TechTarget | SearchSQLServer

E-guide

PRO+ Content

The Ultimate Developer's Guide to SQL Server

Tips, commands, and configurations to improve performance

In this e-guide

Section 1: Table & Column Best Practices

Improve SQL Server performance with these quick tips for creating tables, adding columns, and normalizing your database from expert contributors.

The last article is an excerpt from "A Developer's Guide to Data Modeling for SQL Server, Covering SQL Server 2005 and 2008" by Eric Johnson and Joshua Jones. Learn how to physically implement supertype and subtype tables and clusters on a SQL Server.

Continue reading

In this e-guide

Creating SQL Server tables: A best practices guide

Baya Pavliashvili, Contributor, SearchSQLServer

1. Always **save CREATE TABLE statements**, along with all other statements defining database schema in a secure location. Every time you make a change to a database object, be sure to script the change and check it into version-control software, such as Visual Source Safe.

   With such policy you can easily re-create database schema on the same or different server, if necessary. Also, if you have the same database on multiple servers, it's easy to compare schemas and reconcile any differences that might have crept in over time.

2. Although descriptive, **table names have no performance benefits**. They make databases self-documenting and easier to code against. Table names should reflect their business meaning.

3. Create **user tables on a non-primary filegroup**; reserve the primary file group for system objects. This way the system supplied and user-defined objects do not compete for disk resources.

In this e-guide

4. Create commonly accessed tables on the same filegroup. You can expect performance benefits if the data of commonly joined tables resides on the same disk.

5. Create a **clustered index on every table**. Each table can only have a single clustered index. If a table has a clustered index, its data is physically sorted according to the clustered index key. Clustered indexes in SQL Server have numerous benefits. For example, if you retrieve data from a table using an ORDER BY clause referencing the clustered index key, the data does not need to be sorted at query execution time.

   If two tables have a common column, for example customer_id, and both tables have clustered indexes on customer_id column joining, such tables will be considerably more efficient than joining the same tables based on the same column but without clustered indexes.

6. Ensure the clustered index is built on a column that contains distinct values in each row. This makes the clustered index also a unique index. If the clustered index key(s) contains non-unique values, SQL Server will add a hidden column to your table to make clustered index keys unique.

7. The clustered index should be created on the column(s) that is most commonly used for retrieving data from the table. Since you can only have one clustered index per table, you should carefully examine the data retrieval patterns to choose the most effective key(s) for your

In this e-guide

clustered index.

8. In addition to the clustered index, **create non-clustered indexes**, particularly on those columns used for joining the table to other tables or for filtering the data set to be retrieved.

9. When **adding a primary key constraint**, always specify whether it is clustered or non-clustered. Primary key columns by definition must contain unique values so they're good candidates for clustered indexes. But depending on your data retrieval patterns you might not want the primary key index to be clustered.

10. Be sure to **rebuild or de-fragment your indexes** periodically. SQL Server 2005 Enterprise Edition supports on-line index rebuilds which should reduce index maintenance window/overhead considerably. Nonetheless, maintaining indexes does require considerable system resources. Weigh your index maintenance options carefully.

11. SQL Server uses **table and index statistics** to come up with the most cost-effective query execution plans. SQL Server can maintain statistics on each index automatically, but you can override this option. For example, if you anticipate heavy transactional activity (millions of INSERT, UPDATE, and DELETE statements) during certain hours, you could turn off automatic update of statistics on each index. If so, be sure to periodically update statistics manually.

12. If you have numerous **"lookup" tables with very few rows** in each, consider combining them into a single "master lookup" table. For

In this e-guide

example, you could have numerous "type_lookup" and "category_lookup" tables, each with a dozen of rows. Instead of having to maintain 30 different lookup tables, you can combine them in a single table that has row_identifier, row_type and row_value columns.

Row_type can denote the kind of lookup value found in the row. This could make **developing stored procedures simpler** than trying to remember 30 different table names. Since the master table will have relatively few rows, – a few thousand rows at most – you will not see any performance degradation.

13. Use **table triggers** sparingly. You can often implement trigger functionality with constraints which tend to be considerably more efficient. A simple example is enforcing referential integrity by ensuring the record added to a given table has a corresponding record in a related table. Instead of using triggers, you should enforce such rules through foreign key constraints. Similarly if you wish to validate a string column's format you should use a check constraint, instead of a trigger. For example, social security numbers always have nine digits separated by two dashes, as in 123-45-6789. You should use a check constraint as opposed to a trigger to validate values to be inserted into this column.

14. If a table contains millions of rows and you have multiple disks (or disk arrays) at your disposal, take advantage of **table and index partitioning**. Partitioning can provide considerable query performance improvements. It can also make loading and purging

large data sets from a table very fast.

15. If the table is partitioned, make sure its indexes are aligned; this means indexes are using the same partitioning scheme as the table.

