SQL Server 2000 Handbook

2nd Edition

Author: Stamati Crook
stamati@redware.com
Date: 18 October 2001
Version: 4.0

© REDWARE 1996, 2001.

Shareware Licence

Copyright © REDWARE 1996, 2001.

8 September 2001 - Version 4.0
All rights reserved. This book is **shareware** and may be downloaded and stored on a single computer for 30 days for the purposes of evaluation only. Registration is required by making the appropriate payment at the **redware** website. The book is copyright and no part shall be reproduced, stored in a retrieval system, or transferred by any means: electronic, mechanical, photocopying, recording, or otherwise without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this handbook, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein. For information, please contact:
redware research ltd, 104 Tamworth Road, Hove BN3 5FH, England.
http://www.redware.com

Acknowledgements

Second Edition September 2001

Thank you this time to Victor, Phong and especially James for helping build really big database systems during our roller coaster ride at First Telecom. Thanks also to the job market for letting me take a break and update this book.

First Edition December 1996

Thank you to the technical team at F1 Computing Systems past and present for eight years of implementing FoxPro projects. All of my FoxPro experience has resulted from working with Ian, Phong, James, Danny and David on various projects during my eight years at F1. They still have the best training courses and FoxPro team in the UK.
Thank you especially to James Thornton at F1 Computing who put me straight on a few things regarding SQL Server. Any errors remaining in this book are, of course, down to me and I apologise in advance for them. Please email me with your comments, good and bad.

1. Contents

1. CONTENTS **3**

2. SQL SERVER OVERVIEW **6**

RELATIONAL DATABASE TERMINOLOGY 6
SQL SERVER HISTORY 6
Sybase *6*
SQL Server 4.2 .... *6*
SQL Server 6.0 .... *7*
SQL Server 6.5 .... *7*
SQL Server 7.0 .... *7*
SQL Server 2000 .... *7*
SQL SERVER FEATURES 8
Transactions *8*
Data Dictionary *8*
Constraints *8*
Structured Query Language *8*
Enterprise Networking *9*
Administration *9*
Connectivity *10*
Views *10*
Triggers *10*
Stored Procedures *10*
Replication *10*
User Defined Functions *11*
XML *11*
HTML *11*

3. SQL TOOLS **12**

SERVICE MANAGER 12
ENTERPRISE MANAGER 12
Register the Server *13*
SQL QUERY ANALYSER 14
OSQL 15

4. SQL SYNTAX **16**

PUBS 16
SELECT STATEMENT 16
Field List *17*
WHERE Clause *17*
Wild Cards *18*
FROM Clause *18*
ORDER BY *19*
Natural Join *19*
GROUP BY Clause *19*
HAVING Clause *20*
DISTINCT *20*
Inner (Natural) Join *20*
Outer Join *21*
Sub Queries *21*
UNION *21*
FOR XML mode [, XMLDATA] [, ELEMENTS][, BINARY BASE64] *22*
SELECT .. INTO *22*
INSERT STATEMENT 22
UPDATE STATEMENT 23

© REDWARE 1996,2001.

DELETE STATEMENT........ 23

5. DATABASE DEFINITION........ 25

ENTERPRISE MANAGER........ 25
CREATE A DATABASE........ 26
CREATE A TABLE........ 28
Data Types........ *29*
Nulls and Defaults........ *30*
Table Ownership........ *30*
FIELD PROPERTIES........ 31
Null Values........ *32*
DEFAULT CONSTRAINTS........ 32
CHECK CONSTRAINTS........ 33
CREATE A PRIMARY KEY........ 34
Identity Columns........ *34*
Unique Identifiers........ *35*
PRIMARY KEY CONSTRAINT........ 36
FOREIGN KEYS AND REFERENTIAL INTEGRITY........ 36
USER DEFINED DATA TYPES........ 38
DEFAULTS AND RULES........ 39
Defaults........ *39*
Rules........ *39*

6. INDEXES........ 40

UNIQUE INDEX CONSTRAINT........ 40
CLUSTERED INDEX........ 40

7. VIEWS........ 42

INDEXED VIEWS........ 43
CHECK OPTION........ 43
PARTITIONED VIEWS........ 43
OPENROWSET........ 43
LINKED SERVERS........ 44
TEMPORARY TABLES........ 44

8. STORED PROCEDURES........ 46

EXECUTING A STORED PROCEDURE........ 47
PASSING PARAMETERS........ 49
RETURNING A VALUE........ 50
OUTPUT PARAMETERS........ 50
PROGRAM STRUCTURES........ 51
LOCAL VARIABLES........ 52
SYSTEM VARIABLES........ 53
SCALAR FUNCTIONS........ 53
CASE EXPRESSION........ 54
CURSORS........ 55
SYSTEM PROCEDURES........ 56
EXTENDED PROCEDURES........ 57
EXTENDED MAIL PROCEDURES........ 57
ERROR HANDLING........ 58
TRANSACTIONS........ 59
DISTRIBUTED TRANSACTIONS........ 61

9. TRIGGERS........ 62

TRIGGER PROGRAM STRUCTURE........ 62
FIELD LEVEL VALIDATION........ 64
RECORD LEVEL VALIDATION........ 65
CHECKING VALUES AGAINST ANOTHER TABLE........ 65

PREVENTING CHANGES TO A FIELD 65
REFERENTIAL INTEGRITY CHECKS 66
Checking a Foreign Key *66*
Ensuring Unique Candidate Keys *67*
Checking Referential Integrity on Delete *67*
CASCADING DELETE 67
UPDATING ANOTHER TABLE 68

10. SQL SERVER OPTIMISATION 70

QUERY OPTIMISATION 70
Update Statistics *70*
Index Design *70*
Ordering *72*
Showplan *72*
SQL Trace *73*
Optimiser Hints *74*
CLUSTERED INDEXES 74
INDEX TUNING WIZARD 75
STORED PROCEDURE RECOMPILATION 75
DEFERRED UPDATES 76
LOCKING ISSUES 76

11. CONFIGURATION 78

SERVER CONFIGURATION 78
Memory *78*
Lock Escalation Percentage *79*
Network Packet Size *79*
Open Databases *79*
User Connections *79*
DATABASE CONFIGURATION 80
Size of tempdb *80*
Truncate Log on Checkpoint *80*
Deleting a Database *80*
BACKUPS 80

12. SECURITY 81

SERVER LOGINS 81
DATABASE USERS 82
Permissions *83*

13. INDEX 85

2. SQL Server Overview

Relational Database Terminology

Relational database theory was first defined by Edgar Codd on the principle that relationships between database tables could be defined by the programmer rather than implicitly in the database definition. This improved on the flexibility of the hierarchical database and allowed the programmer to join any two tables together on any common field as required at the application level.

The relational model defines **tables** as a collection of **fields** (domains) which contain values stored as **records** (tuples) in the table. Each record must have a **primary key** which uniquely identifies the occurrence of the record within the table. Fields are defined as numeric, character, date and so forth and may or may not contain values. A relational database can distinguish between a blank or zero and an empty or **null** value.

The programmer may define a **join** between two tables on any common field. This is usually determined by the database designer who includes **foreign key** values in the child datafile that contain primary key values of the parent datafile to allow corresponding records to match up. The programmer may however join tables on any field or fields of the same datatype in both tables to create a **many-to-one** or **one-to-one** relationship. Note that **many-to-many** relationships may not be implemented in a relational database and are implemented with a virtual link table containing foreign key relationships to each of the parent tables.

A Database Management System (DBMS) usually has a **database definition language (DDL)** which allows for the field types and tables to be defined and a **data manipulation language (DML)** which allows for the retrieval and update of data. The manipulation language often comes in several formats allowing access to the database from a variety of programming languages.

Codd went on to define a combined database definition and data manipulation language called **Structured Query Language** or **SQL** (pronounced Sequel). This was implemented in IBM's first relational database product and has now become the standard for most relational database systems. Many older hierarchical and network database management systems also allow data manipulation by interpreting SQL syntax to perform operations on data stored in more traditional logical database formats.

SQL Server History

Sybase

Sybase come into the fray over ten years ago as a pure implementation of an RDBMS taking into account many of the technical refinements of the first generation RDBMS. The implementation is functionally equivalent and an effective competitor to relational database products from Oracle, Ingres and IBM DB2. Sybase runs on many large UNIX computers and is compatible to a large extent with SQL Server.

SQL Server 4.2

Microsoft licensed Sybase technology for use on their operating systems and SQL Server is an implementation of Sybase 4.2 on the OS/2 and Windows NT platforms. The NT version offers the technical advantages of the Sybase implementation coupled with a visual administration tool and very cost effective transaction rates.

© REDWARE 1996,2001.

SQL Server 6.0 Splash Screen

SQL Server 6.0

SQL Server 6.0 is a rewrite of the original SQL Server product that takes advantage of the Windows NT Operating System and allows remote management of a collection of enterprise wide servers. Microsoft are following an independent path from Sybase and have incorporated advanced features such as Replication and support for multi-processor hardware in this version.

SQL Server 6.5

SQL Server 6.5 contains several performance improvements particularly in areas where many users are accessing the same portion of a table for updates. This improves various contention scenarios when many users are attempting to add records and compete to add sequentially to a clustered index for example.

Replication is also much improved and can now replicate with other ODBC data sources as well as interface with Oracle or other more complicated corporate situations.

SQL Server 7.0

SQL Server 7.0 re-engineered the product to use native Windows NT files for a more logical integration with backup systems. Many of the configuration parameters became ‘self-tuning’ to avoid the need for a DBA on smaller systems.

SQL Server 2000

More improvements for the DBA including an index tuning wizard which suggests potential indexes to be placed on tables. Introduction of user defined functions and partitioned views and functionality to support XML.

© REDWARE 1996,2001.

SQL Server Features

SQL Server is a fully fledged relational database server that runs on all versions of Microsoft Windows. The server software is licensed from Sybase and there is a high degree of compatibility with large scale Sybase servers.

Transactions

Many DBMS allow for the concept of a transaction which is a programmer defined unit of work. The programmer defines the beginning and the end of the transaction and any changes made to the data in the database are logged in a transaction log until the programmer completes the transaction with a **Commit** command. The database will then write all of the transactions into the database. If there are any problems in completing the transaction, for example a record locking deadlock occurs with another user, then the DBMS will **Rollback** the database as if the transaction never happened.

The transaction log may also help with database recovery in case of a hardware failure in that the database can be rolled forward using the transaction log from a previously saved state until the last fully completed database transaction.

Correct use of programmer defined transactions allows for the data stored in the database to be correct at all times even if a hardware failure interrupts the program flow.

Data Dictionary

Usually an RDBMS will support a **data dictionary**. This is a set of tables which are stored in each user database and are referred to as **system tables**. SQL Server maintains several system tables each containing information about different parts of a database. For example a system table, called SYSINDEXES, exists in each user database which contains information about all the indexes set-up on tables across the users database.

These system tables can be queried and viewed like any other table but are usually hidden from the user to avoid confusion and can also be accessed using system stored procedures.

Constraints

Constraints may often be defined in a Database to allow data to be checked by the database software before it is added or modified in the database. This has the advantage of ensuring that data is always valid as a program cannot pass in data that breaks a constraint and also allows these checks to be implemented once in the database software rather than in each application that updates the database.

Structured Query Language

The **structured query language (SQL)** used in SQL Server is very similar to the ANSI SQL standard. Following are a few examples of SQL commands, more detailed explanations of the commands available can be found in the Transact-SQL Reference manual supplied with SQL Server.

Creation of a table:

```
CREATE TABLE contact
(contact_id cid,
name varchar(30),
address varchar(60),
telephone_number varchar(20),
rating tinyint)
```

Insertion of a record into a table:

```
INSERT INTO TABLE contact
   (contact_id,
   name,
```

© REDWARE 1996,2001.

```
   address,
   telephone_number,
   rating)
   VALUES
   ('00003215',
   'John Brown',
   '67 North Street, Guildford, Surrey',
   '0327-7384629',
   8)
```

Updating of a record:

```
UPDATE contact
   SET telephone_number = '01327-7384629',
   rating = 9
   WHERE contact.contact_id = '00003215'
```

Deletion of a record:

```
DELETE contact
   FROM contact
   WHERE contact.contact_id = '00003215'
```

In addition to the standard SQL functions, SQL Server supports extensions to the traditional syntax to allow the implementation of outer joins and other enhancements to the language.

SQL Server also implements a programming version of the SQL language known as Transact SQL which is used in the definition of program scripts for triggers and stored procedures.

Enterprise Networking

SQL Server is part of the BackOffice suite of programs designed to run on Microsoft NT Advanced Server. The technology is suitable for Enterprise Networking where many NT Servers are situated throughout an organisation connected together in a Wide Area Network.

SQL Server may be installed on some or all of the servers to provide departmental databases. Programs may access more than one database if required. In addition, a SQL user may be configured as a remote user on another server so that the two servers communicate by automatically logging the user onto the second, remote, server to allow access to data.

Further facilities such as security that is integrated with network security, replication of data between servers, remote administration of servers from a workstation, and integration with electronic mail make SQL Server a good choice for a multi-server networked environment.

Administration

Administration of the server is performed through the SQL Enterprise Manager which allows for the management of any server on the LAN or WAN using client software running on Windows 95 or Windows NT.

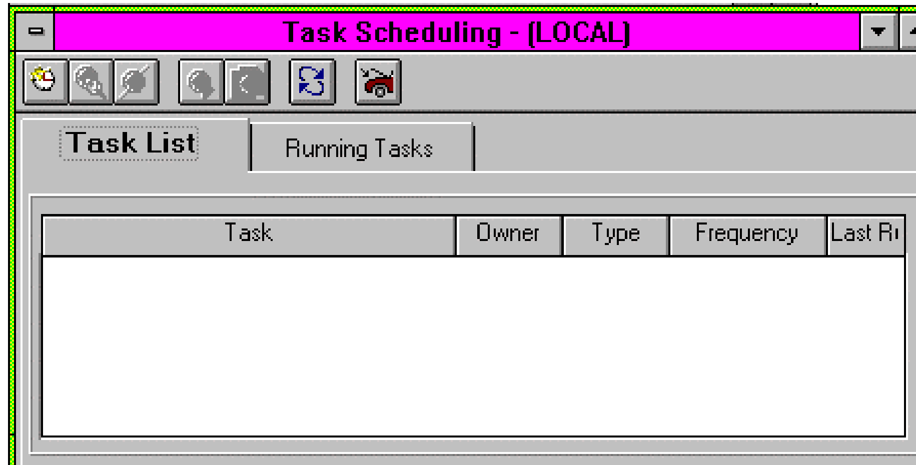
Microsoft have implemented software components which can connect to SQL Server administration functionality for programmatic control of complex administration and system management.

SQL Server has a task management program that schedules activity at regular intervals. This activity includes the implementation of replication triggers which replicate data between servers.

The current Tasks can be viewed with the Tools-Task Scheduling… menu option in the SQL Enterprise Manager.

© REDWARE 1996,2001.

Task Scheduling Window



User security may be automatically inherited from the Network Configuration and the specification of physical devices for the database is fairly straightforward. The database can take advantage of sophisticated operating system features such as RAID fault tolerant disks to supplement the security features of mirrored transaction logs.

Connectivity

Microsoft provide the latest ODBC and OleDB drivers for SQL Server to provide some of the highest connection speeds available from a variety of programming environments. ADO ActiveX data object provide a convinient way to manage OleDB data sources and a well integrated with a SQL Server environment.

Gateways exist to transparently connect a request for SQL Server data through to a Mainframe database.

Views

Views on data may be easily defined to allow local or global corporate database schemas to be defined and yet allow for the underlying local structure to be changed if required without affecting existing programs.

Triggers

Triggers allow programs to be executed on the server whenever data is updated to prevent updates or to perform processing,

Stored Procedures

Stored procedures are programs that run on the server using an enhanced form of SQL called Transact-SQL. This includes program control functionality and the facility to call external programs residing on the server such as electronic mail.

There are considerable benefits in getting the server to perform tasks rather than calling a workstation process. This is particularly relevant in high transaction systems which interface with other components of the computing infrastructure as network traffic is not a bottle-neck when the processing is performed solely on the server,

Replication

Data is published on one server and other servers are defined as subscribers to that data. The SQL Executive copies data regularly during the process of database synchronisation. Replicated data is not modifiable on the subscription databases.

© REDWARE 1996,2001.

User Defined Functions

User defined function (new in SQL 2000) allow Transact-SQL to be used to create a program that returns a single value or one that returns a cursor. The former allows re-usable functions to be used in program implementation and the latter provides a programmatic alternative to defining a View.

XML

XML has considerable support in SQL 2000 and allows fully formed XML files to be returned directly from a server stored procedure without the need of any further middleware.

HTML

SQL Server has been integrated with Internet Information Server to serve directly to an http: request. This can provide powerful functionality when used together with stored procedures return XML in combination with an XLST formatting file.

3. SQL Tools

Service Manager

SQL Server runs as a service on Windows NT Advanced Server and is usually configured to autostart. The SQL Service Manager can be used to start SQL Server running and must be run on the server itself. Simply bring up the window and press the buttons to start and stop the server.

SQL Service Manager

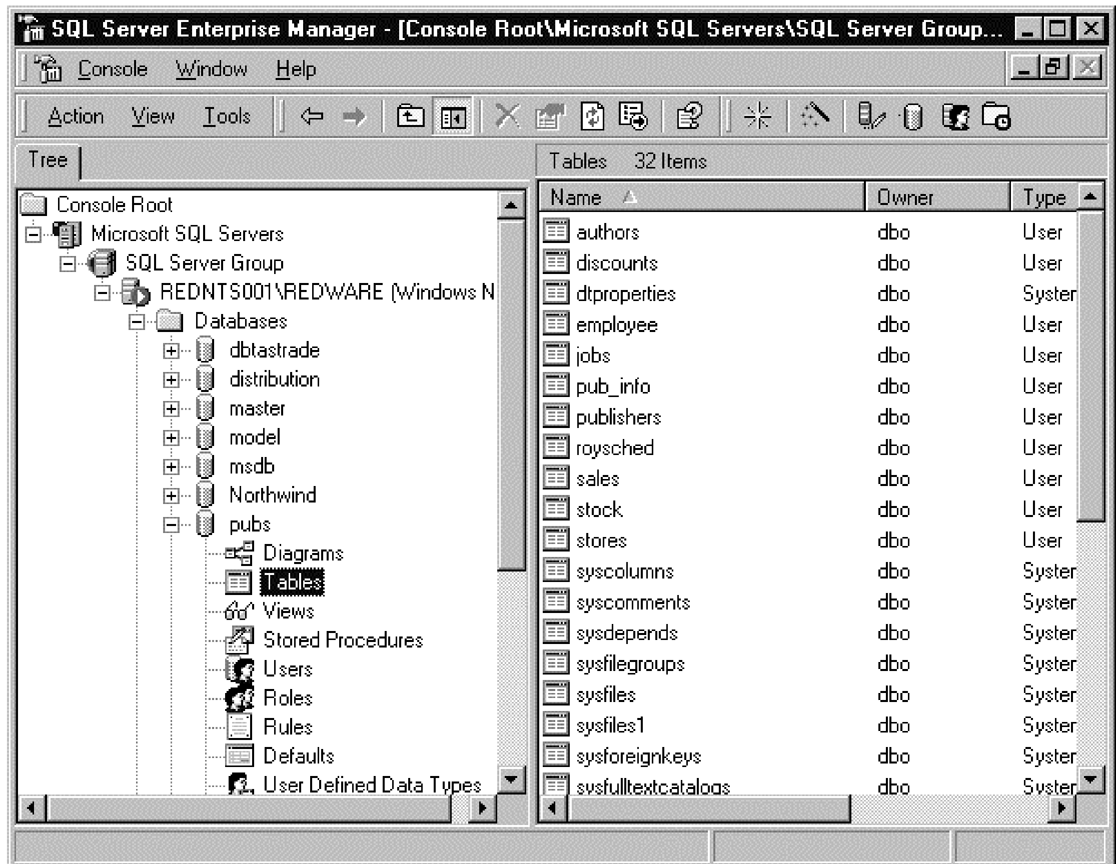
Other services may also be controlled by the service manager by selecting from the Services combo.

Enterprise Manager

The SQL Enterprise Manager is the administrative interface for SQL Server allowing the access to all of the features of SQL Server. SQL Enterprise Manager utilises Microsoft's Distributed Management Framework (DMF) so that any SQL Server installation can be controlled from Windows 95 or Windows NT workstation.

After registering a SQL Server installation on the SQL Enterprise Manager, the Server can be controlled through use of the outline control. Clicking on the Server will bring up an outline which contains the objects controlled by the server.

© REDWARE 1996,2001.



SQL Server Enterprise Manager

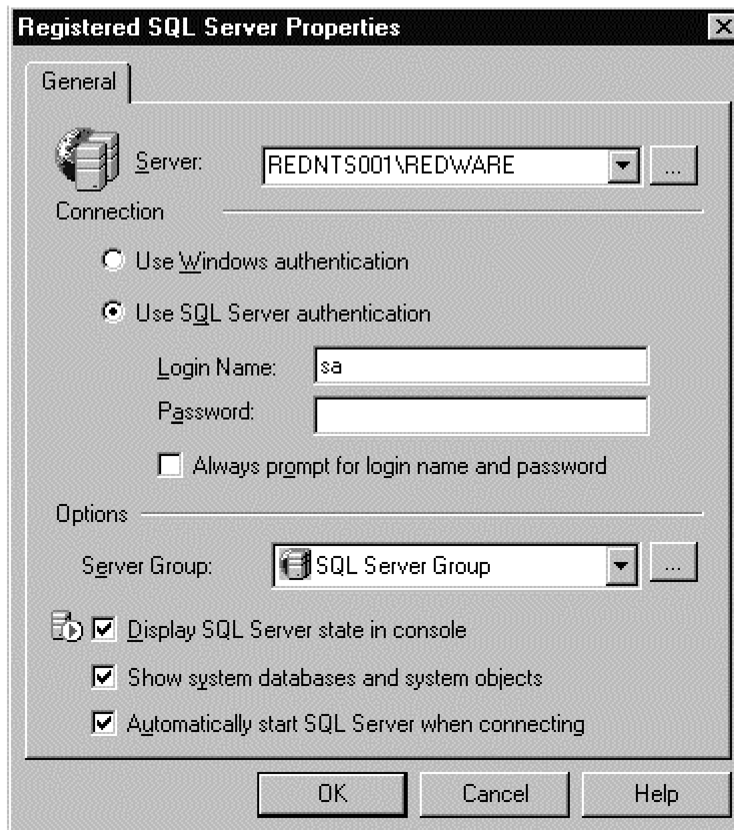
The Enterprise manager can be used to:

- Create a new database and alter database properties.
- Define new tables, views, stored procedures and user defined function within a database.
- Access tools such as the Query Analyser, SQL Profiler, and database diagrams.
- Define users and roles and set permissions on database objects.
- Schedule jobs with the SQL Agent.
- Set up and operate database replication.
- Import and Export data with the DTS.
- Perform database administration tasks such as backups.
- Monitor system activity.

