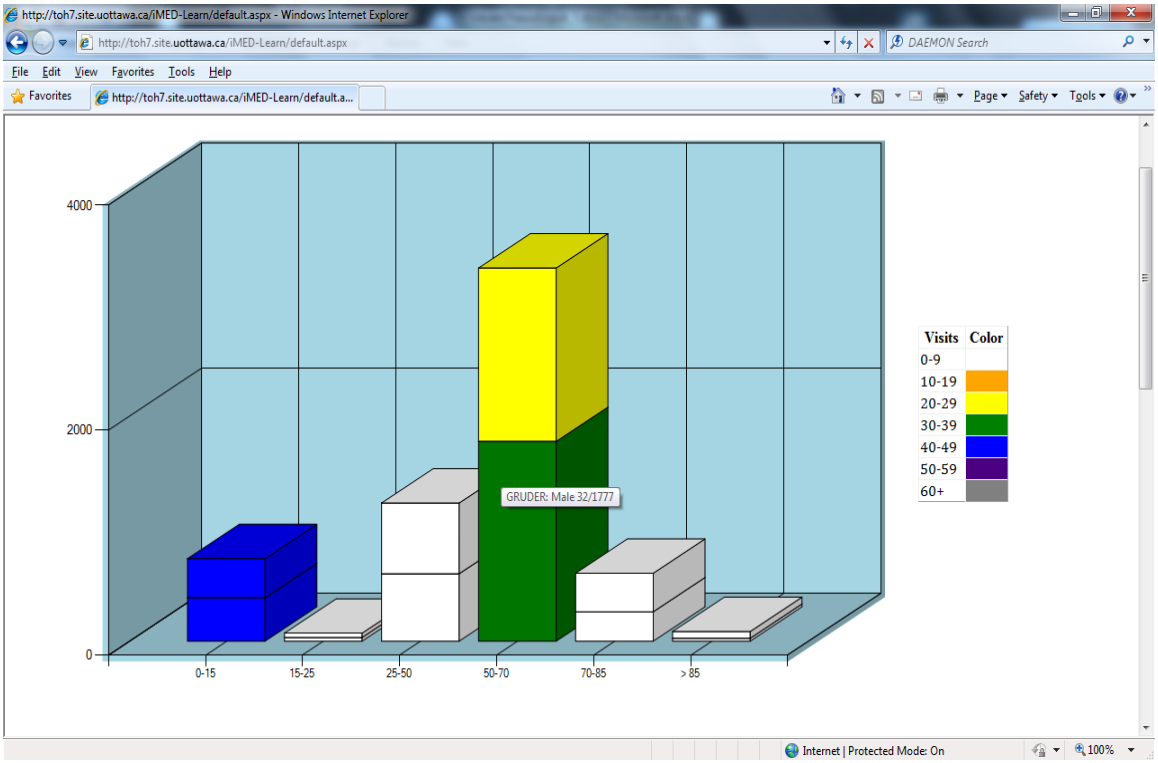


Figure 4.15 ASP.NET chart

The cube generated by Analysis Services can also be exported to Microsoft Excel for viewing charts there. Figure 4.15 is a stacked column chart showing the classes of diagnoses seen by physicians Drs. Al-Madhi, Backo and Briscoe. The chart has filtered to consider only two systems Accident and Cardiovascular.

The main iMED-Learn application has been created in ASP.NET. Figure 4.15 is a screenshot showing one of the charts generated by the application. It shows the distribution of visits across different age groups for both male and female patients as seen by Dr. Gruder. The tooltip in the figure reads: "GRUDER: Male 32/1777" indicating that, out of 1,777 male patients in the 50-70 age group Gruder has seen 32.

Data integrity issues during reporting are currently handled in the application code. One of the reports generated involves showing the top "n" rubrics that are diagnosed during patient visits in a column chart. There are 537 rubrics listed in the OHIP table which makes it difficult to assign that can be easily distinguished from one another in a chart. This results in the user having to constantly reassign colors so that the rubrics that appear in the charts have visually distinct colors. During this process, it becomes difficult to keep track of what colors have been used up, and what are left.

4.4.3 Other Tools/Approaches

In this section, we will consider some integrity problems in iMED-Learn and see how they can be addressed. The first problem arises while loading the data from the clinics into the staging database. It is not unusual for such data to be in no particular format, and an example of a constraint is that the telephone number must be in the format XXX-XXX-XXXX. If the telephone number in the source is 6131234567, it will have to be changed to 613-123-4567. Such a task is usually easy to accomplish in data cleaning

tools since they are designed specifically for ETL. In DQguru, it can be done by using functions like the “Substring” functions to be exact to split the first three characters, then the last four characters, and then putting them back together with the “-“ sign in-between by using “Concat”. In SSIS, the same task can be performed by specifying the logic during profiling. SSIS has rich collection of operators and functions that can be used to construct the logic to be followed while doing the ETL. Telephone numbers can be formatted by using the SUBSTRING() function and the + operator to concatenate the parts together.

Next, consider the constraint that each range of visits in Figure 4.15 must be assigned a unique color. The FD in this case is ColorVisits: Color → Visits. DQguru has no support for functional dependencies. As we saw for the PAL-IS system in Section 3.4.3, this type of dependency is an FD and can be handled in SSIS during profiling by specifying the TableOrView property to ColorVisits, the DeterminantColumns property to Color, and the DependentColumn property to Visits. However, DQguru has no support for dependencies, and only allows deduplication and standardization of the data. In the next section we will see how our approach addresses this issue.

4.4.4 Our Approach

In this section, we show how our approach handles the constraints discussed in Section 4.4.3. The first one is about formatting the telephone numbers in the source data. In or tabular format this constraint can be written as shown in Table 4.3.

Table 4.3 Representing constraint in tabular form

ID	Constraint Type	Antecedent Table	Antecedent Column	…	Description
8	Other	MDs	Phone		Must be in the xxx-xxx-xxxx format.

The trigger for handling this constraint would need to be manually created. The SQL is shown below. We assume that the MDsOnDeck table and MDsIntegrity view have been created. The trigger flags any row that is inserted or updated in the MDs table and copies it to the MDsOnDeck table. It checks the phone number has the right length and has the "-" separator at the right places. A more sophisticated check can be done to ensure that there are no alphabetic characters in the number.

```
CREATE TRIGGER TRIG8 ON MDs AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @phone VARCHAR(25);
  SET @phone = (SELECT Phone FROM INSERTED);
  IF (LEN(@phone) <> 12) OR
     (SUBSTRING(@phone,4,1)<>'-') OR
     (SUBSTRING(@phone,8,1)<>'-')
  BEGIN
      INSERT INTO MDs_OnDeck
      SELECT *,8,'Must be in the xxx-xxx-xxxx format'
      FROM INSERTED
  END
END
```

ETL tools are able to automatically clean telephone numbers if it is simply a matter of translating 1234567890 into 123-456-7890, however if there is the potential for errors in the number (e.g. not enough digits) then CBIM is necessary so that rows can be flagged

for fixing. Note however, that since this is not a dependency constraint, CBIM is unable to automatically generate a trigger. It must be written by hand.