ABOUT THE AUTHOR:

Baya Pavliashvili is a database consultant helping his customers develop highly available and scalable applications with SQL Server and Analysis Services. Throughout his career he has managed database administrator teams and databases of terabyte caliber. Baya's primary areas of expertise are performance tuning, replication and data warehousing. He can be reached at baya@bayasqlconsulting.com.

In this e-guide

Next Article

In this e-guide

Creating SQL Server columns: A best practices guide

Baya Pavliashvili, Contributor, SearchSQLServer

1. Use the **smallest data type necessary to implement the required functionality**. I have worked on several systems which used NUMERIC or FLOAT data types even though not a single row contained digits to the right of the decimal point. An easy way to optimize such systems is to simply change the data type from NUMERIC to INTEGER.

2. Ensure that each column has a **descriptive name**; doing so makes database more self-documenting and easy to understand for those who didn't develop it. Do not call columns "column 2," "ID" or similar names. If you must abbreviate column names be sure to create a data dictionary denoting what data element each column is supposed to store.

3. Many tables have **"natural" keys**; these are columns that have a business meaning, such as customer or account name. Although natural keys are immediately identifiable by business users, they aren't always unique and they tend to change over time. Consider adding **"surrogate" keys** - columns that have no business meaning but can uniquely identify each row. Identity columns are a great

In this e-guide

example of surrogate keys. While you could use a combination of natural keys to uniquely identify a record, joining tables on multiple columns will normally be slower than joining the same tables based on a single column with a small data type (such as INTEGER).

- If you do use surrogate keys be sure that their name includes the table / entity name. For example, do not add a column called "ID" to each table. Instead use "customer_id", "supplier_id", "store_id" and so forth.

4. Each table allows up to 1,024 columns, but normally you don't need nearly as many columns. For transactional systems ensure the data model is highly normalized; this means the same data element (customer address, customer phone number, product description, etc) should not be repeated in multiple tables. For reporting systems you can allow some redundancy, but only if thorough testing confirms that redundant columns improve query performance.

5. If possible and appropriate, use fixed-length as opposed to variable-length data types. For example, if you know product code will always be limited to 50 characters use **CHAR(50) as opposed to VARCHAR(50)**. Variable length columns impose an overhead that isn't always necessary.

6. Use **UNICODE data types (NCHAR, NVARCHAR, NTEXT)** only when necessary. If your database will only contain European characters, then you shouldn't have to use UNICODE data types. Realize that UNICODE data types use twice-as-much storage as their non-

In this e-guide

UNICODE counterparts.

7. Be sure to specify appropriate collation for your string columns. If you don't specify the collation, SQL Server will use the collation defined at the database level. Collation determines the character set and sort order supported by the column. If the correct collation isn't specified, you could see unexpected results when retrieving string data.

8. Attempt to configure **column null-ability** correctly. If the column should always have a value, then configure it as NOT NULL. Using default constraints is more efficient than having columns allowing NULL values. A NULL value isn't equal to anything else - empty string, zero or even other NULL values. A NULL denotes that the value is unknown. With a character column, you can often use a default value of "unknown" as opposed to allowing NULL values.

9. Use large **variable length data types** sparingly. Variable length columns are normally stored on the same data pages as the rest of the record. However, if the combined size of the variable-length columns exceeds 8,000 characters, they're stored on row-overflow data pages, which imposes additional overhead during data retrieval.

10. Avoid using **TEXT, NTEXT and IMAGE data types** for any newly created table columns. These data types are deprecated and might not be available in future versions of SQL Server. Use VARCHAR(MAX), NVARCHAR(MAX) and VARBINARY(MAX) data types instead.

In this e-guide

11. If you must implement multiple large columns, such as **VARCHAR(MAX)** or **VARCHAR(4000)** for example, consider splitting your table in two or more tables with fewer columns. This technique is sometimes called vertical partitioning. These tables will have one-to-one relationships among them and each will have a common primary key column.

12. You cannot use columns with **large variable length data types**, such as VARBINARY(MAX), NVARCHAR(MAX) or VARCHAR(MAX) for clustered or non-clustered index keys. However, consider adding such columns as included columns to your non-clustered indexes. Doing so can help you have more "covered" queries, which can be resolved by seeking through the index as opposed to scanning the table. The included columns are not counted towards the 900 byte limit for index keys. They are also not counted towards the 16 column limit of index keys.

13. The **TIMESTAMP** data type is a misnomer because it doesn't track date or time values. Rather SQL Server uses a column with this data type to track the sequence of data modifications. Instead of checking each column within the table you can simply examine the values of the TIMESTAMP column to determine whether any column values have been modified.

14. Consider using **BIT data type columns for Boolean values**, as opposed to storing "TRUE / FALSE", "yes/no" or other character strings. SQL Server can store up to 8 columns with BIT data type in a single byte. One scenario where this could be handy is when you're

using "soft deletes." Instead of physically removing a record from a table, you simply tag it as deleted, by flipping the bit value of the "deleted" column to 1.

