Jose's Database Programming Corner

Fundamentals of Database Design

This page is an extension of my Introduction to the Relational Database page. We'll investigate the relational model in a little more detail, examining forms of normalization, types of data integrity, and indexing. You may also wish to visit some of the following pages:

- Intro to Jet and Access
  Basics of the Jet engine and using Access as a programming tool.
- Using Jet Data Access Objects
  An introduction to DAO

Here's what will be covered on this page:

- Introduction
- Database Normalization
- Data Integrity
- Indexing
- Review

Return to Top of Page

Introduction

On the Relational Database Model page, I introduced some of the concepts behind the relational database model. Here we will take a closer look at database normalization, examining both why a database should be normalized and how to do it. We'll then move on to a discussion of data integrity and how to use the database engine to help prevent errors in the data. In the section on indexing, we'll take a look at some different types of indexes and what you should and should not index. Finally, we'll review these topics in the summary.

Building a database structure is a process of examining the data which is useful and necessary for an application, then breaking it down into a relatively simple row and column format. There are two points to understand about tables and columns that are the essence of any database:

- Tables store data about an entity
  An entity may be a person, a part in a machine, a book, or any other tangible or intangible object, but the primary consideration is that a table only contain data about *one* thing.
- Columns contain the attributes of the entity
  Just as a table will contain data about a single entity, each column should only contain one item of data about that entity. If, for example, you're creating a table of addresses, there's no point in having a single column contain the city, state, and postal code when it is just as easy to create three columns and record each attribute separately.

One method that I find helpful when initially breaking down the data for tables and columns is to examine the names that are used. I use a plural form of a noun for table names (Authors, Books, etc.) for table names, and a noun or noun and adjective for column names (FirstName, City, etc.). If I find that I'm coming up with names that require the use of the word "and" or the use of two nouns, it's an indication that I haven't gone far enough in breaking down the data. An exception to this would be a table that serves as middle of a many to many relationship, such as BookAuthors, etc.

Before continuing on to discuss normalization, we'll look at some of the common flaws in database designs and the problems they cause. To illustrate these problems, I'll use the following sample table, which I'll call simply "Ugly":

StudentName	AdvisorName	CourseID1	CourseDescription1	CourseInstructorName1	CourseID2	CourseDescription2	CourseInstructorName2
Al Gore	Bill Clinton	VB1	Intro to Visual Basic	Bruce McKinney	DAO1	Intro to DAO Programming	Joe Garrick
Dan Quayle	George Bush	DAO1	Intro to DAO Programming	Joe Garrick	VBSQL1	Client/Server Programming with VBSQL	William Vaughn
George Bush	Ronald Reagan	API1	API Programming with VB	Dan Appleman	OOP1	Object Oriented Programming in VB	Deborah Kurata
Walter Mondale	Jimmy Carter	VB1	Intro to Visual Basic	Bruce McKinney	API1	API Programming with VB	Dan Appleman

Table 1: Ugly

Let's look at some of the problems with this structure:

- Repeating Groups
  The course ID, description, and instructor are repeated for each class. If a student needs a third class, you need to go back and modify the table design in order to record it. While you could add CourseID3, CourseID4, CourseID5, etc., along with the associated description and instructor fields, no matter how far you take it there may one day be someone who wants one more class. Additionally, adding all those fields when most students would never use them is a waste of storage.
- Inconsistent Data
  Let's say that after entering these rows, you discover that Bruce McKinney's course is actually title "Intro to Advanced Visual Basic". In order to reflect this change, you would need to examine all the rows and change each individually. This introduces the potential for errors if one of the changes is omitted or

done incorrectly.

- Delete Anomalies
  If you no longer wished to track Joe Garrick's Intro to DAO class, you would need to delete two students, two advisors, and one additional instructor in order to do it. If you remove the first two rows of the table, all of the data is deleted with the reference to the course.
- Insert Anomalies
  Perhaps the department head wishes to add a new class - let's call it "Advanced DAO Programming" - but hasn't yet set up a schedule or even an instructor. What would you enter for the student, advisor, and instructor names?

As you can see, this single flat table has introduced a number of problems - all of which can be solved by normalizing the table design.