Register the Server

Before we can change the structure of a database we need to register the Server on the current workstation.

This happens automatically the first time the SQL Executive is run or an option can be selected from the Server-Register Server menu to add additional servers.



Registering a Server

> *There may be several separate SQL Server installations running on the same machine.*

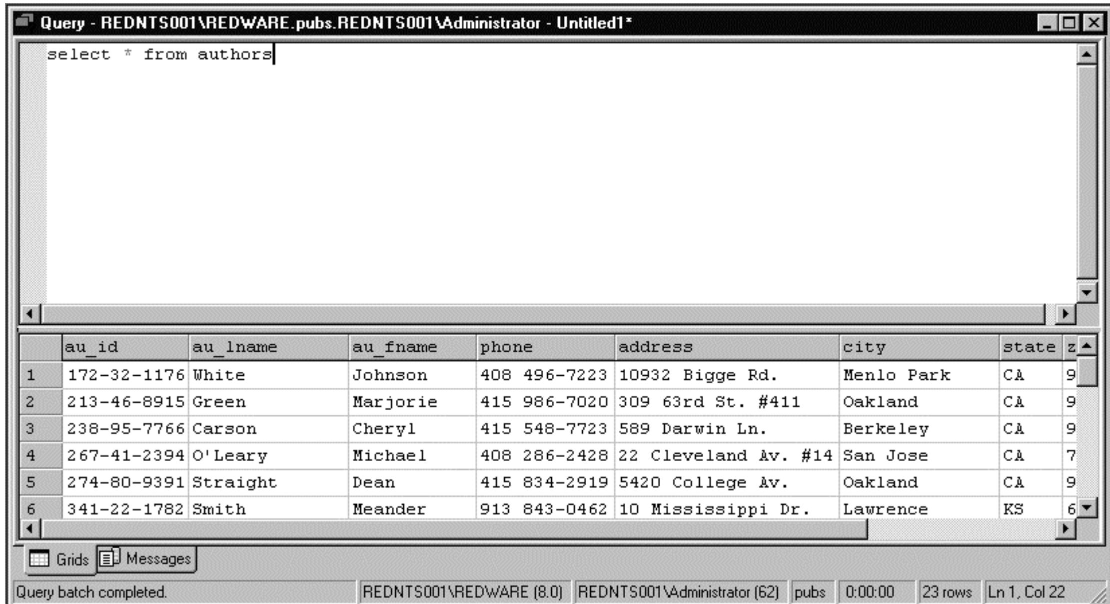
SQL Query Analyser

The SQL Query Analyser is available from the TOOLS.. menu of the Enterprise Manager. The user must log onto the required server and specify the required database using the Database combo at the top of the window. The user can then type in a SQL Server statement in the Query window and press the Execute button to see the results interactively.

The Query Analyser can be used to run any command that SQL Server understands as well as standard SQL queries. Simply and type in the required Transact-SQL commands and press the Green GO triangle icon to run the query and view the results. Note that there are two pages for results, one for results sets returned and the other for messages.

> *Press SHIFT+F1 on any SQL keyword to launch SQL Books Online with the required information displayed.*

SQL Query Analyser

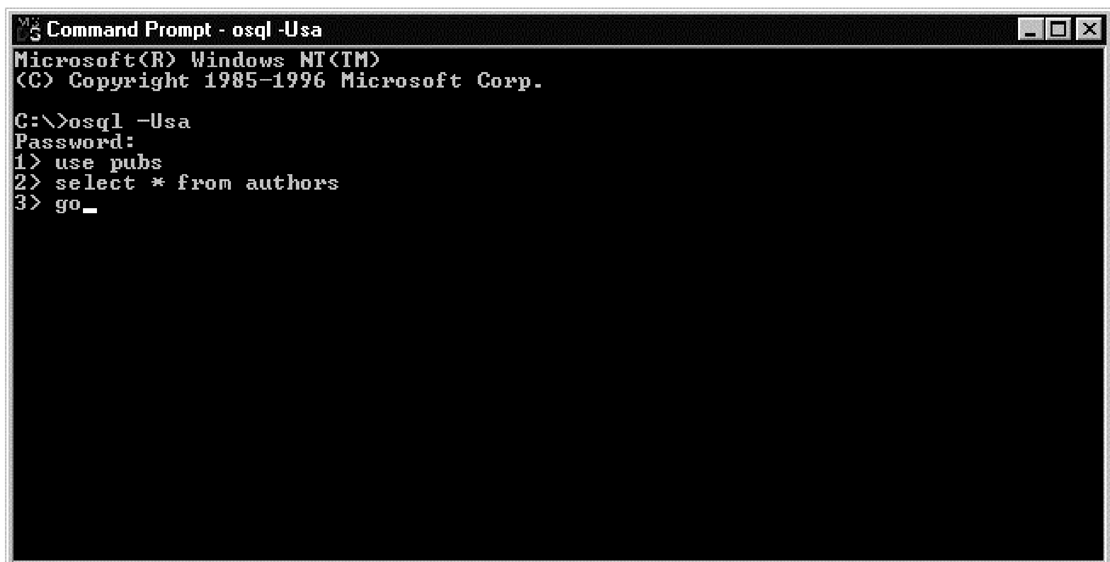


*Transact SQL commands can also be placed in a batch text file with a *.SQL suffix and executed from the OSQL window by selecting the File-Open menu and then executing the query. A batch file may have several sets of SQL statements executed with a "GO" command.*

OSQL

OSQL is a command line utility which interacts with the database server from the DOS command prompt. It can be used to run jobs on the server from a windows standard batch file.

The following example logs onto the server as system administrator to query the authors table in the pubs database. The OSQL window is normally used to run a stored procedure.



4. SQL Syntax

SQL (Structured Query Language) has four main commands for manipulating data:

- SELECT existing data from one or more tables.
- INSERT a new record into a table.
- DELETE one or more records from a table.
- UPDATE existing records in a table.

This section explains these commands with particular reference to the flexibility of the SELECT command. The Query Analyser available from the TOOLS menu of the SQL Executive should be used to run the examples against the PUBS database.

pubs

The pubs database is installed together with SQL Server and is used in most of the examples in this book. This small database contains data referring to book publishers and the titles they publish alongside details of the authors that write the titles and the stores that sell them.

TABLE	PRIMARY KEY	DESCRIPTION
AUTHORS	AU_ID	AUTHOR name and address details.
DISCOUNTS	STOR_ID	Discounts for each STORE.
EMPLOYEE	EMP_ID	EMPLOYEE detail with a link to the JOB description and the PUBLISHER employer.
JOBS	JOB_ID	JOB Descriptions
PUBLISHERS	PUB_ID	PUBLISHER name and address detail.
PUB_INFO	PUB_ID	Image and Text information on the PUBLISHER.
ROYSCHED	TITLE_ID	Royalty information for each TITLE.
SALES	STOR_ID TITLE_ID ORD_NUM	Detail record of the quantity ordered of each TITLE by a STORE.
STORE	STOR_ID	STORE name and address.
TITLE	TITLE_ID	Information on the Book TITLE.
TITLEAUTHOR	AU_ID TITLE_ID	Royalty percentage for each AUTHOR involved in a TITLE.

A SQL batch program called INSTPUBS.SQL is installed with SQL Server to allow the reinstallation of a fresh PUBS database for training purposes. Please ask your system manager to install a new PUBS database if required.

SELECT Statement

The SQL SELECT statement is used to select a set of data from existing tables in the database. The syntax of the command is designed to define a set of data which includes the fields (columns) and the set of records (rows) which should be selected.

The structure of the command is as follows:

```
SELECT [ALL | DISTINCT]
   [TOP n [PERCENT] [WITH TIES]]
   select_list
   [INTO [new_table_name]]
   [FROM {table_name | view_name}[(optimizer_hints)] [ ,...n ]]
```

```
   [WHERE clause]
   [GROUP BY [ALL] expression [ ,...n ] [WITH {CUBE | ROLLUP}]]
   [HAVING clause]
   [ORDER BY clause]
   [COMPUTE clause]
   [FOR XML [ AUTO | RAW | EXPLICIT ]
```

Field List

The select list is the list of fields or expressions that are required in the selected table. These correspond to the fields in the result set:

```
SELECT au_fname, au_lname FROM authors
```

The asterisk can also be used to select all fields from a table.

```
SELECT * FROM authors
```

An alias can be given to the field name to rename the field in the result table:

```
SELECT au_lname AS surname, au_fname AS firstname FROM authors
```

Expressions can also be specified for a field expression:

```
SELECT au_lname + au_fname AS fullname FROM authors
```

Expressions can be used in the select list:

```
SELECT s.*, t.price, qty * price AS qtyprice
   FROM sales s
   INNER JOIN titles t ON s.title_id = t.title_id
```

Scalar functions can be applied to fields or expressions for more complex queries:

```
SELECT UPPER(au_lname), CAST(address + ',' + city + ',' +
   state + space(1) + zip as varchar(45))
   FROM authors
```

WHERE Clause

The WHERE clause is used to narrow down the rows selected for the result table.

```
SELECT * FROM authors WHERE au_lname = 'White'
```

AND and OR and NOT can be used:

```
SELECT * FROM authors
   WHERE (state = 'CA' or state = 'UT')
   AND (NOT contract = 1)
```

The IN clause can be used instead of OR:

```
SELECT * FROM authors
   WHERE state IN ('CA','UT')
```

The BETWEEN syntax can be used also to select between given values:

```
SELECT * FROM titles WHERE price BEWTEEN 10.00 AND 29.00
```

NULL values can be identified in a WHERE clause:

```
SELECT * FROM stores WHERE zip IS NULL
```

The TOP n clause can be used to limit the number of records returned from a SELECT command.

© REDWARE 1996,2001.

Wild Cards

Wild cards can be used in selection criteria, for example to select any AUTHORS containing the letter 'a' in the surname:

```
SELECT * FROM authors WHERE au_lname LIKE '%A%*'
```

The LIKE syntax allows a wide variety of pattern matching templates. The above example utilises the % character as a wild card symbolising any sequence of characters. Angle brackets are used to define a range of characters. The following example to picks out book titles with an identifier beginning with a character in the range B to M:

```
SELECT * FROM titles WHERE title_id LIKE '[B-M]%'
```

The underscore character indicates any single character, # any single digit, angle brackets picks any single character within the brackets, and **`[^]`** will pick any character not within the angle brackets. This example selects Titles which do not have P or M as the first letter of the identifier and have 1 as the third character:

```
SELECT * FROM titles
   WHERE title_id LIKE '[^PM]_1%'
```

An escape character can be defined to allow one of the wild cards to be used as a literal in the expression. This example finds any occurrence of the % character in the PAYTERMS field of the STORES table:

```
SELECT * from sales
   WHERE payterms LIKE '%\%%' ESCAPE '\'
```

> *SQL Server is often configured not to be case sensitive. You may need to check this option or use expressions of the form* ***`WHERE UPPER(au_lname) = 'SMITH'`****.*

FROM Clause

The FROM clause specifies which tables are involved in the SELECT statement. The clause is mandatory:

```
SELECT * FROM authors
```

If more than one table is required in the query then they may be separated by commas:

```
SELECT * FROM titleauthor, authors, titles
   WHERE titleauthor.au_id = authors.au_id AND
   titleauthor.title_id = titles.title_id
```

The tables may be assigned an alias if required to shorten or provide an alternative name in the statement:

```
SELECT * FROM titleauthor ta, authors a, titles t
   WHERE ta.au_id = a.au_id AND
   ta.title_id = t.title_id
```

The alias name also allows for a recursive query that uses the same table twice to show an employees manager for example:

```
SELECT employee.surname, manager.surname ;
   FROM employee, employee manager ;
   WHERE employee.manager = manager.id
```

SQL Server uses a fully qualified name to identify a table. There are four parts to the fully qualified name: **`server.database.owner.table`**. The FROM clause can specify a table that is located in another database or even another table:

```
USE northwind
```

```
SELECT * FROM pubs.dbo.authors
```

ORDER BY

The ORDER BY clause allows for a result table to be ordered in any desired sequence:

```
SELECT * FROM authors ORDER BY au_lname, au_fname
```

The order sequence may also refer to the output fields in the result table using a number indicating the sequence of the field in the select list:

```
SELECT au_lname, au_fname FROM authors ORDER BY 2,1
```

The DESC keyword may be used to reverse the sort order of a column in the ORDER part of the SELECT statement:

```
SELECT title, ytd_sales, type as category
   FROM titles
   WHERE type = 'business'
   ORDER BY 2 DESC
```

Natural Join

WHERE clauses are often used for joining tables together using the primary and foreign keys. Relational databases allow for any two tables to be joined together with an expression in the first table matching any expression in the second table as long as the width and data type of the expression is identical.

Joins allow considerable flexibility to the programmer in joining tables together although, nearly always, the programmer will want to join one table to another using the foreign and primary keys.

```
SELECT * FROM titleauthor ta, authors a, titles t
   WHERE ta.au_id = a.au_id AND
   ta.title_id = t.title_id
```

The more modern syntax for a natural join is to use the INNER JOIN syntax as follows:

```
SELECT * FROM titleauthor ta
   INNER JOIN authors a ON ta.au_id = a.au_id
   INNER JOIN titles t ON ta.title_id = t.title_id
```

> *Specifying two tables in a SELECT statement without specifying a join condition will create a Cartesian product of both tables. This means that a 100 record table joined to a 200 record table with no WHERE clause will create a 20,000 record result table.*

GROUP BY Clause

The GROUP BY command can be used with the aggregate functions to count up the number of occurrences of a value or to summate, average or perform statistical calculations on a table:

```
SELECT title_id, SUM(qty) AS totalsales
   FROM sales  GROUP BY title_id
SELECT title_id, COUNT(*) FROM sales GROUP BY title_id
```

The following example shows the maximum, minimum and average order level for each title together with the total number of records for each title and a count of the number of different stores ordering the title:

```
SELECT title_id, COUNT(*) AS ordercount,
   COUNT(stor_id) AS storecount,
   MAX(qty) AS maxqty,
```

```
   MIN(qty) AS minqty,
   AVG(qty) AS avgqty,
   SUM(qty) AS sumqty
   FROM sales
   GROUP BY title_id
```

The ALL keyword can be used to include all the groupings present in the table even if there are no occurrences selected in the query. The aggregated fields for the additional groups are set as NULL values:

```
SELECT title_id, SUM(qty) AS totalsales
   FROM sales
   WHERE YEAR(ord_date) = 1994
   GROUP BY ALL title_id
```

The CUBE and ROLLUP options on the GROUP BY command are specific to SQL Server and add additional summary records into the selection. This can help in creating results sets for complex management report.

Another aggregation command is the COMPUTE BY clause and remember that the SQL OLAP manager provides full management reporting facilities.

HAVING Clause

The WHERE clause filters the rows that are used in the query. The HAVING clause operates on a query that employs a GROUP BY clause but only after the grouping has been performed.

This allows the summary records to be selected on the basis of their aggregated values. The procedure is similar to performing a second WHERE selection on the final results table.

The following statement selects titles that have sold more than 50 copies:

```
SELECT title_id, COUNT(*) AS ordcount,
   FROM sales
   GROUP BY title_id
   HAVING ordcount > 50
```

DISTINCT

The DISTINCT clause is not often used but may be used to prevent duplicate rows from appearing in the results table.

The following command creates a results table with one record for each Title Type in the TITLES table:

```
SELECT DISTINCT titles.type FROM titles
```

> *A similar result may be obtained with the GROUP BY clause.*

Inner (Natural) Join

A natural join is the operation that joins tables together using a where clause or the more modern INNER JOIN syntax. The following statement will create a view that joins the TITLES and SALES tables:

```
SELECT t.title_id, t.title, SUM(s.qty) AS totalqty
   FROM titles t
   INNER JOIN sales s ON t.title_id = s.title_id
   GROUP BY t.title_id, t.title
```

© REDWARE 1996,2001.

A natural, or inner, Join discussed above will only display records that qualify the join condition and that occur in both tables. A title that has no sales will not be included in the view.

Outer Join

An outer join allows for records to be displayed from either table even if there is no corresponding record on one or other side of the join. The Outer Join may be a left or right outer join depending on whether all the records in the first or the second table are required. A full outer join will include all records from both tables.

The syntax for a left outer join between the TITLES and the SALES tables allows all TITLES to be displayed:

```
SELECT t.title_id, t.title, SUM(s.qty) AS totalqty
   FROM titles t
   LEFT OUTER JOIN sales s ON t.title_id = s.title_id
   GROUP BY t.title_id, t.title
```

Missing values where there are no corresponding SALES records for a Title are represented as NULL values.

> *A LEFT outer join includes all records from the table mentioned in the FROM clause, the RIGHT outer join includes all records from the joined table, and a FULL outer join includes all the records from both tables.*

Sub Queries

Subqueries can be useful in creating views with complex selection criteria. The following example could be expressed as a normal JOIN and GROUP BY but is more clearly expressed as follows:

```
SELECT * FROM titles WHERE title_id IN
   ( SELECT title_id from sales
      GROUP BY title_id
      HAVING SUM(qty)> 25  )
```

Subselections are particularly useful when working with views or temporary sets of data and can be used to check if records are in or not in another table

```
SELECT * FROM titles t
   WHERE title_id NOT IN
   (SELECT title_id FROM sales s
   WHERE t.title_id = s.title_id )
```

This same query could use the less specific NOT EXISTS clause:

```
SELECT * FROM titles t
   WHERE NOT EXISTS
   (SELECT title_id FROM sales s
   WHERE t.title_id = s.title_id )
```

> *A join may always be expressed as a subquery*

UNION

The UNION command can be used to create a single view from two tables with a similar structure. The following example creates a single table from the authors and employee tables:

```
SELECT
```

```
   a.au_id AS cid, a.au_lname AS lastname, a.au_fname AS fname
   FROM authors a
   UNION
   (SELECT e.emp_id AS cid, e.fname AS firstname,
   e.lname AS lastname FROM employee e)
```

Duplicates are removed from the resulting query unless the UNION ALL keyword is specified.

> *Care needs to be taken where the table structures are not identical The CAST or CONVERT scalar functions can be used to change a data type in a SELECT statement.*

FOR XML *mode* [, XMLDATA] [, ELEMENTS][, BINARY BASE64]

SQL Server 2000 allows for rapid creation of XML from standard select statements. This is useful in creating components that use XML to communicate information:

```
SELECT * FROM authors
   FOR XML AUTO
```

The modes are as follows:

- AUTO defines an element with the same name as the table for each record and represents fields at attributes.
- RAW uses an element names <row> instead of the element named after the table.
- EXPLICIT allows precise definition of the XML tree.

XMLDATA specifies full data type information using an external schema.

The ELEMENTS clause is used together with AUTO to include the columns of the select statement as sub-elements instead of attributes in the XML.

```
SELECT * FROM authors FOR XML AUTO, XMLDATA, ELEMENTS
```

> *SQL Server 2000 has additional commands that allow a stored procedure to read an XML file (OPENXML).*

SELECT .. INTO

A new table can be created with a SELECT INTO command provided that the user has CREATE TABLE permissions on the database.

```
SELECT *
   INTO contractauthor
   FROM authors
   WHERE contract = 1
```

INSERT Statement

The INSERT statement allows new records to be added into a table:

```
INSERT [INTO]
   {table_name | view_name} [(column_list)]
   {DEFAULT VALUES | values_list | select_statement}
```

The INSERT statement requires that the values satisfy any validation constraints specified on the table otherwise the transaction will fail.

```
INSERT INTO authors
   (au_id, au_lname, au_fname, contract )
```

```
   VALUES ( "999-99-9001", "Crook", "Stamati", 1 )
```

There are constraints defined on the Authors table that will prevent a new record from being added if the identifier is not unique or if the firstname, lastname, or contract field values are not specified.

The INSERT command may also be used in combination with a SELECT statement to add records into a table:

```
INSERT INTO bestseller
   ( title_id, qty )
   SELECT title_id, SUM(qty) FROM sales s
   GROUP BY title_id
   HAVING SUM(qty) > 25
```

UPDATE Statement

The UPDATE statement allows values in existing records to be changed:

```
UPDATE {table_name | view_name}
SET [{table_name | view_name}]
   {column_list
   | variable_list
   | variable_and_column_list}
      [, {column_list2
         | variable_list2
         | variable_and_column_list2}
         ...    [, {column_listN
                | variable_listN
                | variable_and_column_listN}]]
[WHERE clause]
```

The Update clause can be used to update any field and usually involves a WHERE clause. Take care to specify the WHERE clause carefully or all the records will be updated:

```
UPDATE authors
   SET au_lname = 'Crank',
      au_fname = 'Stanley'
   WHERE au_lname = 'Crook'
```

The WHERE clause is often used in conjunction with the Primary Key expression to update a single record in the table.

The UPDATE command can also set values into a table by making calculations using data from another table:

```
UPDATE sales
   SET qtyprice = qty * (SELECT price
      FROM titles t WHERE sales.title_id = t.title_id )
```

DELETE Statement

The DELETE statement allows for deletion of table records using a WHERE clause to specify the records for deletion:

```
DELETE [FROM] {table_name | view_name}
```

© REDWARE 1996,2001.

```
   [WHERE clause]
```

The DELETE statement must satisfy any referential integrity constraints set up in the database before records are deleted:

```
DELETE authors
   WHERE au_lname = 'Crook'
```

> *The* **`TRUNCATE TABLE authors`** *command could be used to delete all the records in the table. Take care to backup after such a command because a Truncated Delete is not logged.*

© REDWARE 1996,2001.