Keep these pointers in mind when creating and maintaining SQL Server tables. Although creating tables might seem like a trivial exercise, database architects should carefully weigh the consequences of each option when building large scale systems.

Next article

In this e-guide

In this e-guide

SQL Server normalization rules you must follow

Greg Robidoux, Owner, Edgewood Solutions

In this tip we will take a look at database normalization and the advantages and disadvantages of normalization for SQL Server databases.

What is normalization?

Normalization is the process of designing a data model to efficiently store data in a database. The end result is that redundant data is eliminated, and only data related to the attribute is stored within the table.

For example, let's say we store City, State and ZipCode data for Customers in the same table as Other Customer data. With this approach, we keep repeating the City, State and ZipCode data for all Customers in the same area. Instead of storing the same data again and again, we could normalize the data and create a related table called City. The "City" table could then store City, State and ZipCode along with IDs that relate back to the Customer table, and we can eliminate those three columns from the Customer table and add the new ID column.

Normalization rules have been broken down into several forms. People often refer to the third normal form (3NF) when talking about database design.

In this e-guide

This is what most database designers try to achieve: In the conceptual stages, data is segmented and normalized as much as possible, but for practical purposes those segments are changed during the evolution of the data model. Various normal forms may be introduced for different parts of the data model to handle the unique situations you may face.

Whether you have heard about normalization or not, your database most likely follows some of the rules, unless all of your data is stored in one giant table. We will take a look at the first three normal forms and the rules for determining the different forms here.

Rules for First Normal Form (1NF)

Eliminate repeating groups. This table contains repeating groups of data in the Software column.

Computer	Software
1	Word
2	Access, Word, Excel
3	Word, Excel

To follow the First Normal Form, we store one type of software for each record.

Table on next page

In this e-guide

Computer	Software
1	Word
2	Access
2	Word
3	Excel
3	Word
3	Excel

Rules for second Normal Form (2NF)

Eliminate redundant data plus 1NF. This table contains the name of the software which is redundant data.

Computer	Software
1	Word
2	Access
2	Word
3	Excel
3	Word
3	Excel

To eliminate the redundant storage of data, we create two tables. The first

In this e-guide

table stores a reference SoftwareID to our new table that has a unique list of software titles.

Computer	SoftwareID
1	1
2	2
2	1
3	3
3	1
3	3

Software	SoftwareID	Software
1	Word	1
2	Access	2
3	Excel	3

Rules for Third Normal Form (3NF)

Eliminate columns not dependent on key plus 1NF and 2NF. In this table, we have data that contains both data about the computer and the user.

Tables on next page

In this e-guide

Computer	User Name	User Hire Date	Purchased
1	Joe	4/1/2000	5/1/2003
2	Mike	9/5/2003	6/15/2004

To eliminate columns not dependent on the key, we would create the following tables. Now the data stored in the computer table is only related to the computer, and the data stored in the user table is only related to the user.

Computer	Purchased
1	5/1/2003
2	6/15/2004

User	User Name	Hire Date
1	Joe	5/1/2003
2	Mike	6/15/2004

Computer	User
1	1

What does normalization have to do with SQL Server?

To be honest, the answer here is nothing. SQL Server, like any other RDBMS, couldn't care less whether your data model follows any of the normal forms. You could create one table and store all of your data in one

In this e-guide

table or you can create a lot of little, unrelated tables to store your data. SQL Server will support whatever you decide to do. The only limiting factor you might face is the maximum number of columns SQL Server supports for a table.

SQL Server does not force or enforce any rules that require you to create a database in any of the normal forms. You are able to mix and match any of the rules you need, but it is a good idea to try to normalize your database as much as possible when you are designing it. People tend to spend a lot of time up front creating a normalized data model, but as soon as new columns or tables need to be added, they forget about the initial effort that was devoted to creating a nice clean model.

To assist in the design of your data model, you can use the DaVinci tools that are part of SQL Server Enterprise Manager.

Advantages of normalization

1. Smaller database: By eliminating duplicate data, you will be able to reduce the overall size of the database.

2. Better performance:
    a. Narrow tables: Having more fine-tuned tables allows your tables to have less columns and allows you to fit more records per data page.
    b. Fewer indexes per table mean faster maintenance tasks such as index rebuilds.
    c. Only join tables that you need.

In this e-guide

Disadvantages of normalization

1. More tables to join: By spreading out your data into more tables, you increase the need to join tables.

2. Tables contain codes instead of real data: Repeated data is stored as codes rather than meaningful data. Therefore, there is always a need to go to the lookup table for the value.