> *Don't be misled into thinking that normalization is the answer to all your problems in developing a database design. As you will see later, there may times when it's prudent to denormalize a structure. There's a variety of other problems that can be introduced into your data as well as an infinite variety of complex business rules which may need to be applied.*

Return to Top of Page

Database Normalization

Normalization is a process that is too often described using complex theory and jargon. I'm going to (hopefully) try to provide a plain English explanation of what it is, the various forms of normalization, and how to normalize an unnormalized database.

- What is Normalization
- Forms of Normalization
- First Normal Form
- Second Normal Form
- Third Normal Form

What is Normalization?

Normalization is essentially the process of taking a wide table with lots of columns but few rows and redesigning it as several narrow tables with fewer columns but more rows. A properly normalized design allows you to use storage space efficiently, eliminate redundant data, reduce or eliminate inconsistent data, and ease the data maintenence burden. Before looking at the forms of normalization, you need to know one cardinal rule for normalizing a database:

> You must be able to reconstruct the original flat view of the data.

If you violate this rule, you will have defeated the purpose of normalizing the design.

Forms of Normalization

Relational database theorists have divided normalization into several rules called *normal forms*.

- First Normal Form

No repeating groups.
- Second Normal Form
  No nonkey attributes depend on a portion of the primary key.
- Third Normal Form
  No attributes depend on other nonkey attributes.

Additionally, for a database to be in second normal form, it must also be in first normal form, and for a database to be in third normal form, it must meet the requirements for both first and second normal forms. There are also additional forms of normalization, but these are rarely applied. In fact, it may at times be practical to violate even the first three forms of normalization.

Let's take a closer look at each form of normalization and try to explain it in plain English.

First Normal Form

No repeating groups.

> *What we're looking for is repeating groups of columns. The purpose is to reduce the width of the table. This is done by taking the groups of columns and making a new table where the table is defined using the columns that repeat. Rather than having additional columns, the resulting table has more rows.*

OK, so what's a repeating group? Let's look at the columns in our sample table *Ugly*:

- StudentName
- AdvisorName
- CourseID1
- CourseDescription1
- CourseInstructorName1
- CourseID2
- CourseDescription2
- CourseInstructorName2

In the example, columns for course information have been duplicated to allow the student to take two courses. The problem occurs when the student wants to take three course or more. While you could go ahead and add CourseID3, etc., to the table, the proper solution is to remove the repeating group of columns to another table.

> *If you have a set of columns in a table with field names that end in numbers xx1, xx2, xx3, etc., it's a clear warning signal that you have repeating groups in the table. A common exception to this would be a table of street addresses, where you might have AddressLine1, AddressLine2, etc., rather than using a single field for multiple line addresses.*

Let's revisit the design to handle the repeating groups:

Students	**StudentCourses**
StudentID StudentName AdvisorName	*SCStudentID* *SCCourseID* SCCourseDescription

	SCCourseInstructorName

Table 2: First Normal Form
The primary keys are shown in italics.

We've divided the tables so that the student can now take as many courses as he wants by removing the course information from the original table and creating two tables: one for the student information and one for the course list. The repeating group of columns in the original table is gone, but we can still reconstruct the original table using the *StudentID* and *SCStudentID* columns from the two new tables. The new field *SCStudentID* is a foreign key to the *Students* table.

Second Normal Form

No nonkey attributes depend on a portion of the primary key.

> *Second Normal Form really only applies to tables where the primary key is defined by two or more columns. The essense is that if there are columns which can be identified by only part of the primary key, they need to be in their own table.*

Let's look at the sample tables for an example. In the *StudentCourses* table, the primary key is the combination of *SCStudentID* and *SCCourseID*. However, the table also contains the *SCCourseDescription* and the *SCCourseInstructorName* columns. These columns are only dependent on the *SCCourseID* column. In other words, the description and instructor's name will be the same regardless of the student. How do we resolve this problem? Let's revisit the sample tables.

Students	**StudentCourses**	**Courses**
StudentID StudentName AdvisorName	*SCStudentID* *SCCourseID*	*CourseID* CourseDescription CourseInstructorName

Table 3: Second Normal Form

What we've done is to remove the details of the course information to their own table *Courses*. The relationship between students and courses has at last revealed itself to be a many-to-many relationship. Each student can take many courses and each course can have many students. The *StudentCourses* table now contains only the two foreign keys to *Students* and *Courses*.

We're almost done normalizing this small sample, but before taking the last step, lets add a little more detail to the sample tables to make them look something more like the real world.