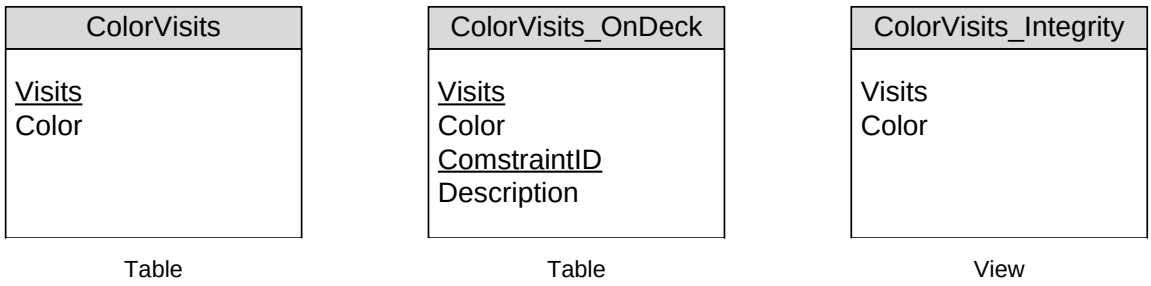


Figure 4.16 On-deck and Data Integrity Zones for ColorVisits

The constraint specified in Section 4.4.3 can be written as the functional dependency ColorVisits: Visits → Color. The first step is to create a shadow table ColorVisits_OnDeck and the view ColorVisits_Integrity as shown in Figure 4.16. The trigger can be generated using our tool and is as follows:

```
CREATE TRIGGER TRIG3 ON ColorVisits
AFTER INSERT, UPDATE AS
BEGIN
  DECLARE @rhs INT;
  SET @rhs =
    (SELECT COUNT(DISTINCT Color) FROM ColorVisits
    GROUP BY Visits HAVING Visits =
    (SELECT Visits FROM INSERTED));
  IF @rhs > 1
    INSERT INTO ColorVisits_OnDeck
    SELECT *,3,'Colors must be unique'
    FROM INSERTED
END
```

The application can be programmed to use the ColorVisits_Integrity table instead of the ColorVisits table. This will ensure that the displayed chart will have unique colors for each age range shown.

4.4.5 Scalability, Performance and Usability

Unlike PAL-IS, the iMED-Learn database has few transactions taking place, since the latter is used mainly as a reporting tool. The only transactions are those that update configuration properties for the application. For example, in the chart shown in Figure 4.5, the user might want to change the configuration so that the columns are displayed for ages 0-10 and 10-25. This can be done by changing the Min and Max values in the AgeRange table. The queries in PAL-IS mainly consist of SELECT statements to retrieve the data being reported on, or just involve updates for controlling how the charts are displayed. Also, the volume of data in iMED-Learn is low when compared to PAL-IS and the structure of the database is similar. So, we decided that it was sufficient to run our scalability experiments on PAL-IS only so that we could focus on testing against different sizes of database, and different numbers of constraints (implemented as triggers).

The key advantage of our approach is to allow the user to interactively fix dirty data rows in the On Deck Zone rather than reject them outright. This is particularly useful in iMED-Learn while loading from clinics into the staging database, as there is no information loss and the dirty data is kept until it can be fixed. It also offers a better way to express constraints than cleaning tools. However, the processes of generating the OLAP database and the data cube require little, if any, management of constraints and existing tools like SSIS are adequate for the job.. During some processes, our tool is thus a complement rather than a replacement for existing tools.

One area where our on-deck mechanism is better than a cleaning tool is during reporting. When users wish to customize reports, constraints can be added to ensure that

the report is being displayed in the correct manner. A rule of thumb to consider would be that our conceptual model of data integrity adds the most value when the data processes involved cannot be automated, for example when handling transactions, working with test data and complex data cleaning.

Chapter 5. Evaluation

In this chapter, we evaluate our CBIM approach for managing data integrity in four different ways. First, we evaluate CBIM using the criteria defined in chapter 3.2, by comparing it with traditional methods that are based on either manual tools for cleaning data to restore data integrity or automated tools for enforcing integrity constraints to prevent data integrity violations from occurring. Next, we conduct experiments using our prototype implementation to test the feasibility of our approach in terms of scalability and performance. Finally we compare our prototype implementation to particular tools including an evaluation of the support for defining constraints and generating triggers.

5.1 Criteria Assessment

Table 5.1 summarizes the benefits of our proposed CBIM approach for managing data integrity within an organization. The most important criterion for an organization is to be able to take a complete approach to data integrity. Integrity constraints on their own can be used to detect inconsistencies, but manual methods are needed to properly resolve inconsistencies. Unfortunately, manual approaches will often miss inconsistencies and not address them at the point of data entry. CBIM provides a logical framework that leverages integrity constraints to flag and manage inconsistencies by clearly delineating a Data Integrity Zone and an On Deck Zone where inconsistencies need to be resolved.

The effort required by CBIM is roughly the same as doing both manual cleaning and using integrity constraints. However, there is potential savings in effort, since constraint violations are systematically managed and can hopefully be addressed closer to

data entry. This certainly results in greater scalability as organizations can effectively track and catalogue the integrity of data for all constraints they are able to define.

Table 5.1 Comparisons of mechanisms for data integrity

Criteria	**Manual Tools**	**Integrity Constraints**	**CBIM**
Completeness	May miss errors that need fixing.	Catch errors without fixing.	Catch and fix errors.
Effort	Manually find errors. Track in an ad hoc manner. Fix manually.	Implement triggers to automatically find errors. Mostly errors are rejected (not fixed).	Implement triggers to automatically find errors. Flag and mange errors systematically with views. Fix manually.
Scalability	Cumbersome as constraints, data and violations increase.	Detection is automatic, but violations are unmanaged.	Detection is automatic. Violations are tracked.
Skills required	Combination of DBA and domain expert in one person.	DBA to implement constraints. Ad hoc resolution of violations.	DBA to implement constraints. Domain expert resolution using views.

