Modification of Integrity Constraints through Knowledge Discovery

Vijayalakshmi Atluri
MS/IS Department
and
Center for Information Management, Integration, and Connectivity (CIMIC)
Rutgers University, Newark, NJ 07102, U.S.A.
atluri@andromeda.rutgers.edu

Abstract
In a conventional database management system, integrity constraints are defined *a priori* and are static in nature. However, in many cases, real-world data is often unpredictable and evolves over time. In order to reflect these changes, there may be a need to modify the constraints. Moreover, it is quite natural that some data may violate the originally defined integrity constraints, but yet there is a need to store such exceptional data in the database. This is because, the schema may be ill-designed, or the world has changed since the design. Therefore, in order to capture the real-world situations, constraint modification is required in many systems. In such systems the constraints evolve based on the knowledge derived from the data and from the exceptions. In this paper, we show how such constraint refinement can be carried out through knowledge discovery mechanisms. We use an attribute-oriented generalization technique to derive knowledge.

1 INTRODUCTION

Constraints are defined in a database system so as to protect the integrity and consistency of the information. These are fundamentally invariant properties of a database state. In other words, to be in a consistent state, the contents of the database must adhere to all of the constraints. There are two fundamental types of constraints, those that are properties defined by the particular data model and those that reflect the data semantics associated with a specific application. For example, key constraints, domain constraints, referential integrity constraints, enforced by a relational database system belong to the first type of constraints. Semantic constraints range from simple type constraints, to more complex expressions, to arbitrary rules or policies of an organization. To assure that a database is consistent, the system enforces the constraints when any modifications to the data are made. Therefore, in a traditional database system, the information entering the database has to conform to the constraints set by the database designers.

However, as the real world is quite unpredictable, irregular and ever changing, the database system also has to be refined to accommodate these changes. Moreover, while designing the database schema, the designers may not have a complete understanding of the data and hence as they acquire new knowledge from data, they might want to modify the schema. Especially, in a design environment, it is not possible to perfectly define all the constraints at the initial stages of the design. Thus, any database system must have some degree of flexibility in order to fully support the activity of its users. Constraints which are part of the schema, also have to evolve depending on the knowledge derived from the data.

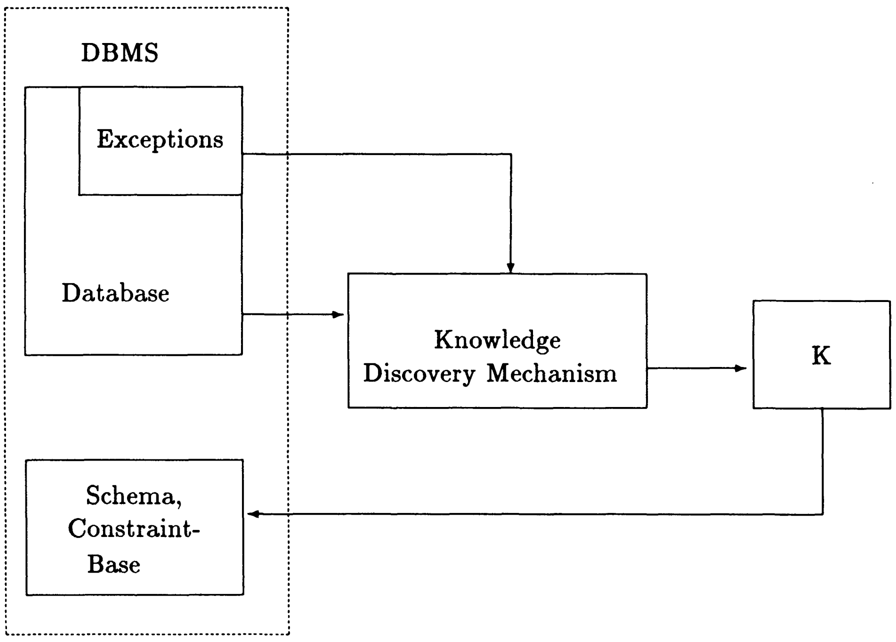

Figure 1 The System Architecture

More explicitly, modification of constraints in a system is required because of two reasons. (1) As the data is ever changing, new knowledge may be derived from this new data and it may be necessary to change some of the constraints initially set by the database designers to suit to the new environment. Hence by learning from the changes in data, constraints have to be modified, if necessary. (2) Suppose the schema is ill-designed or it no more suits to the changed data, the only way to detect that it is so, is to accept and store the exceptional data that violate the constraints. Hence, in order to capture the irregular behavior of the real world, the database system has to accept exceptions. If the exceptional data is enormously large, one has to modify the constraints.