Students	**StudentCourses**	**Courses**
StudentID StudentName StudentPhone StudentAddress StudentCity StudentState StudentZIP AdvisorName	*SCStudentID* *SCCourseID*	*CourseID* CourseDescription CourseInstructorName CourseInstructorPhone

AdvisorPhone		

Table 4: Detail Columns Added

Third Normal Form

No attributes depend on other nonkey attributes.

> *This means that 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. This is really an extension of Second Normal Form - both are used to remove columns that belong in their own table.*

To complete the normalization, we need to look for columns that are not dependent on the primary key of the table. In the *Students* table, we have two data items about the student's advisor: the name and phone number. The balance of the data pertains only to the student and so is appropriate in the *Students* table. The advisor information, however, is not dependent on the student. If the student leaves the school, the advisor and the advisor's phone number will remain the same. The same logic applies to the instructor information in the *Courses* table. The data for the instructor is not dependent on the primary key *CourseID* since the instructor will be unaffected if the course is dropped from the curriculum.

Let's complete the normalization for the sample tables.

Students	**Advisors**	**Instructors**	**StudentCourses**	**Courses**
StudentID StudentName StudentPhone StudentAddress StudentCity StudentState StudentZIP StudentAdvisorID	*AdvisorID* AdvisorName AdvisorPhone	*InstructorID* InstructorName InstructorPhone	*SCStudentID* *SCCourseID*	*CourseID* CourseDescription CourseInstructorID

Table 5: Third Normal Form

There's one other potential modification that could be made to this design. If you look at the *Advisors* and *Instructors* tables, you can see that the columns are essentially the same: a name and phone number. These two tables could be combined into a single common table called *Staff* or *Faculty*. The advantage of this approach is that it can make the design simpler by using one less table. The disadvantage is that you may need to record different additional details for instructors than you would record for advisors. One possible way of resolving this conflict would be to go one further step and create a *Staff* table that records basic information that any organization would retain about an employee, such as name, address, phone, Social Security number, date of birth, etc. Then you would have the *Advisors* and *Instructors* tables contain foreign keys to the *Staff* table for appropriate individual along with any additional details that might need to be stored specifically for that particular role. The benefit of this approach is that if a member of the staff is both an advisor and an instructor (this would often be the case in a college or university), the basic information would not need to be duplicated for both roles.

> *Don't go overboard with Third Normal Form or you'll wreak havoc on performance. If you look at the Students table, you can see that in fact any city and state in the U.S. could be identified by the ZIP code. However, it may not be practical to design the database so that every time you need to get an address you have to join the row from Students to a table containing the (approximately) 65,000 ZIP codes in the U.S. A general guideline is that if you routinely run queries which join more than four tables, you may need to consider denormalizing the design.*

Return to Top of Page

Data Integrity

In this section, we'll discuss types of data integrity, the methods available to protect data, and look at a few scenarios where you might apply data integrity rules.

- Introduction
- Entity Integrity
- Domain Integrity
- Referential Integrity
- User-defined Integrity
- Summary

Introduction

As a database developer, protecting the data in the database is one of your most important responsibilities, perhaps *the* most important. The current generation of database engines, including Jet, provide a powerful array of tools to assist you in making sure the data in your database is accurate and consistent. Although no amount of programming can prevent every type of error that could be introduced, you should use the tools available to you to do whatever you can to guarantee the validity of the data.

The types of data integrity can be broken down into four basic areas:

- Entity Integrity
  No duplicate rows.
- Domain Integrity
  The values in any given column fall within an accepted range.
- Referential Integrity
  Foreign key values point to valid rows in the referenced table.
- User-defined Integrity
  The data complies with applicable business rules.

Let's look at each type in a little more detail.

Entity Integrity

No duplicate rows.

This simply means that in any given table, every row is unique. In a properly normalized, relational database, it's of particular importance to avoid duplicate rows in a table because users will expect that when a row is updated, there are no other rows that contain the same data. If there are duplicate rows, a user may update one of several duplicates and expect that the data has been updated for all instances, when in fact there are duplicates elsewhere that have not been updated. This will lead to inconsistencies in the data.

