**Isolation level in  SQL**

**Transaction Isolation Levels**

Closely tied in with the modes and methods of locking is the transaction isolation level. To understand the new locking behavior, you need to understand the four transaction isolation levels in SQL Server 7.0: Uncommitted Read (also called "dirty read"), Committed Read, Repeatable Read, and Serializable.

**IsolationLevels**   
The isolation level that your transaction runs in determines how sensitive your application is to changes other users' transactions make, and consequently, how long your transaction must hold locks to protect against these changes. The ANSI SQL standard defines four levels of transaction isolation. Although previous versions of SQL Server let you specify all four distinct levels of transaction isolation, there were only three different behaviors because SQL Server internally treated two of the syntactic specifications (i.e., Repeatable Read and Serializable) as synonymous.

You can change the level of isolation that a particular connection is operating in by using the SET TRANSACTION ISOLATION LEVEL command. Keep in mind that the SET command applies only to your current connection, and every time you make a new connection (or open a new window in the Query Analyzer), you'll be back in the default isolation level. I'll use each of the four isolation levels in the examples to follow.

To see how each level behaves, you can use the script in Listing 1, page 20, to create a table with a few rows in it. I'll refer back to this table in examples for each of the four isolation levels.

**UncommittedRead**  
Uncommitted Read, or dirty read, lets a transaction read any data currently on a data page, whether or not that data has been committed. For example, although another user might have a transaction in progress that has updated data, and that transaction is holding exclusive locks on the data, your transaction can read the data anyway, and possibly take further actions based on the values you read. The other user might then decide to roll back his or her transaction, so logically, those changes never occurred. Although this scenario isn't desirable, with Uncommitted Read you won't get stuck waiting for a lock, nor will your reads acquire share locks that might affect others.

Let's see how Uncommitted Read behaves. Use the SQL Server 7.0 Query Analyzer, and start two separate connections. Use the pubs database in each one. In the first connection, begin a transaction, but don't commit it:

BEGIN TRAN

UPDATE ISOLATION\_TEST

SET col2 = 'New Value'

<click the green arrow to execute >

Now, use the second connection, and change your isolation level before trying to access the same table.

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT \* FROM ISOLATION\_TEST

<click the green arrow to execute >

All the values in col1 are 0, even though the transaction in the first connection has not committed yet. In fact, the transaction might never commit. If you took some action based on the fact that all the values are 0, you could regret it if the changes turned out not to be permanent. Back in the first connection, roll back the transaction:

ROLLBACK TRAN

<click the green arrow to execute >

Now rerun the SELECT statement in the second connection to see that all the values are back to what they were before. If you're following along with these examples, make sure you close your connections after each one, so that all outstanding locks are released.

**CommittedRead**   
Committed Read is SQL Server's default isolation level. It ensures that an operation will never read data another application has changed but not yet committed. Because you can never read uncommitted data, if a transaction running with Committed Read isolation revisits data, that data might have changed, or new rows might appear that meet the criteria of the original query. Rows that appear in this way are called phantoms.

So Committed Read behavior has two aspects. To see the first aspect, you can run the above example, without setting the second connection to use isolation level Read Uncommitted. The second connect would then block on the SELECT statement; it can't read the changes the first connection has made but not yet committed (or rolled back). To see the second Committed Read behavior, close all the connections in the Query Analyzer from the previous example, and open two new connections using pubs again. In the first connection, run the following batch:

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

BEGIN TRAN

SELECT AVG(col1) from ISOLATION\_TEST

<click the green arrow to execute >

In the second connection, update the table:

UPDATE ISOLATION\_TEST

SET col1 = 500 WHERE col1 = 50

<click the green arrow to execute >

Notice that the update is successful, even though the first connection is still inside a transaction.

Go back to the first connection and run the same SELECT statement:

SELECT AVG(col1) from ISOLATION\_TEST

<click the green arrow to execute >

The average value is now different. The default isolation level does not prevent another connection from changing data you have read. Because you are not guaranteed to see the same data if you rerun the SELECT within the transaction, the read operations are not guaranteed to be repeatable.

**RepeatableRead**If you want the read operations to be repeatable, choose the third isolation level. The Repeatable Read isolation level adds to the properties of Committed Read by ensuring that if a transaction revisits data or if a query is reissued, the data will not have changed. In other words, issuing the same query twice within a transaction won't pick up any changes to data values that another user's transaction has made. No other user can modify the data that your transaction visits as long as you have not yet committed or rolled back your transaction.

To see Repeatable Read behavior, close all the connections, and open two new ones in pubs. Issue the same two queries as above, but this time, have the first connection SET ISOLATION LEVEL REPEATABLE READ.

The second connection will have to use a slightly different update statement, because the value of 50 for col1 no longer exists:

UPDATE ISOLATION\_TEST

SET col1 = 5000 WHERE col1 = 500

<click the green arrow to execute >

This update will block when it tries to update the ISOLATION\_TEST table. And the first connection will get the same result when it reissues its original SELECT. Preventing nonrepeatable reads is a desirable safeguard, but it comes at a price. The cost of this extra safeguard is that all the shared locks in a transaction must be held until the completion (COMMIT or ROLLBACK) of the transaction.

