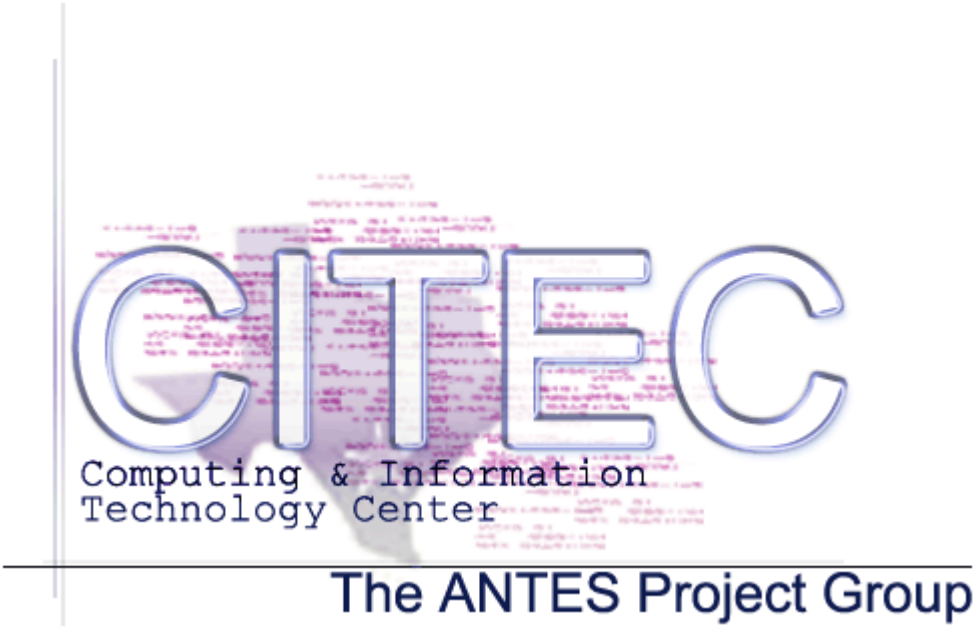

Database Design
June 16, 2003

Kevin Warkentine and Zhixiang Chen

Computing and Information Technology Center (CITeC), and Department of Computer Science, University of Texas-Pan American

Copyright © CITEC 2003

Contents

Copyright © CITEC 2003

1. Introduction

This document describes the database design for the ANTES System.

1.1 Design Overview

The ANTES database is a relational database consisting of 10 tables. The schemas of those tables and their relations are shown in Figure 1.

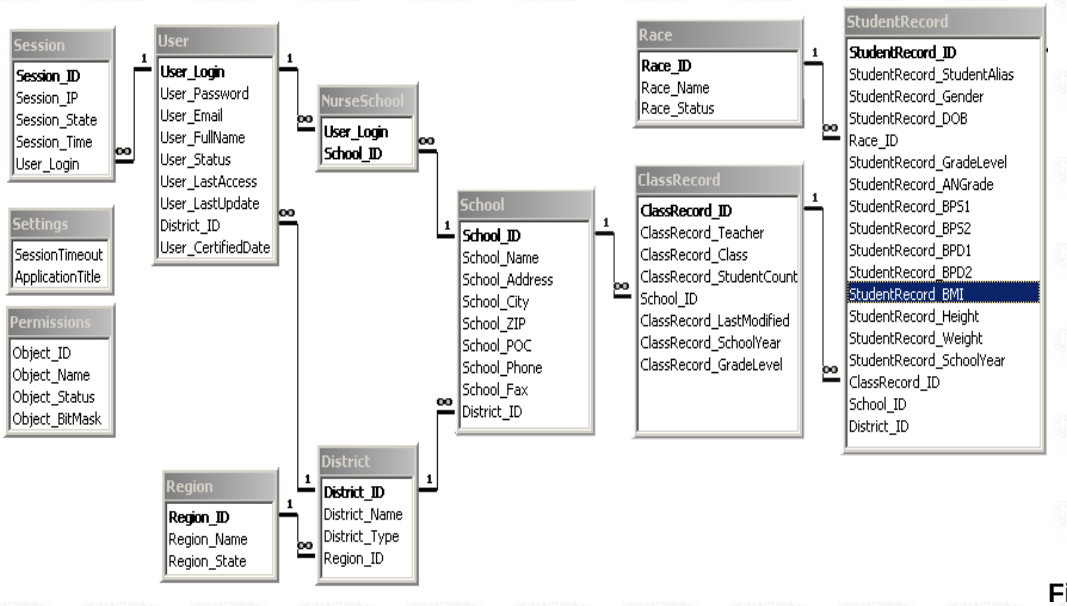

Figure 1: The Relational diagram of ANTES Database

1.2 Table Naming Conventions

For the entirety of this document, the following database table naming convention will be used:

Primary Key: The primary key is defined as one or more fields in a table that can uniquely identify any given data set (or row) in the table. A primary key will be denoted by bold face type and will be named according to the Table-name to which it belongs, followed by an underscore and the logical name of the field. For example, **Region_ID** is the primary key for the table named "Region"; its logical name is "ID."

Foreign Key: A foreign key is any field in a table that is used in an explicit relationship between another table such that the said *field* can uniquely identify records in the related table. For example, Region_ID in the District table can uniquely identify a single "region" record found in the Region table. The naming of a foreign key consists of the

Copyright © CITEC 2003

name of the table to which the foreign key is relating, followed by an underscore and the logical name of the field that it represents. Example: Region_ID is the foreign key in the District table, which relates the District table the Region table.

Table Fields: Any table field (or column name) that is not a primary or foreign key will be written in the same manner as a *primary key*, however, it will not be displayed in bold-face type.

2. Initial Table Structure

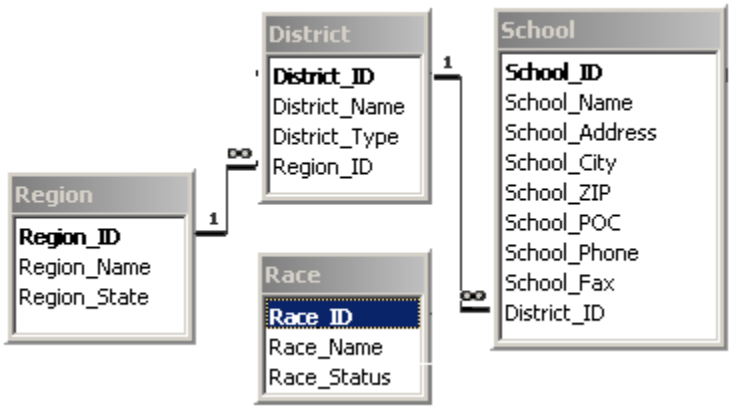

Figure 2: The initial table structure.

The diagram above shows the four initial tables in the design, along with their inter-relationships. A brief description of each table is provided below:

Region:

Description: The Region table stores information of all the regions. Each state can have many regions. And each region can have many districts. The "Region" table is involved in a one-to-many relationship with the District table. This means that one Region may have many "districts."

Region_ID: This is the primary key for the Region table. Type: Integer.

Region_Name: This is the name given to the particular region. Type: Text(20)

Region_State: This is the foreign key linking the Region table to the State table. Type: Text(2)

Copyright © CITEC 2003

District:

Description: The "District" table stores information of all the districts. It is involved in a one-to-many relationship with the School table. This means that one "district" can have many "schools."

District_ID: This is the primary key for the District table. Type: Integer

District_Name: This is the name given to the particular district. Type: Text(20)

District_Type: This is BYTE value represents whether the District contains schools that are "Private" or "Public". If this value is 0, then the District is Public. If the value is a 1, then the District contains only Private schools.

Region_ID: This is the foreign key linking the District table to the Region table. Type: Integer.

School:

Description: The "School" table stores information of all the schools.

School_ID: This is the primary key for the School table. Type: Integer.

School_Name: This is the name given to the particular district. Type: Text(55)

School_Address: This is the Street address of the school. Type: Text(50)

School_City: This is the city the school is located in. Type: Text(30)

School_Zip: This is the zip code the school resides in. Type: Text(10)

School_Phone: This Text(10) value represents the 10-digit phone number of the school.

School_Fax: This Text(10) value represents the 10-digit fax number of the school.