This paper analyzes the situations in which constraint modifications are required and describes methods to carry out these modifications. As shown in Figure 1, the system accepts exceptions and stores this in logically separate files. The data and the exceptions are fed to knowledge discovery mechanisms to discover knowledge (in the form of rules). The discovered knowledge (K) is fed back to the system and is utilized to refine the schema.

Learning from the changes in data and from the exceptions, the system suggests the required modifications to the database schema in order to suit to the new data. Thus schema evolution involves refining constraints, adding or deleting new attributes, discovering new class hierarchies, etc. This paper deals with the evolution of constraint-base and it discusses the mechanisms to modify the constraint-base. The modification process only suggests a set of modifications to the constraints and outputs the same on request or when required. The process is not fully automated and human intervention is required to materialize the actual modification. Thus the system acts as an aid to the database administrator in refining the constraints.

This paper is organized as follows. The remainder of this section investigates the prior work and briefly describes the model assumed in this paper. In section 2, some motivating examples that emphasize the necessity to refine the constraints are enumerated. In section 3, the various types of constraints and the possible modifications in each of the type are described. The mechanism to derive knowledge from data is presented in Section 4. Section 5 and section 6 discuss the techniques to modify the constraints by learning from the changes in data and from exceptions, respectively. The last section points out the limitations of this paper and discusses the required future improvements.

1.1 Prior Work

In (Shepherd & Kerschberg 1986), the authors emphasize that constraint management is an essential part of a knowledge base system for managing both data and knowledge. A concise declarative language for expressing the constraints is provided in (Morgenstern, Borgida, Lassez, Maier & Wiederhold 1987). In (Vianu 1983), the author discusses the evolution process of functional dependencies that are considered as part of the constraint-base. The database schema evolves as new functional dependencies (dynamic constraints) are derived from the changes in the data over time. (Borgida 1985) points out the necessity to allow exceptional data into the database even though they do not conform to the integrity constraints set by the database designers. It discusses numerous examples that occur in real life which are considered as exceptions. It emphasizes that they have to be accepted by the database system and must be stored in order to capture the real life situations. It also discusses the problems encountered in handling these exceptions and the methods to resolve them. In (Borgida, Mitchell & Williamson 1986), the authors describe two methods to derive knowledge from the exceptions: empirical generalization and explanation-based generalization. When prompted by the database administrator or when sufficient evidence is accumulated, the system suggests alternative changes to the schema.

(Cai, Cercone & Han 1991) gives methods to derive knowledge from large relational databases. This presents algorithms to derive characteristic rules and classification rules from the database using an attribute-oriented induction approach. These methods are based on the techniques in "AQ15: Incremental Learning of Attribute-Based Descriptions from Examples"(Hong, Mozetic & Michalski 1992, Michalski, Brakto & Kubat 1997). Discovering classification rules has been implemented in INLEN(Kaufman, Michalski & Kerschberg 1991).

Our approach to modification of constraints using exceptions is similar to that in (Borgida 1985). While the main contribution of this paper is the methodology to modify constraints due to the changes in normal data; we present that due to exceptions for the sake of completeness.

1.2 Model

The data model assumed in this paper is a relational data model. It is important to note that the techniques developed in this paper are applicable to other data models such as object-oriented. Constraints are expressed as first-order logical expressions.

We use first-order predicate calculus as the primitive language for knowledge discovery from databases. From the logical point of view, each tuple in a relation is a formula in conjunctive normal form. For example, the following tuple,

Tname	Dept	Cadre	TDegree	Country
Paul	CS	GTA	B.S	U.S.A.

represents a logic formula,
$\exists x$ ((Tname(x) = Paul) $\wedge$ (Dept(x) = CS) $\wedge$ (Cadre(x) = GTA) $\wedge$ (TDegree(x) = B.S.) $\wedge$ (Country(x) = U.S.A.)

The methodology developed in this paper utilizes a conceptual hierarchy of attribute values. It assumes that the conceptual hierarchy is provided either by human experts like knowledge engineers or domain-specific experts or derived automatically by a conceptual clustering algorithm such as CLUSTER/2 in (Michalski & Stepp 1983). The concept hierarchy consists of different levels of

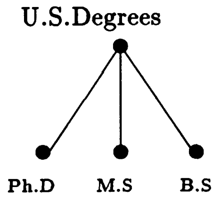

Figure 2 Conceptual Hierarchy

concepts organized as a partial order according to a general-to-specific ordering. The most specific concepts correspond to the specific values of attributes and the most general concept is a null description. For example, {Ph.D, M.S, B.S} $\subset$ U.S.Degree represents a conceptual hierarchy, which can be shown as in Figure 2. Here $\{A_1, \ldots A_n\} \subset B$ indicates that B is a generalization of A_i, for $1 \leq i \leq n$, that is, A_i IS-A B. For the sake of simplicity, this paper assumes that the conceptual hierarchies that represent the IS-A relationships are in the form of a tree. In other words, it assumes that there is no multiple inheritance.