This also provides for a better separation and use of skills within the organization. With manual tools, one effectively needs to combine the skills of both a DBA and a domain expert in one person, which is not easy to do. You need DBA knowledge in order to understand the database scheme and use database tools to find problems, but you need to also be a DBA at the same time, in order to know how to resolve those problems. With CBIM, the DBA can focus on defining integrity constraints that can be automatically applied to flag violations, while the domain expert can focus on resolving issues interacting with the On Deck view of the data which includes documentation on the nature of the constraint violation.

Table 5.2 Evaluation of CIBM

Criteria	Manual Tools	Integrity Constraints	CBIM
Completeness			
• Specify constraints	Ad hoc	Logic	Tabular notation
• Violations flagged	Ad hoc	Reject only	Yes
• Fix violations	Semi-automated	No	No
• Ensure accuracy	Yes	No	No
• Minimize information loss	Ad hoc	No	Yes
Effort			
• Specify constraints	Ad hoc	Logic	Tabular notation
• Deploy constraints	Batch routines	Logic engine	Triggers
• Flag violations	Manual queries	Reject only	Automatic by triggers.
• Fix violations	Semi-automated	Manual	Manual
• Ensure accuracy	Manual	Manual	Manual
• Minimize information loss	Manual	Manual	Automatic
View Convenience			
• Only clean data	Ad hoc	Yes	Yes
• Only dirty data	No	No	Yes
• Why data is dirty	Ad hoc	Ad hoc	Yes
• All data	Ad hoc (info loss)	No	Yes
Skills required			
• Specify constraints	Tool + SQL	Logic + SQL	Table + SQL
• Flag violations	Tool + SQL	N/A	N/A
• Fix violations	Tool + SQL	N/A	N/A

Table 5.2 shows a more detailed evaluation of CBIM by breaking down the following criteria in more detail: completeness, effort, convenience and skills required. Scalability will be evaluated separately in Section 5.2.

Manual tools usually manage completeness on an ad hoc basis. The fixing of violations is semi-automatic since the user can normally select the fix to be applied on the data. Since the user has control over the applied fixes, semantic accuracy can be ensured. Using integrity constraints, however, requires the constraints to be specified logically. They usually reject dirty data outright, and consequently do not fix them. Sometimes, however, criteria such as distance functions are used to fix violations. Integrity constraints cannot ensure semantic accuracy and do not minimize information loss. CBIM, on the other hand, combines the approaches of both manual tools and integrity constraints. It uses a tabular notation to specify constraints that is equivalent to defining them logically, and simpler to understand. It also flags violations through the on-deck mechanism. However, it provides no support for fixing constraint violations and consequently does not ensure accuracy. However, since no data is rejected, the loss of information is minimized in CBIM. Thus, CBIM combines some of the advantages of manual tools with those of integrity constraints. The fixing of constraints can be added as a feature to CBIM by providing user interaction. Overall, CBIM seems to be a better approach than using manual tools or integrity constraints separately.

The effort required to specify constraints using manual tools is ad hoc in nature and depends on how complex the constraint is, and also how sophisticated the tool is. The constraints are usually deployed using batch routines and the flagging of violations requires manual queries to be written. The tools provide support for fixing violations, and

the approach is semi-automated since the user has control over what fixes are applied. In summary the ensuring of accuracy and the minimization of information loss are manual in nature while using these tools. Integrity constraints, on the other hand, are required to be specified using logic and then interpreted by a logic engine. Constraint violations are usually rejected, and these will need to be manually fixed. Just like in manual tools, integrity constraints fail to ensure semantic accuracy and do not prevent information loss. By contrast, CBIM requires minimum effort to specify constraints as it uses a tabular notation. It allows the constraints to be deployed using triggers. As a result, flagging of violations is automatic and information loss is automatically minimized since dirty data is not rejected. However, CBIM has no support for fixing violations and so accuracy of data will have to be ensured manually. So, in terms of effort, using manual tools can sometimes have an advantage, depending on the nature of the process and constraints involved.

With manual tools, it is usually possible to have a view of just the clean data in the system. However, they usually do not provide a view of just the dirty data. Integrity constraints make it possible to view only the clean data, but since the dirty data is rejected it is not possible to see both the clean and the dirty data. CBIM, through its on-deck mechanism, conveniently provides a means to view only the clean data, only the dirty data, or all the data. CBIM is clearly superior in this regard.

Manual tools, depending on their sophistication, require some skills in using the them and a knowledge of SQL to specify constraints, flag violations, and fix those violations. Some constraints will need to be specified using expressions and built-in functions. Integrity constraints require an understanding of the logic behind the

constraints and also some SQL to specify them in a database. CBIM, on the other hand, requires an understanding of the tabular notation for constraints in addition to SQL.

5.2 Scalability and Performance

In this section we report on experiments that were run with our prototype implementation of CBIM to measure the potential impact of our conceptual model of data integrity on performance and scalability.

We conducted experiments using the PAL-IS test databases (described in Section 4.3) in order to evaluate our approach in terms of the scalability and performance. As described in Section 4, we had a simple test database (with 226 patients and 4,314 encounters) and we had a large database (with 1,000 patients and 100,000 encounters). The first experiment tests the overhead incurred for query operations using our implementation of On Deck and Data Integrity Zones, while the second and third tests evaluate the overhead incurred by processing triggers for transactions. We did this by simulating respectively the tasks involved in creating a new patient and filling in an initial assessment form using PAL-IS.

View Support (On-Deck, Data Integrity) when Querying

We tested our approach by measuring the time needed to run the Forms Not Finalized report in PAL-IS. This simply queries the list of the forms that have not been finalized (formally signed by a nurse or physician). This is a very common report run every day by the team manager to ensure that all forms related to patient care are completed and finalized in a timely fashion. It involves joining the Patients table with the Encounters table (which records finalized or not finalized regardless of Form Type). We