School_POC: This is the name of the "point of contact" for this school. Type: Text(30)

District_ID: This is the foreign key linking the School table to the District table. Type: Integer.

Race:

Description: This table stores all the Names of the different Races used in the forms. It shall be used to help eliminate typographical errors in the code and to maintain data consistency.

Copyright © CITEC 2003

Race_ID: This is the primary key for the Race table. It is of type Byte.

Race_Name: This is the name given to the particular race.

Race_Status: This BYTE value determines whether the given Race is visible or not in the user data entry form for student records. If Race_Status = 0, then that particular race will not show up in the form, otherwise it will be an available choice as a Race/Ethnicity. If a Race is to be deleted by an Administrator, then its status is set to 0 so that is invisible to the users.

3. Extended Table Structures

The following Table Diagram shows the rest of the database design. These tables, in particular, ClassRecord and StudentRecord, make up the core of the database.

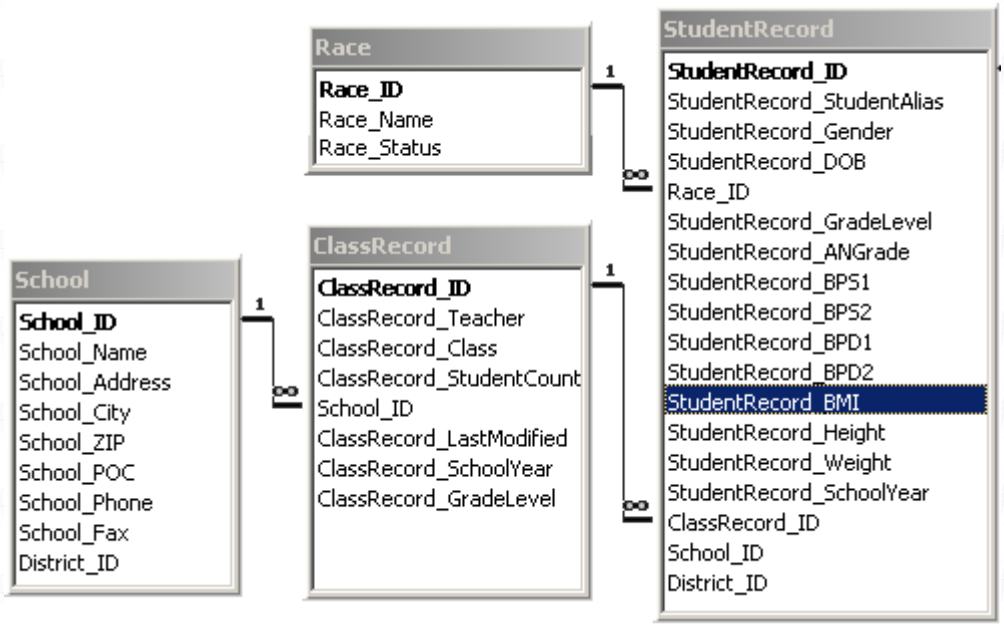

Figure 3: The second half of the database structure. Recall from Figure 1 that *School* is also linked to the District table.

3.1. Data Retrieval Efficiency

To maximize data retrieval efficiency, we would like to store as much useful information as possible in one "master" table. This way we would minimize the amount of time required to query across multiple tables. Of course this method creates certain redundancy. However, with this design, we can retrieve medical records from any school, district, or class (assuming the Class_ID is known) without the need of cross-table querying. Cross-table querying would be *required* in following scenarios:

Copyright © CITEC 2003

- Performing statistical queries based on "Class Size" – This would require a link to the ClassRecord table.
- Performing queries based on Teacher/Class/School – Requires a link to the ClassRecord table and School table (if School_ID is not known)
- Retrieving data based on Region or State – Requires a link to the District and Region tables.

3.1.1 Table Design

ClassRecord:

Each entity in the ClassRecord table serves to represent a single "ANTES Consolidated Data Form." Descriptions of the fields in this table are provided below:

ClassRecord_ID: The primary key. Type: LONG

ClassRecord_Teacher: The name of the teacher that the given student set belongs to. This is of type Text(30), so it can vary between schools.