2 SOME MOTIVATING EXAMPLES

This section presents two categories of examples: (1) examples that explain the necessity for the modification of the constraints based on the knowledge derived from data and (2) examples that emphasize the need to store exceptional data.

2.1 Dynamic constraints-Examples

The following two examples illustrate the necessity to have dynamic constraints in database systems rather than static constraints. These examples show that the constraints have to be changed based on the changes in data from time to time.

1. Let us consider a *University Database* and assume that it maintains data about *Career development*. Also assume that there is a constraint that all students with CS major have to take a predefined set of core courses (say c_1, c_2 and c_3). This constraint can be expressed as follows:
   $\forall(x)(student(x) \wedge major(x) = CS \Rightarrow course_1(x) = c_1 \wedge course_2(x) = c_2 \wedge course_3(x) = c_3)$

Suppose we derive knowledge from the data in *Career development* which says that all students of CS major who have taken a particular course c_4 are fully employed. Then we might want to modify the constraint by including this course also into the set of core courses. Thus the constraint has to be modified as follows:
$\forall(x)(student(x) \wedge major(x) = CS \Rightarrow course_1(x) = c_1 \wedge course_2(x) = c_2 \wedge course_3(x) = c_3 \wedge course_4(x) = c_4)$
This example suggests that there is a need to modify the constraints in the system depending on the knowledge derived from the data.

2. Consider the constraints encountered in a simple banking schema. One of the integrity constraints that most of the banks observe is,
balance > 0.
That means, if any customer tries to withdraw more than his/her balance, then that transaction would immediately be aborted. In some cases the bank my even fine the customer for doing so. If the customer has a very good history and if he is a valuable customer to the bank, then the bank might want to allow that customer to over draw. To facilitate this, it has to change the integrity constraint to either
balance > -100,
or, it might have to include another attribute such as ***history*** to the constraint which indicates whether he/she is a "*good* " customer or not and change the constraint to
If (history = good) then balance > -100.

2.2 Exceptions in databases

In most practical situations, there will occasionally be information which a user wants to store, even though it contradicts the constraints defined in the schema. A database should have the flexibility to accommodate such special cases. Let us consider the database of an organization. Here, some examples that are treated as exceptional information are enumerated. The examples are from (Borgida & Williamson 1985).

1. Suppose there is a constraint that "no person should earn more than 100K." But if few persons in the organization may have salary of more than 100K, the system immediately rejects such updates. If a higher bound is used in the constraint, then the error checking capability of the constraint will be lost.
2. Let there be a domain constraint on the attribute **degree** of each employee: **degree** $\Rightarrow$ {HSGD, BS, MS, PHD}. Some employee may have received a foreign degree which is not specified in the domain and is not equivalent to any American degree. It is important to store such a value in the database as there may be some decisions based on this. On the other hand, it is

not possible to add all possible foreign degrees at the time of designing the schema.

3. There may be a constraint which says "no employee should earn more than his/her supervisor." But in some special cases, a person may earn more than the supervisor especially when the supervisor is either on leave or a part-time employee or if the employee has worked over-time.
4. Suppose the system encounters a person outside the U.S.A., whose address is not in the standard form street, city, state, zip-code but might be having different attributes. In such a case, it is not practical to describe all forms of addresses in the world in the database but still one would like to store such data.

All these examples show that exceptional data is quite common in real-world and the database should be able to accommodate such occasional exceptions.

The next section discusses the various types of constraints encountered in database systems and the possible modifications that can be done to each type of the constraint.

3 CONSTRAINT MODIFICATION

Constraints in the database system are of many categories: constraints defined by the model and the constraints defined by the semantics of the data. In this context, as we are concerned with the semantics of the data, refinement of the constraints is due to the changed environment and circumstances of the real-world. To capture these changes and to suit to the new requirements, the constraints defined by the semantics of the data have to be changed. Modification of constraints is possible in either of the two ways: by relaxing the constraints or by strengthening them.

Semantic constraints that are encountered in a system can be of various kinds: (1) type constraints, (2) domain constraints, (3) value constraints, and (4) complicated rules defined in the system expressed as logical expressions, etc. Constraints also include the IS-A relationships as well as functional dependencies. Type constraints declare the type of a value that can be assigned to an attribute. Relaxing or strengthening such a constraint could be changing the type to a more general type or to a more specific type, respectively. Domain constraints specify the domain of the values each attribute can assume. Relaxing such a constraint is simply increasing the region of the domain and strengthening such a constraint is to constrict the domain. A range of values can be introduced or the length of the string can be increased in order to relax a constraint. Value constraints can be defined over one or more attributes and they may also contain logical expressions. A value constraint can be of the form,

$< expression >< comparison\ operator >< value >$.