3. Data model is difficult to query against: The data model is optimized for applications, not for ad hoc querying.

Summary

Your data model design is both an art and a science. Balance what works best to support the application that will use the database and to store data in an efficient and structured manner. For transaction-based systems, a highly normalized database design is the way to go; it ensures consistent data throughout the entire database and that it is performing well. For reporting-based systems, a less normalized database is usually the best approach. You will eliminate the need to join a lot of tables and queries will be faster. Plus, the database will be much more user friendly for ad hoc reporting needs.

Next Article

In this e-guide

Supertype and subtype tables in SQL Server

Eric Johnson & Joshua Jones, Authors of *A Developer's Guide to Data Modeling for SQL Server, Covering SQL Server 2005 and 2008*

Implementing Supertypes and Subtypes

We discuss supertypes and subtypes in Chapter 2. These are entities that have several kinds of real-world objects being modeled. For example, we might have a supertype called phone with subtypes for corded and cordless phones. We separate objects into a subtype cluster because even though a phone is a phone, different types will require that we track different attributes. For example, on a cordless phone, you need to know the working range of the handset and the frequency on which it operates, and with a corded phone, you could track something like cord length. These differences are tracked in the subtypes, and all the common attributes of phones are held in the supertype.

How do you go about physically implementing a subtype cluster in SQL Server? You have three options. The first is to create a single table that represents the attributes of the supertype and also contains the attributes of all the subtypes. Your second option is to create tables for each of the subtypes, adding the supertype attributes to each of these subtype tables.

In this e-guide

Third, you can create the supertype table and the subtype tables, effectively implementing the subtype cluster in the same way it was logically modeled.

To determine which method is correct, you must look closely at the data being stored. We will walk through each of these options and look at the reasons you would use them, along with the pros and cons of each.

Supertype Table

You would choose this option when the subtypes contain few or no differences from the data stored in the supertype. For example, let's look at a cluster that stores employee data. While building a model, you discover that the company has salaried as well as hourly employees, and you decide to model this difference using subtypes and supertypes. After hashing out all the requirements, you determine that the only real difference between these types is that you store the annual salary for the salaried employees and you need to store the hourly rate and the number of hours for an hourly employee.

In this example, the subtypes contain very subtle differences, so you could build this subtype cluster by using only the supertype table. For this situation, you would likely create a single employee table that contains all the attributes for employees, including all three of the subtype attributes for salary, hourly rate, and hours. Whenever you insert an hourly employee, you would require that data be in the hourly rate and hour columns and that the salary column be left NULL. For salaried employees, you would do the exact opposite.

In this e-guide

Implementing the types in this way makes it easy to find the employee data because all of it is in the same place. The only drawback is that you must implement some logic to look at the columns that are appropriate to the type of employee you are working with. This supertype-only implementation works well only because there are very few additional attributes from the subtype's entities. If there were a lot of differences, you would end up with many of the columns being NULL for any given row, and it would take a great deal of logic to pull the data together in a meaningful way.

Subtype Tables

When the data contained in the subtypes is dissimilar and the number of common attributes from the supertype is small, you would most likely implement the subtype tables by themselves. This is effectively the opposite data layout that would prompt you to use the supertype-only model.

Suppose you're creating a system for a retail store that sells camera equipment. You could build a subtype cluster for the products that the store sells, because the products fall into distinct categories. If you look only at cameras, lenses, and tripods, you have three very different types of product. For each one, you need to store the model number, stock number, and the product's availability, but that is where the similarities end. For cameras you need to know the maximum shutter speed, frames per second, viewfinder size, battery type, and so on. Lenses have a different set of attributes, such as the focal length, focus type, minimum distance to subject, and minimum aperture. And tripods offer a new host of data; you need to store the minimum and maximum height, the planes on which it can pivot, and the type

In this e-guide

of head. Anyone who has ever bought photography equipment knows that the differences listed here barely scratch the surface; you would need many other attributes on each type to accurately describe all the options.

You are reading part 5 from "Physical data storage in SQL Server 2005 and 2008," excerpted from *A Developer's Guide to Data Modeling for SQL Server, Covering SQL Server 2005 and 2008*, by Eric Johnson and Joshua Jones, copyright 2008, printed with permission from Addison-Wesley Professional.

The sheer number of attributes that are unique for each subtype, and the fact that they have only a few in common, will push you toward implementing only the subtype tables. When you do this, each subtype table will end up storing the common data on its own. In other words, the camera, lens, and tripod tables would have columns to store model numbers, SKU numbers, and availability. When you're querying for data implemented in this way, the logic needs to support looking at the appropriate table for the type of product you need to find.

Supertype and Subtype Tables

You have probably guessed this: When there are a good number of shared attributes and a good number of differences in the subtypes, you will probably implement both the supertype and the subtype tables. A good example is a subtype cluster that stores payment information for your customers. Whether your customer pays with an electronic check, credit card, gift certificate, or cash, you need to know a few things. For any

In this e-guide

payment, you need to know who made it, the time the payment was received, the amount, and the status of the payment. But each of these payment types also requires you to know the details of the payment. For credit cards, you need the card number, card type, security code, and expiration date. For an electronic check, you need the bank account number, routing number, check number, and maybe even a driver's license number. Gift cards are simple; you need only the card number and the balance. As for cash, you probably don't need to store any additional data.