5. Database Definition

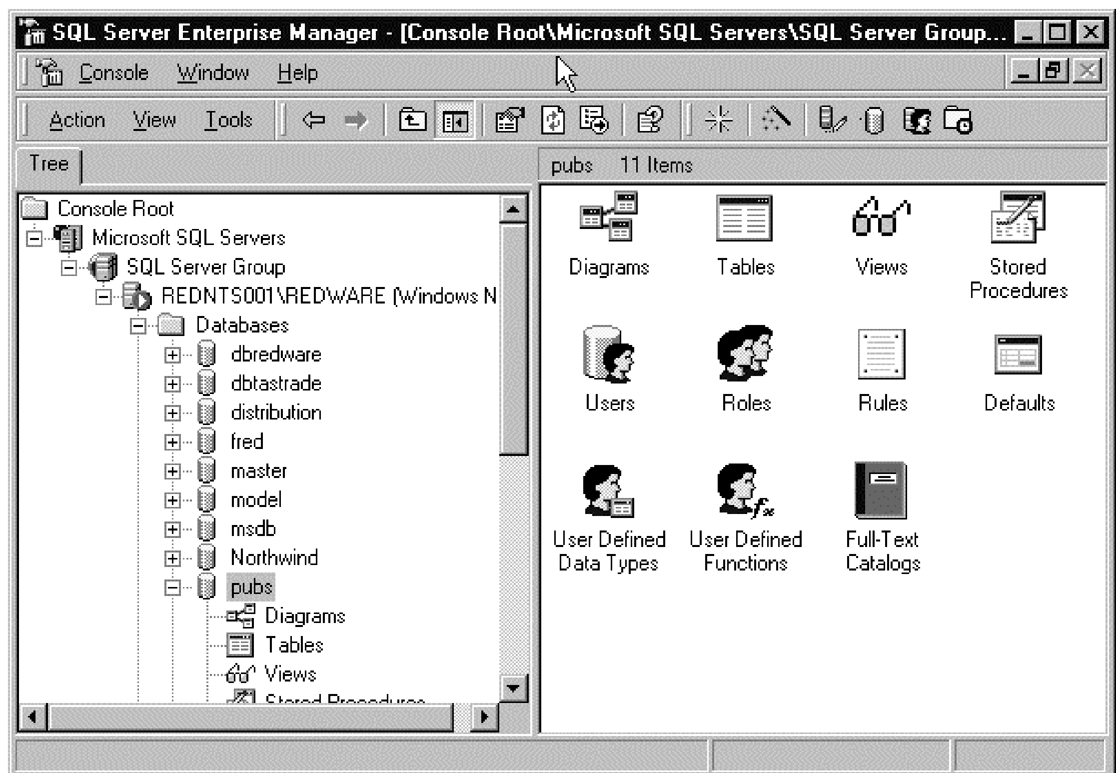
This section introduces the Enterprise Manager as a means of defining a database and its constituent tables. The following procedures are covered in detail:

- Create a new database and set database parameters.
- Create a Table and constituent fields.
- Define additional properties on a field.
- Add default and check constraints to a field.
- Define a primary key.
- Define a foreign key and set up referential integrity constraints between tables.
- Create a user defined data type.
- Generate a SQL Script for the database objects.

Enterprise Manager

The Enterprise manager can be used to manage a number of SQL servers throughout the enterprise. Most options are made available by navigating the tree structure of the Enterprise Manager and right clicking on the desired option.

Enterprise Manager



Each installation of SQL Server has several system databases:

- MASTER contains many configuration details of the server including details of the schema for each database. It is important that this database is backed up whenever changes are made to the structure of any of the databases as it is very difficult to repair a system if the MASTER database is missing.
- The MODEL is used as the template to create a new database.
- MSDB is used by the SQL Agent for holding details of scheduled jobs created to do housekeeping tasks such as backing up databases.

© REDWARE 1996,2001.

- TEMPDB is used for temporary storage during everyday use of the database. This includes temporary tables created in stored procedures and other working tables.

> *If you are working with large amounts of data you will want to increase the size of the TEMPDB database.*

Create a Database

SQL Server is usually installed on a NT Server machine which runs SQL Server programs as a network service. The Client workstation makes a request to the SQL Server service running on the NT Server which performs the database retrieval before returning the required data to the workstation.

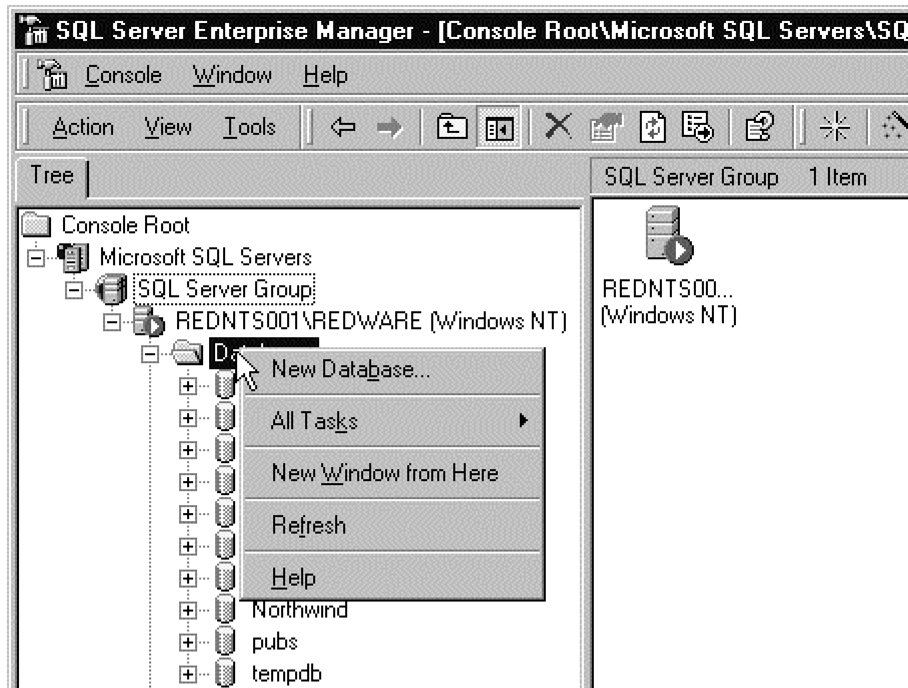
The database files are usually stored on the server machine and a database needs to be set up to hold the physical files for each database application. This task is often performed by the database administrator so that a new empty database is provided for the programmer to configure.

Each database has two physical files. One contains the data and the other contains the log. Data is written to the database and a record of each transaction is made in the log file. If the database file becomes corrupted, it can be restored from a backup and the transaction log rolled forward to restore the database.

> *It is recommended that the Database and the Log are stored on separate physical devices. This allows the database to be recreated from a backup and the transaction log if the physical disk containing the database files is corrupted.*

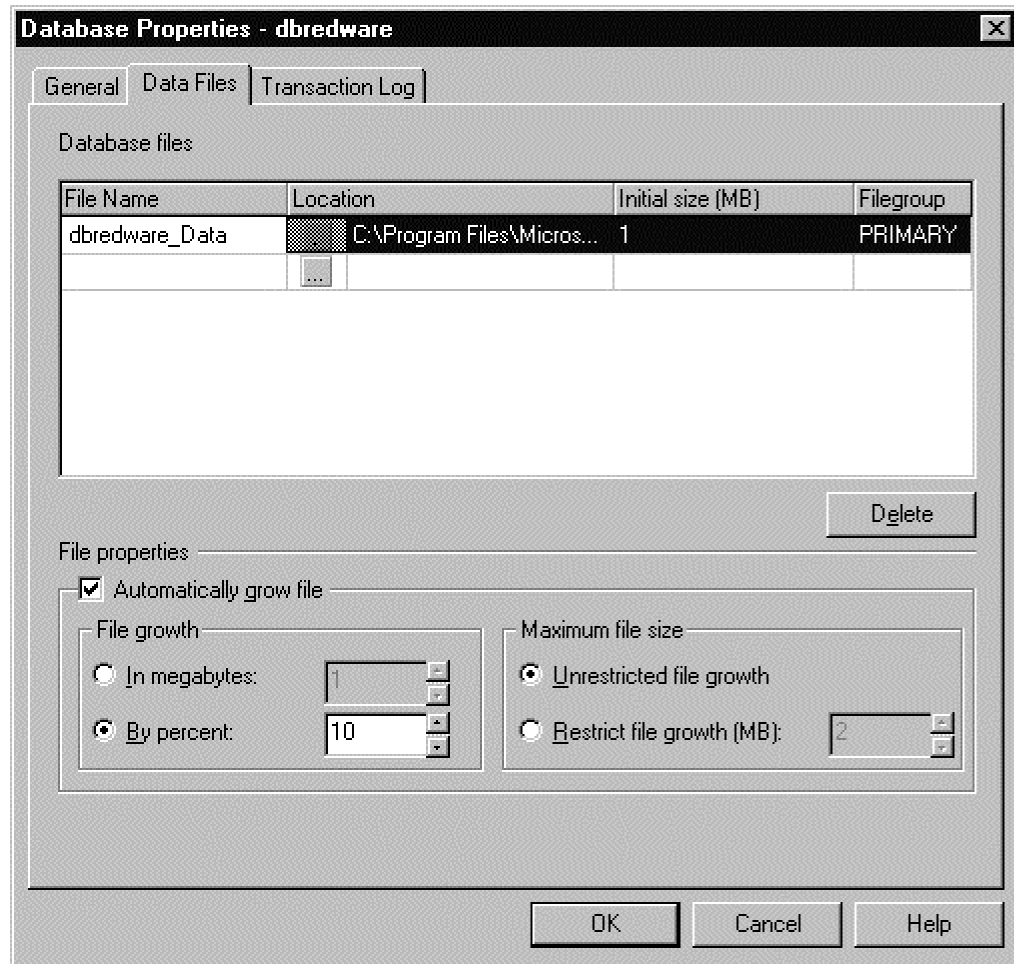
A new database is created by right clicking on the databases option in the tree view of the enterprise manager and selecting the New Database.. option (also available from the Action menu option).

Create a New Database



Provide an appropriate size for the database and preferably define a maximum size. Allow the database to grow by a reasonable amount each time so the system is not constantly redefining the database size. Place the transaction log on a separate physical disk dive if possible. The log is usually 10-15% of database size.

Create a new database



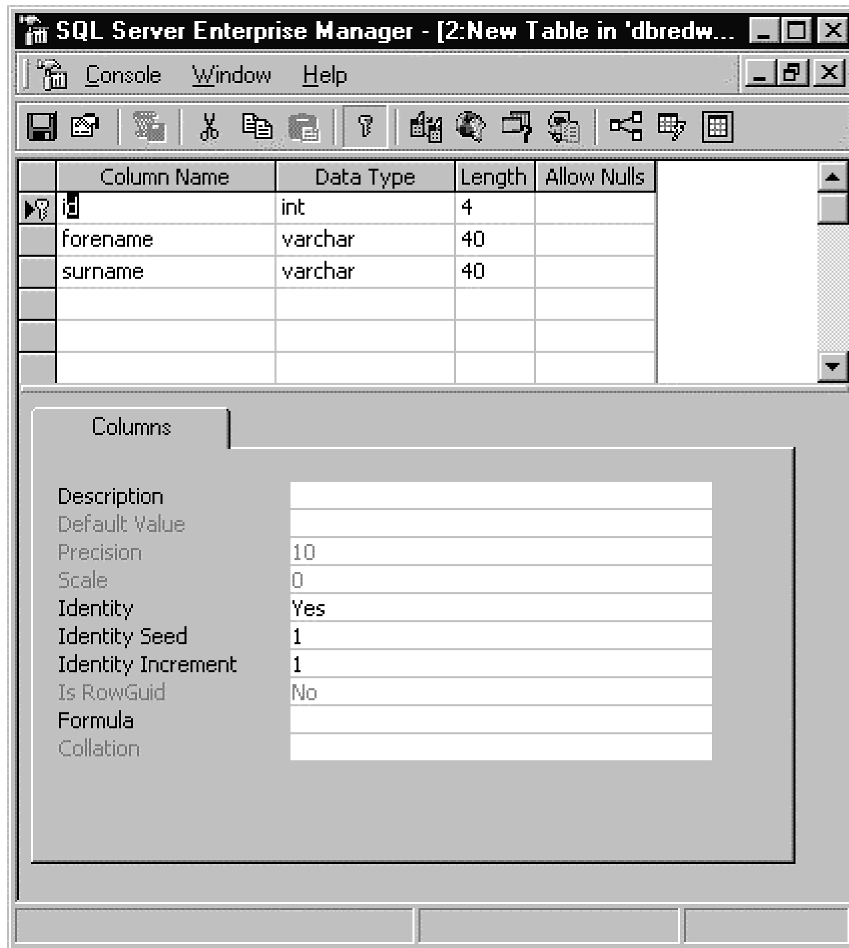
*The **CREATE DATABASE** command can also be used to create a new database.*

It is advisable to backup the MASTER database each time changes are made to any database on the server.

Create a Table

Tables are created by expanding the tree view for the database and right clicking on the TABLE tree to select the NEW TABLE option.

Create a Table with SQL Enterprise Manager



Data Types

SQL Sever has predefined data types one of which must be used for each field:

- Integer: int, smallint, tinyint, bigint.
- Exact Numeric: decimal, numeric.
- Approximate Numeric: float, real.
- Money: money, smallmoney.
- Character data: char(n), varchar(n).
- Binary data binary(n), varbinary(n).
- Date and time: datetime, smalldatetime.
- Text and Image text, image.
- Other: bit, timestamp, sql_variant, user defined.
- Unicode: nchar(n). nvarchar(n), ntext.
- Identifier uniqueidentifier

> *SQL Server field types are copied from the MODEL database each time a new database is created and may be added to by defining user defined data types.*

Selection of the correct data type is very important and requirements will vary according to the business requirements of the application:

- A market research database may have very large amounts of data and numeric field types should be selected with a view to the storage requirements. A SMALLINT integer field will need less space than a FLOAT field for example. Be careful when using TINYINT however as the maximum allowable value is 255.
- Numeric accuracy is important in financial applications and allowance should be given for the requirement to store fractions as decimals particularly for stock market applications. Some fractions may require many decimal places to be stored accurately. DECIMAL is often the most precise data type for numerical values.
- The MONEY data type is normally the best for storing currency values
- The VARCHAR data type allows character data of variable length to be stored at the cost of an extra bit for each value to store the width of the field. VARCHAR is not appropriate for very short field lengths or for fields where the width is relatively constant throughout the table (a single byte is used to store the length of each value).
- TEXT and IMAGE fields store data in 2K chucks by default to a maximum of 8,000 bytes. It is recommended that Null values are permitted if there are a large number of empty values in the table. In some applications it may be more efficient to store the data as external files and use network protocols to access them.
- Unicode data types use two bytes for each character and allow a variety of international characters to be stored.

Nulls and Defaults

Properties can be defined against each field. It is recommended that the ALLOW NULLS option is deactivated for each field to prevent problems when inserting records. A default value for each field should be defined in the properties for the field as shown below.

Field Properties

Columns

Description	
Default Value	('')
Precision	0
Scale	0
Identity	No
Identity Seed	
Identity Increment	
Is RowGuid	No
Formula	
Collation	<database default>

Other options allow the precise format of numeric fields to be set and unique identifier functions to be set on candidate primary leys for the table.

FORMULA is used to set up a calculated field and COLLATION is used to set a specific sort order on the field. These options are rarely used.

Table Ownership

Tables are referred to with a four part qualifier:

```
SELECT * FROM servername.database.owner.table
```

The qualifiers can normally be omitted depending on the context. The following code ensures that the context is the PUBS database and so the database need not be qualified:

```
USE pubs
SELECT * FROM authors
```

Selecting information from a different database required that at least the database is qualified:

```
USE master
SELECT * FROM pubs..authors
```

Tables can be created by various database users and ownership is recorded for the table. This could lead to a situation where there are two tables with the same name but with different owners. The owner qualifier would need to be specified each time the table is referenced and this could lead to considerable confusion.

A system stored procedure can be used to change the ownership of an object as follows:

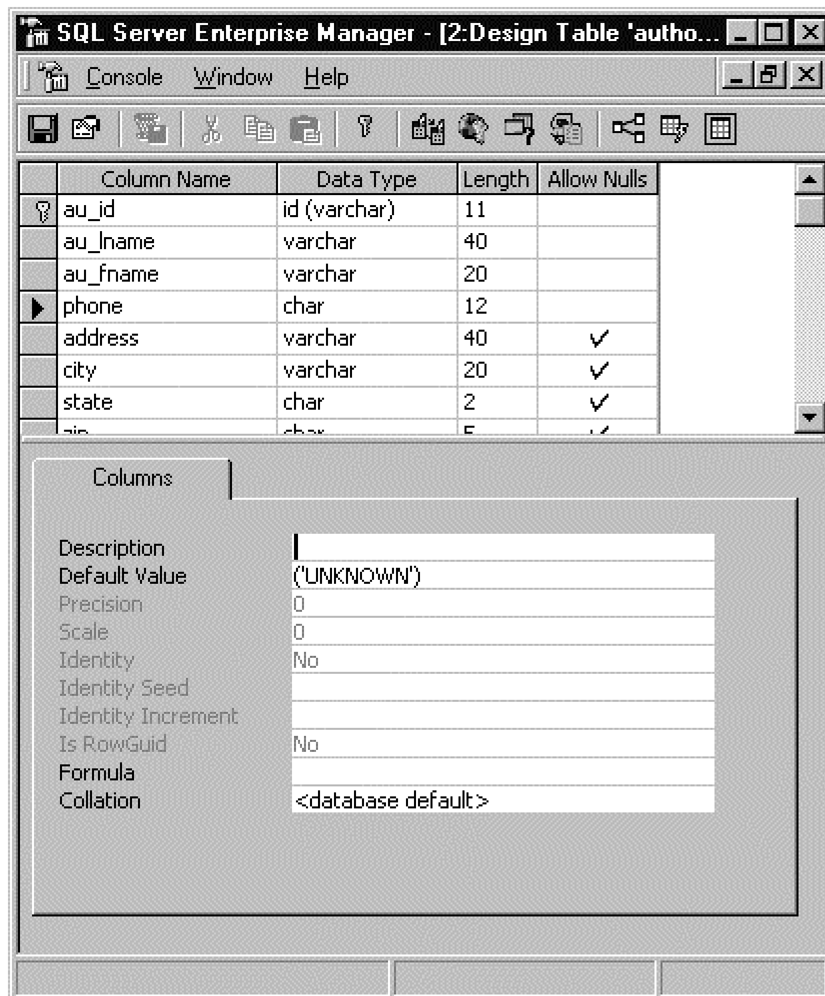
```
EXECUTE sp_changeobjectowner authors, dbo
```

> *It is recommended that all objects in a database are owned by the database owner: dbo.*

Field Properties

Fields are defined by selecting a Table and right-clicking on the DESIGN TABLE..option to bring up the DESIGN TABLE window.

Table Designer



> *There is a maximum of 1,024 fields per table with SQL 2000 and a maximum of 8,060 bytes per record.*

© REDWARE 1996,2001.

New fields are added by entering the new field definition at the bottom of the window. The name of existing fields can be changed and their widths altered by changing the appropriate values. Be careful when shrinking the field size as data may be lost.

Defaults can be specified for a field by entering a constant in the column for defaults against the field. These are constant values specified for the individual field only.

Pressing the Save Table button saves any changes to the table. Definitions of keys and other table settings can be displayed with the Advanced Features option later in this chapter.

Null Values

Relational database theory differentiates between a null value for a field and a zero value. SQL Server can determine whether a value has never been entered against a numeric value (a null) or whether a zero has been entered by the user.

Null values are important in relational theory but can cause problems in application development if not used correctly. Arithmetic and Boolean operation on Null values can yield unexpected results.

Specifying that nulls are not allowed will fail an insert transaction that attempts to add a record without a specific value entered against particular fields. This is useful for forcing entry of numeric values, foreign key or lookup fields, and status flags.

> *Null fields often cause confusion and it is a good idea to specify fields not to allow nulls and to specify a default value.*

Default Constraints

Defaults may be defined against a field to automatically enter a value when a new record is inserted into the table if the application has not entered a value.

Defaults can be useful with fields that are defined as NOT NULL as SQL Server will supply a value for the field if not specified by the application.

Defaults may be set by typing a value in against the DEFAULT field property when defining a field with the DESIGN TABLE window. The default must be specified as a constant value.

Specifying a Default Value

Columns

Description	
Default Value	('UNKNOWN')
Precision	0
Scale	0
Identity	No
Identity Seed	
Identity Increment	
Is RowGuid	No
Formula	
Collation	<database default>

Defining a default value with the DESIGN TABLE window will automatically create a default constraint on the table. A field default constraint can also be added to (or dropped from) a table with a SQL data manipulation command as follows:

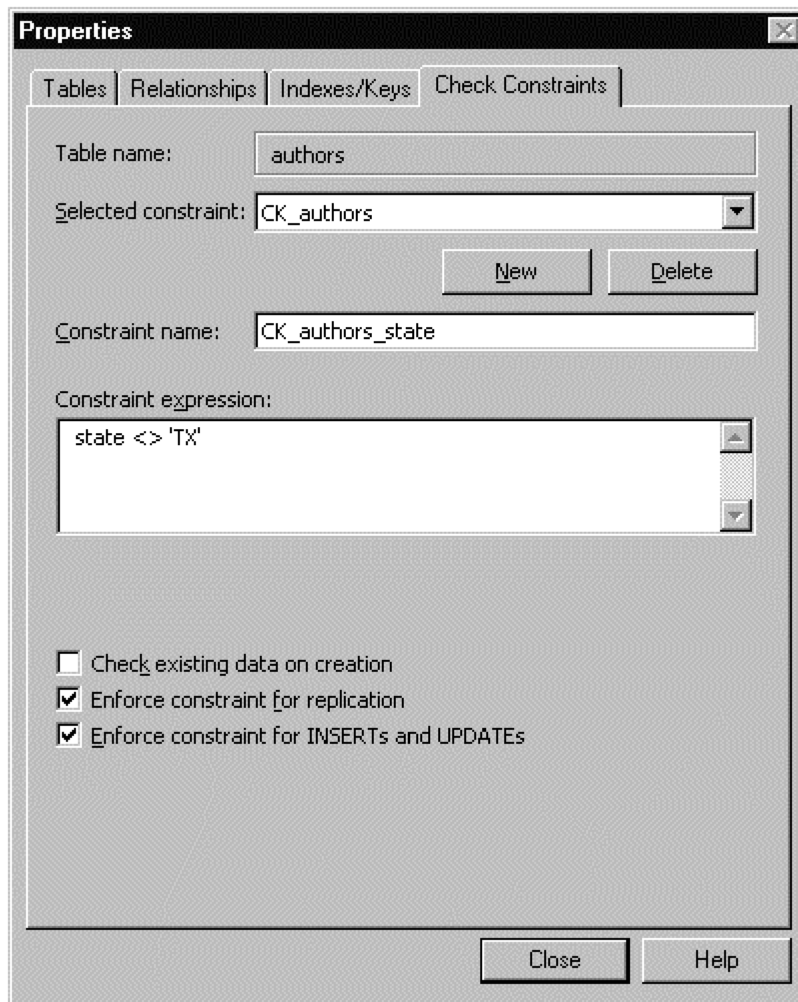
```
ALTER TABLE [authors] WITH NOCHECK ADD
   CONSTRAINT [DF_authors_state] DEFAULT ('CA') FOR [state]
```

Check Constraints

Check Constraints allow the definition of a simple piece of logic to check the values to be entered in a field. The AUTHORS table could be altered to prevent further entry of any authors from Texas by setting a constraint which prevented the state field from being set to TX.

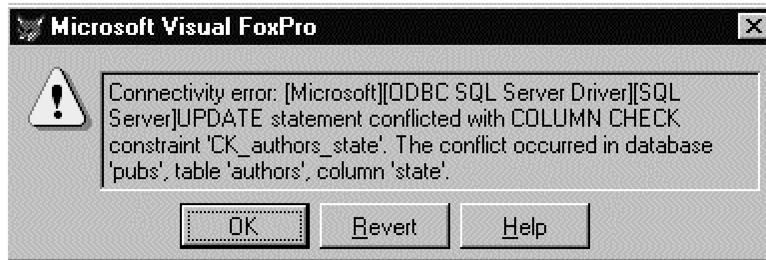
Check constraints may be entered with the SQL Enterprise Manager by selecting the appropriate Table object and rightclicking to edit the table. Pressing the ADVANCED FEATURES push button will bring up a Page which allows various constraints to be set. The Check Constraints page allows for simple checks to be made on the data.