The $< expression >$ can be either an arithmetic or a logical expression.

Relaxation(strengthening) of such a constraint is to increase(decrease) the range of the values satisfied by the expression either by increasing or decreasing the $< value >$.

Finally, rules can be expressed as
if $< expression >$ then $< action >$.

The $< expression >$ here is a logical expression. Relaxing a constraint is carried out by including a *new literal* in the following form:
$< expression \vee new\ literal >$.

Similarly, strengthening a constraint involves inclusion of a *new literal* such that,
$< expression \wedge new\ literal >$.

As described earlier, modification of constraints is performed in two cases. First, knowledge derived from the exceptional data may lead to the refinement of constraints. Second, new knowledge derived from the changed data, may lead to the modification of constraints.

If there is modification of a constraint because of the exceptions, it obviously results in relaxing the constraint. After the modification, the exceptions are no longer considered as exceptions but are considered as normal data.

However, if there is a modification of a constraint due to the knowledge derived from the new data, this modification might be either to relax the constraint or to strengthen it.

While refining the constraints, the consistency of the constraint-base has to be maintained. That means, no new constraint should invalidate an already existing constraint. In other words, the system of constraints should not yield a logical contradiction. While this paper does not address the issue of resolving such conflicts, interested readers may refer to (Yoon & Kerschberg 1993).

4 THE KNOWLEDGE DISCOVERY MECHANISM

This section describes a knowledge discovery mechanism from data stored as a relational database. This is an attribute-oriented, tree-ascending generalization technique, which adopts the AI learning techniques of "learning from examples." In order to derive knowledge from data one has to specify the task. The steps involved in the knowledge discovery process are as follows:

1. First step is to select the task-relevant data. This can be extracted by performing selection, projection and join operations on the relevant relations. In this process only the attributes relevant to the task have to be selected.
2. This step involves an attribute-oriented induction process, which generalizes each attribute value in the table by a higher-level concept. Hence,

by climbing the conceptual hierarchy, generalization is performed. But, if there is a large set of distinct values for a particular attribute, then that attribute has to be dropped. For example, the key attributes have to be dropped as they are distinct for each tuple and do not contain any knowledge. The generalization can be performed until a pre-specified threshold number of tuples is reached. The resulting relation is called the *generalized relation.*

3. This step involves simplification of the generalized relation. If several tuples contain the same attribute values except one, then they can be reduced to one by taking the distinct values of this attribute as a set.
4. This step transforms the generalized relation into a logic formula. Each tuple is represented as a logic formula in conjunctive normal form. If multiple tuples exist in the simplified generalized relation, they are represented as disjunctions of several conjunctive normal logic formulae.

This algorithm assumes that the conceptual hierarchy is in the form of a tree. But if there is multiple inheritance, there will be more than one higher-level concept. Therefore, the algorithm has to select the most relevant higher-level concept from the set of generalizations.

5 MODIFICATION OF CONSTRAINTS BY LEARNING FROM CHANGES IN DATA

This section discusses the modification of constraints based on the knowledge derived from the data which changes from time to time. This has to be accomplished in two phases. The first phase is to derive knowledge from the data and the second phase is to refine the constraints, if required, by integrating the derived knowledge into the constraint-base.

5.1 Knowledge Discovery from data

To discover knowledge from data, an attribute-oriented, tree-ascending generalization technique presented in section 4 is employed. The generalization is carried out basically by substituting attribute value of the lower-level concept by the corresponding higher-level concept. In order to derive knowledge, one has to specify the *target* of deriving such a knowledge. From the specified *target*, the *target class* tuples are extracted from the data. The process of generalization is performed only on the *target-class.* In this process, different tuples may be generalized to the same concept and it results in a reduced number of tuples. These small number of tuples can be transformed into a logic formula, which describes the rule derived from this *target-class.* The knowledge discovery process is illustrated with the following example.