Entity integrity is normally enforced through the use of a primary key or unique index. However, it may at times be possible to have a unique primary key for a table and still have duplicate data. For example, if you have a table of name and address information, one user may enter a row for the name "Robert Smith" and another user enter a row for the same individual but enter the name as "Bob Smith". No form of primary key or unique index would be able to trap this type of violation. The only solution here is to provide the user a means of performing a search for existing data before new rows are created. (Don't be fooled into thinking that you can get away with putting a unique index on a combination of first, middle, and last names to avoid duplicates or your design will collapse when you need to enter rows for two people named John David Smith.) Another method of finding this type of situation before the data is entered is by using a "soft" search algorithm, such as a Soundex search.

> *Soundex is an algorithm that will produce a code which describes a phonetic equivalent of a word. Soundex codes are widely used with name searches because they will detect matches even if names are misspelled. Microsoft SQL Server comes with a built-in function to determine the Soundex code for a word. You can also build your own if the database engine you are using does not provide one. The algorith is publicly available almost everywhere. A search of the Web for the keyword "Soundex" should produce the algorith within the first few matches.*

Domain Integrity

The values in any given column fall within an accepted range.

It's important to make sure that the data entered into a table is not only correct, but appropriate for the columns it's entered into. The validity of a domain may be as broad as specifying only a data type (text, numeric, etc.) or as narrow as specifying just a few available values. Different database engines will have varying means of enforcing the validity of the domain of a column, but most will permit the entry of at least a simple Boolean expression that must be True for the value in a column to be accepted. (Jet calls this a "Validation Rule".) If the range of acceptable values is large or changes frequently, you can also use a lookup table of valid entries and define a referential integrity constraint between the column and the lookup table. Although this approach is more complex to implement, it allows you to modify the domain without changing the table design. (Changing the table design often requires that you have exclusive access to the table - something which can be nearly impossible with a production database in use by even a modest number of users.)

In addition to single column rules, the domain of an entry may be dependent on two or more columns in a table. For example, the range of valid entries for a sub-category column may be dependent on the value entered in an associated category column. Unfortunately, it can be difficult or impossible to express this type of rule as a simple Boolean expression. If you're fortunate enough to be working with a database engine which provides the capability of writing insert, update and delete triggers you can code complex logic to handle this type of validation.

> *Unfortunately, Jet does not provide the capability of writing trigger code for data events. If you need to force some code to run when a row is inserted, updated, or deleted, you'll need to go through a few hoops to do it. One method that I've applied is to revoke all permissions except Read on the table, then write functions to insert, update, or delete rows which use either an owner access query or a priviledged workspace to change the data.*

Let's look at some examples of columns which could have domain integrity rules applied:

Column	Data Type	Domain
Social Security Number	Text	In the format xxx-xx-xxxx, where x is a number from 0 to 9. *I tend to use text rather than numeric columns for formatted data such as this even if the value contains only numbers. This not only avoids the possibility of performing mathematical operations on the data, but saves the trouble of formatting it each time you need to present it if you save the data with the formatting in the column.*
Grade Level	Text	Exists in the following list: Freshman, Sophmore, Junior, Senior.

		This type of list is probably stable enough that a "hardcoded" list of values is sufficient.
Denomination	Text	Exists in the following list: Catholic, Episcopalian, Lutheran, Methodist, etc. *This type of list is probably long enough and dynamic enough that it would be better handled using a lookup table of valid entries. To do this, you would need to create a table that holds a unique list of valid denominations and define a relationship between the lookup table and the Denomination column.*
Insurer	Text	If the Insured column is True, then Insurer is not null. *This type of multi-column validation typically can be defined at the table level with most database engines. With Jet, for example, the expression might be:* *If Insured Then Not IsNull(Insurer)* *Where Insured is a Boolean column which cannot be null.*
Salary	Numeric	Falls within the range defined by the job title. *Unless you have trigger capability, this can be difficult to implement. A workaround is to use a lookup table of valid salaries by job title and a multiple-field relationship between the table containing the salary and the lookup table.*

One thing to remember when working with domain integrity rules is that you not only need to understand the design of the database, you also need to understand the type of expected data and the business rules which apply to it. In many cases, you will have several choices available to you for the manner in which you implement a rule. My approach is normally to use the method which provides the most flexibility for future changes.