Specifying a Check Constraint for a Table



SQL Server will not allow a value to be entered or modified that conflicts with the check constraint and will generate an appropriate error message if an attempt is made to violate the constraint:

Error Generated by SQL Server when the Constraint is violated



> *Constraints are new with SQL Server 6.0 and replace the previous notion of defining Defaults that are bound to fields. They are compatible with the latest SQL ALTER TABLE syntax.*

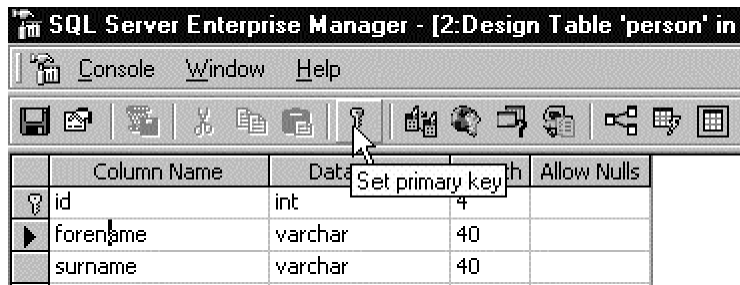
Constraints can also be added with a SQL script by utilising the ALTER TABLE command:

```
ALTER TABLE authors
   ADD CONSTRAINT CK_authors_city CHECK( city<>'gotham')
```

Create a Primary Key

Primary keys can be created easily from the Table design window by using selecting the required field and using the SET PRIMARY KEY button represented by the key shape at the top of the table design window. Multiple fields may be selected to create a composite Primary Key.

Set the Primary Key in the Table Design Window



SQL Server automatically defines the required primary key index. The penultimate button on the right of the Table Designer toolbar allows the user to MANAGE INDEXES/KEYS.. and can be used to refine the primary key definition.

> *Fields that allow Nulls may not be specified as Primary Keys.*

Identity Columns

Field properties can be used to define Integer fields as an Identity column which is automatically incremented each time a new record is added to create a unique value suitable as a primary key on a table. An initial seed value and an increment can also be set.

Specifying Identity Column Properties

Columns

Property	Value
Description	
Default Value	
Precision	10
Scale	0
Identity	Yes
Identity Seed	1000
Identity Increment	1
Is RowGuid	No
Formula	
Collation	

Identity columns are typically used as primary keys and have the advantage of being small in size and therefore fast for SQL Server to use when joining tables.

Identity columns also provide a degree of data independence and can be a better design option than alternate candidate keys. An employee table, for example, has both the Staff Identifier and the National Insurance Number of the person as candidate keys. However the person cannot be added into the table until a Staff Identifier has been allocated causing problems in entering data before the person actually starts work. Similarly a National Insurance Number may not be immediately available or may change, forcing key values to be cascaded down any dependent tables. An independent identity column is a better primary key than either of the obvious candidate keys.

There are some useful system functions for obtaining information about identity columns.

- The @@IDENTITY system variable will indicate the value of the primary key of the previously inserted record.
- IDENT_CURRENT('person') will return the latest identity column value for the person table. Similarly IDENT_SEED and IDENT_INCR will return the seed and increment for the required table.
- DBCC CHECKIDENT('person') will check that the values in the identity column are correctly defined for the table.
- SET IDENTITY_INSERT OFF is required when inserting records where the value of the identity column is already known.

Only one identity column is allowed for each table and can be specified in a select statement as follows:

```
SELECT IDENTITYCOL, surname FROM person
```

Unique Identifiers

There are some problems with Identity Column keys particularly if a table is distributed over several servers and needs to be replicated. The UNIQUEIDENTIFIER data type has similar properties but is a 16 character globally unique identifier that is automatically defined by setting the default value to NEWID() in the field properties.

Unique Identifiers can be represented with the field name or the ROWGUIDCOL keyword in a SELECT command:

```
SELECT ROWGUIDCOL, surname FROM person
```

> *UNIQUEIDENTIFIER fields are larger than integer fields and care should be taken using them on very large tables.*

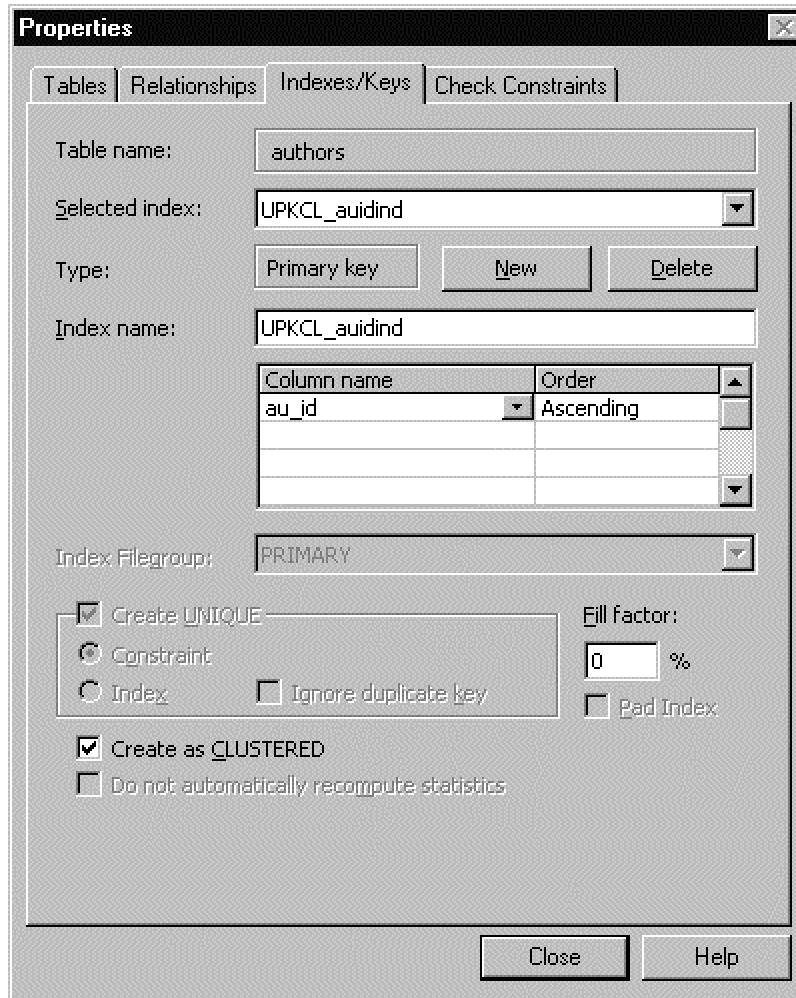
© REDWARE 1996,2001.

Primary Key Constraint

An index and primary key constraint is automatically created when a primary key is defined using the Table Designer. The MANAGE INDEXES/KEYS.. button on the right of the Table Designer allows all indexes, including primary keys, to be defined in more detail.

The Primary Key is comprised of one or more fields that do not allow null values. Composite Keys can be defined by selecting a second column for the Primary Key. Select the Clustered option if the table is to be physically ordered in the sequence of the Primary Key.

Defining a Primary Key with the Manage Indexes/Keys..Window



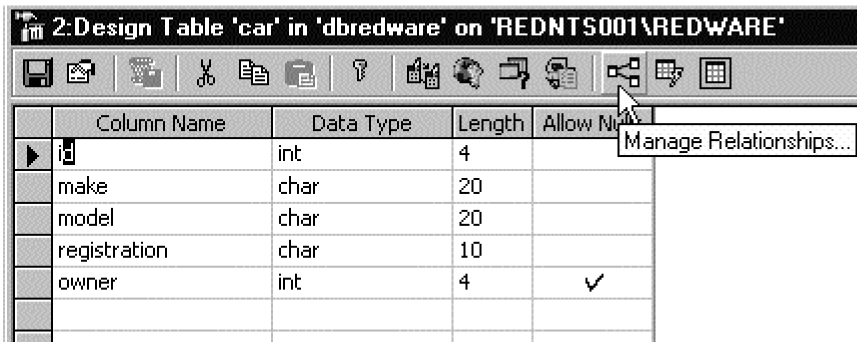
Primary Key constraints can also be defined using SQL Server data manipulation language which automatically creates the appropriate index:

```
ALTER TABLE person
   ADD CONSTRAINT PK_person
   PRIMARY KEY CLUSTED (id)
```

Foreign Keys and Referential Integrity

Foreign Keys are the other half of a relationship between tables and link a child table to a parent table. The Foreign Key value should match directly to the value of the Primary Key.

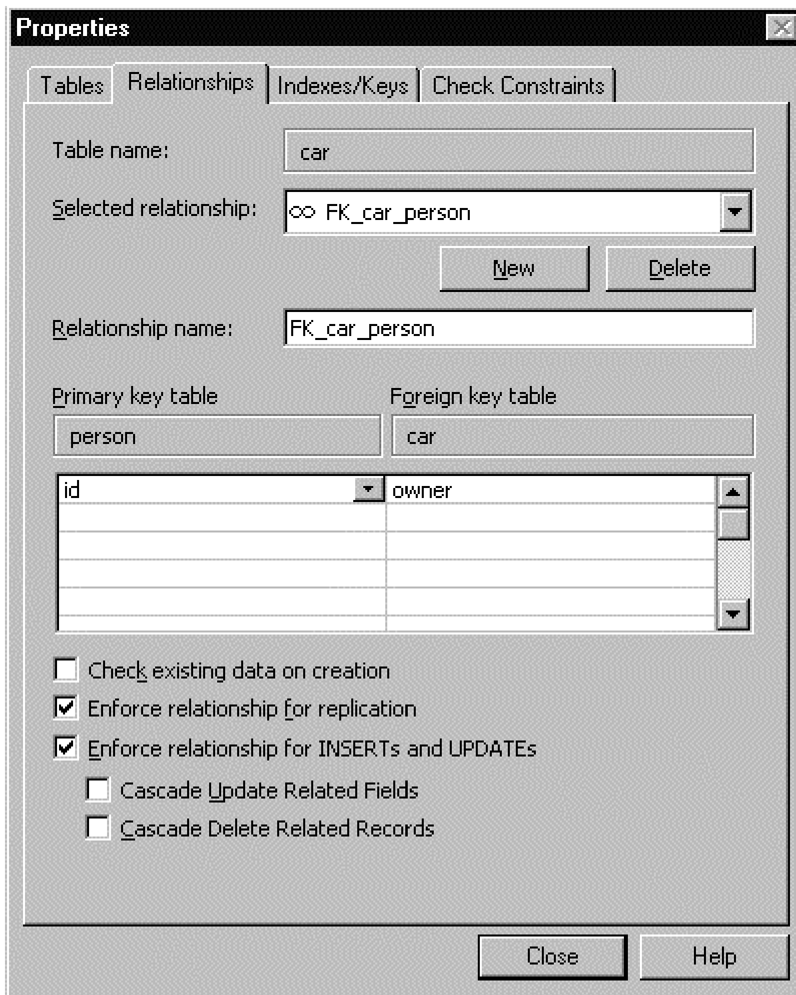
Foreign Keys can be defined in SQL Server to automatically maintain the referential integrity of the table. Select the MANAGE RELATIONSHIPS option in the DESIGN TABLE window to define a Foreign Key.



The fields for the Primary Key Table and the Foreign Key Table are entered in the respective columns on the RELATIONSHIPS page of the TABLE window to create the relationship.

Enforcing the relationship for INSERTS and UPDATES will create the referential integrity constraint so that no OWNER of a CAR can be entered without a corresponding record in the PERSON table.

Defining a Foreign Key



The OWNER field in this example allows NULL values so a CAR can be defined without an OWNER by leaving the value as a NULL.

A cascading delete will automatically delete linked records in the Foreign table if the record in the Primary table is deleted. Similarly, a cascade update allows a key value to be changed in both the Primary and Foreign tables if the value of the identifier changes.

Triggers are no longer required to perform cascading deletes

Foreign Key constraints can also be added using SQL:

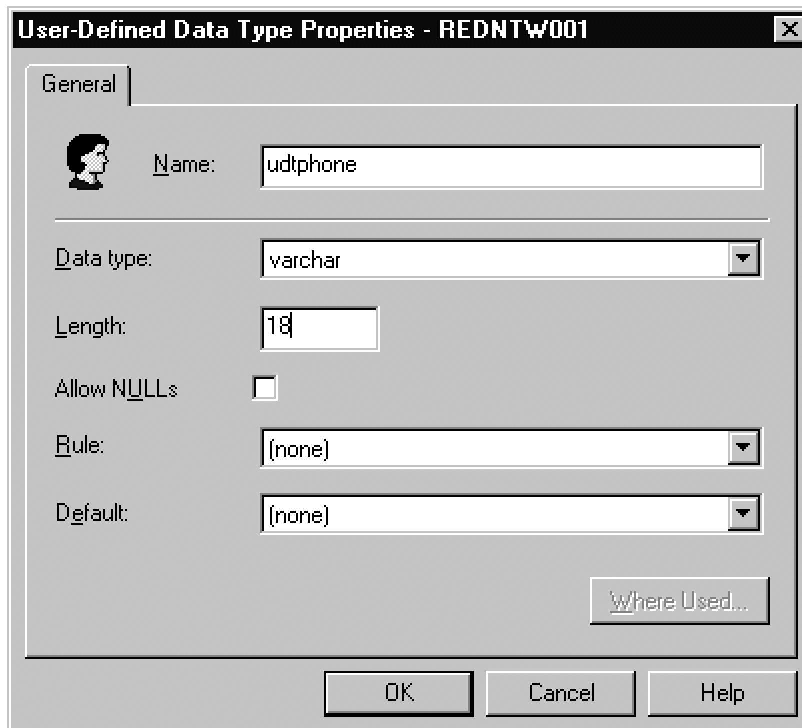
```
ALTER TABLE car ADD
   CONSTRAINT FK_car_person FOREIGN KEY
   ( owner ) REFERENCES person ( id )
   ON DELETE CASCADE
```

User Defined Data Types

SQL Server allows the definition of user defined data types within a database. These can be useful to prevent inconsistencies in large database schemas where similar fields occur many times. A user defined datatype might be defined for telephone and fax numbers, for example, to make sure all occurances where of the same width throughout the database.

The outline view of the SQL Enterprise Manager can be used to define a user defined data type. The datatype is then available for use when defining fields with the Table Designer.

User Defined Data Type Window



Alternatively, system stored procedures can be used to define or drop user defined types from the database:

```
EXECUTE sp_addtype udtphone varchar(20)
EXECUTE sp_droptype udtphone
EXECUTE sp_help udtphone
```

© REDWARE 1996,2001.

> *User defined datatypes may be defined in the MODEL database and are then automatically copied into each new database.*

Defaults and Rules

These features were in popular use in earlier versions of SQL Server but have now been replaced with the use of constraints that have the advantage of being ANSI compatible and easier to define. Their use is not recommended.

Defaults

A collection of default values may be defined independently in the database and then bound to individual fields or to a user defined data type. Unfortunately, it seems that changing the value of the default requires all the default and all bindings to be dropped and recreated. The use of defaults has been replaced by ANSI compatible default constraints.

Rules

Simple validation rules (that reference only constant values) can also be defined in the database and bound to user defined data types or directly onto a field. These are now replaced with ANSI compatible check constraints.

© REDWARE 1996,2001.

6. Indexes

Unique Index Constraint

A table may have alternate candidate keys that uniquely identify each occurrence in the table. A staff or employee table, for example, may have a Staff Identifier and a National Insurance number entered for all staff. Each of these fields is unique to each occurrence and creating a unique index constraint on the field will ensure that duplicate values do not occur.

Indexes may be defined on a single field or a set of fields as a **composite key** which may be useful for optimising queries or for sorting data for a report or batch processing tasks.

> *If several fields are often involved in a query selection is good for performance to create a composite key. If some selections only use one or two of the fields then they should be defined as the first columns of the index otherwise the index will not be selected by the query optimiser.*

```
ALTER TABLE person
   ADD CONSTRAINT IX_person
   UNIQUE NONCLUSTERED (id)
```

Clustered Index

A single **clustered** index may be defined on a table that physically sorts the records into the index order. This can speed up performance on a table if sequential access to a set of records is often needed in the sequence of the clustered index. Retrieval of individual records is not improved by a clustered index.

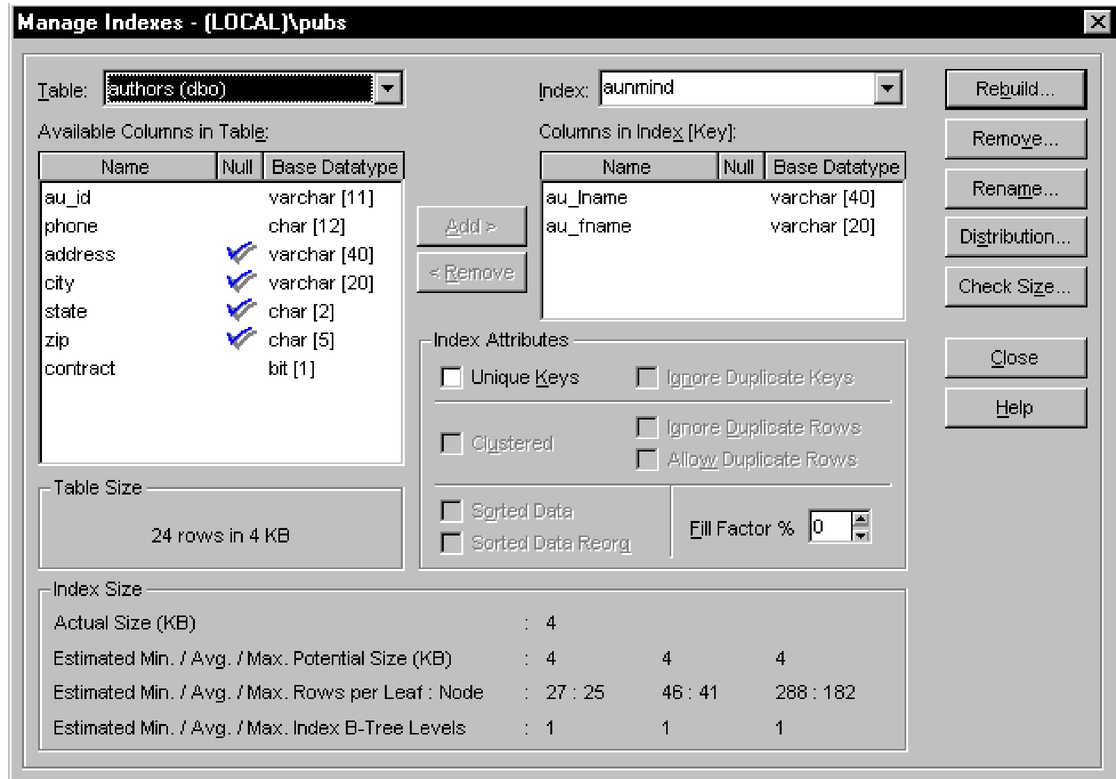
> *Take care not to cause contention problems when creating a clustered index. Many users entering new records simultaneously with similar clustered index values will cause a performance bottle-neck as they all need to access the same part of the clustered index. This occurs particularly with date or timestamp values or incrementing primary keys when they are used as clustered indexed. The situation has improved with SQL Server 6.5 but is still not recommended.*

There are more advanced options, discussed below, that can be defined against an index but are initially best avoided as they may result in part of a transaction failing with no indication to the user that some records have been ignored by the database engine.

Indexes are defined with the Manage Indexes window available by rightclicking on the required Table in the SQL Enterprise Manager and selecting the Indexes option. The fields for the Table are displayed and can be moved into the Index by clicking the Add button. More than one field can be added to create composite keys.

© REDWARE 1996,2001.

Manage Indexes Window



SQL Server indexes can be defined to ignore certain problems when inserting new records into a table. The **Ignore Duplicate Row** option causes records with duplicate rows, where a clustered index has been defined, to be ignored during a transaction without failing the whole transaction. In this case, duplicate rows are not inserted into the table but the remaining records are processed.

Similarly with the **Ignore Duplicate Key** option, attempts to insert a record with a duplicate key that has been defined as a unique index, will ignore only that record and continue with the remainder of the transaction.

The **Allow Duplicate Rows** option contradicts the principle of a primary key for each record and is required only in unusual circumstances.

> *Relational database tables should theoretically always have a unique primary key and therefore no duplicate rows. Microsoft Access, for example, will not allow updates on a table without a primary key.*

Clustered Indexes allow for the Sorted Data checkbox to be specified so processing time is not wasted sorting the Index. The Index is not created however if the data is not sorted correctly.

7. Views

SQL Views employ a SELECT statement to create a new virtual table that behaves in a similar fashion to the real tables in the database.

Views can be used to hide the complexity of the underlying database structure or to show a subset of data. They are useful in presenting summary or aggregated information to users for a Decision Support or Reporting Application. The view can be redefined if any changes are made to the underlying table structures without affecting any of the existing management reports.

Views are also useful in implementing security and performance requirements. A view can be defined to allow read/write access to a subset of data to which users are otherwise denied access.

A partitioned view allows several tables to be joined together (with the UNION command) and processing to be spread over different databases or servers for parallel processing and improved performance.

SQL Enterprise manager or the CREATE VIEW command is used to create views. Use the ALTER VIEW command to change existing views that have references made to them in stored procedures or triggers.

```
CREATE VIEW [<database_name>.][<owner>.]view_name
   [ (column[,...n])]
   [ WITH <view_attribute> [ ,...n ] ]
   AS
   select_statement
   [ WITH CHECK OPTION ]
```

The following example creates the TITLEVIEW view by joining three tables together. The view can be used in exactly the same manner as a normal table and will update the underlying tables.

```
CREATE VIEW titleview
AS
SELECT title, au_ord, au_lname, price, ytd_sales, pub_id
   FROM titleauthor
   INNER JOIN authors ON authors.au_id = titleauthor.au_id
   INNER JOIN titles ON titles.title_id = titleauthor.title_id
```

> *Care should be taken with the ownership of Views an the underlying tables. In general, it is best to have the database owner (dbo) as the owner of all views and tables. Use the sp_changeobjectowner system stored procedure to change ownership.*

Any standard SELECT statement can be used including complex queries with UNION, GROUP BY, and HAVING. An ORDER BY clause however is not allowed unless used in conjunction with the TOP clause.

> *Views with aggregate or computed fields in the SELECT syntax cannot be modified.*

The **`sp_depends viewname`** and **`sp_helptext viewname`** system procedures will display the dependent columns and the syntax of the view respectively.

© REDWARE 1996,2001.