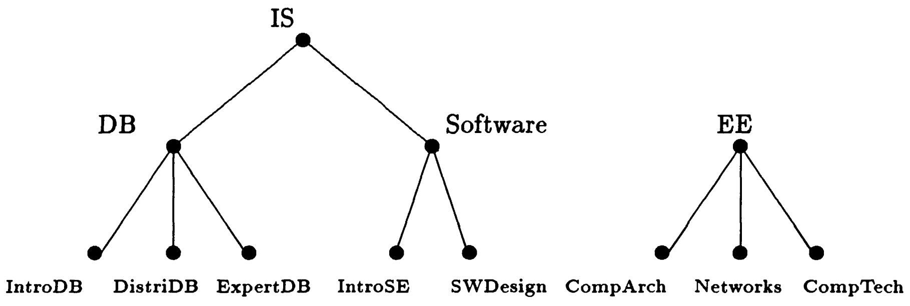

Figure 3 Concept Hierarchy

Course-history

Sname	Major	$Course_1$	$Course_2$
John	CS	CompArch	ExpertDB
Tom	CS	Networks	DistriDB
Paul	CS	Networks	IntroSE
Sam	CS	Networks	SWDesign
Ram	CS	CompArch	IntroDB

Employment

Sname	Status
John	employed
Tom	employed
Paul	unemployed
Sam	unemployed
Ram	employed

Consider a simple *University Database.* Assume that it maintains data about *Career Development* and contains two relations "Course-history" and "Employment" as shown below.

Let there be the following constraint on the students of "CS" majors, which mandates that all students of CS major have to take two courses outside their major area.

$\forall(x) Student(x) \wedge \text{major}(x) = \text{CS} \Rightarrow \text{Course}_1(x) \in \text{EE} \wedge \text{Course}_2(x) \in \text{IS}$

The concept hierarchy relevant to this example also shown in Figure 3 is as follows:

{IntroDB, DistriDB, ExpertDB } $\subset$ DB

{IntroSE, SWDesign } $\subset$ Software

{DB, Software } $\subset$ IS

{CompArch, CompTech, Networks} $\subset$ EE

Once, the user specifies the *target-class* which is nothing but the class of the previous students who are employed, say "employed-students." A join operation is performed to derive the required data and then the ***target-class*** is extracted from the derived relation. The resulting relation is as shown in below, which shows the tuples of the ***target-class*** separated from the rest of the data.

Course-history $\bowtie$ Career-development

Sname	Major	Course_1	Course_2	Status
John	CS	CompArch	ExpertDB	employed
Tom	CS	Networks	DistriDB	employed
Ram	CS	CompArch	IntroDB	employed
Paul	CS	Networks	IntroSE	unemployed
Sam	CS	Networks	IntroSE	unemployed

Now, the generalization is performed on this data, and the rule/rules that characterize the em target class are derived using the tree-ascending process. The generalization would results in the following tuple and is shown below.

generalized-relation

Major	Course_1	Course_2	Status
CS	EE	DB	employed

During the induction process, the key attribute "Sname" is dropped as it does not contain any knowledge. Therefore, this tuple can be logically expressed as,
$\forall(x)$ employed-students$(x) \Rightarrow \text{Course}_1(x) \in \text{EE} \wedge \text{Course}_2(x) \in \text{DB}$
which is nothing but knowledge discovered in the form of a rule, representing the ***target-class.***

The knowledge discovery methodology described here is similar to that in (Cai et al. 1991). At this point, the next step is to make use of this derived knowledge to refine our constraints, if necessary. The system has to see whether the derived knowledge is relevant or not to modify the constraints and if so it also has to integrate this knowledge into the previous schema and refine the constraints only that are relevant. If there is any modification required to the constraint-base, the system has to signal the same and has to output those changes so that the system administrator can take a final decision on materializing these changes. The following subsection discusses these aspects.

5.2 Constraint Refinement

There are two steps involved in performing the task of constraint refinement. The first step is to find the set of constraints that needs modification depending on the discovered knowledge. The second step is to integrate the discovered rule into the system.

1. Regarding the first task of determining the relevant set of constraints, the following are the two ways of doing it.

   (a) The first approach is to attach reasons to each constraint. These reasons are nothing but the descriptions of the existence of the constraint. In other words, one has to reason about the constraints. These rules have to be stored in the knowledge base. If we return to the constraint in the example in the previous subsection on the course requirement, these reasons could be (i) employment (ii) state-stipulated rule (iii) school policy (iv) department chairman's decision etc. Now the system compares the *target* with the existing reasons attached to each constraint by scanning through the reason-set and picks up the relevant set of constraints that matches with the *target.*

   (b) The second approach is a brute-force method that involves comparison of the attributes in the derived rule and those in each of the constraints. Whenever there is a constraint that contains any of the attributes in the rule, that constraint has to be included into the set of the relevant constraints.