- Data Type
  Simple data type rules, such as "Value is numeric" should always be defined by specifying the appropriate data type for the column when you create it. Remember when working with numeric values to specify the appropriate type of number. In most cases, you will have a choice of whole number ranges (byte, integer, long integer), floating point (single and double precision), and financial. Be sure to use a financial data type (money, currency, etc.) when working with financial data. Most database engines use special data formats to provide greater consistency and accuracy than the equivalent floating point data type.
- Formatted Data
  For columns such as social security numbers, phone numbers, and other values which can be guaranteed to be in a fixed format, I use a formatting rule and store the data as text. In general, I prefer to use text data types for formatted data even if the values are all numerals. While you can use a numeric data type and perhaps save a few bytes of storage, you will need to deal with formatting issues each time you present the data. Additionally, by storing the data as text, you can allow the database engine (rather than your application) to enforce that the data is entered in the correct format.
  Note: If you are building an application for international use or which contains international data, be sure to provide flexibility in the rules to account for the locality. Values such as phone numbers, postal codes, etc., vary from country to country.
- Nullability
  This can nearly always be defined as a Boolean expression. An exception would be if the nullability of a column is defined by the value in another column. In this case, the possible range of values for the second column would determine if you can use a simple expression or would need to define a lookup table.
- Value List
  Unless I have a very high degree of certainty that the list will not change, I use a lookup table. Examples of lists which could be "hardcoded" would be items such as military rank, states or provinces within the U.S. and Canada, school grade level, etc. If there is a possibility that the list is or could easily become dynamic or excessively long, I prefer to use a lookup table and a referential integrity constraint. Keep in mind that data which appears stable when you begin to build a design could easily become dynamic if the business rules change.

Remember that whenever possible, you should let the database engine do the job of enforcing the integrity of a domain.

Referential Integrity

Foreign key values point to valid rows in the referenced table.

A foreign key is a value in one table that references, or points to, a related row in another table. It's absolutely imperative that referential integrity constraints be enforced. While it is possible (likely, in fact) that foreign key values may be null, they should never be invalid. If a foreign key is entered, it **must** reference a valid row in the related table. If you allow unenforced references to exist, you are inviting chaos in your data. Referential or relational integrity is really a special form of domain integrity. The domain of a foreign key value is all of the valid primary key values in the related table.

Depending on the database engine you are using, you may have several choices available for how and in what manner referential integrity constraints will be enforced. Many database engines (including Jet) allow you to use *Declarative Referential Integrity* rather than triggers or application code to enforce constraints. With this approach, you define a relationship between the columns in two tables and the database engine enforces the constraint for inserts, updates, and deletes.

Assuming a one-to-many relationship, here are the constraints which will be imposed:

- When a row is inserted on the many side, a foreign key that is entered (pointing to a row on the one side) must reference a valid row.
- If the foreign key is updated on the many side, the new value must point to a valid row on the many side.
- A row on the one side cannot be deleted if related rows exist on the many side.
- The primary key from the one side cannot be updated if related rows exist on the many side.

Some database engines also allow you to specify how to handle changes to the one side of the relationship:

- Cascading Updates
  If cascading updates are specified, a change to the primary key on the one side of a relationship will be propogated through related rows on the many side so that all foreign keys which point to the row on the one side are updated.
- Cascading Deletes
  If cascading deletes are specified, deleting a row on the one side will delete any related rows on the many side.

  Microsoft's Jet engine allows you to specifiy either cascading updates, deletes, or both.

Use caution when specifying cascading deletes. This can be a dangerous practice if you are using a lookup table to enforce a domain. Consider what would happen if you had a table of several thousand names and address related to a table of the 50 U.S. states (using the table of states to enforce that the state entered in the address is valid), then deleted California from the table of states. If cascading deletes were requested, you would delete not only the California row from the states table, but also delete all rows from the name and address table in the state of California.

Also note that if you are using some type of autonumbering column as a primary key, there is generally no point in specifying cascading updates, since this type of column typically can't be updated anyway.

This applies to both Jet Counter fields and MS SQL Server IDENTITY columns.

User-defined Integrity

The data complies with applicable business rules.

User-defined integrity is a kind of "catch-all" for rules which don't fit neatly into one of the other categories. Although all of the other types of data integrity can be used to enforce business rules, all business rules may not be able to be enforced using entity, domain, and referential integrity. The types of rules that apply will of course vary by application. If possible, you should take advantage of the database engine to apply whatever constraints it is capable of, but some may need to be enforced using application code.