Indexed Views

Views that contain summary information need to retrieve the underlying information each time they are used by the calling application. Creating an index on the View forces SQL Server to retrieve and permanently store the index in the database vastly improving performance.

The SCHEMABINDING option must be used on a View before indexing is permitted:

```
CREATE VIEW
CREATE INDEX…
```

Maintaining an index on a View adds an overhead and should not be used on very volatile data that is frequently updated. Careful design of the index can yield fruitful results as the new Index can be used by the query optimiser in any query even if the View itself is not involved.

Check Option

Views are a great way to provide limited access to data for selected users. A View on an Employee table may be defined without any salary details and permissions denied on the original table to simplify security access for this sensitive data.

The CREATE VIEW syntax has a WITH CHECK OPTION that prevents data being added or modified within the view that cannot subsequently be retrieved from the view.

The following example creates a view that only shows authors with contracts and will not allow an author to be added or modified without the contract field having a value of one:

```
CREATE VIEW authorscontracts AS
   SELECT * FROM authors
   WHERE contract = 1
   WITH CHECK OPTION
```

Partitioned Views

A special case of Views that UNION several tables of identical structure is known as a partitioned view. These tables can be local, within a single database, or distributed on several databases, perhaps even on different servers.

The data is usually partitioned on some logical basis such as the inclusion of a country code in the table and a check constraint is set on each table so that the query optimiser can determine which tables to look at for a typical query.

The view is then created by UNIONing all the tables and an updateable partitioned View results. The advantage of spreading each table over different databases or servers allows the query to run in parallel on multiple processors or servers and can speed performance on very large databases.

> *A partitioned view over several databases or servers, with an index, can provide very powerful parallel processing facilities for very large databases.*

OPENROWSET

SQL Server can use OleDB/ODBC middleware to connect to external datasources directly from the server. The following example uses an ODBC datasource defined on the server to connect and retrieve data from a FoxPro table:

```
select * from openrowset( 'MSDASQL',
    'DSN=dsnfoxtastrade',
    'select * from shippers where company_name like ''U%''')
```

The **OPENROWSET** command is used for ad hoc queries and is much more flexible when a connection string is used rather than a pre-defined ODBC datasource.

Linked Servers

A linked server can be defined using the Security-Linked Servers option of the SQL Executive or the **sp_addlinkedserver** system stored procedure. This defines a permanent relationship between the SQL Server and another SQL Server or external datasource.

The following example adds a linked server, called FOXTASTRADE, to the current SQL Server using an existing ODBC datasource:

```
EXECUTE sp_addlinkedserver
   @server='foxtastrade',
   @srvproduct='foxpro',
   @provider='MSDASQL',
   @provstr='DSN=dsnfoxtastrade'
```

> *The MSDASQL is the generic driver to connect to ODBC datasources. More specific drivers can be easily defined e from the Security-Linked Servers option in the SQL Executive.*

The OPENQUERY() function can be used to execute a pass through query directly on the linked server and return a result:

```
SELECT * FROM
   OPENQUERY(foxtastrade,
   'select * from category where category_name like ''B%''')
```

> *The sp_serveroption system stored procedure may be required to set the default database options for the linked server (collation sequence, etc).*

Distributed queries can also be run on linked server by using the full four part object reference:

```
select * from linkedserver01.pubs.dbo.authors
```

> *Information on the database schema contained inside a linked server can be obtained with the relevant system stored procedure: sp_linkedservers, sp_catalogs, sp_indexes, sp_tables_ex, sp_columns_ex.*

Temporary Tables

Temporary tables can be created on the server using a SQL SELECT statement. These temporary tables can be used for reporting purposes or to perform interim calculations as part of a batch process.

Temporary Tables have their name prefixed with # or ## for local and global tables respectively. Local tables are only available for the current user session or perhaps just within the scope of a single stored procedure. Global temporary tables are available to all users of the database and are deleted only when the last session that refers to the table is closed.

The following command sets up a temporary table which all users can access:

© REDWARE 1996,2001.

```
select *
   into ##contractauthor
   from authors
   where contract = 1
```

The SELECT .. INTO .. syntax may also be used to create a new permanent table provided that the SELECT INTO/BULKCOPY database option is set to True:

exec sp_dboption 'pubs', 'select into', TRUE

8. Stored Procedures

SQL Server allows the programmer to write programs which can be executed repetitively with a simple instruction to the server. The programs are called Stored Procedures and consist of a series of Transact-SQL commands with structures to control program flow, receive parameters and return values and other features representing familiar programming principles.

One advantage of Stored Procedures is that they are executed on the server, performing a series of actions, before returning the results to the client. This allows a repetitive series of actions to take place with minimum network traffic and can considerably improve performance in many cases.

Security permissions can keep the underlying data hidden from the programmer to aid in more complex security requirements. For example, a stored procedure could be used to add bank account details to the database with access to the underlying table denied to the user.

Stored Procedures also have direct access to server resources and can call programs residing on the server to integrate with other systems or parts of the computer infrastructure.

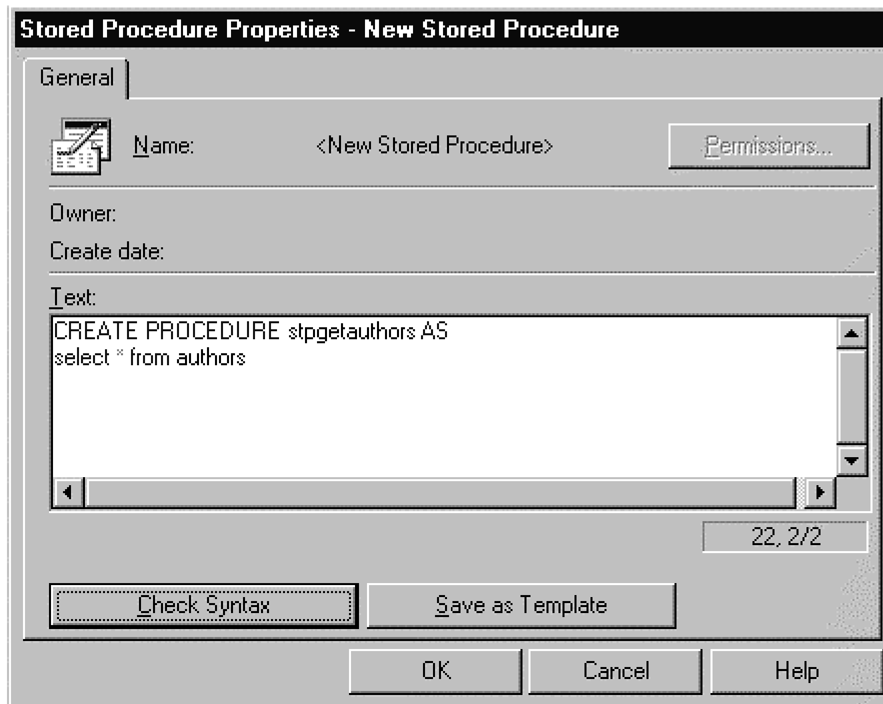
Stored Procedures are created with the CREATE PROCEDURE command:

```
CREATE PROCedure [owner.]procedure_name[;number]
      [(parameter1 [, parameter2]...[parameter2100])]
   [{FOR REPLICATION} | {WITH RECOMPILE}
      [{[WITH] | [,]} ENCRYPTION]]
   AS sql_statements
```

Local and Global Temporary procedures can be created by prefixing the procedure name with a # or ## respectively.

SQL Enterprise Manager can be used to create and maintain stored procedures instead of using the CREATE, ALTER, and DROP PROCEDURE statements.

© REDWARE 1996,2001.



Creating a Stored Procedure

> *Stored Procedures may be renamed with the sp_rename system procedure.*

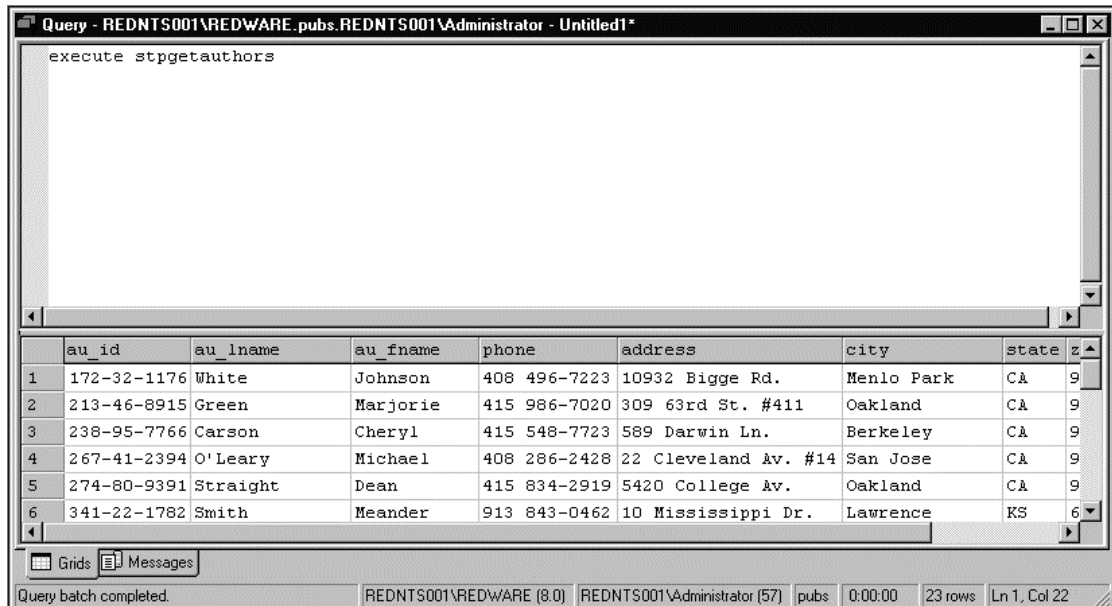
Executing a Stored Procedure

```
[[EXECute]
{[@return_status =]
   {[[[server.]database.]owner.]procedure_name[;number] |
      @procedure_name_var}
   [[@parameter_name =] {value | @variable [OUTPUT]
      [, [@parameter_name =] {value | @variable [OUTPUT]}]...]
   [WITH RECOMPILE]
```

Stored procedures may perform an action or sequence of actions and return a single value or a result set. To execute a stored procedure immediately you can use the SQL Query Analyser tool. Type in the keyword **EXECUTE** followed by the procedure name and any parameters. The result or result set is displayed in the Result window:

```
EXECUTE stpgetauthors
```

Executing a Stored Procedure



	au_id	au_lname	au_fname	phone	address	city	state	z
1	172-32-1176	White	Johnson	408 496-7223	10932 Bigge Rd.	Menlo Park	CA	9
2	213-46-8915	Green	Marjorie	415 986-7020	309 63rd St. #411	Oakland	CA	9
3	238-95-7766	Carson	Cheryl	415 548-7723	589 Darwin Ln.	Berkeley	CA	9
4	267-41-2394	O'Leary	Michael	408 286-2428	22 Cleveland Av. #14	San Jose	CA	9
5	274-80-9391	Straight	Dean	415 834-2919	5420 College Av.	Oakland	CA	9
6	341-22-1782	Smith	Meander	913 843-0462	10 Mississippi Dr.	Lawrence	KS	6

> *Stored procedures normally return text messages indicating how many records have been selected along with other information. This can be suppressed by issuing the SET NOCOUNT ON command at the beginning of the stored procedure.*

Stored procedures can also be executed from inside triggers or other stored procedures using the **EXECUTE** command. This is useful as common code can be placed out into a stored procedure for software reuse. Nesting and recursion is allowed down to 32 levels and the @@NESTLEVEL system variable indicates how many levels down the application has passed.

The EXECUTE command can also parse a string or a variable to execute code that can vary according to the context. The following example will select from a table specified in a local variable:

```
DECLARE @tname varchar(20)
SELECT @tname='authors'
EXECUTE ('select * from ' + @tname)
```

Multiple commands can be executed with the EXECUTE command:

```
EXECUTE( 'set nocount on;' + 'execute stpgetauthors' )
```

The procedure name may even be placed inside a variable so that automated tasks can be performed from a table:

```
DECLARE @pname varchar(20)
SELECT @pname='byroyalty'
EXECUTE @pname 40
```

© REDWARE 1996,2001.

> *One very important feature of Stored Procedures is that a procedure on a remote server may be run simply by specifying the server name in the procedure execute command. The remote server needs to be defined by the SQL administrator so that the servers can communicate but there is no need to log onto the second server as the local server will handle the communication.*

Passing Parameters

Stored procedures can accept parameters and these are held in variables preceded with an @ symbol. These variables need their type defined explicitly in the stored procedure.

The following example accepts a parameter and returns a results set with the Authors selected by Surname according to the parameter passed.

```
CREATE PROCEDURE stpgetauthors
   @surname varchar(30)
 AS
BEGIN
   SELECT * FROM authors
      WHERE au_lname LIKE @surname
END
```

The parameter is passed to the procedure as follows:

```
execute stpgetauthors '[a-d]%'
```

> *Be careful when using SELECT * in a stored procedure as the fields are stored when the procedure is created or altered and may not reflect recent changes to the table structure.*

Procedures can be created with default values for the parameters if none are entered by the user. The following example defaults the parameter to null and causes an error message if no parameter is passed to the function.

```
CREATE PROCEDURE stpgetauthors
   @surname varchar(30)=null
 AS
BEGIN
   IF @surname = null
   BEGIN
      RAISERROR( 'No selection criteria provided !', 10, 1)
   END
   ELSE
   BEGIN
      SELECT * FROM authors
         WHERE au_lname LIKE @surname
   END
END
```

A stored procedure may have more than one parameter declared and values are passed to the procedure in the order that they are declared:

```
CREATE PROCEDURE stpMathTutor @x int =1, @y int =1 AS
BEGIN
   …
```

```
END
```

The procedure may be executed with values of 2 for x and 3 for y as follows:

```
EXECUTE stpMathTutor 2,3
```

Missing out the second parameter with the execute will cause the variable to take up the default value and so the following example will run the procedure with x as 2 and y as the default value of 1.

```
EXECUTE stpMathTutor 2
```

Parameters may also be declared explicitly in the Execute command allowing for them to be specified independently of the order in which they have been declared. The following example gives y a value of 3 and leaves x undefned to take the default value of 1:

```
EXECUTE strMathTutor @y = 3
```

Returning a Value

Stored procedures also have the ability to return a value. This is done by using the return command in the procedure:

```
CREATE PROCEDURE stpMathTutor
   @x int =1 ,
   @y int =1
AS
BEGIN
   RETURN @x + @y
END
```

The value is returned by assigning the procedure to the variable as follows:

```
DECLARE @equals int
EXECUTE @equals = stpMathTutor 2,3
SELECT @equals
```

SQL Server will default the return value to zero. The returned values are typically used to return a status flag from the stored procedure with a non-zero value usually indicating failure during processing.

Returned values are difficult to access using ODBC their use is recommended only to return a success or failure of the stored procedure when communicating with other stored procedures.

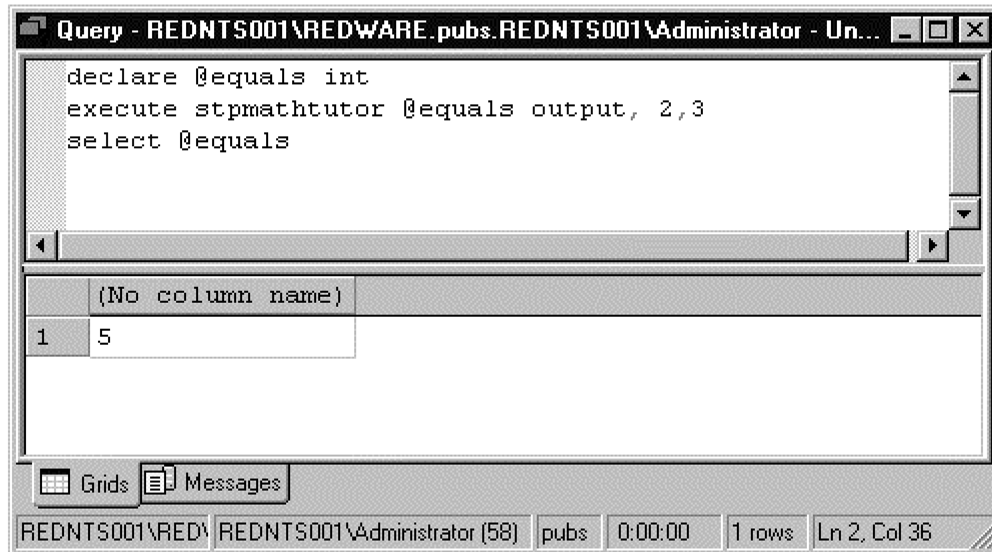
Output Parameters

Values may also be returned into an output parameter by a stored procedure in a similar fashion to other programming languages returning a value by reference. This is achieved by including the 'output' command after the output parameter in the procedure definition.

```
ALTER PROCEDURE stpMathTutor
   @result int output,
   @x int =1 ,
   @y int =1
AS
BEGIN
   set @result = @x + @y
END
```

The returned value from a procedure can be stored in a variable for later use in the calling procedure or trigger. The variable is called in the command line with a "output" modifier:

```
EXECUTE stpMathTutor @equals output, 2, 3
```



Executing the Math Tutor

Program Structures

Transact SQL is primarily a set based language designed for processing sets of data using SQL statements. The language does contain control of flow structures similar to other programming languages.

The BEGIN…END statements are used to create a statement block around a series of Transact SQL statements.

```
BEGIN
    {sql_statement | statement_block}
END
```

The IF…ELSE structure is used extensively inside triggers and stored procedures.

```
IF Boolean_expression
   {sql_statement | statement_block}
[ELSE [Boolean_expression]
   {sql_statement | statement_block}]
```

The structure can use a SELECT statement to perform complex interrogations on data:

```
IF (SELECT SUM(qty) FROM inserted) > 500
BEGIN
   …
END
```

Remember the BEGIN…END structures around blocks of code otherwise only the first line is taken as part of the program flow.

```
WHILE Boolean_expression
   {sql_statement | statement_block}
   [BREAK]
   {sql_statement | statement_block}
   [CONTINUE]
```

The WHILE statement can be used to perform a loop to process a cursor for example. This might be useful when complex sets of different updates and actions need to be performed for each record in a table or when a server cursor is processed.

GOTO can be useful in controlling program flow. A label is defined in the code by placing a line with a label name and a colon and the GOTO command will move program flow to the label. This is especially useful with complex triggers and stored procedures where rollback and updates need close control:

```
IF (SELECT SUM(qty) FROM inserted) = 0
   GOTO noprocessing
   …
noprocessing:
   …
```

A stored procedure can be terminated at any time with the RETURN statement that returns an integer value (default is zero) to the calling program.

Comments can be placed in a stored procedure either with /* */ delimiters or with a double hyphen (--) to make the rest of a line into a comment.

```
/* Test the quantity */
IF @quantity = 0
   RETURN (-1) -- No processing required.
```

Local Variables

```
DECLARE @variable_name datatype
   [, @variable_name datatype...]
```

Local variables are used to store values within Transact-SQL. They must be declared before use and a datatype assigned.

The SELECT statement is then used to assign values to the variables.

```
SELECT @variable = {expression | select_statement}
   [, @variable = {expression | select_statement}...]
   [FROM table_list]
   [WHERE search_conditions]
   [GROUP BY clause]
   [HAVING clause]
   [ORDER BY clause]
```

Values can be assigned in a similar fashion to most programming languages:

```
DECLARE @xvalue int
SELECT @xvalue = 22
```

Values can be determined by a SELECT statement which queries the database and returns a single value:

```
DECLARE @titleqty int
SELECT @titleqty = (SELECT SUM(qty)
   FROM sales WHERE title_id = @titleid)
```

The SET statement can be used instead of SELECT to assign a value to a variable:

```
DECLARE @dialcountry varchar(20)
SET @dialcountry =
   CASE @dialprefix
```

```
        WHEN '44' THEN 'UK'
        WHEN '01' THEN 'USA'
        ELSE 'OTHER'
    END
```

If the SELECT command used with a local variable as the last line of a stored procedure then the value of the variable is returned as a one record results set to the calling application:

```
SELECT @xvalue AS x, @yvalue AS y
```

System Variables

System variables exist which are automatically determined by SQL Server and do not have to be declared. These are always available and indicate a variety of values.

For example:

@@error	Error number
@@identity	Latest identity value of newly inserted record
@@language	Language currently in use
@@max_connections	Maximum connections allowed to the server
@@rowcount	Number of records affected by last command
@@rowcount	Number of rows affected by last statement
@@servername	SQL Server name
@@version	Version number of SQL Server

Other system information is returned from scalar functions:

DB_NAME()	Database Name
SUSER_SNAME()	NT User Name
USER_NAME()	SQL Server User Name

All of these system variables can be used as required within any Transact SQL code as shown in the following example:

```
CREATE PROCEDURE stpserverinfo AS
   select db_name(), user_name(),suser_sname(), @@servername,
   @@max_connections, @@version, getdate()
```

Scalar Functions

Scalar functions can also be used to perform an operation and return a single value. There are many examples some of which are listed below:

Mathematical and trigonometric functions:

- `abs(-22.33)`
- `pi()`
- `sin(30)`
- `cos(30)`
- `tan(30)`
- `rand(22)`
- `round(3.4456,2)`

Date functions:

- `datepart(yyyy,getdate())`
- `year(@datevalue)`

- `dateadd( yyyy, 2, @datevalue)`
- `datediff( yy, @datevalue,getdate() )`
- `month( @datevalue )`

String functions:

- `left('abc123',3)`
- `ltrim('   abc')`
- `replace( 'abc','b','z' )`
- `soundex('abc' )`
- `substring( 'abc',2,1 )`
- `upper( 'abc' )`

Miscellaneous:

- `cast('abc' as varchar(5))`
- `convert(int, 22.33)`
- `columnproperty( object_id('authors'),'city', 'allowsnull')`
- `isnull( @xvalue,0,1 )`

> *Look at the entry on 'scalar functions' in the online books for more information.*

CASE Expression

```
CASE expression
   WHEN expression1 THEN expression1
   [[WHEN expression2 THEN expression2] [...]]
   [ELSE expressionN]
END
```

The CASE expression is very useful for assigning different values according to an expression for each record of a table. The following example will evaluate a description of the Authors contract status as a field in the results set according to the defined conditional rules:

```
SELECT *,
   'Contract Status' = CASE
      WHEN contract = 1 THEN 'Contracted'
      WHEN contract = 0 THEN 'No Contract'
      END
   FROM authors
```

The CASE expression can be used anywhere where an expression is required including in an Update statement to set values into a field.