ClassRecord_Class: This is the "class identifier" which tells us which classroom/course number that the student data set is associated with. This is a Text(20) field so it can vary between schools (i.e., Bio 101, PE101.23, 1001, etc).

ClassRecord_StudentCount: This is a BYTE value (0..255) that represents the total number of students that were screened in the given data set. This value would be useful for statistical purposes.

ClassRecord_LastModified: This is the date that the record was first created, or last modified.

ClassRecord_SchoolYear: This INTEGER value represents the academic school year in which the data set for the given class was first collected. Example: 2001 represents the academic year of "2000-2001."

School_ID: This is the foreign key that relates ClassRecord entities to the "School" they belong to. This field is of type INTEGER.

StudentRecord:

The StudentRecord table is the "master table" of the database, which stores all the raw data that is collected each year. It is a combination of Student data, Medical data, and "School Location (School/District)" information.

Copyright © CITEC 2003

StudentRecord_ID: This is the primary key for the table, of type LONG. Note: It could also be possible to set Student_Alias and Class_ID as the primary key.

StudentRecord_ANGrade: This is a BYTE value (0 – 4) that represents the grade of the AN (Acanthosis Nigricans).

StudentRecord_BPS1: BYTE value that represents the first systolic blood pressure reading.

StudentRecord_BPS2: BYTE value that represents the second systolic blood pressure reading.

StudentRecord_BPD1: BYTE value that represents the first diastolic blood pressure reading.

StudentRecord_BPD2: BYTE value that represents the second diastolic blood pressure reading.

StudentRecord_Height: DECIMAL value representing the student's height.

StudentRecord_Weight: DECIMAL value representing the student's weight.

StudentRecord_Alias: This Text(15) value is the "student ID" used internally by the given school/class the student belongs to.

StudentRecord_Gender: Byte value (1=male, 0 = female)

StudentRecord_DOB: Date value representing student date of birth.

Race_ID: This foreign key is a BYTE value representing values (1 ~ 20) that correspond to a Race_ID in the Race table.

StudentRecord_SchoolYear: Date value that represents the school year that the data was collected during. Example: 2002 represents the academic year "2001-2002."

StudentRecord_GradeLevel: BYTE value that represents the student's grade level at the time of collection.

ClassRecord_ID: This LONG value is the foreign key that relates to the ClassRecord table. This tells us that there is a One-to-Many relationship between ClassRecord and StudentRecord: A ClassRecord can have many StudentRecords, however, we must also enforce that no ClassRecord has more than one StudentRecord for the same StudentRecord_Alias.

School_ID: This INTEGER value is the foreign key that relates to the School table. This allows for easy (faster) searching of data according to school.

Copyright © CITEC 2003

District_ID: This INTEGER value is the foreign key that relates to the District table. This allows for easy searching of data according to District.

3.1.2 Design for Optimization

To optimize the performance of the database, we will enforce an additional table naming convention and require a set of periodic structural maintenance procedures. These requirements, listed below, will optimize performance of any data queries or insertions relating to the current year of data collection.

1. The logical tables *ClassRecord* and *MedicalRecord* will, at any given time, only store data for the current academic year for which data is still being collected. These tables will be given the names *ClassRecord_Current* and *MedicalRecord_Current*, respectively.

2. At the end of each year, or whenever all data collection for the *current* year has been completed, each record-set in both *ClassRecord_Current* and *MedicalRecord_Current* will be moved into a set of *Archive* tables: *ClassRecord_Archive* and *MedicalRecord_Archive*.

3. Logically, these *Archive* tables are the same as the logical *ClassRecord* and *MedicalRecord* tables. However, since we cannot automatically guarantee that duplicate entries may be formed when moving new data into these archive tables, the primary key of the *ClassRecord_Archive* table will be the set of two fields **ClassRecord_ID** and **ClassRecord_SchoolYear**.

3.2. Optional Extensibility

This part of the design will probably not be implemented into the final product; however, a design methodology is provider where the user would be able to add specific attributes to the Survey Form.

3.2.1. User-defined “Text Attributes”

To allow for the addition of user-defined input fields, we create an “Attribute” table that acts as a Bridge between the *attribute mapping* table and the StudentRecord. In this case the attribute type is “Text.” However, if the user desires to have Integer values, then we would just create two new tables: IntegerAttribute and IntegerAttributeMap.