This situation calls for implementing both the supertype and the subtype tables. A Payment table could contain all the high-level detail, and individually credit card, gift card, and check tables would hold the information pertinent to each payment type. We do not have a cash table, because we do not need to store any additional data on cash payments beyond what we have in the Payment table.

When implementing a subtype cluster in this way, you also need to store the subtype **discrimination**, usually a short code or a number that is stored as a column in the supertype table to designate the appropriate subtype table. We recommend using a single character when possible, because they are small and offer more meaning to a person than a number does. In this example, you would store CC for credit card, G for a gift card, E for electronic check, and C for cash. (Notice that we used CC for a credit card to distinguish it from cash.) When querying a payment, you can join to the appropriate payment type based on this discriminator.

If you need data only from either the supertype or the subtype, this method offers two benefits: you need go to only one table, and you don't retrieve

In this e-guide

extraneous data. However, the flip side is that you must determine which subtype table you need to query and then join both tables if you need data from both the supertype and a subtype table. Additionally, you may find yourself needing information from the supertype and multiple subtypes; this will add overhead to your queries because you must join multiple tables.

Supertypes and Subtypes: A Final Word

Implementing supertypes and subtypes can, at times, be tricky. If you take the time to fully understand the data and look at the implications of splitting the data into multiple tables versus keeping it tighter, you should be able to determine the best course of action. Don't be afraid to generate some test data and run various options through performance tests to make sure you make the correct choice. When we get to building the physical model, we look at using subtype clusters as well as other alternatives for especially complex situations.

In this e-guide

Section 2: Commands

Restoring a database from another SQL Server is simple -- matching up the logins and users again is not. Get the commands you need to restore one database from another.

Continue reading

In this e-guide

Restoring a database from another SQL Server

Greg Robidoux, Owner, Edgewood Solutions

There are many reasons that you would want to move a database from one server to another, disaster recovery, refreshing a test environment, data analysis or maybe something else. Restoring the database is the easy part, but then you may be left with mismatched logins and database users. In this tip we will look at some of steps to go through when restoring a database from a different server.

The restore

The first step in the process is to restore the database. We talked about restoring databases in a previous tip and the different commands that are used to restore databases. We will take a look at a couple of the commands that can be used to look at the contents of the backup files as well as the commands to perform the restore. Restores can be accomplished by either using T-SQL commands or using Enterprise Manager. We will take a look at how the restore can be accomplished using T-SQL.

In this e-guide

RESTORE HEADERONLY

To determine what is stored on the backup files you can run this command in Query Analyzer.

```
RESTORE HEADERONLY FROM DISK = 'C:\SQL\Backup\North.bak'
```

This command allows you to see the backup header information for all backup sets on a particular backup device. The information included in this is shown in the following table. This command is useful when you have multiple files to work with or if you get a backup file from a server that you do not manage.

Table on next page

In this e-guide

BackupName	North backup
BackupDescription	NULL
BackupType	1
ExpirationDate	NULL
Compressed	0
Position	1
DeviceType	2
UserName	TestServ\GRobidoux
ServerName	GRobidoux\TEST
DatabaseName	NORTH
DatabaseVersion	539
DatabaseCreationDate	5/5/05 10:30
BackupSize	3980800
FirstLsn	290000000077700000.00
LastLsn	290000000078000000.00
CheckpointLsn	290000000077700000.00
DifferentialBaseLsn	10000000026400000.00
BackupStartDate	5/26/05 14:18
BackupFinishDate	5/26/05 14:18
SortOrder	52
CodePage	228
UnicodeLocaleId	1033
UnicodeComparisonStyle	196609
CompatibilityLevel	80
SoftwareVendorId	4608
SoftwareVersionMajor	8
SoftwareVersionMinor	0
SoftwareVersionBuild	760
MachineName	GRNBNEW
Flags	0
BindingId	{6765CCC0-972C-4552-8607-2CD5818B8109}
RecoveryForkId	{6765CCC0-972C-4552-8607-2CD5818B8109}
Collation	SQL_Latin1_General_CP1_CI_AS

In this e-guide

RESTORE FILELISTONLY

This is another command that can be run on the backup files.

```
RESTORE FILELISTONLY FROM DISK = 'C:\SQL\Backup\North.bak'
```

This command allows you to see a list of the database and log files contained in the backup set, what filegroup they are in and also the size of the data and log files. For our restore process the LogicalName and PhysicalName are key data elements. These will be used in the next step when we actually perform the restore.