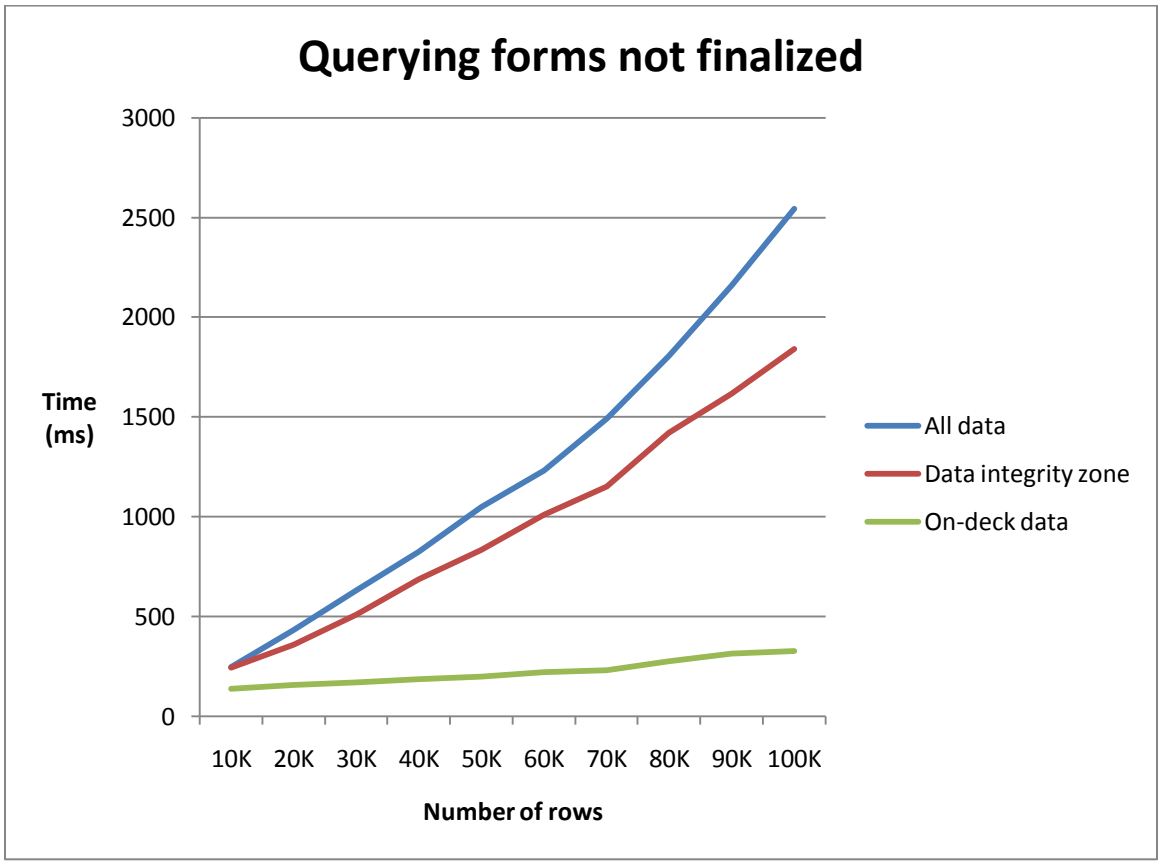
ran three queries based on using either Data Integrity Zone patients (PatientsClean view), On Deck Zone patients (PatientsDirty table) or all-data patients (Patients table). In all three cases, we simply joined against all-data in the Encounters table since there were no constraint violations for the Encounters table (it's a fairly simple table tracking meta-data associated with the specific forms tables). Here are the three queries for retrieving the list of forms not signed from the entire database (Patients), the Data Integrity Zone (PatientsClean) and the On Deck Zone (PatientsDirty) respectively:

```
SELECT FirstName, LastName, Label as [Form],
Encounters.AddedBy as [Added By], EncounterDate,
Patients.PatientID
FROM Patients, Encounters
WHERE Encounters.PatientID=Patients.PatientID and
Encounters.Active='True' and Encounters.IsFinal = 'False'
ORDER BY LastName, FirstName, Form

SELECT FirstName, LastName, Label as [Form],
Encounters.AddedBy as [Added By], EncounterDate,
PatientsClean.PatientID
FROM PatientsClean, Encounters
WHERE Encounters.PatientID=PatientsClean.PatientID and
Encounters.Active='True' and Encounters.IsFinal = 'False'
ORDER BY LastName, FirstName, Form

SELECT FirstName, LastName, Label as [Form],
Encounters.AddedBy as [Added By], EncounterDate,
PatientsDirty.PatientID
FROM PatientsDirty, Encounters
WHERE Encounters.PatientID=PatientsDirty.PatientID and
Encounters.Active='True' and Encounters.IsFinal = 'False'
ORDER BY LastName, FirstName, Form
```

Figure 5.1 shows the running time for each view as the number of rows in the Encounters table was varied from 10,000 to 100,000 (this was done by simply deleting rows from our large database). We started with 100,000 rows and ran the three queries to measure the

execution time. We ran each of the queries ten times and measured the average running time to report on all data, only data in the Data Integrity Zone, and only data in the On Deck Zone. We fixed the percentage of rows in the On Deck Zone to be 10% of the total number of rows in the table. As a result, the remaining 90% of the data were in the Data Integrity Zone.

Next we removed 10,000 rows from the Encounter table and 1,000 rows from the EncountersTable table, and measures the query time. We continued this process till there were 10,000 rows remaining in Encounter and 1,000 rows in EncounterDirty. The database was static while executing these queries.



Number of rows	Time (in milliseconds)		
	All data	Data Integrity Zone	On Deck Zone
10K	245	242	138
20K	434	359	156
30K	632	510	170
40K	824	686	187
50K	1048	835	199
60K	1231	1009	222
70K	1493	1151	232
80K	1805	1420	276
90K	2159	1618	315
100K	2543	1841	328