2. The next task is to modify the set of relevant constraints. This can be done by simply replacing the right hand side of the relevant constraints by that of the derived rule. In our example, the constraint is modified as,
   $\forall(x)(\mathit{Student}(x) \wedge \text{major}(x) = \text{CS}) \Rightarrow (\text{Course}_1(x) \in \text{EE} \wedge \text{Course}_2(x) \in \text{DB})$

6 MODIFICATION OF CONSTRAINTS BY LEARNING FROM EXCEPTIONS

Once the system starts accepting exceptional data, number of problems crop up. In order to store the exceptional data many modifications to the data as well as to the constraints have to be made. If the exceptional data accumulates to a large quantity, then there is a need to modify the constraints. In this section, mechanisms to handle exceptions and those to modify the constraints are discussed (originally proposed in (Borgida & Williamson 1985)).

6.1 Handling Exceptions

This subsection explains the problems encountered in handling exceptions and the methods to resolve them. The problems that are usually encountered by allowing exceptions are the following:

1. How to store the exceptional data.
2. What to do if some other user accesses the exceptional data.
3. How to continue checking for future violations without causing any false alarms due to the past violations.

Basically, new data enters the database by one of the following operations: **create, insert, modify**. If there is a violation of an integrity constraint, usually the system responds to it by signaling a violation. In such cases, if the user still insists on storing that fact, the system allows it and stores that instance as an exception. This exceptional data can be entered by a special set of operators, **exnal-create, exnal-modify, exnal-insert**. This data is stored in logically separate files.

Similarly, if any user wants to retrieve this exceptional data, he/she will be cautioned that it is an exceptional data. This can be retrieved only by operations such as **exnal-retrieve**. Thus the user can determine whether normal procedures apply in such circumstances or some special actions have to be taken.

If there is a violation of a constraint, the system continues to signal alarms as long as the violated data is existing. Therefore, the constraint will be inconsistent with the database and it will not be able to distinguish between any later violations and the false alarm due to the old exception. Therefore, the system has to avoid these false alarms and must be able to continue to detect any future violations. And also, the system must be able to handle the situations when this exceptional data are accessed. Therefore, to handle this, all constraints in the system are modified into the following form:

e-constraint$_i$:
$\forall(x)$ SPECIALconstraint$_i(x) \vee$ constraint$_i(x)$,

where x is a sequence of variables, constraint$_i$ is the original form of the constraint and SPECIALconstraint$_i(x)$ is a predicate which prevents the actual condition being evaluated for exceptional cases. Initially, SPECIALconstraint$_i(x)$ is false, but as exceptions are encountered and excused for various argument tuples t_1, t_2, ..., then SPECIALconstraint$_i(x)$ becomes,

SPECIALconstraint$_i(x) \Leftrightarrow x = t_1 \vee x = t_2$ ...

In the following subsection, a technique for characterizing the class of exceptions and thereby refining constraints is presented.

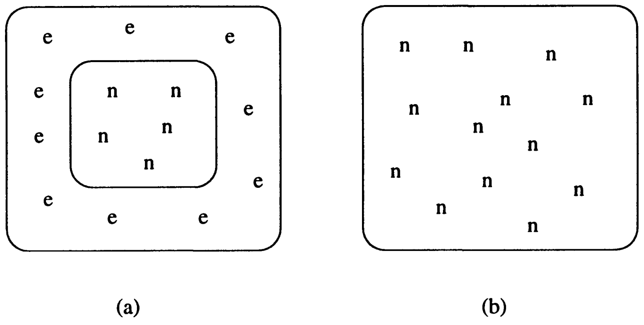

Figure 4 Exceptional and Normal Data

6.2 Constraint Modification

Constraint modification from exceptions always results in relaxing the constraint. Relaxation of a constraint is done in such a way that the previously known exceptional data is now considered as normal data. Figure 4(a) shows the normal data (n) and the exceptional data (e). After the modification of the constraint, all the exceptional data are considered as normal, which is shown in Figure 4(b).

Relaxing a constraint can be done either as described in section 3 or can be done by making the constraint to be checked only in certain restricted circumstances. To modify the constraints based on the knowledge derived from exceptions, the attribute-oriented generalization technique discussed in section 4, which makes a specific-to-general search is employed. The process of modification of constraints is explained with an example. Assume that the specified threshold level is "one."

Consider once again the *University Database* and assume that it contains a relation *Teaching-Staff* that store the information of all its teaching-staff. Also assume that there exists a domain constraint on the attribute of "TDegree," which is as follows:

$(\text{TDegree} \Rightarrow \{\text{Ph.D, M.S, B.S}\}) \vee \text{SPECIALconstraint}_{\text{TDegree}}(x)$

As mentioned earlier the constraint has already been modified to accommodate exceptions by including $\text{SPECIALconstraint}_{\text{TDegree}}$ to the actual constraint.

