

Optimum Database Design: Using Normal Forms and Ensuring Data Integrity

by Patrick Crever, Relational Database Programmer, Synergex

Printed: April 2007

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

The software described in this document is the proprietary property of Synergex and is protected by copyright and trade secret. It is furnished only under license. This manual and the described software may be used only in accordance with the terms and conditions of said license. Use of the described software without proper licensing is illegal and subject to prosecution.

© Copyright 2001, 2004, 2007 by Synergex

Synergex, Synergy, Synergy/DE and all Synergy/DE product names are trademarks of Synergex.

All other product and company names mentioned in this document are trademarks of their respective holders.

DCN WP-04-0003

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

<http://www.synergex.com>

phone 916.635.7300

fax 916.635.6549

Contents

Data Access Keys	1
Normal Forms (Logical Model Design)	2
First Normal Form	4
Second Normal Form.....	6
Third Normal Form	8
Denormalization	10
Data Integrity	10
Entity Integrity.....	11
Referential Integrity.....	11
Domain Integrity.....	12
User-defined Integrity.....	13
Stored Procedures and Triggers.....	14
Conclusion.....	14

Optimum Database Design:

Using Normal Forms and Ensuring Data Integrity

Database design plays a vital role in the operation of reliable and efficient database systems. A well-designed database allows a Synergy™ application using SQL Connection or xfODBC to work most efficiently. But even the fastest ODBC driver cannot make up for database design deficiencies.

Programmers are routinely searching for solutions to problems that derive, ultimately, from a poorly-designed database. Design problems can lead to inaccurate and inconsistent data, data modification problems, data access problems, reduced flexibility when changing business rules, and performance degradation—just to name a few. By applying the rules of normalization and data integrity described in this article, database designers can eliminate many problems that result from a database that was not designed for maximum efficiency and flexibility.

Data Access Keys

Database keys play a vital role in directly accessing one or more rows of data. Keys eliminate the need to sequentially read an entire table, filtering out rows not meeting a search criteria. Applying the techniques of data normalization determines which columns become keys or whether a unique identifying column should be added to a table. Before discussing data normalization techniques, we need to have a general understanding of the three basic key types.

Primary keys are the most important of all the key types. Primary keys consist of one or more columns that in combination form an attribute (or value) that uniquely identifies a single row from all other rows. An example of a primary key is a student ID in a student information table, which is used to uniquely identify each student and allow direct access to a student's information. Another table, course information, contains a course ID that uniquely identifies each course. Primary keys are used to extract data from one or more tables as specified by the WHERE and FROM clauses of an SQL command.

Secondary keys are used to optimize sorting of the final data set. Further ordering of the data is performed as specified by an ORDER BY or GROUP BY/HAVING clause to create the result-set. A well-tuned query combines both primary keys and secondary keys to generate presorted data-sets, thus eliminating the sort process and improving performance. An example of a secondary key is a zip code column in the student information table.

Relationships between tables are created using both primary and foreign key constraints. A foreign key constraint is a column in a table that is the same as the primary key column or a unique column in another table, creating a relationship between the two tables. A table can have more than one foreign key. Foreign key constraints work in conjunction with unique or primary key constraints to establish and ensure data—or *referential*—integrity between specific tables. Referential integrity and unique constraints are discussed below.

Database design literature commonly refers also to a candidate key. Although not really a key, candidate keys have the potential to become primary or secondary keys. Candidate keys consist of one or more columns that uniquely identify a row in the table. During the database design phase, candidate keys become apparent as tables are restructured into a normalized form.

An important aspect to remember is that keys increase read performance while decreasing write performance. Therefore, in situations where INSERT, UPDATE, and DELETE commands are used extensively, defining a large number of keys will substantially impact performance. Each time a write command is performed, the indices storing the key values must be updated. In these tables, you may want to specify only a primary key.

Query performance is dependent on how keys are defined and used to access data. Columns used in a WHERE clause should make use of primary keys. Columns used in an ORDER BY and GROUP BY/HAVING clause should utilize secondary keys. A column rarely used for accessing data should never be a part of a key.