Figure 5.1 Forms Not Finalized report

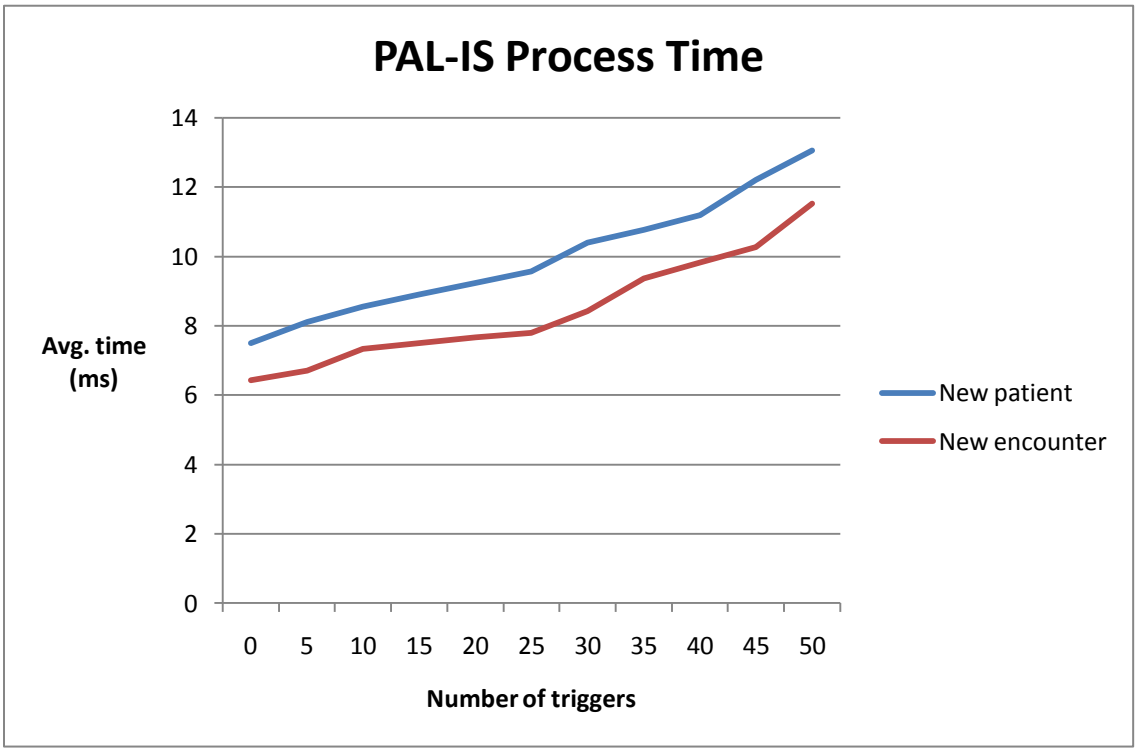
The graph shows that there is no significant impact to the running time in retrieving data from our flagging or view mechanisms. The running time for each of the three zones (on deck, data integrity, all data) varies only based on the number of rows returned. Note that the "All data" query is the exact same query that would be run if our data integrity model was not used; i.e. if there was no flagging mechanism and there is no view. The view of clean data runs quicker than the view of all data roughly in proportion to the fact that there is roughly 10% fewer rows returned. The on-deck query returns only 10% of the row. One could argue that the CBIM approach is more scalable because it effectively separates our clean data from dirty data allowing one to only view the data one is interested in.

Constraint Checking Overhead

To test the overhead when updating or inserting into the database with our conceptual model for data integrity, we ran two transactions related to creating a new patient and filling out an encounter form. Note that Appendix A shows the script for the transaction that simulates the creation of a new patient in PAL-IS. It checks whether there

is already a patient with the same first and last names, and date of birth. If not, a new entry is made in the Patients and PatientContacts tables. Information about the patient's ESAS threshold, care team, and diagnosis are also entered during the process.

The script for the transaction that simulates a new encounter is shown in Appendix B. A row is added to the Encounters table and the InitialConsultationForms table. The patient's allergies, symptoms and prescribed drugs are also added.



Number of triggers	New patient	New encounter
0	7.50	6.43
5	8.10	6.70
10	8.56	7.33
15	8.90	7.50
20	9.23	7.66
25	9.56	7.80
30	10.40	8.43
35	10.78	9.36
40	11.20	9.83
45	12.20	10.26
50	13.06	11.53

Figure 5.2 New patient/encounter

Figure 5.2 shows the average running time to create a new patient or encounter as the number of triggers increases. We started with no triggers and measured the time taken to create a new patient and encounter. As in the previous experiment, we ran the process ten times and took the average time. Next we added five triggers and measures the process time. This was continued until the number of triggers reached 50. In this experiment, all the data was in the Data Integrity Zone. That is all triggers are executed every time a trigger is run, but not violations were found. That is we are only measuring the overhead of testing the constraint. We are not measuring the overhead of creating a flagged entry in the On Deck Zone (the rational being that organizations are happy to incur overhead in flagging a constraint once found, but want to minimize any overhead associated with checking clean data). The running time is on the order of a few milliseconds and increases almost linearly for both transactions as the number of triggers increases. Without triggers, the running time was around 7 milliseconds, which increased to around 12 milliseconds when the number of triggers was increased to 50. This corresponds to about a 70% increase. In practice, however, the number of triggers on a single database table depends on the number of constraints on that table, and it is reasonable to expect no more than around 5 triggers per table (unless the number of columns is really large). Whether the overhead in the running time is acceptable depends on the application and how important the constraints are.

5.3 Comparison with other tools

In this section, we compare CBIM to two data cleaning tools, SSIS and DQguru. Both tools are semi-automated tools (both manual and automatic support) for finding, managing and fixing violations with support for defining constraints.

SSIS is a part of Business Intelligence Development Studio (BIDS) that comes with the SQL Server DBMS. We chose SSIS for comparison because of the wide use of SQL Server in organizations today and because of its sophistication. DQguru was selected because it is an open-source tool that works with a variety of DBMSs including SQL Server.

SQL Server offers data quality solutions through Integration Services (SSIS) by providing support for profiling, cleansing and auditing. SSIS works by specifying data flows. A data flow consists of a source, transformation and destination. The data source can connect to an SQL Server table or a flat file. The flow can be constructed using various kinds of tools for transforming the data. Profiling works during ETL by checking the number of rows, determining referential integrity (foreign key/inclusion dependency) errors, check for missing files in the source data, monitoring schema changes in the source and flagging data values that appear to be suspicious. The support for data cleaning includes checking for missing values, data duplication and inconsistent data formats. Finally, auditing provides services for performing a data lineage trail, data validation and data execution statistics. It provides audit tables to store details like records that contain errors and details about the errors.

DQguru, on the other hand, is marketed specifically as a "Data Cleansing & MDM Tool". It is a Java-based platform-independent tool that can be used with a variety of DBMSs including SQL Server. It provides support for removing duplicate records based on rules that can be specified, data cleaning, and address correction. In Section 5.4 we compare CBIM with DQguru and SSIS based on their constraint-handling capabilities.

Table 5.3 shows a comparison of DQguru and SSIS with a prototype implementation of CBIM. The comparison is based on the ability to specify constraints, to flag constraints, and to fix those violations. CBIM allows constraints to be specified in a tabular notation. In both DQguru and SSIS, the constraint will have to be specified in an ad hoc manner using expressions and functions. CBIM flags all violations in the On Deck Zone and shows the constraints causing them. SSIS maintains an Audit Errors table containing details about all violations. DQguru, however, has no mechanism for flagging violations and errors are handled on an ad hoc basis. Finally, CBIM has no mechanism for fixing constraint violations. DQguru and SSIS allow the user to express rules for fixing the violations.