LogicalName	NORTH_Data	NORTH_Log
PhysicalName	C:\Program Files\Microsoft SQL Server\MSSQL$TEST\data\NORTH_Data.MDF	C:\Program Files\Microsoft SQL Server\MSSQL$TEST\data\NORTH_Log.LDF
Type	D	L
FileGroupName	PRIMARY	NULL
Size	5111808	53673984
MaxSize	35184372080640.00	35184372080640.00

RESTORE

To restore the database we may need to use the MOVE option to move the physical files to a different location and also the NORECOVERY option if we want to restore multiple backup files (i.e. full, differential and logs).

In this e-guide

Let's say we are moving the database from one server to another and the logical drives are different. In our output above the data files were located in the C: \Program Files\Microsoft SQL Server\MSSQL$TEST\data directory, but we now need to restore to the D: and E: drive on a different server. The command would look like this.

```
RESTORE DATABASE NORTH FROM DISK = 'C:\SQL\Backup\North.bak' WITH MOVE
'NORTH_Data' TO 'D:\SQL\Data\North_Data.mdf', MOVE 'NORTH_Log' TO
'E:\SQL\Log\North_Log.ldf'
```

If we also need to restore and move both a full, differential and log backups the commands would look like this.

```
RESTORE DATABASE NORTH FROM DISK = 'C:\SQL\Backup\North.bak' WITH
NORECOVERY, MOVE 'NORTH_Data' TO 'D:\SQL\Data\North_Data.mdf', MOVE
'NORTH_Log' TO 'E:\SQL\Log\North_Log.ldf' RESTORE DATABASE NORTH FROM
DISK = 'C:\SQL\Backup\North_Diff.bak' WITH NORECOVERY, MOVE
'NORTH_Data' TO 'D:\SQL\Data\North_Data.mdf', MOVE 'NORTH_Log' TO
'E:\SQL\Log\North_Log.ldf' RESTORE LOG NORTH FROM DISK =
'C:\SQL\Backup\North_Log.bak' WITH RECOVERY, MOVE 'NORTH_Data' TO
'D:\SQL\Data\North_Data.mdf', MOVE 'NORTH_Log' TO
'E:\SQL\Log\North_Log.ldf'
```

In this e-guide

Users

Now that the database has been successfully restored we need to make sure our users and logins match on the server.

When you restore backups from another server their will most likely be mismatched login and user information. The login information is stored in the sysxlogins table in the master database. This table contains a column that holds the SID (Security Identifier) which is tied to a specific login. A corresponding table sysusers is stored in each user database and uses the SID to determine if a login has database access. So even though you may already have the same logins on your new server the SID may not match. What makes this even more confusing is that you can see the actual names of the logins and users, so you would think they would automatically match up, but the SID is what is actually tying the security together not the name.

So if the standard login already exists on the server and the user exists in the database you can use this stored procedure to relink the standard login and user.

```
sp_change_users_login
```

This stored procedure has three options Auto_Fix, Report and Update_One. Below is a brief description, but more can be found in SQL Server 2000 Books Online.

In this e-guide

- The Auto_Fix option will link users and logins that have the same name.
- The Report option will show you a list of users in the current database that are not linked to a login.
- The Update_One option allows you to link a user and login that may not have the same exact name.

Note: To link Windows logins see more information under Moving Logins section below.

REMOVING USERS

If there are users in the database and there is not a corresponding login on the server, you can use the following command to remove the user from the database and cleanup the user list.

```
sp_revokedbaccess 'NorthDomain\Mike'
```

LOGINS

On the other end of the security there are logins. To find out what access a login has on your server you can run the following command. This will display a list of databases the login has access to as well as other info about

In this e-guide

the login such as the SID.

```
sp_helplogins 'NorthDomain\Mike'
```

If there is a user in your database and there is not a corresponding login on the server you can use the following commands to create the new login.

If you are creating a standard SQL Server login you would use this command to create the login.

```
sp_addlogin
```

If you are creating a login that will use Windows authentication use this command.

```
sp_grantlogin 'NorthDomain\Mary'
```

To change the default database for a login use this command for both Windows or standard logins.

```
sp_defaultdb 'NorthDomain\Mary', 'master'
```

In this e-guide

MOVING LOGINS

In some cases you may need to duplicate the entire login list from one server to another. This can be achieved manually by running the above commands, by using DTS or you can use the stored procedure that was developed by Microsoft (see link below). When you create the logins manually or use DTS the original SID is not kept and a new SID is created as the login is being created, therefore you will need to link the logins and users for your database. See Microsoft Knowledge Base article 246133 How to: Transfer logins and passwords between instances of SQL Server for more information.