This sophisticated example, shown blow, is taken from the SQL On-Line reference and shows a SELECT statement used within a CASE to change the expression shown:

```
SELECT a.au_lname Surname, a.au_fname Forename,
   "Royalty Category" =
   CASE
      WHEN (SELECT AVG(royaltyper) FROM titleauthor ta
         WHERE t.title_id = ta.title_id) > 60
         THEN 'High Royalty'
      WHEN (SELECT AVG(royaltyper) FROM titleauthor ta
         WHERE t.title_id = ta.title_id)
         BETWEEN 41 and 59
         THEN 'Medium Royalty'
```

```
      ELSE 'Low Royalty'
   END
   FROM authors a, titles t, titleauthor ta
   WHERE a.au_id = ta.au_id AND ta.title_id = t.title_id
   ORDER BY 1,2,3
```

Cursors

Stored procedures often need to process each record in a table and perform an action. For example a housekeeping program might run through all the new orders in a sales database and send email messages to the account manager in instances where the delivery date is more than five days from the date of order.

Cursors allow for the selection of the records in a stored procedure and the sequential processing of each record. Scrollable cursors also allow movement forwards and backwards through the table.

```
DECLARE cursor_name CURSOR
   [ LOCAL | GLOBAL ]
   [ FORWARD_ONLY | SCROLL ]
   [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
   [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
   [ TYPE_WARNING ]
   FOR select_statement
   [ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```

The cursor must first be declared before the FETCH command can be used to move up and down the cursor.

```
FETCH [[NEXT | PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n]
FROM] cursor_name
[INTO @variable_name1, @variable_name2, ...]
```

The @@FETCH_STATUS variable is set to zero after a successful FETCH and should always be checked before processing. A value of -1 indicates that the results set has been exceeded and -2 indicates that the cursor record is no longer a member of the original table.

Cursors are relatively slow and should not be used if more traditional set based processing is possible. Complex expressions can be created with the CASE expression within a SELECT statement and should be used in preference to a cursor whenever possible.

The following example illustrates the use of a cursor to process the records in a table one by one. The cursor is created from a SELECT statement and the OPEN command used to open the cursor. The value are FETCHed into variables that have already been defined and a WHILE loop used to process each record. The logic for each record is contained in the loop and mails a simple message.

```
CREATE PROCEDURE cursortest AS

DECLARE @id varchar (12)
DECLARE @firstname varchar(40)
DECLARE @surname varchar (40)
DECLARE @message varchar (80)

DECLARE curAuthors CURSOR LOCAL
   FOR SELECT au_id, au_fname, au_lname
```

```
   FROM authors WHERE contract = 1
   FOR READ ONLY

EXECUTE master..xp_startmail

OPEN curAuthors
FETCH NEXT FROM curAuthors INTO @id, @firstname, @surname
WHILE (@@fetch_status <> -1)
BEGIN
   IF (@@fetch_status <> -2)
   BEGIN
      SELECT @message = @id + @firstname + @surname
      EXECUTE master..xp_sendmail 'stamati crook', @message
   END
   FETCH NEXT FROM curAuthors INTO @id, @firstname, @surname
END
DEALLOCATE curAuthors
```

Take care to CLOSE or DEALLOCATE a cursor when you have finished it to prevent using too many server resources.

An alternative to a WHILE loop is to use a program marker and the GOTO statement.

System Procedures

The MASTER database contains several system procedures which are created when SQL Server is installed. These procedures have 'sp_' as a prefix to their name and are used mainly for administration purposes. They can be accessed from any database as long as the user has access rights to the master database. The procedures can be copied into your own database and edited to suit your needs.

One example is the system procedure **sp_depends** which returns the dependencies of a SQL Server table, view or procedure object. The procedure returns a result set indicating all the objects upon which the object depends and all those that depend on it.

```
EXECUTE sp_depends 'authors'
```

There are many system procedures affecting all aspects of the database and server configuration. For example, batch scripts may be created to add users to a database. See the Transact SQL Reference manual for details.

There are hundreds of system stored procedures described in the online help. Some more useful system procedures are described below:

sp_helpdb	Details of the databases defined on the server.
sp_helpdb pubs	Details of the pubs database.
sp_help authors	Provides details on any database object.
sp_helptext byroyalry	Provides the text of a stored procedure.
sp_depends authors	Details of all objects that debend on the specified object.
sp_changeowner	Change the owner of an object (usually to dbo).
sp_rename	Rename an object.

Extended Procedures

Extended Procedures are used to call programs residing on the server automatically from a stored procedure or a trigger run by the server.

The extended stored procedures are held in the MASTER database and may be used to interact with the server. The following example is used to log an event in the NT event log of the server without raising any errors on the client application:

```
declare @logmessage varchar(100)
set @logmessage = suser_sname() + ': Attempted to access the bingo system.'
exec master..xp_logevent 50001, @logmessage
```

The XP_CMDSHELL command will run an operating system command on the server:

```
EXECUTE master..xp_cmdshell 'dir e:\*.*'
```

This functionality is very dangerous in the wrong hands as files may be deleted on the server or worse havoc caused.

An example extended procedure might call a Visual Basic program that runs on the server whenever an order is entered into the database which reads the SQL database in order to enter data into a FoxPro system that is used for Order Processing. This functionality could also be performed from the original application but implementing at the server level allows for orders to be created in a variety of front end implementations and yet always perform the required transactions.

DLLs may be created on the server and called within SQL Server as an extended procedure after registering the procedure with the sp_addextendedproc function.

Extended Mail Procedures

SQL Server includes extended procedures that facilitate the integration with Microsoft Mail. This allows an update trigger, for example monitoring stock levels, to generate an electronic mail message whenever the stock level falls below the reorder level.

SQL Server can be configured to “Auto Start Mail Client” when the SQL Server Service is started or Mail can be run on the server before starting the SQL Server service. Alternatively the mail client may be started on the server with the following extended procedure:

```
EXECUTE master..xp_startmail
```

The startmail extended procedure can accept username and password to start a particular mail session if the setup defaults are not acceptable.

Mail may be sent to a mail user as a simple mail message or with the attachments of a file or results from a SQL Query:

```
EXECUTE master..xp_sendmail 'stamati crook', 'Reorder Disks
20303 Please!'
```

> *The mail procedures require the full user name as parameters. The shortened mail name will create an error.*

SQL Server can also read mail to form part of an integrated Mail - Database Information strategy. There are extended procedures to read mail and to process queries attached to mail messages and attach the results set into a mail message and so on.

Error Handling

Stored procedures return a zero value by default. The convention is to return a zero value if the stored procedure is successful and a non-zero value for a failure.

```
declare @returnvalue int
exec @returnvalue = stpgetauthors
if @returnvalue <> 0
begin
```

The RAISERROR command is used to create error messages from the server which are returned to the application. The command will return an error number and a message to the calling application error handle.

```
RAISERROR ({msg_id | msg_str}, severity, state
    [, argument1 [, argument2]] )
    [WITH LOG]
```

The severity is a number from 0 to 25 although only system administrators should use values above 18. The convention is as follows:

- 10 is for information only
- 11-16 is for errors that can be corrected by the user
- 17 is where system resources are exceeded
- 18 is a non fatal system error

Severity levels 17 and above should be notified to the system administrator. The state is a number from 0 to 127 that can be used as you like.

Additional arguments can be included in an error message to provide additional information for a specific instance of the error. The following example raises an error and includes details of the author identifier and the number or records retrieved in the error message. It also records the error in the NT Event log of the server.

```
if @@rowcount <> 1
begin
    raiserror (
        'stpgetauthordetail: %s :Incorrect (%i) number of records found !',
        16,1,@authorid, @@rowcount ) with log
    return (2)
end
```

The default error number for a user created error is 50000. All user created errors should have an error number greater than 50000.

Error messages may be added into the database catalogue of error messages with the **sp_addmessage** stored procedure which stores a message against an error number and severity:

```
sp_addmessage 52001,16,'%s : Incorrect Parameters !'
```

The error is called with the RAISERRROR command without the need to supply the message text each time. This also allows messages to be displayed in multiple languages or system alerts to be defined to notify the system administrator immediately a particular error occurs.

```
if @authorid = null
```

```
begin
   raiserror(52001,16,1,'stpgetauthordetail-authorid')
   return (-1)
end
```

The @@ERROR system variable can be used to control errors a little more closely. The following stored procedure updates the value of the ZIP field in the authors table. This field has a constraint and will only allow a five numeric value to be applied. The @@ERROR value is used to trap for an error and return an explanatory error message to the client.

```
CREATE PROCEDURE stpsetauthorzip
@authorid id,
@zip char(5)
AS
update authors
   set zip = @zip
   where au_id = @authorid
if @@error <> 0
begin
   raiserror( 'Invalid ZIP code: %s',16,1,@zip)
   return (2)
end
```

> *The above example will return two error messages to the client. The first error is generated by SQL Server to indicate a failure of the constraint and the second generated by the user.*

The error can also be trapped for in a stored procedure that traps the result code returned from another stored procedure:

```
declare @return int
exec @return = stpsetauthorzip '267-41-2394','74722'
if @return <> 0
   print 'failue'
else
   print 'ok'
```

Transactions

Stored Procedures often implement a series of transactions that update the database. Occasionally one of these transactions may fail, perhaps because another user has locked the resource or because a database constraint is activated, and an error is generated within the stored procedure. The programmer may need to control the transactions within the stored procedure to ensure that all or none of the transactions are written to the database.

Transactions are controlled with three commands:

- BEGIN TRANSACTION starts a transaction and also allows for nested transactions.
- COMMIT TRANSACTION will write all of the current transactions to the database.
- ROLLBACK TRANSACTION will undo all of the changes to the database for the current transaction.

> *SQL Server will 'write ahead' any changes to the database allowing for another user to read uncommitted data if they use the NOLOCK option of a SELECT statement.*

© REDWARE 1996,2001.

The programmer can check for errors after each database update and then rollback the transaction if required. The following stored procedure adds an order to the SALES table provided that there is enough stock for the particular title indicated in the STOCKLEVEL field of the STOCK table.

Errors are monitored after each database update using the @@ERROR global variable and the whole transaction rolled back if an error occurs. This prevents the stock values from being debited if the subsequent order record cannot be created (for example if the order number is not unique).

```
CREATE PROCEDURE stpaddorder
    @storeid char(4),
    @orderid varchar(20),
    @titleid tid,
    @quantity int,
    @orderdate datetime,
    @payterms varchar(20) = 'Standard'
AS

/*
Author: Stamati Crook
Date:   6 October 2001
Name:   stpaddorder
Purpose:
Adds an order into the sales table after checking that there is sufficient
stock.
*/

DECLARE @errortrap int
SET @errortrap = 0


IF NOT EXISTS (SELECT title_id FROM stock WHERE title_id = @titleid )
BEGIN
    RAISERROR ( 'No Stock Record',16, 1)
    RETURN (-100)
END

IF (SELECT stocklevel FROM stock WHERE title_id = @titleid ) < @quantity
BEGIN
    RAISERROR ( 'Not enough stock',16, 1)
    RETURN (-101)
END

BEGIN TRANSACTION

UPDATE stock
    SET stocklevel = stocklevel - @quantity
    WHERE title_id = @titleid

SET @errortrap = @@error

IF @errortrap = 0
BEGIN
    INSERT INTO sales
        ( stor_id, ord_num, ord_date, qty, payterms, title_id )
        VALUES
        (@storeid, @orderid, @orderdate, @quantity, @payterms, @titleid )
    SET @errortrap = @@error
END

IF @errortrap = 0
BEGIN
    COMMIT
END
ELSE
BEGIN
    RAISERROR ( 'Error updating sales or stock table',16, 1)
    ROLLBACK
    RETURN (-102)
END
```

```
RETURN 0
```

Distributed Transactions

Distributed transactions can be controlled by the programmer to allow transactions on different SQL Servers to be committed or rolled back:

```
DECLARE @result int
BEGIN DISTRIBUTED TRANSACTION
EXECUTE @result = stpaddorder '7066', '240','PC1035',1,'2001-
05-23'
IF @result = 0
EXECUTE @result = remote.pubs.dbo.stpaddorder '7066',
'240','PC1035',1,'2001-05-23'
IF @result = 0
   COMMIT DISTRIBUTED TRANSACTION
ELSE
   ROLLBACK DISTRIBUTED TRANSACTION
RETURN @result
```

This type of functionality is often created as a middle tier component on Microsoft Transaction Server which can also control distributed transactions using the same transaction coordinator that SQL Server uses.

9. Triggers

Triggers allow for more sophisticated validation to be defined against a table than can be provided through the field level Rules. They may also be used to ensure **referential integrity**, to cascade changes throughout related records, and to determine the changes made to a field and perform appropriate action.

Three triggers may be defined against each table which operate when a new record is inserted, modified, or deleted respectively. Triggers are programmed to prevent **insert, update, and delete anomalies** that may occur when updating the database.

> *SQL Server also allows the definition of Primary and Foreign Keys to enforce referential integrity at the database level.*

An example of an insert anomaly might be the creation of an Invoice record which does not have a valid foreign key relationship with the Customer table. If this were to occur there would be no means of determining the address for the Invoice.

Update anomalies, which should be checked when inserting a record, often enforce business rules as well as referential integrity issues. An Invoice date cannot be changed after it has been issued, or the total outstanding amount cannot exceed the credit limit for the customer.

Delete anomalies often involve referential integrity issues where a record may not be deleted if there are related records in another table. This is usually performed by setting up referential integrity with primary and foreign keys.

Triggers are created with the MANAGE-TRIGGERS window for the appropriate database using a programmable form of SQL known as **Transact SQL**. Transact SQL is an extension of SQL that allows program flow and the use of variables to determine the actions performed for a trigger.

A typical trigger will perform some validation and rollback the transaction with appropriate error messages of there is a violation of the database validation rules. More sophisticated triggers may update values or delete records in other database tables. Typical examples are given below and further information on the available syntax for Transact SQL can be obtained from the reference manual.

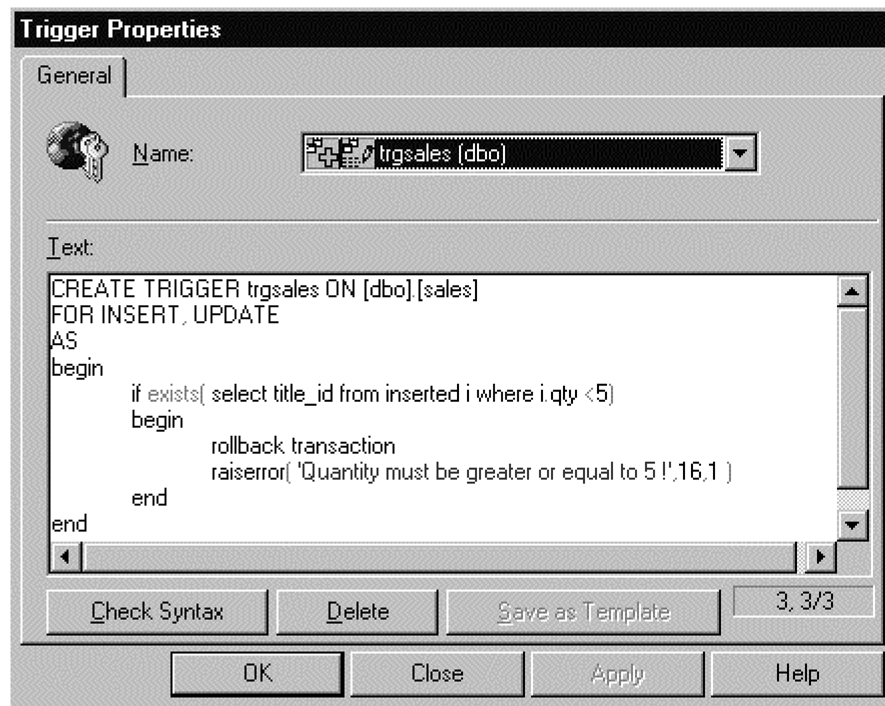
> *Multiple triggers can be defined on a table and the sequence of execution controlled with the sp_settriggerorder system stored procedure.*

Trigger Program Structure

Triggers are created with the MANAGE-TRIGGER menu which brings up the trigger window.

```
CREATE TRIGGER [owner.]trigger_name
   ON [owner.]table_name
   FOR {INSERT, UPDATE, DELETE}
   [WITH ENCRYPTION]
   AS sql_statements
```

© REDWARE 1996,2001.



Creating a Trigger with the SQL Executive

Triggers employ two **conceptual tables, 'deleted'** and **'inserted'** to allow for Transact SQL to determine the values in a record before and after a transaction. An Insert transaction will create a record in the 'inserted' table before completing a transaction, a delete transaction will create a record in the 'deleted' table, and an Update will place the old values for the record in the 'deleted' table and the new values in the 'inserted' table.

> *These virtual tables may, in some circumstances, contain more than one record. It is therefore necessary to structure any processing to process all the potential records The system variable @@rowcount indicates the number of records to process.*
>
> *The examples use set based processing to cater for several records. Some processing may require the use of Cursors as described in the Stored Procedures section..*

Program flow can be controlled with an IF statement normally used in conjunction with a select statement that returns a single value. The next Transact SQL statement is executed if the condition is true. If more than one line of Transact SQL is required for a particular condition they must be enclosed in a BEGIN...END construct.

The UPDATE() function is often used to indicate if a field has been changed by the application. If the field has not been changed the function returns a false value and the program flow will skip the next line or BEGIN..END section.

A count of the records is usually made to test if any records have failed the required business rules.

> *The IF EXISTS (SELECT….) clause is more efficient than IF (SELECT COUNT(*) …) >0 because processing stops after the first record is found.*

Error messages need to be returned to the application if appropriate. Transact SQL allows for a PRINT command to display error messages but the RAISERROR command is better for creating a smooth interface with the application. SQL Server error numbers above 50,000 are reserved for applications and the RAISERROR command can generate an error number in the calling application as well as supplying error text.

If a rule causes a transaction to fail it should be rolled back with the ROLLBACK TRANSACTION command. SQL Server will not update the database and the application will need to perform appropriate actions to recover from the error or attempt to resubmit the changes.

Triggers should be commented well and cross-referenced with the data dictionary documentation. Comments are enclosed with a forward slash and asterisk as shown in the example.

> *Trigger functionality often needs to be implemented both as INSERT and as UPDATE triggers to prevent incorrect data from being entered.*

Typical syntax for a trigger is shown below.

```
CREATE TRIGGER truOrderUpdate ON tabOrders FOR UPDATE AS
IF UPDATE( fieldname )
BEGIN
   /* This is a Comment */
   IF EXISTS (SELECT * FROM inserted WHERE expression ) > 0
   BEGIN
      ...
      Transact SQL Statements...
      ...
      RAISERROR 52001 'This is an error message!'
      ROLLBACK TRANSACTION
   END
END
```

> *Remember to place BEGIN and END statements after a program flow statement if more than one Transact SQL statement is required within the IF statement.*

Field Level Validation

Some field validation can be performed using a Rule. The advantage of a rule is that it is defined independently of the field and may be applied consistently to several fields in the database.

Update triggers may be used to check field values in a similar manner to a rule. The form of the Transact SQL is as follows:

```
CREATE TRIGGER truorder ON dbo.taborder
FOR UPDATE
AS
BEGIN
```

```
    IF EXISTS (SELECT * FROM inserted
       WHERE freight > 50.0 )
    BEGIN
       ROLLBACK TRANSACTION
       RAISERROR 52001 'Freight must be less than £ 50 !'
    END
END
```

Notice that the Transact SQL is designed to work with more than one record in the inserted table.

Record Level Validation

The trigger is more flexible than a rule and can check other values in the record or in other tables as well as performing complete queries and complex sequences of operations.

The following example of an Update Trigger prevents the Freight field from being greater than the Limit field in the same record.

```
IF UPDATE(Freight) OR UPDATE(Limit)
BEGIN
   IF EXISTS (SELECT * FROM inserted WHERE Freight>Limit)
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 52002 'Freight must be less than Limit!'
   END
END
```

Avoid Null values in numeric fields as they will not default to zero for validation purposes. Define a zero default for the field.

Checking Values against another Table

Triggers may also be used to check values against another table.

Inserting a new Invoice Item record may require a stock check against the relevant product record. Changes to the line item are prevented in the Update trigger so this check is only required on Insert.

Preventing Changes to a Field

The UPDATE() function will indicate if a particular field has been updated during a transaction. A typical trigger would check if a field has been updated and rollback the transaction after error handling.

This is particularly useful for primary and foreign key values which should not be changed after inserting the record as shown in the following trigger code:

```
IF UPDATE(Order_ID)
BEGIN
   ROLLBACK TRANSACTION
   RAISERROR 52003 'No update allowed on Order Identifier!'
END
```

© REDWARE 1996,2001.

> *Security may be defined at the field level to prevent certain users from updating certain fields.*

Referential Integrity Checks

SQL Server 6.0 implements referential integrity checks automatically when Foreign Keys are defined against the corresponding Primary Key. The integrity check is mandatory and it is not possible to enter empty values against a foreign key field if the relationship is optional.

Implementation of referential integrity in database triggers is also possible and its the method employed before SQL Sever 6.0. Greater control is possible over error messages and optional relationships or cascading deletes.

> *Referential Integrity checks are easy to implement with Foreign Keys and this should be used wherever possible. The referential integrity is checked by the server before the triggers are fired and the programmer may prefer that all errors are processed together. In this case referential integrity must be implemented as triggers.*

Referential integrity often involves the setting up of several triggers. A typical scenario for a mandatory relationship between parent and child tables would involve an insert trigger on the child table to check the foreign key, an insert trigger on the parent table to ensure unique primary keys, a delete trigger on the parent table to avoid orphaned child records, and update triggers on both the parent and child tables to ensure that the primary and foreign key values are not updatable.

Checking a Foreign Key

Triggers may be used to check referential integrity when entering a new record. This usually involves ensuring that the new record has a corresponding occurrence in another table.

Foreign Keys may be implemented with a trigger that allows empty values or checks the referential integrity of the entered value. The trigger needs also to be defined on Update or changes to the field prevented.

```
CREATE TRIGGER triOrders ON tabOrders FOR INSERT AS
BEGIN
   /* Check Customer-Orders Integrity */
   IF (SELECT COUNT(*) FROM inserted, tabCustomers
      WHERE (inserted.Customer_ID = tabCustomers.Customer_ID)
      <> (SELECT COUNT(*) FROM inserted))
   BEGIN
      RAISERROR 52005 'Orders must have a valid or blank
Customer!'
      ROLLBACK TRANSACTION
   END
END
```

> *It is easier to define Foreign keys with the Foreign Key feature of the Manage Tables window.*

© REDWARE 1996,2001.

Ensuring Unique Candidate Keys

Primary keys are usually enforced by defining an index that is unique however this functionality may also be performed with a trigger for candidate keys where values may also be left blank.

```
/* Check that Primary Key is Unique */
IF EXISTS (SELECT id FROM inserted, taborder
   WHERE inserted.id = tabOrder.id )
BEGIN
   RAISERROR 51004 'Order Identifier is not Unique!'
   ROLLBACK TRANSACTION
END
```

> *SQL Server will check the unique index before checking the insert trigger and the trigger validation error will not occur. Unique primary key indexes are the recommended approach for ensuring the uniqueness of a key value.*

Checking Referential Integrity on Delete

Referential integrity checks on deletion prevent records from being deleted if there are dependent records in related tables. An example is shown below where the Order table has many Order_Details records linked on the common Order_ID field.