Copyright © CITEC 2003

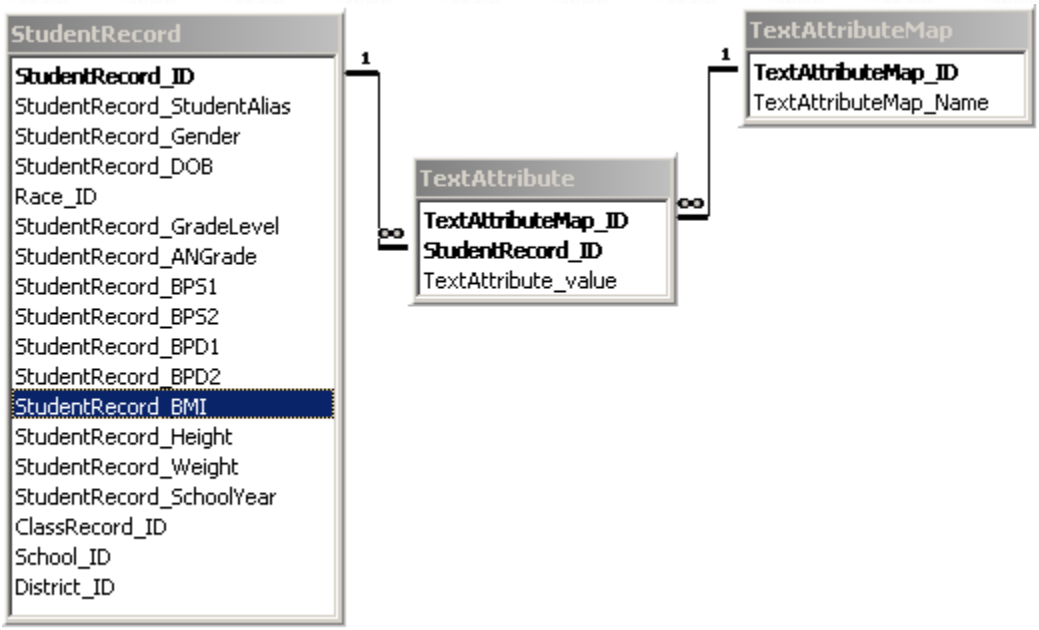

Figure 4 – This diagram shows the additional tables that could be added to provide the user to add custom fields to be added to the "Consolidated Data Forms" recorded in the database.

The mapping table maps an "attribute ID" (for example MapID = 12; Map_Name = "Hair Color") to the "name" of the attribute being created. The TextAttribute table stores all the text attributes of each StudentRecord (if any).

4. Implementation-Specific Database Design

This section describes the tables that are depended upon only by the *front-end* application to provide appropriate access and data retrieval to the users of the database.

4.1. User Access Control

To enforce certain user access control, the User and UserSchool tables are used to map each user access data of one or more schools. The *User* table maintains that only authorized individuals have access to *privileged* data in the database. The *UserSchool* table is used to determine the extent of data that the given user is allowed to access (at the school level).

Copyright © CITEC 2003

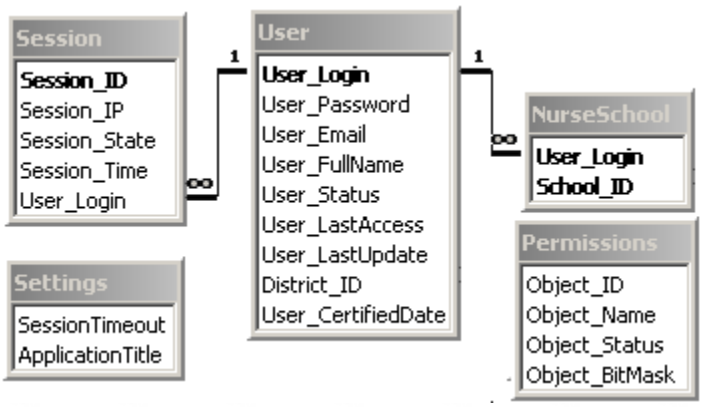

Figure 5: The *User* table and the *UserSchool* tables determine which *schools* each *user* has access to. The Session table is used to maintain user sessions for the application. The optional *Settings* table is used to store various application specific settings. The optional *Permissions* table is used to provide a mechanism for assigning *object access* permissions to the different types of users of the system.