Table 5.3 Comparison of tools in terms of support for constraints

Criteria	**CBIM**	**DQguru**	**SSIS**
Specify constraints	Tabular notation	Ad hoc	Ad hoc
Flag violations	Yes	Ad hoc	Yes
Fix violations	No	Semi-automated	Semi-automated

5.3.1 Support for Finding, Managing and Fixing Integrity Violations

In this section, we compare the tools with our CBIM prototype in terms of what it is like for an organization to use the tools to find, manage and fix integrity violations. Table 5.4 compares CBIM to DQguru and SSIS. In terms of ease of use, DQguru is the easiest. CBIM is harder since it requires tables and views to be set up properly, and constraints to be written in a tabular form. SSIS is the hardest to use, since it requires data flows to be created before profiling the data. Using CBIM to find integrity violations is

advantageous since it works internal to the database and can be integrated seamlessly into the database. This offers a kind of "real-time" management. Both DQguru and SSIS are required to be executed separately on the data. However, they have the advantage of providing direct support for fixing constraint violations, while CBIM requires no support for fixing violations. Finally CBIM has the advantage in managing violations since it does not reject any data and, at the same time, allows the clean data to be used for processing.

Table 5.4 Comparison of tools in terms of support

Support	**CBIM**	**DQguru**	**SSIS**
Ease of use	Medium	Easy	Hard
Finding integrity violations	Internal	External	External
Fixing violations	No	Yes	Yes
Managing violations	Yes	No	No

5.3.2 Working with Constraints

In this section, we evaluate how easy or hard it is to define and manage constraints. We compare CBIM with DQguru and SQL Server with respect to their constraint handling capabilities. Table 5.5 shows the criteria used to make the comparison. CBIM has the ability to specify arbitrary constraints, in addition to dependencies, that an expert can enforce in a DBMS. In addition, CBIM allows functional, inclusion and conditional dependencies to be specified, and generates triggers to flag rows that violate them. The ability to specify dependencies on data is possible in

CBIM which also provides support for generating triggers that can be used to flag violations. DQguru and SQL Server require the user to specify rules that are then validated against the data. However, unlike DQguru and SQL Server, CBIM lacks a mechanism to fix errors, as the data is only flagged and kept waiting for an expert to make the required fixes. CBIM currently lacks an easy to use user interface that allows a user to select the tables and columns relevant to a dependency.

Table 5.5 Comparison of tools for constraint-handling

Criteria	CBIM	DQguru	SSIS
Ability to specify arbitrary constraints on data	Yes	No	No
Ability to specify cleaning operation	No	Yes	Yes
Ability to specify dependencies	Yes	No	No
Easy to use user interface	No	Yes	Yes
Flagging violations	Yes	No	No
Managing violations and minimizing information loss	Yes	No	No

Both DQguru and SSIS have interfaces that are convenient to use. The "on-deck" mechanism is supported only by CBIM. Although SSIS does show rows that have integrity problems, it does not segregate them so that only the data within the Data

Integrity Zone is used in processing. Finally, CBIM uses triggers that automatically flag violations while SSIS requires profiling to be done in order to show violations.

In conclusion, CBIM seems to complement data cleaning tools by providing the ability to specify constraints. In processes like ETL, using a tool like SSIS seems to be more convenient particularly with its own user interface. They also have the option to manage cleaning of constraint violations which can be used when appropriate.

Chapter 6. Conclusions

6.1 Summary of Contributions

Integrity constraints are a powerful mechanism for defining data integrity and are effective for flagging or blocking inconsistencies. In this thesis, we introduced a conceptual model of constraint-based integrity management to show how to leverage constraints to manage inconsistencies in a systematic manner. We illustrated the potential of our approach using our case studies on PAL-IS and iMED-Learn, and discussed how to best implement the conceptual model. Our research contributions are along three dimensions:

- Gap Analysis: We provided a systematic analysis of the types of processes for which organizations need to manage data integrity and analyzed the data integrity problems that are encountered during those processes and the gaps faced by current approaches for managing data integrity. This is a representative list and typical of the kind of processes that are encountered in an organization that handles data. We also classified the different types of integrity constraints that can be used to ensure data integrity. The integrity of data depends on the context or process for which it is used, and is based on organizational requirements.
- Conceptual Model: We provided a conceptual model for dynamically managing data integrity in an organization. Our model introduces the notion of a Data Integrity Zone and an On Deck Zone. This allows the management of data integrity dynamically by protecting clean data while allowing dirty data into the database so that it can be cleaned. The model can be implemented in a database by using triggers to create views, and separately see the data integrity (clean) and

on-deck (dirty) data, as well as an integrated view of all the data. The flagged rows also show which constraint caused the violation, making it easier to fix them. However, we do not provide support for fixing violations, and an expert is required to review the violations in the On Deck Zone and fix them manually. The on-deck mechanism thus facilitates the processing of clean data by keeping the dirty data isolated, and also avoids rejecting data thus minimizing information loss.

- Tool Support: We provide a table-based approach for specifying constraints in a compact notation. Our tool uses this table to generate SQL statements for generating triggers in a DBMS that implement the flagging of constraint violations using our on-deck and data integrity views. Although our table can be used to specify any constraint, the tool currently only works for dependencies that can be written in a logical notation.

6.2 Thesis Limitations

In this thesis, we assumed that all constraints on the data have already been defined. We have not discussed the possibility of adding new constraints to the data. That will require checking whether the set of constraints is satisfiable, a problem which we mentioned to be NP-complete in Section 2.1. Also, adding new constraints may violate rows in the Data Integrity Zone and require clean data to be identified again. This can cause the amount of clean data to reduce as new constraints are added. Actually, the thesis is focused on the problem of managing integrity as data is updated and processed in a database rather than the problem of changing constraints.

Our tool support for generating SQL triggers from constraints currently works only for functional, inclusion and conditional dependencies. However, this could be extended to incorporate support for join dependencies. We also assume that functional dependencies and their conditional counterparts have a single column in their consequent. So, a constraint such as $[\text{Country} = \text{"Canada"}, \text{PostalCode}] \rightarrow [\text{Address}, \text{City}]$ will have to be entered as two separate dependencies: $[\text{Country} = \text{"Canada"}, \text{PostalCode}] \rightarrow [\text{Address}]$ and $[\text{Country} = \text{"Canada"}, \text{PostalCode}] \rightarrow [\text{City}]$.

6.3 Future Work

Our concept of data integrity and on deck zones, in effect, systematically segments data into clean and unclean data and ensures the maintenance of an ongoing clean view of data that cannot be compromised. This approach shows promise, although more work is needed in tool support for expressing constraints and maintaining constraints (including how to handle adding existing constraints to new databases). A more complete set of case studies and more work on implementation are needed to fully validate the approach.