Normal Forms (Logical Model Design)

Normalization theory is built around the concept of normal forms. In 1972, Dr. E. F. Codd presented three normal forms. Later, Boyce and Codd jointly proposed the Boyce-Codd form as a special case of the third normal form. Two additional normal forms proposed later, fourth and fifth normal form, are based upon multi-value and join dependencies. This paper focuses on the first three types of normal forms.

A database is said to be in a particular normal form if it satisfies a certain specified set of constraints. Constraints are rules that define a set of values a column may be assigned. An increasing level of normal-form conformance implies adherence to rules of the previous level. Thus, relational models meeting the third level of conformance must also satisfy the second and first levels.

Data normalization rules refer to attribute properties. A non-key attribute is any column that does not participate in the primary key of the relation. In other words, non-key attributes in each relationship must provide a fact about the entity or relationship that is being identified by the key. An example is a student phone number in a student information table. The student's phone number is a member, or entity, of a student's information but is not a key or part of a key.

Another term used is mutual independence. Two or more attributes are mutually independent if none of the attributes concerned is functionally dependent on any of the others. We consider the primary keys of the student information and course information tables to be mutually independent.

The Campus Database below is our non-normalized example table. It is a single large table containing information on students, counselors, instructors, and courses. As we discuss each normal form, we will also transform Campus Database from a non-normalized flat-file to a database in third-normal form. What starts out as a single large table, consisting of many columns and a redundancy of data, becomes a collection of related tables with few columns, eliminating duplicate data. Our Campus Database begins with following structure and sample data.



In the examples below, **PRIMARY KEY COLUMNS** are all caps, **Secondary Keys** have initial caps only, and *foreign keys* are in italics. Our sample database represents only a small subset of an actual campus database. The actual data base would also include addresses, phone numbers, and other types of information.

Campus Database					
TYPE ID	Name	Abbr	Class	Instructor	Counselor
STUDENT	Cheryl Carson	C. Carson	NULL	NULL	E. Wells
STUDENT	Dirk Stringer	D. Stringer	NULL	NULL	H. McBaden
STUDENT	Jane Brown	J. Brown	NULL	NULL	E. Dixon
STUDENT	Charley Locksley	C. Locksley	NULL	NULL	E. Wells
INSTRUCTOR	Patrick Crever	P. Crever	NULL	NULL	NULL
INSTRUCTOR	Sheri Pantely	S. Pantely	NULL	NULL	NULL
INSTRUCTOR	Alfred Ringer	A. Ringer	NULL	NULL	NULL
INSTRUCTOR	Fred Taylor	F. Tayler	NULL	NULL	NULL
COUNSELOR	Ernest Wells	E. Wells	NULL	NULL	NULL
COUNSELOR	Henrietta McBaden	H. McBaden	NULL	NULL	NULL
COUNSELOR	Elouise Dixon	E. Dixon	NULL	NULL	NULL
COURSE	C++ Programming	NULL	CIS32	P. Crever	NULL
COURSE	Network Technology	NULL	CIS63	A. Ringer	NULL
COURSE	Intro to Computers	NULL	CIS2	S. Pantely	NULL
COURSE	TCP/IP Protocol	NULL	CIS64	S. Pantely	NULL
COURSE	Data Structures	NULL	CIS40	staff	NULL
ROSTER	Cheryl Carson	NULL	CIS32	NULL	NULL
ROSTER	Cheryl Carson	NULL	CIS63	NULL	NULL
ROSTER	Dirk Stringer	NULL	CIS2	NULL	NULL
ROSTER	Jane Brown	NULL	CIS64	NULL	NULL
ROSTER	Jane Brown	NULL	CIS40	NULL	NULL
ROSTER	Jane Brown	NULL	CIS2	NULL	NULL
ROSTER	Dirk Stringer	NULL	CIS32	NULL	NULL

Potential problems (or storage anomalies) in this non-normalized table include the following:

- ▶ Simple data entry errors are propagated through the database and are difficult and time-consuming to fix.
- ▶ Creating common reports, such as a student roster or list of a counselor's students, require sequential access of the database multiple times.
- ▶ Name changes for students, counselors and instructors are very difficult to implement, since more than one person may have identical first and last names.
- ▶ Duplicate information is spread throughout the database.
- ▶ Because many columns are empty (NULL), there is substantial wasted storage space.