4.1.1. Table Structures

User: The User table contains the following attributes:

User_Login: The *login* credential by which the user's access is authenticated. Type: Text(25)

User_Password: The *password* credential by which the user's access is authenticated. Type: Text(15)

User_Email: The email address the user specifies to be used for password retrieval purposes. Type: Text(30).

User_FullName: The display name given to the user. Type: Text(30)

User_Status: This BYTE value can be 1 of 5 values as defined as follows:

0 – Inactive. Indicates that the user cannot access the database.

1 – System Administrator. A user of this status will have full database administration rights.

2 – Supervisor. Generally, a user of this status can view data at the *District* level.

3 – District Nurse Coordinator (DNC). DNC related activities are granted to a user of this status.

Copyright © CITEC 2003

4 – Nurse. Nurse-related privileges are granted to a user of this status.

5 – Teacher. Teacher-related privileges are granted to a user of this status.

User_CertifiedDate: This DATE/TIME value represents the date the "nurse" was certified. It is useful for calculating when the certification expires.

User_LastAccess: This TIME/DATE value will store the last access time the user logged into the database system.

User_LastUpdate: This DATE/TIME value represents the time the user was either first created, or last modified.

District_ID: This INTEGER value is a foreign key to the District table and represents the district that the given user has "access" to. If the user is a Nurse, then he/she can only be assigned to schools in the same district (or a private school in the same region). If the user is a DNC, he/she can only assign nurses to schools in the district he/she coordinates. For users that are administrators, this field is not important.

NurseSchool: The NureseSchool table contains the following two attributes:

User_Login: This Text(25) value is one of the primary keys that also serve as a foreign key linking to the User table.

School_ID: This INTEGER value is part of the dual primary key that determines which *Schools* the given user (nurse) has access to.

Session (optional): The Session table is provided as an alternative method (independent of the web server) of recording and managing user sessions. The table has the following set of attributes:

Session_ID: This is a 16-Byte Global Unique Identifier (GUID) that serves as the primary key for the session table. It is generated either when the user first *accesses* the website, or when they first *logon* through the website. This choice is solely based upon preference. The first option would allow the assignment of certain privileges to "default" users. And it would also allow one to log/track the amount of unique visitors at any point in time. The second option would require less storage and session ID's would not need to be tracked unless the user logs on; thus, non-privileged pages would not require *session-tracking* code. The GUID may be automatically generated by the database, or on the web server (via ASP code), provided that the resulting GUID cannot be easily guessed, or spoofed.

Session_IP: This Text(15) value represents the IP address associated with the user logged in under the given Session_ID. Although the Session_ID is an adequate primary key, the

Copyright © CITEC 2003

Session_IP should also be used as a *logical* primary key, in conjunction with the Session_ID, to provide greater security. Requiring that both Session_ID and Session_IP be used for page request validations likewise requires that a would-be 'hacker' must spoof both the Session_ID *and* the user's IP address in order to gain unauthorized access.

User_Login: This Text(25) value serves as a foreign key that relates to the Login table. If Session_State (as defined in the following paragraph) = 1, then User_Login is used to determine which user is currently associated with the given Session_ID. This information can then be used to determine the access rights of that user.

Session_State: This BYTE value represents the 3 possible states of the session:

0 : The status of this session is "logged out"

1 : The status of this session is "logged in"

2 : The status of this session is "Expired"

Session_Time: This is the timestamp at which SessionID was generated, or at which point SessionID was "renewed" (i.e.: Session_State changes from 0 to 1). The Session_Time should be used by the application upon each page request to determine if the session needs to be "expired".

Permissions: The Permissions table is provided to allow permission rights to be assigned to individual "objects" in the application. For example, certain queries or reports can be given a set of permissions that allow access only to the specified user types or groups (administrator, DNC, nurse, etc). The two fields in this table are described below.

Object_ID: This BYTE value is the primary key, and represents a numeric "code" assigned to the given object.

Object_Name: This Text(30) value represents the name of an object that has been assigned a given set of permissions.