If your NT login and user names do not match you can use the following commands to link your NT login and user information. Microsoft does not recommend updating system tables directly, so you should only use this if you totally understand what you are doing and how to recover if there is a problem. You also need to change the server setting to allow updates to system tables.

```
DECLARE @sysxlogins_sid VARBINARY(85) SELECT @sysxlogins_sid = sid
FROM master.dbo.sysxlogins WHERE name = 'NorthDomain\Joe' UPDATE
sysusers SET sid = @sysxlogins_sid WHERE name = 'Joe'
```

In this e-guide

Changing database owner

The last thing you may want to do is change the owner of the database after the restore has been completed. When a database gets restored the user running the restore command becomes the owner of the database. To change the owner, issue this command.

```
sp_changedbowner 'sa'
```

Summary

As you can see there is a little more to the restore process then just restoring your backup files. Keeping logins and users in sync is not always a simple task, unless the machines are completely identical, which rarely happens. The above steps along with the commands should be all you need to get your databases restored and your users linked properly. For additional information you can always refer to SQL Server 2000 Books Online.

Next section

In this e-guide

Section 3: Configuring Aliases

Aliased names can save you from changing your code and configuration files when you need to point your application to another server or move the location of database objects. In this section, you'll find options and strategies for aliasing database servers and database objects.

Continue reading

In this e-guide

Configure aliases for SQL Server databases and servers

Roman Rehak, Principal Database Architect, MyWebGrocer

Most SQL Server applications access objects in the same database, but once in a while you need to step outside the database to another database on the same server or on another SQL Server instance. In either scenario, you can utilize aliased names of database servers and objects in other databases, instead of using the actual names. This article will demonstrate different options for using aliases in SQL Server. I'll show how aliases can be beneficial to reduce the number of changes you need to make in your code or configuration files when the physical location of your distributed objects needs to be changed.

First, let's look at the **server-level alias**. A SQL Server alias is a mechanism that allows you to create an aliased name for a SQL Server instance. You can think of server aliasing as a mapping between the aliased name and the actual instance name or the IP address of the SQL Server instance.

In addition to the names, you can also define the port number and the network protocol. Database connections that use the aliased name will be connected to the mapped server instance using the network protocol specified in the alias definition. Figure 1 shows you a sample alias - I am aliasing the IP address of a SQL Server instance to the name *Inventory*, plus specifying the port number 6379.

In this e-guide

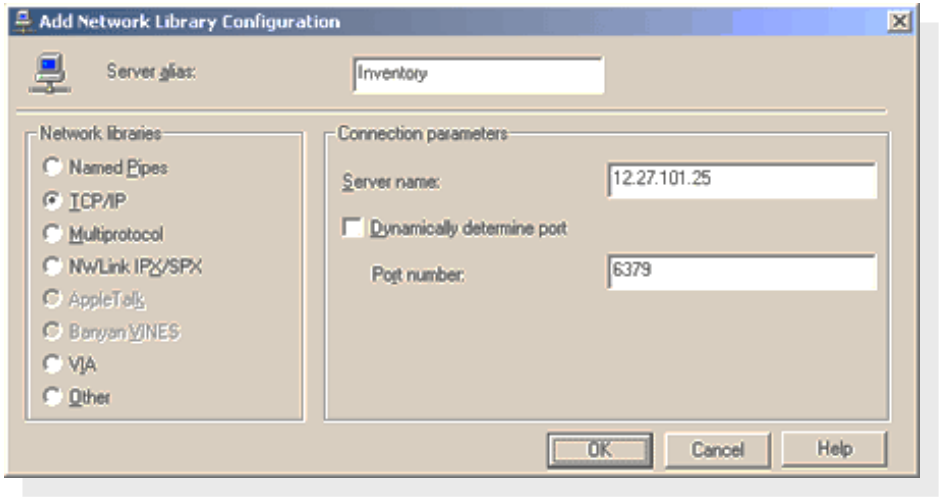

Figure 1: Sample alias of the IP address of a SQL Server instance.

Server aliases can be very beneficial in many different scenarios:

- If you have a distributed environment with multiple servers that reference each other using Linked Servers, you can define your linked servers using aliases rather than the actual names. If, in the future, you need to move databases from one server to another, you will not have to modify linked server names in your T-SQL code, just modify where the alias is pointing to.

- You can alias an IP address to a friendly name.

In this e-guide

- You can enforce the Named Pipes connection protocol instead of the default TCP/IP. This is useful if you want to connect using Windows Authentication to a SQL Server in another domain or over the Internet using VPN. If the TCP/IP connection gives you the "Cannot generate SSPI context" error, you might be successful by creating an alias and specifying the Named Pipes protocol.

- Quickly redirect application connections from the primary to the standby SQL Server in your high-availability scenarios. If your primary server is down, you will not have to modify all connection strings in all applications. Instead, all you have to do is alias the name of the primary server to point to the secondary server. If you need to go back in the other direction, you can delete the alias or point the server to itself.