First Normal Form

First normal form, or 1NF, deals with the basic structure of a relation, splitting up the data to remove repeating groups. A database is said to be in 1NF if and only if it satisfies the constraint that it contains atomic values only. Atomic value refers to the smallest part of an element, and the element, in our case, is a *logical group of data* within a database.

Breaking up the Campus Database into atomic elements requires determining a single identity in which to group the data. An obvious candidate in our example is the TYPE ID column containing the following domain of acceptable values: STUDENT, INSTRUCTOR, COUNSELOR, COURSE, and ROSTER. To satisfy the 1NF rule, Campus Database is transformed from a single table into five tables determined by the TYPE ID. Now each table contains only atomic values of a logical group of data.

Student Table (1NF)		
NAME	Abbr.	Counselor Abbr.
Cheryl Carson	C. Carson	E. Wells
Dirk Stringer	D. Stringer	H. McBaden
Jane Brown	J. Brown	E. Dixon
Charley Locksley	C. Locksley	E. Wells

Instructor Table (1NF)	
NAME	Abbr.
Patrick Crever	P. Crever
Sheri Pantely	S. Pantely
Alfred Ringer	A. Ringer
Fred Taylor	F. Tayler

Counselor Table (1NF)	
NAME	Abbr.
Ernest Wells	E. Wells
Henrietta McBaden	H. McBaden
Elouise Dixon	E. Dixon

Course Table (1NF)		
COURSE ID	Course Title	Instructor
CIS32	C++ Programming	P. Crever
CIS63	Network Technology	A. Ringer
CIS2	Intro to Computers	S. Pantely
CIS64	TCP/IP Protocol	S. Pantely
CIS40	Data Structures	staff

Roster Table (1NF)		
COURSE ID	STUDENT NAME	Instructor Name Abbr.
CIS32	Cheryl Carson	P. Crever
CIS63	Cheryl Carson	A. Ringer
CIS2	Dirk Stringer	S. Pantely
CIS64	Jane Brown	S. Pantely
CIS40	Jane Brown	staff
CIS2	Jane Brown	S. Pantely
CIS32	Dirk Stringer	P. Crever

Storage anomalies of the 1NF tables include the following:

- ▶ Creating common reports, such as a student roster or a list of each counselor's students, requires sequential access of the database multiple times.
- ▶ Name changes for students, counselors and instructors are very difficult to implement, since more than one person may have identical first and last names.
- ▶ Duplicate information is spread throughout the database.
- ▶ Searches make extensive use of primary keys that are based on character values, requiring string comparison routines for find matches. Numeric keys are far more efficient. Where two people have the same first and last name, additional difficulty is introduced.

But 1NF does begin the process of normalization. Looking over the structure of each table, we are able to determine candidate keys or a lack thereof. An obvious candidate key is the Course ID in the course table. Student, Instructor, Counselor Tables do not have any candidate keys; therefore, a column must be added to each table to satisfy the rule that primary keys must be unique. We can also determine the natural relationship between the tables—counselor-to-student, instructor-to-course, etc. This knowledge is used to transform the Campus Database from 1NF to second normal form.

Second Normal Form

Applying the rules of first normal form splits our Campus Database into individual tables, but fails to address the data redundancy issue. The rules of second normal form (2NF) address this problem.

A database is in 2NF if every non-key attribute is fully dependent on the primary key. The concept of 2NF requires that all attributes not part of a primary key be fully dependent on each primary key. If there are columns that can be identified by only part of the primary key, they need to be moved to their own table.