Object_Status: This BYTE value can be one of two values:

0 – The object is hidden from the user interface and cannot be edited

1 – The object is visible from the administration interface and the permissions bit mask may be edited.

Object_BitMask: This BYTE value represents the user permissions allowed to access the given object. The last 4 or 5 (depending on whether we include "public" as a "user type") bits in the BYTE are used as the bit mask to determine if a given "user type" has access to the object in question.

Copyright © CITEC 2003

4.1.2. BitMask

BitMask Overview

The table below shows all the possible BitMasks (ignoring the 1st 4 bits) that could be used to provide access rights to 4 types of users. The decimal equivalent of each bit mask is listed with the most common access rights shown in bold.

		BitMask (8 bits)							
Decimal	Access Rights	-	-	-	-	Admin	DNC	Nurse	Public
0	Nobody has access	0	0	0	0	0	0	0	0
1	**Public**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
2	**Nurse**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
3	Nurse, Public	0	0	0	0	0	0	1	1
4	**DNC**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
5	DNC, Public	0	0	0	0	0	1	0	1
6	DNC, Nurse	0	0	0	0	0	1	1	0
7	DNC, Nurse, Public	0	0	0	0	0	1	1	1
8	**Admin**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**
9	Admin, Public	0	0	0	0	1	0	0	1
10	Admin, Nurse	0	0	0	0	1	0	1	0
11	Admin, Nurse, Public	0	0	0	0	1	0	1	1
12	**Admin, DNC**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**
13	Admin, DNC, Public	0	0	0	0	1	1	0	1
14	**Admin, DNC, Nurse**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**0**
15	Everyone has access	0	0	0	0	1	1	1	1

Figure 6 – The first possible BitMasks to assign access rights (up to 4 user types) to an object.

Creating a new User Access BitMask

The process of creating an object's BitMask is quite simple, if we are starting from *scratch.* This can be done by summing the decimal values of each *User-Type* that is allowed to access the object. For example, suppose we want to create a BitMask that would allow both DNC and Admin to access an object. The resulting bitmask would be:

```
Creating a New Bitmask

  DNC          ( 4)
+ Admin        ( 8)
-------------------
  DNC,Admin    (12)
```


Copyright © CITEC 2003

Figure 7: Creating a new bitmask

However, this method only works correctly when the components of each UserType do not "overlap." For example, if we tried to add the UserTypes "DNC, Admin" (decimal 12) and "DNC, Nurse" (decimal 6), we would get the BitMask 0001 0010, which would grant access only to the UserType Nurse. The correct way to compute the BitMask would be 12 + 2 = 14, or 4 + 8 + 2 = 14.

Updating an object's existing User Access BitMask

The most basic method of *updating* the existing User Access BitMask of an object is to decompose the BitMask into its UserType components and then "re-create" it (using the method in above section), incorporating the new UserTypes into the new BitMask. Although this method works, it requires that you *know* the individual UserType components of the existing BitMask. However, it is possible to *blindly* add a UserType to an existing BitMask without decomposing the existing BitMask. This is accomplished via the bitwise OR operation. For example, suppose we have an existing User Access bitmask of 3, and we want allow access to two more UserTypes: Admin (decimal 8) and DNC (decimal 4), regardless of whether they already have access. To do this, we OR the

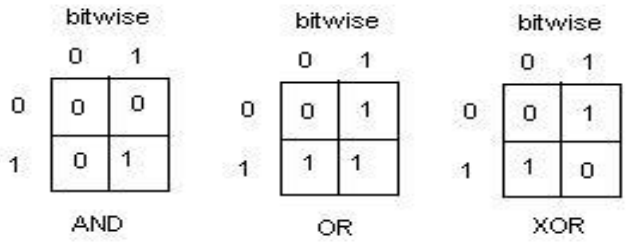

Figure 8 –Truth tables of bitwise operators used

decimal values of Admin and DNC to the decimal value of the existing User Access bitmask. Another method is to SUM the UserTypes you wish to "add" access to, then OR that result with the existing User Access BitMask. This can be seen in the diagram below:

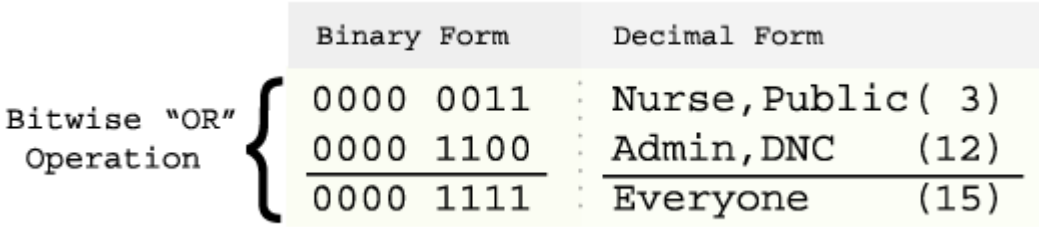

Figure 9: Blindly updating an existing User Access bitmask.

Copyright © CITEC 2003

Removing UserTypes from an Existing BitMask:

Removing UserYpes from an existing BitMask is slightly more complicated, involving an additional step. First we must negate the *User-Type* mask by XOR'ing it with the bits 1111 1111 (decimal 255). Then we AND the result with the current BitMask of the object. In the example below, we are removing the Nurse and Public UserTypes from the existing BitMask Everyone (decimal 15).

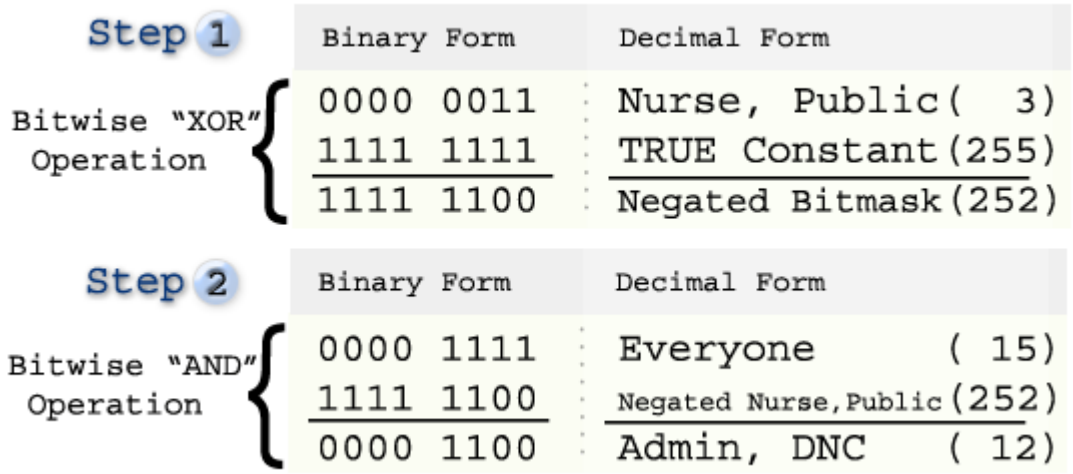

Figure 10: Blinding removing a UserType from a User Access BitMask.

5. Application Settings (optional)

To provide an easy method of updating the content of the user-interface, the Settings table is provided as a central location to store and retrieve interface settings. .

5.1 Suggested Settings

Session_Timeout: This BYTE value would represent the number of minutes before a sessionID should expire.

Content_Footer: Text/Memo field containing the HTML that makes up the footer of each page.

Content_AboutUs: Text/Memo field containing the HTML that briefly describes the project.

Copyright © CITEC 2003

6. Security Issues

Although the majority of security issues lie in the scope of the application design, some aspects of the database design can affect the security of the application, namely the option to store the data in encrypted form. However, the added encryption could have negative effects on the performance of data retrieval. Perhaps a more reasonable solution would be to encrypt key elements in the database that could be considered as "sensitive," for example, the *password* field in the *User* table.

6.1. Password Storage

Concerning passwords, an *application specific* option would be to store an MD5 hash of the password in the database, as opposed to the actual password. Then, to validate that the user has entered the correct password, we compare the MD5 of the "password" to the value stored in the database. This would present a more secure method of transmitting the users password over a network, without requiring the use of SSL encryption. However, this method falls within the application design and is only mentioned here as an alternative to database encryption.

Copyright © CITEC 2003