The concept hierarchy that is relevant to this example also shown in Figure 5 is as follows:

$\{\text{Ph.D, M.S, B.S}\} \subset \text{U.S.Degree}$

$\{\text{India, Pakistan}\} \subset \text{Asia}$

Let us assume that the relation *Faculty* is as shown in the following table.

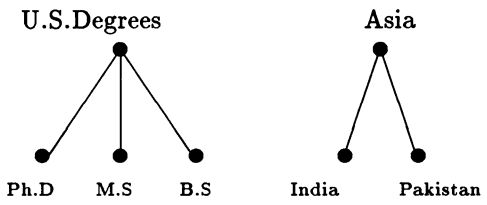

Figure 5 Conceptual Hierarchy

Faculty

Tname	Dept	Cadre	TDegree	Country
John	CS	GTA	Ph.D	U.S.A.
Tom	CS	GTA	M.S	U.S.A.
Paul	CS	GTA	B.S	U.S.A.

Now if a new tuple
Exnal-Faculty

Ravi	CS	GTA	M.Tech	India

enters the database which violates the constraint stated above, then it is stored as an exception. Suppose another exceptional tuple enters given below enters into the database.
Exnal-Faculty

Ahmed	CS	GTA	M.Tech	Pakistan

Then the generalization algorithm converts this into the generalized relation as shown below. (The exceptional data is considered as the task-relevant data.)

Generalized-relation

Dept	Cadre	TDegree	Country
CS	GTA	M.Tech	Asia

As we can see here, in the induction process, the attribute "Tname" is dropped as it does not contain any information. (Usually the key attributes are eliminated as they are distinct for each tuple and do not contribute to discover any knowledge.) In this process, the predicate $\text{SPECIALconstraint}_{\text{TDegree}}(x)$ is also modified as,
$\text{SPECIALconstraint}_{\text{TDegree}}(x) \Leftrightarrow x = \text{Ravi} \vee \text{x} = \text{Ahmed}$

If such kind of exceptional information mounts to some threshold value, the system signals a modification to the constraints. As the system already stores the relevant constraint violated by this exceptional data, that particular constraint is modified. As mentioned earlier, modification can be done in two ways.

The constraint can be generalized by increasing the domain of the attribute "TDegree." Thus, the modified constraint is as shown below.

(TDegree $\Rightarrow$ { Ph.D, M.S, B.S, M.Tech }) $\vee$ SPECIALconstraint$_{\text{TDegree}}(x)$

The predicate SPECIALconstraint$_{\text{TDegree}}(x)$ is still there to accommodate further exceptions.

The second approach to modify the constraint is to restrict the circumstances in which it is checked. In other words, make the system check this constraint unless "Country $\in$ Asia." This can be done as follows: From the above table, the following rule can be derived.

$\forall(x)$ ***Exnal-Faculty(x)*** $\Rightarrow$ (Dept = CS $\wedge$ Cadre = GTA $\wedge$ TDegree = M.Tech $\wedge$ Country $\in$ Asia)

This is compared with the rule derived from the normal data. This rule can be derived just the same way as that of exceptional data using the generalization technique. The generalized relation is as shown in below.

Generalized-relation

Dept	Cadre	TDegree	Country
CS	GTA	U.S.Degree	U.S.A.

$\forall(x)$ ***Faculty(x)*** $\Rightarrow$ (Dept = CS $\wedge$ Cadre = GTA $\wedge$ TDegree $\in$ U.S.Degree $\wedge$ Country = U.S.A.)

Then the attributes that have a different value in these two rules is identified with respect to the constraint on the attribute "TDegree." Therefore, the tuples that are being compared are,

Generalized-relation

Dept	Cadre	TDegree	Country
CS	GTA	M.Tech	Asia
CS	GTA	U.S.Degree	U.S.A.

Therefore, it results in "Country $\in$ Asia." Hence, the constraint is modified as,

(TDegree $\Rightarrow$ {Ph.D, M.S, B.S}) $\vee$ SPECIALconstraint$_{\text{TDegree}}$ $\vee$ Country $\in$ Asia

Whenever there is a modification to a constraint is performed on a constraint, the system outputs that suggested modification. The actual modification is made by the database administrator. The same approach can be employed in case of a simple value constraint or in case of a constraint represented as a first-order logical expression.

7 CONCLUSION

In order to capture the real-world situations and to fully serve to the users' needs, constraint modification is required in many systems. As the data entering the system is ever-changing, the refinement of constraints is required in order to suit to the environment. Whenever there is a change in the data, the changes in the data are such that it may be required to modify the constraint-base. And also, if there is an accumulation of a large amount of exceptional data, refinement of constraints is required. The modifications in both these cases is performed based on the knowledge derived from the data and from the exceptions. The derived knowledge is actually in the form of rules, and is derived using a generalization technique based on attribute-oriented, specific-to-general search. The derived rule is integrated into the constraint-base and refines the relevant set of the constraints either by relaxing them or by strengthening them.