Since Campus Database in first normal form does not have any keys with multiple columns, the fundamental rule of 2NF is satisfied. To address the most obvious redundancy issues, we will specify foreign keys. Foreign keys allow us to use the primary key access information that is not stored that table. Using foreign keys eliminates duplicate information and provides an ability to update information directly related to the primary key in one place.

Transforming our Campus Database to 2NF requires a number of changes. Added to each of the Student, Instructor, and Counselor Tables is an ID column. The ID column contains a unique identifier assigned when the record is initially created. An ID column is used as a primary key in its own table and as a foreign key for formally defining the relationship between the tables.

Adding an ID to the Student Table changes the Roster Table's primary key. In the first normal form, the Roster Table's primary key is comprised of Course ID plus Student Name. As mentioned above, using a Student Name does not allow two students to attend the campus with identical first and last names. Combining the Course ID and Student ID to form a primary key eliminates this problem. This combination of columns to form the primary key also prevents students from enrolling in the same class twice since the primary key components (Course ID + Student ID) already exist. Moreover, a numeric key is far more efficient for the database engine than an alpha-numeric key when performing a search, and it removes the possibility of a data entry error caused by incorrectly typing a student's name.

Student Table (2NF)			
STUDENT ID	Name	Abbr.	<i>Counselor ID</i>
99000123	Cheryl Carson	C. Carson	110343
99000435	Dirk Stringer	D. Stringer	116545
99001647	Jane Brown	J. Brown	112610
99000987	Charley Locksley	C. Locksley	110343

Instructor Table (2NF)		
INSTRUCTOR ID	Name	Abbr.
100734	Patrick Crever	P. Crever
102265	Alfred Ringer	A. Ringer
105410	Sheri Pantely	S. Pantely
104874	Fred Taylor	F. Taylor

Counselor Table (2NF)		
COUNSELOR ID	Name	Abbr.
110343	Ernest Wells	E. Wells
116545	Henrietta McBaden	H. McBaden
112610	Elouise Dixon	E. Dixon

Course Table (2NF)		
COURSE	Course Title	<i>Instructor ID</i>
CIS32	C++ Programming	100734
CIS63	Networking Tech.	102265
CIS2	Intro to Computers	105410
CIS64	TCP/IP Protocol	104874
CIS40	Data Structures	NULL

Roster Table (2NF)		
<i>COURSE ID</i>	<i>STUDENT ID</i>	<i>Instructor ID</i>
CIS32	99000123	100734
CIS63	99000123	102265
CIS2	99000435	105410
CIS64	99001647	104874
CIS40	99001647	NULL
CIS2	99002647	105410
CIS32	99000435	100734

Third Normal Form

Our Campus Database in second normal form fully addresses the redundancy of data—with one exception. The Roster Table contains a column unrelated to the table's primary key. Applying the rules of 3NF addresses this issue.

A database is in 3NF when all the columns in the table contain data about the entity that is defined by the primary key. The columns in the table must contain data about only one thing. In other words, no attributes depend on other non-key attributes, thus eliminating sequential database reads to locate one or more rows.

Only a slight modification of the second normal-form of the Campus Database is required. Removing the Instructor ID column from the Roster Table satisfies the rules of 3NF.

Student Table (3NF)			
STUDENT ID	Name	Abbr.	<i>Counselor ID</i>
99000123	Cheryl Carson	C. Carson	110343
99000435	Dirk Stringer	D. Stringer	116545
99001647	Jane Brown	J. Brown	112610
99000987	Charley Locksley	C. Locksley	110343

Instructor Table (3NF)		
INSTRUCTOR ID	Name	Abbr.
100734	Patrick Crever	P. Crever
102265	Alfred Ringer	A. Ringer
105410	Sheri Pantely	S. Pantely
104874	Fred Taylor	F. Taylor

Counselor Table (3NF)		
COUNSELOR ID	Name	Abbr.
110343	Ernest Wells	E. Wells
116545	Henrietta McBaden	H. McBaden
112610	Elouise Dixon	E. Dixon

Course Table (3NF)		
COURSE	<i>Instructor ID</i>	Course Title
CIS32	100734	C++ Programming
CIS63	102265	Networking Tech.
CIS2	105410	Intro to Computers
CIS64	105410	TCP/IP Protocol
CIS40	NULL	Data Structures