```
CREATE TRIGGER trdorder ON taborder FOR DELETE AS
BEGIN
   /* Check Order Item Integrity */
   IF EXISTS (SELECT id FROM deleted, taborderitem
      WHERE deleted.id = taborderitem.order )
   BEGIN
      ROLLBACK TRANSACTION
      RAISERROR 52004 'Order Items Exist!'
   END
END
```

Cascading Delete

Delete validation usually prevents deletion of a parent record if any child records exist in a related table. It is possible for the deletion trigger to automatically delete the child records when the parent record is deleted.

This Cascading Delete prevents orphaned child records existing in the database and removes the need for the application to delete the child records before attempting to delete the parent record.

```
CREATE TRIGGER trdorder ON taborder FOR DELETE AS
BEGIN
   /* Cascading Delete to ensure Order_Item Integrity */
   DELETE taborderitem FROM deleted, taborderitem
      WHERE taborderitem.order = deleted.id
END
```

Further validation is often required. The select statement might only delete order detail records if there were no outstanding shipments to be performed and rollback the transaction if this were not possible. In general, application level deletion of the child records before deletion of the parent record allows greater control and these deletions could be placed into a single transaction with a Rollback to prevent the possibility of data anomalies.

*A similar approach can be used to allow for changes to a primary key to cascade down and update foreign key values in child tables, this is called **cascade update**. This approach is not recommended and can be circumvented by using a second candidate primary key in the parent table for the application to change and providing automatic primary keys that are used by the application.*

Updating another Table

Triggers may be used to change values in related tables. Our example will automatically update the YTD_SALES field in the Titles table which indicates the number of titles sold so far this year.

Skilful use of triggers allow for calculated fields to be stored in the database and updated outside of application control. Remember these fields should not be updatable by the application. Insert and update triggers may be used to check values against another table as well as performing foreign key validation.

Calculated field may sometimes be implemented as Views but there are some restrictions on SQL Server views.

The Insert Trigger on the Sales table will automatically add the sales quantity onto the YTD_SALES fields for the appropriate titles. Notice that the trigger logic can process several inserted records at one time.

```
CREATE TRIGGER trisales ON dbo.sales
FOR INSERT
AS
BEGIN
     UPDATE titles
       SET titles.ytd_sales = titles.ytd_sales +
       inserted.qty
       FROM titles, inserted
       WHERE titles.title_id = inserted.title_id
END
```

In this instance referential integrity is handled with a foreign key constraint in the database and does not need to be implemented at the trigger level.

The Delete Trigger must subtract the value of the deleted Sales from the Titles records:

```
CREATE TRIGGER trdSales ON dbo.sales
FOR DELETE
```

```
AS
BEGIN
UPDATE titles
SET titles.ytd_sales = titles.ytd_sales - deleted.qty
FROM titles, deleted
WHERE titles.title_id = deleted.title_id
END
```

The Update Trigger needs to handle the difference between the original and the new sales record:

```
CREATE TRIGGER truSales ON dbo.sales
FOR UPDATE
AS
BEGIN
UPDATE titles
SET titles.ytd_sales = titles.ytd_sales - deleted.qty +
inserted.qty
FROM titles, deleted, inserted
WHERE titles.title_id = deleted.title_id AND
titles.title_id = inserted.title_id
END
```

Finally, updating the YTD_SALES can be prevented in a trigger in the Update trigger of the TITLES table:

```
CREATE TRIGGER truTitles ON dbo.titles
FOR UPDATE
AS
BEGIN
 IF UPDATE( ytd_sales )
 BEGIN
       RAISERROR 52010 'YTD_SALES may not be updated!'
       ROLLBACK TRANSACTION
 END
END
```

Be careful with more complex validations in conjunction with altering table values. An UPDATE command issued after a ROLLBACK TRANSACTION will still update the database. The GOTO command may be useful here in sending trigger processing to the end of a trigger.

10. SQL Server Optimisation

This section describes how application design can affect server performance and provides general hints for optimising SQL Server performance.

Query Optimisation

Many database applications retrieve a results set of data from the server using a SELECT statement and then selectively update individual records as they are changed by the user.

SELECT statements are passed through a query optimiser which determines the most efficient way to optimise the query. The optimiser will look at the size of the table and the indexes defined as well as information on the distribution of records within each index and will determine which index to use to improve performance.

Some queries will require a table scan which passes through all records in the table before determining which records are required in the query. Other more complex queries will perform combinations of index searching and table scanning to create the final results set.

Update Statistics

SQL Server looks at the distribution of records within each index before determining the optimisation plan. The distribution available to the optimiser is created when the index is first put on the table. Many indexes are created on empty tables and after some months of operation the optimiser will still not have any knowledge of the distribution of data and may be considering Table Scans instead of index searches.

It is therefore extremely important that these index statistics are updated after the structure of the index is changed by the addition of modification of an amount of data. This is particularly true in the first months of system use and should also be performed regularly by the system manager.

The UPDATE STATISTICS command is used to update the index distributions statistics on a table and may be called from ISQL:

```
UPDATE STATISTICS authors
```

A batch file could be created and run periodically from the ISQL or from a maintenance program written using pass through queries.

> *Everybody gets caught out by UPDATE STATISTICS. Remember to perform this operation on every table after uploading test data.*

Index Design

Definition of appropriate indexes is the single most important performance optimisation technique. The indexes should reflect the expressions used in the WHERE clause of the most frequently used queries.

The following query based on the Authors surname requires an index on the AU_LNAME field for an index as opposed to a table scan to be used:

```
SELECT au_fname forename, au_lname surname
   FROM authors
   WHERE au_lname LIKE 'C%'
```

> *Many queries in SQL Server are not case sensitive by default which simplifies query design.*

© REDWARE 1996,2001.

The expressions used in the Where clause must be recognisable by SQL Server as part of an index. The above example would not recognise that the surname index could be used if it had been written in this form:

```
SELECT au_fname forename, au_lname surname
   FROM authors
   WHERE substring(au_lname,1) = 'C'
```

Take care with wildcard characters at the beginning of expressions because they will not allow an index search:

```
SELECT au_fname forename, au_lname surname
   FROM authors
   WHERE au_lname LIKE '%opulos'
```

The Not Equal To operator is not optimised in SQL Server so the following query will not use an index defined on the price for a search:

```
SELECT * FROM titles WHERE price <> 10.00
```

It should be replaced with:

```
SELECT * FROM titles WHERE price < 10.00 OR price > 10.00
```

More complex queries will still use a single index for optimising the query. A search on surname and forename will still use only a single index. Thus a compound index with both the surname and firstname field will help to optimise the following query:

```
SELECT au_fname forename, au_lname surname
   FROM authors
   WHERE au_lname LIKE 'C%' AND au_fname LIKE 'S%'
```

In general the field that has the largest range of distinct values should be chosen as the first field in the index provided that both fields are used in the major queries.

It is important that the expressions that are used in most queries are placed as the first columns in the index. Suppose a third query on the Authors table required the CONTRACT field as an expression:

```
SELECT au_fname forename, au_lname surname
   FROM authors
   WHERE au_lname LIKE 'C%' AND au_fname LIKE 'S%'
   AND contract = 1
```

Adding the Contract field into the index would further optimise the query. However perhaps a fourth query requiring only the surname and contract fields is also required:

```
SELECT au_fname forename, au_lname surname
   FROM authors
   WHERE au_lname LIKE 'C%' AND contract = 1
```

For this query to be fully optimised an index on Surname and Contract only is required. The previously defined index contains the Surname, Firstname, and Contract fields but the Firstname field is not used in this query and will prevent the optimiser from fully optimising the query. The optimiser is likely to use the index to perform the search on the Surname field but will then have to scan through all the Authors whose Surnames begin with C. A second index defined on Surname and Contact fields will allow this query to be optimised.

© REDWARE 1996,2001.

There is a performance penalty associated with having too many indexes on a table particularly when large numbers of transactions are being made on a table. A trade off between adding new indexes and having partially optimised queries is a necessary part of database design.

Some queries are able to use an index to process a query without looking at the table data. These covered indexes contain all the fields specified in the fields clause and the where clause and are particularly useful for summary calculations.

The following select statement would be satisfied by a covered index on the product type and the quantity:

```
SELECT SUM(qty) FROM sales GROUP BY prodtype
```

Ordering

The Order By clause can destroy performance on the server particularly if a large results set is required.

SQL Server will create a worktable for sorting in the tempdb database if the ordering criteria is not fully satisfied by the index used by the optimiser for the selection criteria. This index needs to be determined or specified by the programmer to ensure efficient ordering.

The clustered index is used often in the ordering process to prevent the requirement for a worktable. An analysis of the order required on a table will often lead to the choice of clustered index and a restriction on the ordering permitted for efficient processing.

The server may chose to use the clustered index rather than the specified index in an order by clause which prevents the server from returning the results set until the selection is completed. The FASTFIRSTROW optimiser hint allows the server to use the nonclustered index and returns the first row faster. This is useful for asynchronous queries.

Showplan

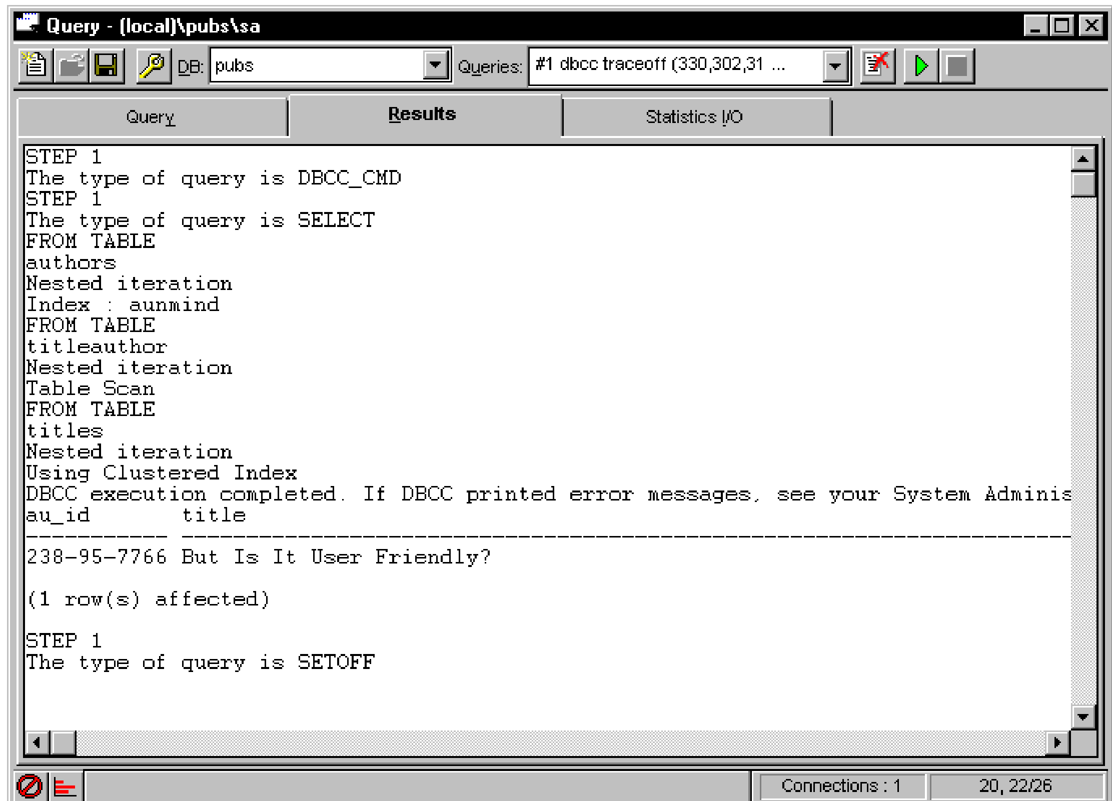
The ISQL tool allows access to the plan that the optimiser is making to optimise a query. Once a query has been identified as a potential bottleneck this window can be used to see the exact optimisation plan that SQL Server is following.

Copy the relevant SELECT statement out from the application and paste into the QUERY Page of the ISQL tool ensuring that the correct Database is selected.

Select the QUERY OPTIONS menu item from the QUERY menu and chose the SHOW QUERY PLAN option. The Results Page will show the tables scans or indexes used to perform the query and indicate if indexes may need to be added to optimise the query.

© REDWARE 1996,2001.

Showplan Results



Selecting NO EXEC from the QUERY menu will create the optimisation plan without performing the query.

The STATISTICS I/O Page will show the logical and physical disk reads and if the query takes a long time and can be specified in the QUERY menu option.

The DBCC TRACEON command may be issued to show additional query optimisation statistics. This can selectively display the join order chosen by the optimiser, the estimated costs in terms of disk reads. The following command will show an abundance of optimiser information when performed with the Statistics and Execution Options set:

```
dbcc traceon (330,302,310,3604)
select authors.au_id, titles.title, titles.ytd_sales
   from authors, titleauthor, titles
   where authors.au_id = titleauthor.au_id
   and titles.title_id = titleauthor.title_id
   and au_lname like 'C%'
```

SQL Trace

SQL Server 6.5 boasts a new utility that allows monitoring of SQL Server operation whilst a live application is running. This can be useful for analysing which queries are being run in a production environment.

© REDWARE 1996,2001.

Monitoring Server Operation with SQL Trace

```
SQL Trace - (local) - [sa monitor]
File  Edit  View  Tools  Window  Help
-- 12/19/96 08:01:01.106 Filter Started (ID=4, SPID=14, User=sa(POSEIDON\Adminis

-- 12/19/96 08:01:00.113 Active connections (ID=4, SPID=14, User=sa(POSEIDON\Adm
-- 12/19/96 06:46:56.146 Active connections (ID=3, SPID=11, User=sa(POSEIDON\Adm
-- 12/19/96 07:59:42.913 Active connections (ID=2, SPID=13, User=sa(POSEIDON\Adm
-- 12/19/96 07:59:16.816 Active connections (ID=1, SPID=12, User=sa(POSEIDON\Adm
-- 12/19/96 08:01:17.720 SQL (ID=3, SPID=11, User=sa(POSEIDON\Administrator), Ap
select suser_name()
go
-- 12/19/96 08:01:18.140 SQL (ID=3, SPID=11, User=sa(POSEIDON\Administrator), Ap
set showplan on
go
-- 12/19/96 08:01:18.460 SQL (ID=3, SPID=11, User=sa(POSEIDON\Administrator), Ap
select authors.au_id, titles.title, titles.ytd_sales
        from authors, titleauthor, titles
        where authors.au_id = titleauthor.au_id
        and titles.title_id = titleauthor.title_id
        and au_lname like 'C%'
go
-- 12/19/96 08:01:19.030 SQL (ID=3, SPID=11, User=sa(POSEIDON\Administrator), Ap
set showplan off
go
Active filters 2
```

Optimiser Hints

Optimiser hints may be specified on a SELECT statement that force the query optimiser to use the specified index. The cost criteria the SQL Server assigned to a query are complex and it is likely that the statistics have not been updated or that an appropriate index has not been defined if SQL Server is selecting the wrong query path.

The optimiser hints exist to override the optimiser and helps to overcome holes in the design of the optimiser where unoptimised queries slip through. Many of these have been patched and the optimiser should be better at determining the best access path to data than all but the most experienced database administrators. It is recommended that optimiser hints are only employed where it proves impossible for the optimiser to recognise the required indexes/

> *One trick to force the optimiser into considering alternative query paths is to change the join order of a multi table query. The optimiser determines the join order early in the optimisation process and changing the join order may force it to reconsider and choose a faster query path. Alternatively split the query up into several simple selects to give the optimiser a set of less complex join optimisations.*

Clustered Indexes

Each table may have a clustered index which physically sorts the table into a particular order. This is useful where many processes require that the table is processed in a particular order. Transactions in an accounting system for example are often accessed in date order and a clustered index on the date might seem to be appropriate at first glance.

© REDWARE 1996,2001.

> *Clustered Indexes are useful for optimising queries on a range of values such as a date range.*

Adding new records to any table will require updates on the table and all associated indexes. Data is stored on the disk in logical pages and once that appropriate page is found, the page is locked and the data written in the correct place. Each page however may contain data relating to more than one record and several users may require to write to the same page at the same time. The server will of course handle this contention.

Clustered indexes will place sorted data into the pages so that sorted records may be next to each other in the same page. When two new records are added in a table with a sorted index on the date, for example, they may be added into the same page. The users are competing for the same resource and a bottleneck occurs.

Clustered indexes should not therefore be defined on a sort order that is the same or similar to the sort order of new records as they are added as this would cause contention in high transaction systems.

> *SQL Server 6.5 improves on this bottleneck but care is still required when selecting the clustered index.*

Index Tuning Wizard

Stored Procedure Recompilation

Queries are often used within stored procedures to perform a variety of tasks. The query plan is determined when the stored procedure is initially compiled and is not updated even when the UPDATE STATISTICS command is run on a table that is used in the query.

The procedure may be executed with a recompile option to force recompilation of the procedure at runtime. This will ensure that the optimiser is used in any queries but is costly as the server must perform extra work each time the procedure is executed:

```
exec byroyalty 40 with recompile
```

Another option is to define the Stored Procedure with the Recompile option by adding the WITH RECOMPILE keywords in the CREATE PROCEDURE syntax of the stored procedure definition.

This forces the stored procedure to be recompiled each time it is executed and is suitable if the stored procedure is called infrequently and requires different query plans each time. This might occur if parameters were passed that sometimes access only a few records and other times the whole table.

> *Recompilation each time is inefficient and not suitable for stored procedures that are in constant use.*

A system stored procedure can be used to automatically recompile each stored procedure that references a particular table. This stored procedure is best run after the statistics for the table have been updated.

The following example will update statistics on the Authors table and recompile the query plans for any stored procedure that used the Authors table:

```
UPDATE STATITICS authors
sp_recompile authors
```

Creating a Stored Procedure with Recompile

Manage Stored Procedures - (LOCAL)\pubs

Procedures: byroyalty (dbo)

```
if exists (select * from sysobjects where id = object_id('dbo.b
        drop procedure dbo.byroyalty
GO

CREATE PROCEDURE byroyalty @percentage int
WITH RECOMPILE AS
SELECT au_id FROM titleauthor
WHERE titleauthor.royaltyper = @percentage
GO

GRANT  EXECUTE  ON dbo.byroyalty  TO public
GO

GRANT  EXECUTE  ON dbo.byroyalty  TO guest
GO
```

Sophisticated programmers may find that the optimisation for a stored procedure differs according to the parameters passed to the procedure if the number or records selected varies considerably for example. Several stored procedures may be written and run with parameters defined to optimise the query for that particular type of parameter value. Alternatively parameters can be used in the SELECT statements so the optimiser cannot determine the query plan in advance of execution.

Deferred Updates

Many SQL Server update commands use a deferred update where the changes are made the transaction log and the old record deleted and the new one inserted. This can have an adverse effect on a high transaction system.

Non-deferred, or Direct, updates 'in place' can be designed that update the record without deleting and inserting records. This requires that no changes are made to the clustered index or to variable width fields so that the changed record can remain on the same page. The table should not have a trigger that updates other table nor should columns involved in referential integrity be updates.

Direct updates not-in-place use a delete followed by an update but are performed in a single pass. These require that no join is specified and that the index used to select the records is not updated.

Locking Issues

SQL Server, in common with other servers, will use optimistic record locking by default.

SQL Server performs all internal locking at the page level. A page contains 2K worth of information and may therefore contain more than one table record. A shared lock is placed on a page whenever it is read by a user program. This shared lock is upgraded to an Update lock when the server plans to write data to a record and then to an Exclusive lock before data is actually written to any record within the page. The locking manager will resolve any deadlocks and issues where different users attempt to update the same record at the same time.

Read only databases benefit from having the read only option set so that any shared locking overhead is removed.

Locking issues come into play on high transaction systems with the potential for many deadlocks. Minimising the records read in a single transaction will reduce the number of shared locks placed on each table. This then allows SQL Server to place Update locks more efficiently and allow the programmer to program a locking strategy for critical transactions.

The BEGIN TRANSACTION statement matched with the COMMIT TRANSACTION or ROLLBACK TRANSACTION statements help to define smaller size transactions in a stored procedure.

The optimiser hints of a SELECT statement allow a programmer to specify specific and override the default use of shared locks. The HOLDLOCK option will ask the server to maintain the shared locks on a set of records until the end of the transaction.

The UPDLOCK optimiser hint will specify that a SELECT statement uses Update locks immediately instead of shared locks and can be used when the program is attempting to update all records in the select query.

Page locks are automatically escalated to table level if a specified number are issued on a single table. The SELECT statement can be used to request a shared table lock immediately without using the escalation level if all records in a table are to be processed:

```
BEGIN TRAN
SELECT count(*) FROM taborder (TABLOCK HOLDLOCK)
```

Alternatively, the PAGLOCK hint will prevent escalation to a shared table lock if this is not desirable in a high transaction system.

The Database Administrator may need to set the configuration on the database to cope with some locking scenarios.

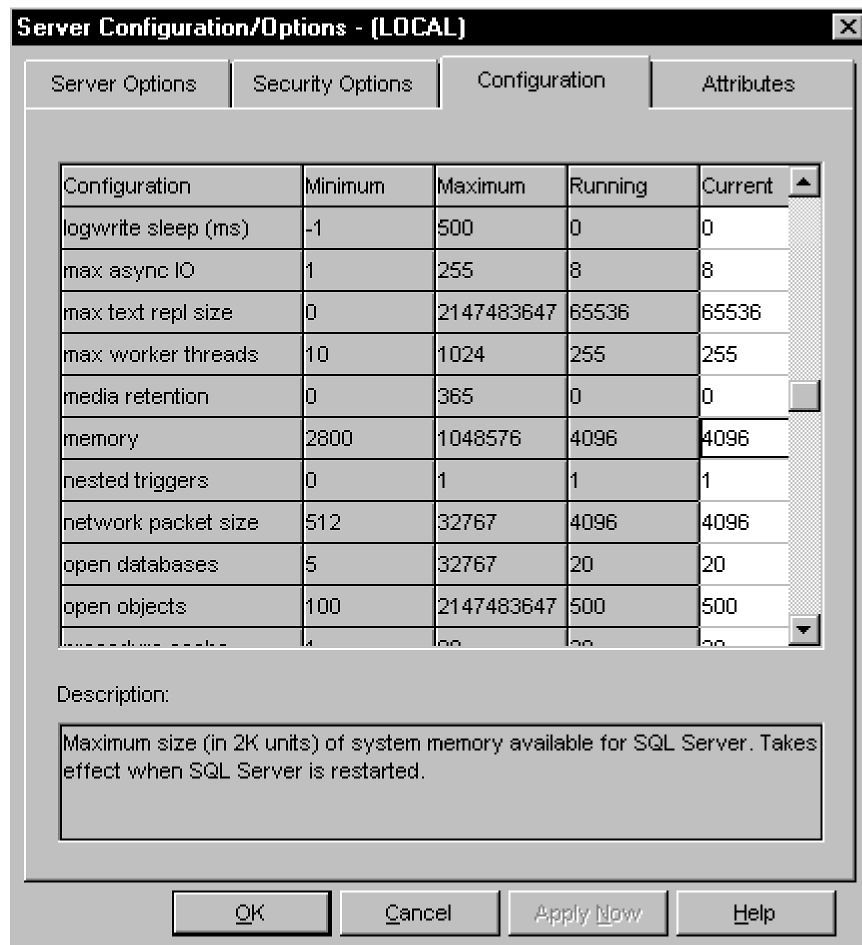
11. Configuration

Configuration in most organisations is best left to an experienced Database Administrator as there are many ramifications from setting each option. These notes are not intended as hints for configuration of a production database but provide some hints for initial configuration problems experienced when first starting with SQL Server in a development environment.

Server Configuration

Various configuration parameters may be set in the SQL Enterprise manager by selecting the required server and rightclicking to select the CONFIGURE window:

Server Configuration

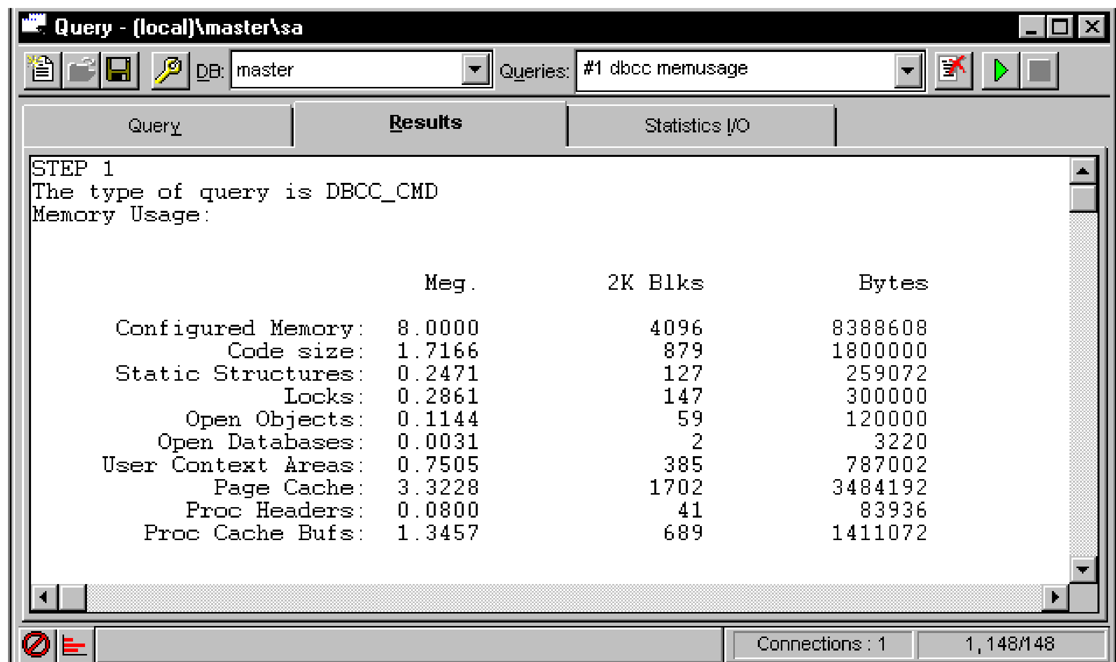


Configuration	Minimum	Maximum	Running	Current
logwrite sleep (ms)	-1	500	0	0
max async IO	1	255	8	8
max text repl size	0	2147483647	65536	65536
max worker threads	10	1024	255	255
media retention	0	365	0	0
memory	2800	1048576	4096	4096
nested triggers	0	1	1	1
network packet size	512	32767	4096	4096
open databases	5	32767	20	20
open objects	100	2147483647	500	500

Memory

The memory specified for SQL Server can be determined by using the DBCC MEMUSAGE command in the ISQL window.

DBCC MEMUSAGE



Do not specify too much memory because NT still needs to run.

Lock Escalation Percentage

Lock escalation determines the number of shared pages locks issued before the server upgrades to a table lock. The lock escalation percentage is a useful initial property to set if there are problems with locking to escalate when a certain percentage of pages in the table are locked rather than an absolute number.

Network Packet Size

The default packet size is 4096 and may require changing in specific circumstances.

This is a connection property which can be specified by the ODBC driver.

Open Databases

The default maximum number of database for simultaneous connection is 12.

User Connections

The default user connections is 15. Some applications may use more than one connection and this is a common problem initially. Take care to minimise the connections used by the application because each connection specified in the configuration uses about 40K of memory.

Database Configuration

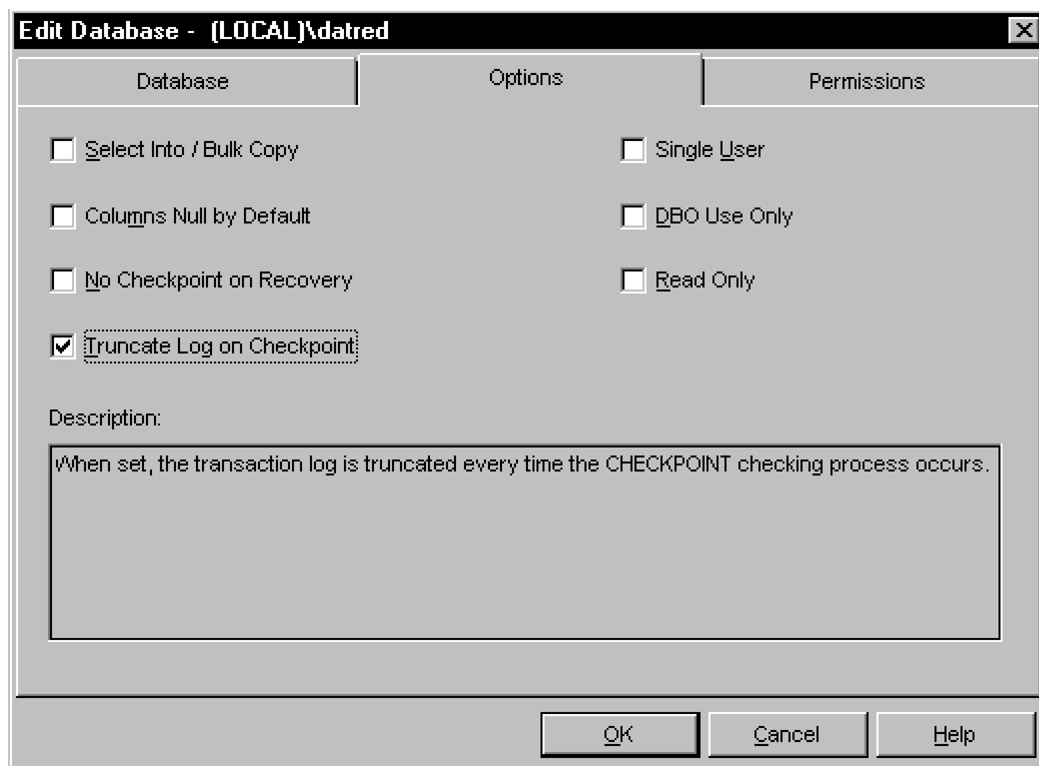
Size of tempdb

The tempdb database is used by SQL Server for all worktables. The default size is only 2MB and this will need to be enlarged before any sensible workload even for a single user can be performed.

Truncate Log on Checkpoint

Development databases do not often need a transaction log as there are too many changes made for a backup policy to be effective. The transaction log may fill up if many records are changed or uploaded and specifying the Truncate Log option on the database will stop SQL Server from creating a large Log file as it will be automatically truncated at every opportunity.

Truncate Log on Checkpoint



Deleting a Database

The corresponding .DAT file is not deleted when a database is dropped. Use the operating system to delete this file but take care.

Backups

Create initial backups of the MASTER, MODEL, and MSDB databases for recovery purposes as well as the development databases.

The MASTER database contains changes to database configurations and system tables and should be backed up after any changes to the database metadata.

The NT registry should also be backed up to help with a disaster situation.

© REDWARE 1996,2001.

12. Security

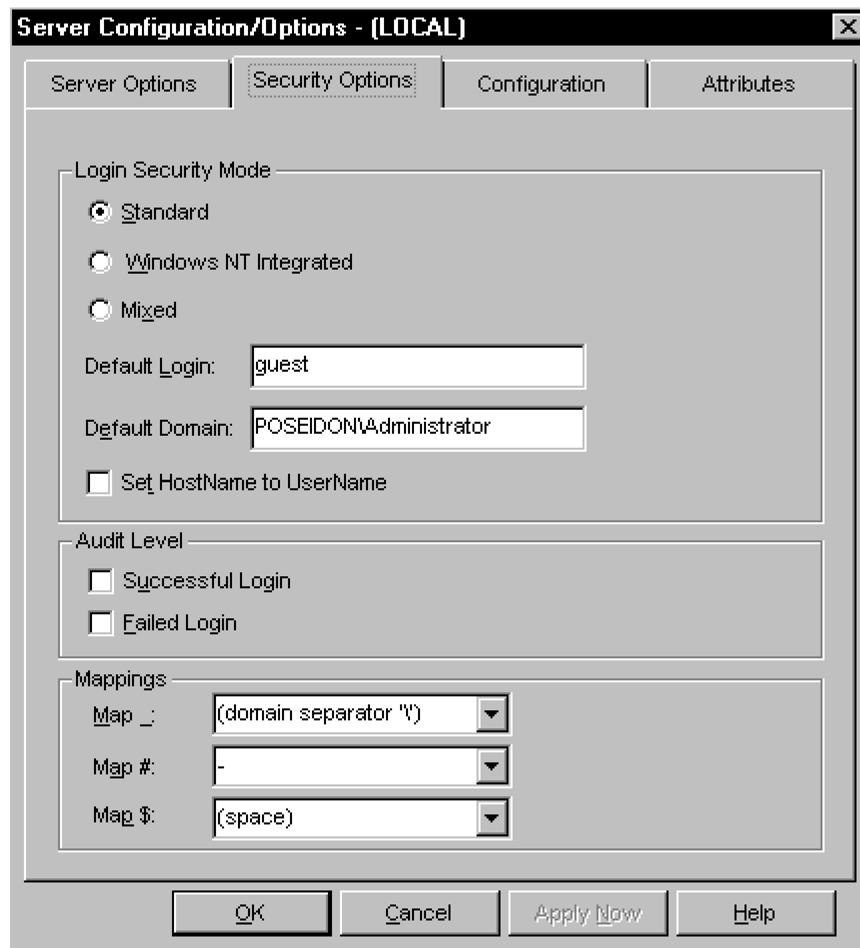
SQL Server has an internal security mechanism that offers sophisticated layers of control to all objects within a database.

Microsoft have provided an integrated security mode which integrates security with NT security. This involves the creation of two NT user groups called SQLAdmins and SQLUsers and the maintenance of SQL Server security using the SQL Security Manager application.

This handbook originally described integrated security. However from a developers point of view, security is simpler if controlled using SQL Server standard security and consequently discussion in this basic overview is limited to SQL Server security.

Security is specified in the Server Configuration window.

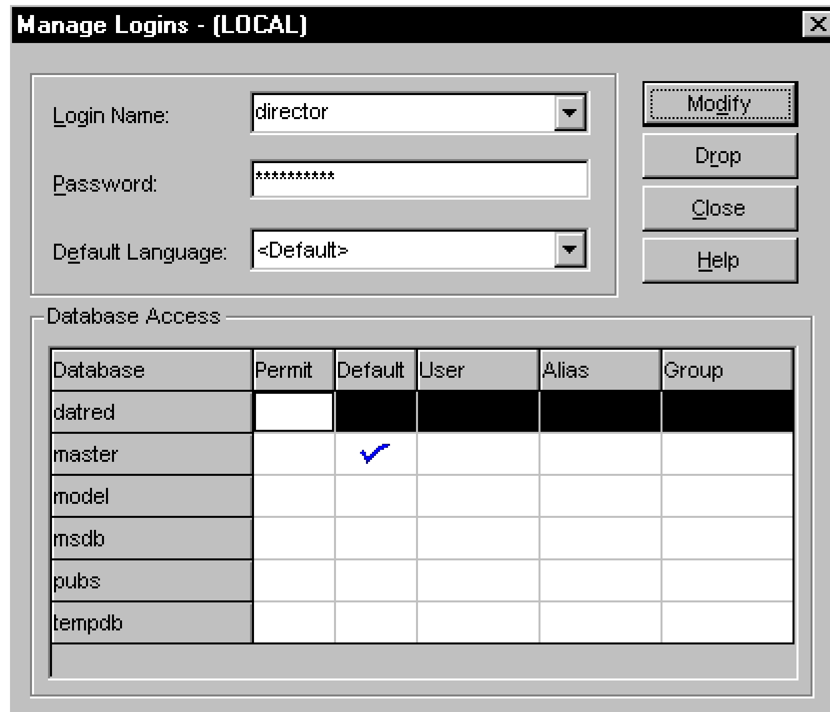
Standard Security



Server Logins

The SQL Server application requires Logins to be defined for each type of user that requires access to the database server. Logins are global to the Server and are mapped to individual users specified for each database.

Specifying a New Login



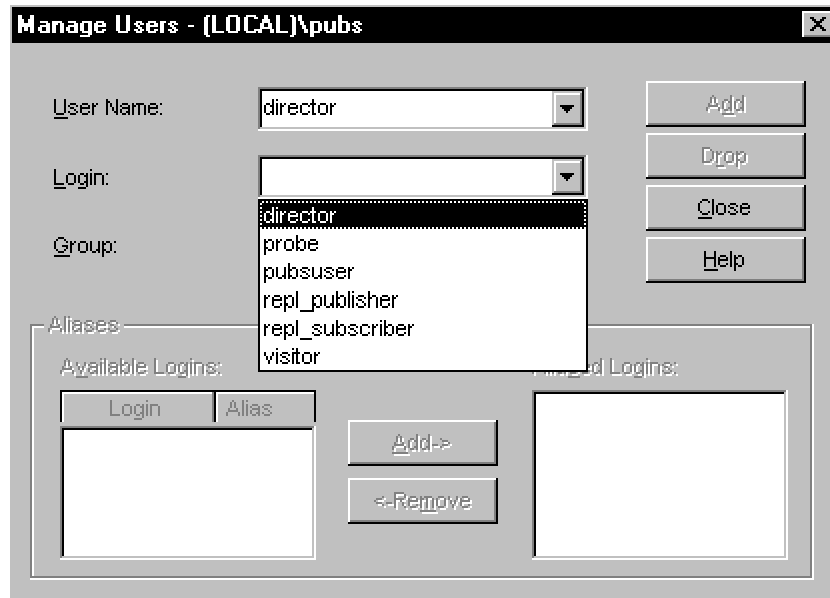
Server Logins are global to each server and are the user identifier and password that are used to log onto the server with the ODBC datasource.

Database Users

Users are defined for each Database and determine access to the database. The creator of the database is defined as the DBO user and has all privileges on the database.

Additional users can be created from the SQL Executive by selecting the Database and expanding the outline for the Groups/Users option and righclicking to select the New Users option.

Creating a new User for a Database



Each Database User is mapped to a Server Login when adding the user so that SQL Server can automatically determine the Database User when a Server Login logs onto the system. The Database User and Server Login can be given the same names to avoid confusion.

> *The Aliased Logins mover in the Manage Database Users window may be used to add several Server Logins to an individual Database User.*

Permissions

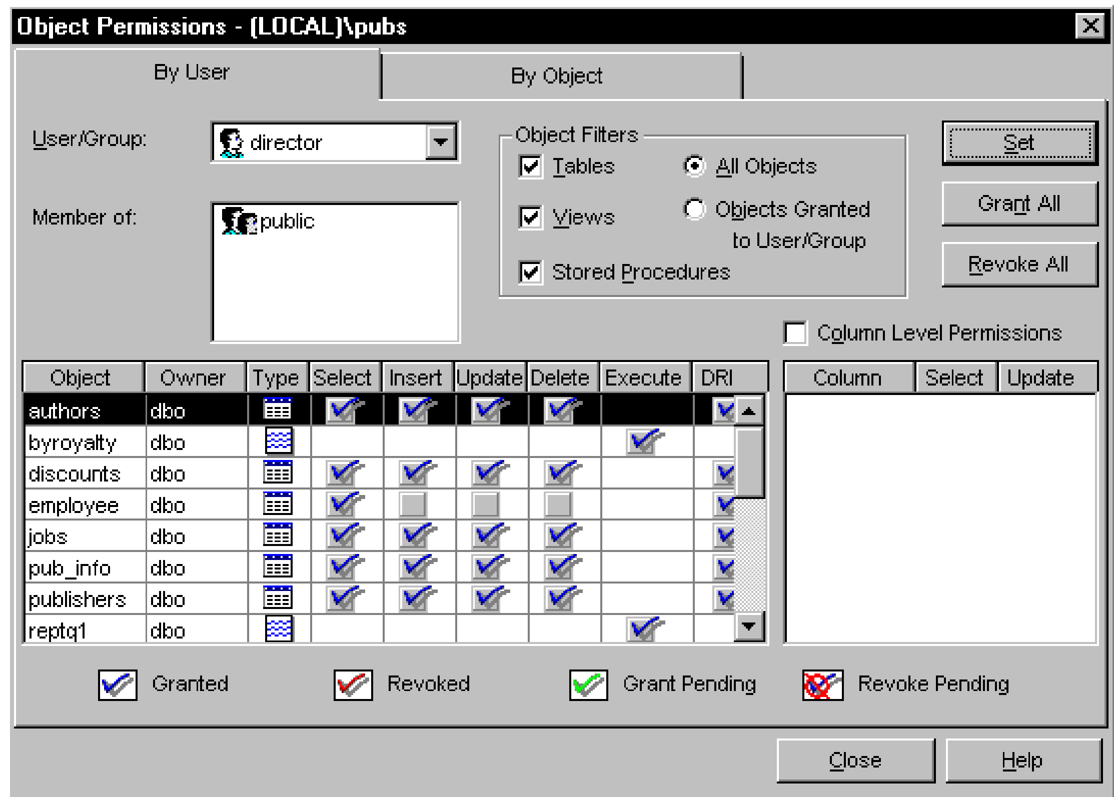
A database initially has permissions set for the DBC Database User who has all available permissions and for the Public user group who have the default permissions.

The permissions for the PUBLIC group can be revoked using the Object Permissions window available in the SQL Enterprise Manager. To revoke all permissions for PUBLIC, select the BY USER Page and select the PUBLIC Group and then press the REVOKE ALL button followed by SET to remove any permissions from the Public group.

Permissions for a specific user may be granted with the Grant All button. The Object Permissions window can be used to select all Tables, Views, or Stored Procedures and grant and revoke permissions accordingly.

© REDWARE 1996,2001.

Setting Permissions for a User



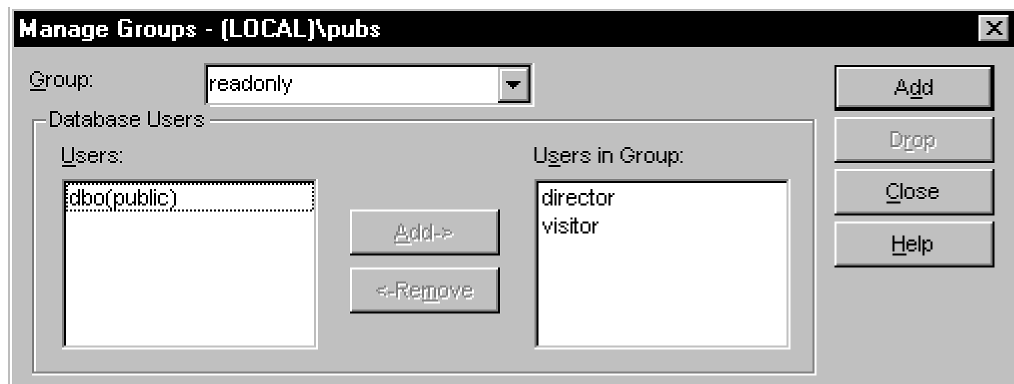
Individual Tables can be selected for each user and SELECT, INSERT, UPDATE and DELETE permissions applied. Each field on each table may also have Select and Update permissions individually applied per User.

Stored Procedures may have EXECUTE permissions granted or revoked to allow only certain users to run them.

Database Groups

A Database Group may be defined to define overall security for a Group of Database Users. The individual permission differences for a specific user may then be granted or revoked for the individual user.

Creating a Database Group



13. Index

@@FETCH_STATUS 55
@@max_connections 53
@@rowcount... 53, 63
@@version 53
ALTER TABLE 34
BackOffice... 9
BEGIN... 64
CASE.. 54
Constraints... 8, 33
covered indexes 72
CREATE PROCedure 46
CREATE TRIGGER 62
Cursors 55
Data
  Types 31
data manipulation language 6
database definition language 6
DBCC TRACEON 73
DECLARE... 52
Defaults 32
Delete
  Trigger Validation 67
DELETE.. 23
END... 64
EXECute.. 47
Extended Procedures 57
  Mail 57
FETCH 55
Field
  Types 29
  Validation in Trigger 64
Foreign Key... 36, 66
GOTO.. 52
Group.. 84
IF…ELSE.. 51
Indexes
  Allow Duplicate Rows 41
  Clustered... 36, 74
  Ignore Duplicate Key 41
  Ignore Duplicate Row.............................. 41
  Optimisation 70
INSERT... 22
integrated security 81
Locking.. 76
Login 81
Manage Tables Window.............................. 32
  Advanced Features 33
MODEL... 29
Natural Join..19
Null 32
ODBC 10
Optimiser Hints..74
Outer Join...20
Permissions 83
Primary key..67
RAISERROR 58, 64
Referential Integrity 66, 67
Registering the Server..................................13
Replication 10
ROLLBACK TRANSACTION....................64
SELECT..16, 52
  DISTINCT 20
  FROM 18
  HAVING...20
  ORDER BY 19
  WHERE 17
Showplan 72
sp_depends...56
SQL Enterprise Manager 9, 12
SQL Server 6.0 ....7, 34
SQL Server 6.5 7
SQL Service Manager..................................12
standard security 81
Stored Procedure
  Parameters..49
  Parameters passed by Reference 50
  Recompilation 75
  Returning Values 50
Stored Procedures 10, 46
Sybase 6
System Procedures 56
Task Scheduling...9
Temporary Tables 44
the SQL Enterprise Manager...........................9
Trigger 62
  Program Structure 62
Triggers..10
UPDATE..23, 65
UPDATE STATISTICS................................70
User..82
Views 10
WHERE 70
WHILE...52
xp_cmdshell 57
xp_sendmail 57
xp_startmail 57

© REDWARE 1996,2001.