In this approach, the system only suggests the required modifications but does not modify the constraints on its own. Therefore, a human being, usually the database administrator, is involved in the evolution process of the constraint-base. It is the job of the database administrator to decide what modifications have to be incorporated from the set of suggestions for each constraint. This makes the modification more tractable. Thus there is no automatic evolution process employed over here.

Our approach to the modification of constraints due to changes in normal data, however is limited to only those constraints that represent the characteristic rules of the data. More research is need to modify more general constraints including those that require comparison of more than one attribute, and those involve aggregate predicates. In the refinement process of the constraints discussed in section 5.2, in order to find the relevant set of constraints that need modification, scanning the entire set of reasons attached to the constraints is required. This is quite an expensive process especially when the set of rules attached to each constraint is enormously large. In such a case, one has to employ an optimization technique that identifies the relevant set of constraints more efficiently.

Performing modifications to the constraint-base might yield a logical contradiction. In order to maintain the consistency of the constraint base, the system should check for such conflicts and has to resolve them before materializing any modifications.

This paper assumes that the conceptual hierarchy is in the form of a tree. But, in real situations, there may be some concepts with multiple inheritance.

The techniques have to be modified as suggested in section 4, to suit to such more general systems.

REFERENCES

Borgida, A. (1985), 'Language features for flexible handling of exceptions in Information Systems', *ACM Trans. on Database Systems* **10**(4).

Borgida, A., Mitchell, T. & Williamson, K. (1986), Learning Improved Integrity Constraints and Schemas from Exceptions in Databases and Knowledge Bases, *in* M. L. Brodie & J. Mylopoulos, eds, 'On Knowledge Base Management Systems', Springer-Verlag.

Borgida, A. & Williamson, K. E. (1985), Accommodating Exceptions in Databases, and Refining the Schema by Learning from them, *in* 'he Proceedings of the 11th VLDB conference', Stockholm.

Cai, Y., Cercone, N. & Han, J. (1991), Attribute-Oriented Induction in Relational Databases, *in* regory Piatetsky-Shapiro & W. J. Frawley, eds, 'Knowledge Discovery in Databases', AAAI/MIT Press.

Hong, J., Mozetic, I. & Michalski, R. S. (1992), AQ15: Incremental Learning of Attribute-Based Descriptions from Examples, *in* 'The Method and User's Guide'.

Kaufman, K. A., Michalski, R. S. & Kerschberg, L. (1991), Attribute-Oriented Induction in Relational Databases, *in* regory Piatetsky-Shapiro & W. J. Frawley, eds, 'Knowledge Discovery in Databases', AAAI/MIT Press.

Michalski, R. S., Brakto, I. & Kubat, A. (1997), *Machine Learning and Data Mining: Methods and Applications*, John Wiley & Sons.

Michalski, R. S. & Stepp, R. E. (1983), 'Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonomy', *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1(4).

Morgenstern, M., Borgida, A., Lassez, C., Maier, D. & Wiederhold, G. (1987), Constraint-Based Systems: Knowledge about Data, *in* L. Kerschberg, ed., 'Expert Database Systems: The Second International Conference', Benjamin/Cummings Publishing Company, Menlo Park.

Shepherd, A. & Kerschberg, L. (1986), Constraint Management in Expert Database Systems, *in* L. Kerschberg, ed., 'Expert Database Systems: The First International Conference', Benjamin/Cummings Publishing Company, Menlo Park.

Vianu, V. (1983), 'Dynamic Constraints and Database Evolution', *ACM Transactions on Database Systems* .

Yoon, J. & Kerschberg, L. (1993), 'A Framework for Knowledge Discovery and Evolution in Databases', *IEEE Transactions on Knowledge and Data Engineering* .

BIOGRAPHY

Vijayalakshmi Atluri is an Assistant Professor of Computer Information Systems in the MS/CIS Department at Rutgers University. She received her B.Tech. in Electronics and Communications Engineering from Jawaharlal Nehru Technological University, Kakinada, India, M.Tech. in Electronics and Communications Engineering from Indian Institute of Technology, Kharagpur, India, and Ph.D. in Information Technology from George Mason University, USA. Her research interests include Information Systems Security, Database Management Systems, Workflow Management and Distributed Systems. In 1996, she was a recipient of the NSF CAREER Award to investigate issues related to incorporating multilevel security into database management systems for advanced application domains.