Roster Table (3NF)	
<i>COURSE ID</i>	<i>STUDENT ID</i>
CIS32	99000123
CIS63	99000123
CIS2	9900435
CIS64	99001647
CIS40	99001647

Transforming the Campus Data into 3NF includes the following benefits:

- ▶ Changing a person's name is easy to perform and is immediately distributed throughout the database.
- ▶ Reports are generated by efficiently joining tables using primary, secondary and foreign keys rather than by scanning a non-normalized table many times sequentially.
- ▶ Redundancy of data has been eliminated.
- ▶ Data entry errors are not propagated within the database to other columns. When data entry errors occur, they can be easily fixed in a single place.

3NF finalizes our transformation of the Campus Database into the relational model. Roster Table is the main, top-level table, referred to as the parent. The Course ID foreign key points to the course information (a child of the Roster Table) and cascades (relates) further to the Instructor Table. Likewise, the Student ID foreign key points to the Student Table and cascades again to the Counselor Table.

Transforming the database from non-normalized to 3NF achieved a number of major goals: removing redundant information, reducing the amount of wasted space, and eliminating the need for sequential access of the database to generate common reports.

Denormalization

In most cases, the benefits of maintaining a high level of normalization outweigh the drawbacks. But in a few cases, *denormalization* may be justified to increase performance. Denormalization is a common technique used to improve performance of SQL commands by converting a logical database design from the third normal form or higher to some lower form of normalization.

One of the most common arguments for denormalization occurs when a table join is used extensively. In this case, query performance may increase when the contents of the joined tables are merged by design versus constantly performing the join using SQL commands.

Data Integrity

Once the logical model has been developed, our final task is to define rules to ensure that the data remains within a predefined range and foreign key relations do not point to non-existing data. For the data to have integrity, the data must remain accurate and consistent when shared by all database clients. Data may become corrupt in many ways, some of which include:

- ▶ Dates for an order are entered incorrectly.
- ▶ Data entry incorrectly enters a discount rate as 30% when it should be 10%, resulting in a customer receiving an invoice with a balance due lower than actual cost.
- ▶ A new hire is given an incorrect department number causing the new hires record to become “lost.” Not finding the record causes someone to create a new record. Now there are two records for one employee.

An unfortunate aspect of legacy databases is that the data is often corrupted (such as numeric fields that are blank or contain alphabetic characters, dates that contain all zeros, etc.) As you build your applications, you may want to develop a utility to verify the correctness of all columns in every row of the database. Unless steps are taken to ensure the data is correct, applications may not operate as expected and produce incorrect results.

The actual implementation of data integrity depends entirely on the data storage engine you choose and your application design requirements. Synergy DBMS is a high-powered data storage engine that reads and writes data to the database and maintains the integrity of the primary and secondary keys. However, the application that uses Synergy DBMS must provide custom code to handle constraints, foreign key deletions, and other data integrity issues. On the surface, this may appear to be a limitation but there are benefits. Since the database engine is merely reading and writing data without incurring the overhead of data validation, Synergy DBMS operations are much faster than relational database server operations.

In contrast, a relational database engine enforces data integrity within the database itself, guaranteeing that data integrity rules are followed. Because all rules are stored with the data and the single control point for data consistency is on the server, the need to program these rules in every application is eliminated. In an enterprise environment, data integrity rules may be replicated from a main corporate server to remote offices automatically.

In some instances, application-based data integrity may also be appropriate for relational databases. This commonly occurs when the available mechanism for implementing and maintaining constants, foreign keys, stored procedures, and triggers are unavailable, inadequate, or far too complicated to use.

There are four types of data integrity: Entity Integrity, Domain Integrity, Referential Integrity, and User-defined Integrity. As mentioned above, from a Synergy DBMS point-of-view, data integrity is an application-driven issue. It is the responsibility of an application to perform all necessary data integrity checks.