> *A textbook example of a business rule which must be enforced is a funds transfer in a banking application. If a customer wishes to transfer funds from a savings account to a checking account, you must deduct the withdrawl from the savings account **and** add the deposit to the checking account. If you successfully record the withdrawl without recording the deposit, the customer will have "lost" the amount (and not be pleased). If you record the deposit without the withdrawl, the customer will get a "free" deposit (and the bank will be unhappy). To enforce this type of rule, you'll need to use a Transaction in your application code.*

Summary

As a database developer, you are ultimately responsible for ensuring the accuracy of the data in a database. In addition to normal entity, domain, and referential integrity constraints, you are also expected to have your application enforce appropriate business rules against the data and to protect the data as much as possible from data entry and other end-user errors.

Keep in mind that the users are normal, fallible, humans. While it is expected that people should make every effort to enter data correctly, the database design and application code should also be able to anticipate many errors. While some errors may be impossible to trap systematically (an incorrectly typed name or address), any business rules which can be defined should be enforced. Avoid the temptation to rely on vigilance by end-users to ensure accurate data.

Although application performance, features, and capabilities are important concerns, the application is worthless if the data cannot be counted on to be correct. Use the tools provided by your database engine to enforce rules where possible and your application code to enforce those rules that can't be enforced by the database engine.

Return to Top of Page

Indexing

In order to get acceptable performance from a database application, you not only need to write efficient code, you also need to provide the database engine with the tools it needs to do its job well. The most important step you can take in that direction is the proper use of indexes.

- Introduction
- Types of Indexes
- What to Index
- What Not to Index
- Summary

Introduction to Indexing

Like an index in a book, an index on a table provides a means to rapidly locate specific information - but a table index is used by the database engine rather than the user. The proper use of indexes can yield significant performance improvements in a database application because indexes can assist the database engine in avoiding costly disk i/o when searching a table for specific data. Most importantly, proper indexing can prevent the database engine from using the most costly of

database operations: the table scan.

Database engines often employ a form of a binary search algorith when retrieving rows from a table. If you're familiar with the binary search, you know that a binary search can isolate a specific item in a collection of thousands of items with only a handful of comparisons. In a database application, comparisons become seeks in a disk file (or a disk cache if you're lucky). File seeks are inherently expensive since even the fastest disks on the market today are orders of magnitude slower than RAM operations. A primary concern for improving performance is to minimize the number of disk reads when querying the database. Well chosen indexes are the key to reducing file i/o operations because they reduce the number of data pages which must be read to locate a row in a table. When searching for a row, the database engine can read a small number of pages in the index rather than reading every row in the table until it finds a match.

It's important to remember, however, that just as an index will improve read performance, it will degrade write performance. When a new row is written to a table, the database engine must not only write the row, it must also update any associated indexes. If you decided to maximize read performance by indexing every column in a table and creating a multiple column index for every possible search or sort, performance of inserts, updates, and deletes will come to a grinding halt.

Types of Indexes

Nearly every database engine available will support two fundamental types of indexes.

- Unique Indexes
  There are no duplicate entries in a unique index. Unique indexes are most often used for the primary key of a table.
- Non-unique Indexes
  Non-unique indexes may have duplicate values and are used anywhere than an index will provide a performance improvement in the application.

  Although it is the most common use, a unique index does not necessarily need to be the primary key of a table. A technique I've used to simplify query designs is to specify a unique index on a combination of columns that could serve as a primary key, then add an autonumbering column to the table and specify that column as the primary key. This makes queries simpler because only one column needs to be joined. Additionally, it can help eliminate redundancy since only one value would need to be stored as a foreign key to the table. I've never analyzed the performance impact of adding an extra column for this purpose, but in some situations, a performance cost (if one in fact exists) is worth the price in the simplicity of the design.

Some database engines can also create clustered or non-clustered indexes.

- Clustered Index
  In a clustered index, the actual rows in the table reside on the leaf pages of the index.
- Non-clustered Index
  In a non-clustered index, the leaf pages of the index are pointers to the data pages containing the rows in the table.

Clustered indexes can offer significant performance advantages over non-clustered indexes if your database engine supports them (many - Jet for example - do not). Since the actual table data resides at the lowest level of the index, there is always one less seek involved in the file. However, keep in mind that there can only be one clustered index per table, so if a clustered index is to be used, it must be chosen carefully. Additionally, dropping or changing a clustered index is a costly operation because the data in the table must be entirely rewritten.

Another specification that is sometimes available is ascending or descending indexes. This is as simple as it sounds. In an ascending index, the index is written in ascending order. In a descending index, its written in descending order. Not all database engines can create descending indexes (SQL Server doesn't), but most can use an ascending index to speed up a descending sort.