Another issue is a change in the constraints that need to be enforced. Adding a new constraint might invalidate rows within the Data Integrity Zone. Also, if the new constraints conflict with any of the existing ones, it might result in a large amount of data in the On Deck Zone requiring an expert to then study the constraints and decide which one is causing the conflict.

A comprehensive analysis of the different types of constraints normally used by an organization is required. Such a list would be useful in better managing integrity. Our list of constraint types is comprehensive as far as dependencies are concerned. There are

other types of constraints for which there is no standard notation and may need to be explained in a sentence. For example, a check constraint on a column table can be easily added to a database table, but may not be possible to write in the standard form of a dependency.

There are also some constraints such as the primary key in a table that, when violated, must force a fix immediately before bringing the data in or the data must be rejected outright. A list of such constraints needs to be enumerated. More work needs to be done on how to manage user interaction, so that such violations can be fixed immediately when they are flagged, rather than simply moving them to the On Deck Zone to be fixed later. We have also seen that CBIM can be used as a complement to data cleaning tools for certain processes because of the latter's ability to clean data.

References

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

[2] Agrawal, R., and R. Srikant. "Fast Algorithms for Mining Association Rukes." *VLDB*, 1994: 487-489.

[3] Arenas, M., L. E. Bertossi, and J. Chomicki. "Consistent Query Answers in Inconsistent Databases." *PODS.* 1999. 68-79.

[4] Batini, C., and M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques.* Springer, 2006.

[5] Batini, C., C. Cappiello, C. Francalanci, and M. Maurino. "Methodologies for Data Quality Assessment and Improvement." *ACM Computing Surveys*, 2009: 41(3).

[6] Bayardo, R. J., and R. Agrawal. "Mining the Most Interesting Rules." *KDD*, 1999: 145-154.

[7] Benjelloun, O., H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. "Swoosh: A Generic Approach to Entity Resolution." *VLDB*, 2009: 18: 255-276.

[8] Bertossi, L., and J. Pinto. "Specifying Active Rules for Database Maintanance." *FMLDO*, 1999: 65-81.

[9] Bohannon, P., W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. "Conditional Functional Dependencies for Data Cleaning." *ICDE*, 2007: 746-755.

[10] Bravo, L., W. Fan, and S. Ma. "Extending Dependencies with Conditions." *VLDB*, 2007: 243-254.

[11] Bravo, L., W. Fan, F. Geerts, and S. Ma. "Increasing the Expressivity of Conditional Functional Dependencies without Extra Complexity." *ICDE*, 2008: 516-525.

[12] Brin, S., R. Motwani, J. D. Ullman, and S. Tsur. "Dynamic Itemset Counting and Implication Rules for Market Basket Data." *SIGMOD*, 1997: 255-264.

[13] Caruso, F., M. Cochinwala, U. Ganapathy, G. Lalk, and P. Missier. "Telcordia's Database Reconciliation and Data Quality Analysis Tool." *VLDB*, 2000: 615-618.

[14] Chaudhuri, S., and U. Dayal. "An Overview of Data Warehousing and OLAP Technology." *SIGMOD*, 1997: 26(1): 65-74.

[15] Chiang, F., and R. J. Miler. "Discovering Data Quality Rules." *VLDB*, 2008: 1166-1177.

[16] Dasu, T., T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. "Mining Database Structure; Or, How to Build a Data Quality Browser." *SIGMOD*, 2002: 240-251.

[17] *DataFlux Data Quality.* http://www.dataflux.com/Solutions/Technology-Solutions/Data-Quality.aspx.

[18] *DBE Software.* http://www.dbesoftware.com/products.

[19] Deutsch, A. "FOL Modeling of Integrity Constraints (Dependencies)." *Encyclopedia of Database Systems*, 2009: 1155-1161.

[20] Dong, X., and F. Naumann. "Data Fusion - Resolving Data Conflicts for Integration." *VLDB*, 2009: 1654-1655.

[21] Elfeky, M. G., V. S. Verykios, and A. K. Elmagarmid. "TAILOR: A Record Linkage Toolbox." *ICDE*, 2002: 17-28.

[22] *EMC Consulting Blogs: SSIS Junkie.* http://consultingblogs.emc.com/jamiethomson/archive/2008/03/03/ssis-data-profiling-task-part-7-functional-dependency.aspx.

[23] Fan, W., F. Geerts, and X. Jia. "A Revival of Integrity Constraints for Data Cleaning." *VLDB*, 2008: 1(2): 1522-1523.

[24] Fan, W., F. Geerts, and X. Jia. "Semandaq: A Data Quality System Based on Conditional Functional Dependencies." *VLDB*, 2008: 1460-1463.

[25] Fan, W., F. Geerts, V. S. Lakshmanan, and M. Xiong. "Discovering Conditional Functional Dependencies." *ICDE*, 2009: 1231-1234.

[26] Fehily, C. *Visual Quickstart Guide: SQL.* Second Edition: Peachpit Press, 2005.

[27] Galhardas, H., D. Florescu, D. Shasha, and E. Simon. "AJAX: An Extensible Data Cleaning Tool." *SIGMOD*, 2000: 590.

[28] Galhardas, H., D. Florescu, D. Shasha, E. Simon, and C. A. Saita. "Declarative Data Cleaning: Language, Model, and Algorithms." *VLDB*, 2001: 371-380.

[29] Geng, L., and H. J. Hamilton. "Interestingnes Measures for Data Mining: A Survey." *ACM Computing Surveys*, 2006: 38(3).

[30] Golab, L., H. Karloff, F. Korn, D. Srivastava, and B. Yu. "On Generating Near-Optimal Tableaux for Conditional Functional Dependencies." *VLDB*, 2008: 1(1): 376-390.

[31] Gupta, A., and I. S. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications." *IEEE*, 1995: 18(2): 3-18.

[32] Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram. "Design Science in Information Systems Research." *MIS Quarterly*, 2004: 75-105.

[33] Hipp, J., U. Guntzer, and U. Grimmer. "Data Quality Mining - Making a Virute of Necessity." *DMKD*, 2001.

[34] Huhtala, Y., J. Karkkainer, P. Porkka, and H. Toivonen. "TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies." *Computer Journal*, 1999: 100-111.

[35] *InfoSphere QualityStage.* http://www-01.ibm.com/software/data/infosphere/qualitystage/.