Entity Integrity

Entity integrity defines each row as unique. The primary key satisfies the entity integrity rule by allowing only one primary key attribute per table. Entity integrity is a by-product of database normalization. When possible, entity integrity should be enforced at the lowest level using primary keys and unique constraint rules.

Referential Integrity

Referential integrity means that the foreign key in any referencing table must always refer to a valid row in the referenced table. Referential integrity ensures that the relationship between two tables remain synchronized during updates and deletes.

Referential integrity preserves the relationships between tables when rows are added and deleted. It preserves the relations between tables by ensuring primary and foreign keys remain consistent across tables. At no time are rows orphaned (foreign keys pointing to nonexistent data). When a database engine is not designed to manage referential integrity, as is the case with Synergy DBMS, that responsibility falls on the application. Insertion of a new child row must ensure that the relationship to a parent exists. Likewise, deletion of a parent row must include deletion of all associated children.

Tables may be related in three ways:

- ▶ One-to-one relationship
- ▶ One-to-many relationship
- ▶ Many-to-many relationship

One-to-one relationship

A relationship between two tables, in which a single row in the first table can be related only to one row in the second table, and a row in the second table can be related only to one row in the first table. This type of relationship is not commonly used.

One-to-many relationship

A relationship between two tables in which a single row in the first table can be related to one or more rows in the second table, but a row in the second table can be related only to one row in the first table. An example of a one-to-many relationship is the relation between instructors and courses. One instructor may be assigned to teach more than one course, but a course can only be assigned one instructor. An instructor's unique ID (primary key) is stored as a foreign key in the course information table, thus defining the relation between the two.

Many-to-many relationship

A relationship between two tables in which rows in each table have multiple matching rows in the related table. Many-to-many relationships are maintained by using a third table called a junction table. Our class roster table is an example of this type of table. Each entry in the roster table cascades in two directions. The left side points to the course information, which in turn points to the instructor's information. The right side points to the student, which in turn points to a counselor.

Domain Integrity

Domain integrity defines acceptable values for each column. In its most basic form, domain integrity does not allow numeric or date values to be written to character columns. A more advanced level controls the values stored in a column. Some examples include: specifying an acceptable range of dates for a date column or only allowing valid product codes to be stored in a product code column.

There are two primary mechanisms for enforcing business rules and data integrity: triggers and constraints. The primary benefit of triggers is that they can contain complex processing logic that uses SQL code. Therefore, triggers can support all of the functionality of constraints; however, triggers are not always the best method for a given feature. Triggers and stored procedures are discussed in more detail later in this section.

Constraints may be applied at both a table and a column level during table creation or at a later time by altering table and column properties. There are five types of constraints:

- ▶ NOT NULL Constraint
- ▶ CHECK Constraint
- ▶ UNIQUE Constraint
- ▶ PRIMARY KEY Constraint
- ▶ FOREIGN KEY Constraint
- ▶ Default values

NOT NULL constraint

The NOT NULL constraint is a rule to specify whether or not null values are allowed in the column. If NOT NULL is not specified, the column can contain null values, and its default value is either the null value or a value provided by the WITH DEFAULT clause. Using the WITH DEFAULT clause in conjunction with NOT NULL ensures that a column value remains in an acceptable range.

CHECK constraint

The CHECK constraint is a condition that must be met before data can be assigned to a column during an INSERT or UPDATE statement. In the most basic form, a column's data type determines the kind of information that can be stored. More complex rules can be defined to control the range of the values. Numeric data types can be assigned a range of values. For character data types, a set of character values can be defined as acceptable types of information. A CHECK constraint cannot perform a comparison with another table. This

is the function of triggers and stored procedures. Postal code verification, for example, is a prime candidate for implementing a stored procedure or trigger to look up a given postal code in a postal code table.

A column can have more than one CHECK constraint. The database applies the CHECK constraints as defined. It is up to the designer and database manager to ensure that no inconsistencies, duplicates, or equivalent conditions are defined.