One more consideration in creating an index is the nullability of the index. This may be driven by the index definition or simply inherited from the definitions of the

underlying columns. It should be noted, however, that the treatment of indexes which allow nulls can vary from one database engine to another. For example, if you create a unique index in a Jet database which allows nulls, Jet will essentially ignore the null values from the index and allow any number of rows with null entries. Contrast that with MS SQL Server, where if an index allows nulls, only one null entry is permitted in a unique index. This applies only to primarily to multicolumn indexes since it would be rare to create a nullable unique index on a single column. To illustrate this, let's look at an example of the difference.

Column A	Column B	SQL Server Result	Jet Result
1	1	Allowed - unique entry	Allowed - unique entry
1	Null	Allowed - unique entry	Allowed - unique entry
2	1	Allowed - unique entry	Allowed - unique entry
2	Null	Allowed - unique entry	Allowed - unique entry
2	Null	Disallowed - duplicate value in unique index	Allowed - duplicate null ignored

What to Index

The following list indicates the types of data which are good candidates for indexing.

- Columns used in joins
  Joins will almost always benefit from having an index available on both sides of the join.
- Columns used in a query WHERE clause
  If a column is part of the selection criteria for a frequently run query, an index may improve query performance. For less frequently used queries, you will need to consider the cost in terms of inserts, updates, and deletes against the gain in select queries.
- Columns used in a query ORDER BY clause
  Sort performance can be improved substantially by indexing the columns used in the ORDER BY clause of the query. However, sorts are inherently slow unless a clustered index is used for the sort (in which case the data will be stored in the sorted order).
- Columns used in a query GROUP BY clause
  Grouping operations will be enhanced by indexing especially in situations where the range of values being grouped is small in relation to the number of rows in the table.

What Not to Index

This list indicates columns that may not benefit from indexing.

- Tables with a small number of rows
  If a table has only a handful of rows, the query optimizer for the database engine might determine that a table scan is more efficient than using an index. In this case, the index would only serve to slow down inserts, updates, and deletes in the table.
- Columns with a wide range of values
  If the data in a column is different for nearly every row (such as a table of addresses) the index may not be useful because the database engine might determine that scanning the table is more economical than traversing such a large index. An exception would be using a clustered index on a column or columns that are used in sorts. If a clustered index is created using the same sorting order as a query, the data would be stored in sorted order in the table.

- Tables with heavy transaction loads but limited decision support load
  If a table has a lot of insert, update, and delete activity but there are few SELECT queries run against it, added indexes will probably result in a net penalty in overall performance of the application.
- Columns not used in queries
  Columns which are rarely retrieved do not need to be indexed since the index only enhances performance for SELECT queries where the column is part of the WHERE, GROUP BY, or ORDER BY clause. Even if the column is part of many queries but is never included in criteria, sorting, etc., it will not benefit from an index.

Summary

It may take some experimentation to discover what columns to index. If your database engine can provide you with activity logs or real time information on queries, you should take advantage of these tools to determine what columns to index. If these types of tools aren't built into the database engine, you can still build your own profiling code to test variations of indexing strategies to see what works best. Remember that when testing you should try to either test with a full user load or simulate a full user load on the database as best you can. There can be substantial difference in performance with a single user versus having dozens or hundreds of users querying a database. Also keep in mind that general network traffic can have an impact on the performance of an application as well.

Return to Top of Page

Review

Let's review a few key points:

- Create tables that store the attributes of one entity, and columns that store a single attribute.
- Build a properly normalized database to improve data consistency, accuracy, and application performance.
- The integrity of the data in a database is your most important concern as a database developer. Use the power of the database engine to enforce entity, domain, and relational integrity, as well as appropriate business rules.
- Use appropriate indexing to enhance the performance of select queries, but keep in mind that indexes will create a performance penalty when inserting, updating, or deleting rows in a table.

Building a solid database application is often an iterative process, so be prepared to handle changes in the structure of the database as the application is in development. While it would make life easy as a developer if the requirements for an application could be "carved in stone" when the project begins, the reality is that you often can't determine what you need to record until after you begin recording it. However, a properly normalized database design where the data which is available is accurate will ultimately prove to be the most flexible approach. Beware of taking shortcuts in the database structure - what seems easy and expedient today may prove to be costly in the future.

Return to Top of Page