[36] Jiang, L., A. Borgida, and J. Mylopoulos. "Towards a Compositional Semantic Account of Data Quality Attributes." *ER*, 2008: 55-68.

[37] Johnson, T., and T. Dasu. "Data Quality and Data Cleaning: An Overview." *SIGMOD*, 2002: 681.

[38] Kimball, R., and M. Ross. *The Data Warehouse Toolkit.* Second Edition: Wiley Computer Publishing, 2002.

[39] Kolahi, S., and L. Libkin. "On Redundancy vs Dependency Preservation in Normalization: An Information-Theoretic study of 3NF." *PODS*, 2006: 114-123.

[40] Lee, M. L., T. W. Ling, and W. L. Low. "IntelliClean: A Knowledge-Based Intelligent Data Cleaner." *KDD*, 2000: 290-294.

[41] Lenzerini, M. "Data Integration: A Theoretical Perspective." *PODS*, 2002: 233-246.

[42] Mannila, H., and K-J. Raiha. "Dependency Inference." *VLDB*, 1987: 155-168.

[43] Medina, R., and L. Nourine. "A Unified Hierarchy for Functional Dependencies, Conditional Functional Dependencies and Association Rules." *ICFCA*, 2008: 98-113.

[44] Motro, A., and I. Rakov. "Estimating the Quality of Databases." *FQAS*, 1998: 298-307.

[45] Naumann, F., and M. Roth. "Information Quality: How Good Are Off-The-Shelf DBMS?" *IQ*, 2004: 260-274.

[46] Nevill-Manning, C. G., G. Holmes, and I. H. Witten. "The Development of Holte's 1R Classifier." *ANNES*, 1995: 239-242.

[47] Pipino, L. L., Y. W. Lee, and R. Y. Wang. "Data Quality Assessment." *ACM*, 2002: 45(4): 211-218.

[48] Rahm, E., and H. H. Do. "Data Cleaning: Problems and Current Approaches." *IEEE*, 2000: 23(4): 3-13.

[49] Raman, V., and J. M. Hellerstein. "Potters Wheel: An Interactive Data Cleaning System." *VLDB*, 2001: 381-390.

[50] *Redgate: Tools for SQL Developers.* http://www.red-gate.com/products/sql-development/.

[51] *SQL Power DQguru.* http://www.sqlpower.ca/page/dqguru.

[52] *SQL Server Data Quality Solutions.* http://msdn.microsoft.com/en-us/library/aa964137(v=sql.90).aspx.

[53] *Trillium Software.* http://trilliumsoftware.com/home/products/index.aspx.

[54] *UCI Machine Learning Repository.* http://archive.ics.uci.edu/ml/datasets.html.

[55] Widom, J. "Trio: A System for Integrated Management of Data, Accuracy, and Lineage." *CIDR*, 2005: 262-276.

[56] *WinPure Data Cleaning.* http://www.winpure.com/Article--DataCleaningTool.html.

[57] Yakout, M., A. K. Elmagarmid, J. Neville, and M. Ouzzani. "GDR: A System for Guided Data Repair." *SIGMOD*, 2010: 1223-1226.

Appendices

The SQL scripts that execute during two processes are shown here. They were created in SQL Server for simulating the creation of a new patient and a new encounter.

A. New Patient

```
DECLARE @firstName VARCHAR;
DECLARE @lastName VARCHAR;
DECLARE @birthDate DATETIME;
DECLARE @count INT;
SET @firstName = 'Vikram';
SET @lastName = 'Mallur';
SET @birthDate = '6/30/2011';
SET @count = (SELECT COUNT(PatientID) FROM PATIENTS WHERE FirstName =
@firstName AND LastName = @lastName AND BirthDate = @birthDate);
IF @count < 1
BEGIN

INSERT INTO Patients (PatientID, AddedDate, AddedBy, FirstName,
LastName, Birthdate, Active)
VALUES ('c0e74260-6156-4499-8217-0009ace62579', '6/30/2011', 'vmallur',
'Vikram', 'Mallur', '6/30/2011', 'true');

INSERT INTO PatientContacts (PatientID, Address, City, PostalCode)
VALUES ('c0e74260-6156-4499-8217-0009ace62579', '123 Fake St.',
'Ottawa', 'K1W 3E5');

INSERT INTO ESASThresholds (PatientID, Pain, Tiredness, Nausea)
VALUES ('c0e74260-6156-4499-8217-0009ace62579', 5, 5, 5);

INSERT INTO CareTeams (AddedDate, AddedBy, PatientID, Type)
VALUES ('6/30/2011', 'vmallur', 'c0e74260-6156-4499-8217-0009ace62579',
'Family Physician');

UPDATE Patients set diagnosis = 'Cancer'
WHERE patientid = 'c0e74260-6156-4499-8217-0009ace62579';

UPDATE Patients SET diagnosisType = 'GI.BLADDER'
WHERE PatientID = 'c0e74260-6156-4499-8217-0009ace62579';

END
```

B. New Encounter

```
INSERT INTO Encounters (FormID, AddedDate, AddedBy, PatientID,
EncounterDate, Label, EncounterType, Active, IsFinal) VALUES
('c0e74260-6156-4499-8217-0009ace6257a', '6/30/2011', 'vmallur',
'c0e74260-6156-4499-8217-0009ace62579', '6/30/2011', 'JPG Image', 3,
'True', 'False');

INSERT INTO InitialConsultationForms (Formid, Patientid) VALUES
('c0e74260-6156-4499-8217-0009ace6257a',
'c0e74260-6156-4499-8217-0009ace62579');

INSERT INTO Allergies (AddedDate, AddedBy, FormID, AllergyName) VALUES
('6/30/2011', 'vmallur', 'c0e74260-6156-4499-8217-0009ace6257a',
'Penicillin');

INSERT INTO Symptoms (FormID, SymptomName) VALUES
('c0e74260-6156-4499-8217-0009ace6257a', 'Pain');

INSERT INTO DRUGS (FormID, Name) VALUES
('c0e74260-6156-4499-8217-0009ace6257a', 'Betamethasone');

UPDATE InitialConsultationForms SET PPSScore = 60
WHERE FormID = 'c0e74260-6156-4499-8217-0009ace6257a';

UPDATE InitialConsultationForms SET Diagnosis = 'Non-Cancer'
WHERE Formid = 'c0e74260-6156-4499-8217-0009ace6257a';

UPDATE InitialConsultationForms SET DiagnosisType = 'ALS'
WHERE FormID = 'c0e74260-6156-4499-8217-0009ace6257a';
```