Creating a server alias depends on what client tools are installed on the computer. If you have SQL Server 2000 client tools, use SQL Client Network Utility and select the *Alias* tab. If you have SQL Server 2005 tools, open SQL Server Configuration Manager and expand the SQL Native Client Configuration node to find the node for *Aliases*. Most application servers don't have SQL Server tools installed, but luckily the client network utility is also included in the MDAC pack install. You can run it from DOS by executing "cliconfg" from the command prompt.

In this e-guide

Synonyms in SQL Server 2005

Now let's discuss aliasing of database objects using the new feature in SQL Server 2005 called Synonyms. A synonym is a database level object that allows you to define an alternate name for another database object. The aliased object can reside in the same database, in another database on the same server, or even on another server. You can create synonyms for these database objects: tables, views, stored procedures and user-defined functions.

T-SQL has two commands for working with synonyms - CREATE SYNONYN and DROP SYNONYM. For whatever reason, there is no ALTER equivalent so you have to drop and recreate the synonym if you need to make a change. Figure 2 shows how to create a synonym in the AdventureWorks database for a table in the AdventureWorksDW database:

Figure 2 on next page

In this e-guide

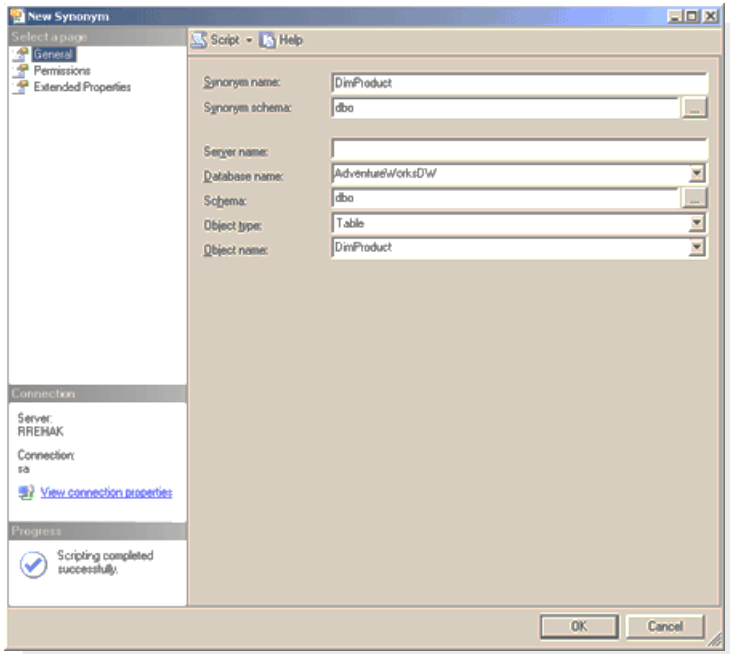

Figure 2: Creating a synonym in AdventureWorks. (*Click on image for enlarged view.*)

In this e-guide

The following T-SQL creates the same synonym:

```
CREATE SYNONYM [dbo].[DimProduct]
FOR
[AdventureWorksDW].[dbo].[DimProduct]
```

If you now execute "SELECT * FROM DimProduct" in the AdventureWorks database, SQL Server returns data from the AdventureWorksDW.dboDimProduct table.

Creating database aliases provides the following advantages:

- You don't have to use three-part names when referencing objects in another database on the same server or four-part names for objects on another server. Instead, you reference them as if they resided in the same database.
- If the location of the object changes, you can just create a new alias or modify the existing one instead of changing the code in your application. However, keep in mind that when you move a table to another server, creating a synonym will not help you avoid dealing with the idiosyncrasies and restrictions of using linked servers. For example, you will have to make sure that DTC is correctly configured and running. While you may not need to modify your object names, you still should thoroughly test your distributed application and make

In this e-guide

sure that cross-server data retrievals and updates are still working.

- You can create synonyms in the same database to provide backward compatibility for older applications when you drop or rename objects.

SQL Server synonyms are loosely bound to the referenced objects. This means you can delete aliases without getting any notification that other database objects are referencing it. Also, keep in mind that synonym chaining is not allowed. If MyTable2 is a synonym for MyTable, you cannot create MyTable3 as a synonym for MyTable2.

You have different options for referencing database servers and database objects by aliased names. When the physical structure of your distributed environment needs to be changed, use aliasing to minimize changes in your code and application configurations.

Further reading

In this e-guide

About SearchSQLServer

We're a free informational hub for enterprise database developers and administrators working with Microsoft's relational database management system, SQL Server. Our site provides news, expert advice, tips, digital learning guides, definitions, research, and quizzes on all aspects of SQL Server database design, development, administration, troubleshooting, and management.

Our editors and contributors compare new releases, editions, and platforms before and after they're released – including pricing, licensing, features, and support. We're dedicated to serving SQL Server DBAs with helpful information to fine-tune SQL Server performance, ensure data security, improve productivity, unify data sources, and successfully improve data quality to yield more actionable insights and results.

For further reading, visit us at http://SearchSQLServer.com/

Images; Fotalia