UNIQUE constraint

The UNIQUE constraint provides entity integrity for a given column or columns through a unique index. The identified columns must be defined as NOT NULL. Each column name must identify a column of the table and the same column must not be identified more than once. A table can have multiple UNIQUE constraints.

PRIMARY KEY constraint

The PRIMARY KEY constraint enforces entity integrity for a given column or columns through a unique index. The clause must not be specified more than once and the identified columns must be defined as NOT NULL. Each column name must identify a column of the table and the same column must not be identified more than once. Only one PRIMARY KEY constraint can be created per table.

FOREIGN KEY constraint

The FOREIGN KEY constraint provides referential integrity for the data in a column or columns. FOREIGN KEY constraints require that each value in the column exist in the corresponding referenced columns in the referenced table. FOREIGN KEY constraints can reference only columns that are PRIMARY KEY or UNIQUE constraints in the referenced table.

WITH / DEFAULT values

WITH / DEFAULT specifies the value provided for a column when a value is not explicitly supplied during an insert operation. DEFAULT definitions can be applied to any columns except those with the IDENTITY property. Only a constant value—such as a character string, a system function, or NULL—can be used as a default. Omission of DEFAULT from a column definition results in the use of the null value as the default for the column.

User-defined Integrity

User-defined integrity refers to business rules that do not fall within any of the other data integrity types.

Developers can define their own data types to supplement the system-supplied data types. For example, state_type can be defined as a two-character data type to store the legal abbreviations for each of the 50 states in the United States of America. The advantages of user-defined data types is that definitions of NULL values, defaults, and rules can be shared by all table definitions or tailored as needed.

Stored Procedures and Triggers

Stored procedures are predefined commands stored in a database, pre-compiled and ready to execute. The database manages stored procedures as database objects. Stored procedures are run as stand-alone programs on the database server via internal or external requests. The request may originate within the database system by other stored procedures, triggers or agents. External requests may originate from remote client applications, local server applications, or other database systems.

Database servers centrally enforce complex business rules with a special type of stored procedure called a trigger. The main difference between stored procedures and triggers is the method in which it's invoked. Stored procedures must be explicitly called to be executed, whereas triggers are automatically invoked, or triggered, by the database server attempting to insert, delete, or update a data field. Triggers are managed by the relational database server to directly monitor changes in the data, providing a data-driven integrity implementation as opposed to application-driven integrity.

Triggers prevent incorrect, unauthorized, or inconsistent changes to data. Triggers accomplish this by performing any number of actions whenever a specified change to a data object is attempted. Triggers can be nested: for example, a trigger can cascade a change throughout related tables in a database, roll back transactions, and send messages to a user.

Triggers run automatically as changes are made to the database, regardless of who requested the change. They are most useful when the constraints that a database supports do not meet the needs of the application. Two of the most important reasons for using triggers include the issue of FOREIGN KEY constraints not supporting cascading activities (in other words, the database engine does not support referential integrity automatically), and column value requiring validation against a column in another table before allowing an insert to occur.

Our Campus Database can make use of stored procedures and triggers in a couple of ways. A stored procedure can be used to generate a unique Student, Instructor, and Counselor ID. A trigger can be assigned to the abbreviation column in the Student, Instructor and Counselor tables. When a new entry is added to these tables, the trigger uses the name column to generate an abbreviated form. A trigger can be created to delete all related child rows when a student drops out of college. When a student is deleted from the Student Table, all related rows in the Roster Table are deleted as well, ensuring that referential integrity is enforced automatically.

Conclusion

Applying normalization and data integrity rules can greatly reduce application development cost because server-enforced business rules are stored in a central location and shared by all applications. Although data normalization is not a panacea for database design, a properly designed database reduces data redundancies and helps to eliminate the data anomalies resulting from those redundancies. Database reliability can be increased because data integrity is maintained independently from application programs. Database security also increases because rules, as well as database tables, are protected from unauthorized access.

Another windfall of good database design is the ability to change business rules without impacting applications. In the fast moving business application market, the capacity to adapt and evolve can increase a product's opportunity to become, and remain, an industry leader. Solid database design can play an important role.