However, Repeatable Read isolation doesn't prevent all possible changes. It protects only the data that you have read. The following example shows you what this protection means. Close all connections, and open two new ones connecting to pubs. In the first connection, start a transaction in Repeatable Read isolation level and look for all rows that meet a certain condition.

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

BEGIN TRAN

SELECT \* FROM ISOLATION\_TEST

WHERE col1 BETWEEN 20 AND 40

<click the green arrow to execute >

In the second connection, insert a new row:

INSERT INTO ISOLATION\_TEST

VALUES (25, 'New Row')

<click the green arrow to execute >

Go back to the first connection, and reexecute the SELECT:

SELECT \* FROM ISOLATION\_TEST

WHERE col1 BETWEEN 20 AND 40

<click the green arrow to execute >

The second time you execute the same statement, the new row appears. Because the row doesn't even exist the first time you run the SELECT statement, it isn't locked. This new row that appears is called a phantom. You can prevent phantoms with the fourth isolation level.

**Serializable**   
The Serializable isolation level ensures that if a query is reissued, no data will have changed and no new rows will appear in the interim. In other words, you won't see phantoms if the same query is issued twice within a transaction. Rerun the example from the Repeatable Reads section, inserting a row with a col1 value of 35. But this time, set your isolation level to SERIALIZABLE. The second connection will block when you try to do the INSERT, and the first connection will read exactly the same rows each time.

You pay a price to prevent phantoms. In addition to locking all the data you have read, enforcing the Serializable isolation level requires that SQL Server also lock data that doesn't exist! The Serializable level gets its name from the fact that running multiple serializable transactions at the same time is the equivalent of running them one at a time—that is, serially—regardless of sequence.

Controlling the Isolation Level SQL Server's default isolation level is Committed Read, but as you've seen, you can override this setting within your application. The most straightforward way is by using the SET command:

SET TRANSACTION ISOLATION LEVEL

[READ UNCOMMITTED | READ COMMITTED | REPEATABLE

READ | SERIALIZABLE]

Previous versions of SQL Server treated Repeatable Read and Serializable as synonymous. I thought the difference was that Repeatable Reads prevented UPDATE operations, and Serializable prevented INSERTs and DELETEs. But the difference is in what data is locked. Repeatable Read locks only the data that has been read. With Serializable, SQL Server has to guarantee complete serializability, so it locks ranges of data.

Previous versions of SQL Server gave you no way to distinguish between these two levels, because the smallest unit of locking was an entire page. Locking the page containing the col1 values between 20, 30, and 40 prevented changes to those specific rows but also prevented inserts anywhere on the same page. With row-level locking, SQL Server 7.0 can lock only the accessed rows. SQL Server 7.0 also introduced a new kind of lock to lock the ranges of data between two rows, which I'll discuss in an upcoming article.

**What are User Defined Types? and how do you use them?**

**Built-in data types**

In Microsoft SQL Server 2000, each object (such as column, variable,

or parameter) has a related data type, which is an attribute that

specifies the type of data that the object can hold.

SQL Server 2000 ships with 27 built-in (system) data types. They are:

| **Data Types** | **Description** |
| --- | --- |
| bigint | Integer data from -2^63 through 2^63-1 |
| int | Integer data from -2^31 through 2^31 - 1 |
| smallint | Integer data from -2^15 through 2^15 - 1 |
| tinyint | Integer data from 0 through 255 |
| bit | Integer data with either a 1 or 0 value |
| decimal | Fixed precision and scale numeric data from -10^38 +1 through 10^38 -1 |
| numeric | Fixed precision and scale numeric data from -10^38 +1 through 10^38 -1 |
| money | Monetary data values from -2^63 through 2^63 - 1 |
| smallmoney | Monetary data values from -214,748.3648 through +214,748.3647 |
| float | Floating precision number data from -1.79E + 308 through 1.79E + 308 |
| real | Floating precision number data from -3.40E + 38 through 3.40E + 38 |
| datetime | Date and time data from January 1, 1753, through December 31, 9999, with an accuracy of 3.33 milliseconds |
| smalldatetime | Date and time data from January 1, 1900, through June 6, 2079, with an accuracy of one minute |
| char | Fixed-length character data with a maximum length of 8,000 characters |
| varchar | Variable-length data with a maximum of 8,000 characters |
| text | Variable-length data with a maximum length of 2^31 - 1 characters |
| nchar | Fixed-length Unicode data with a maximum length of 4,000 characters |
| nvarchar | Variable-length Unicode data with a maximum length of 4,000 characters |
| ntext | Variable-length Unicode data with a maximum length of 2^30 - 1 characters |
| binary | Fixed-length binary data with a maximum length of 8,000 bytes |
| varbinary | Variable-length binary data with a maximum length of 8,000 bytes |
| image | Variable-length binary data with a maximum length of 2^31 - 1 bytes |
| cursor | A reference to a cursor |
| sql\_variant | A data type that stores values of various data types, except text, ntext, timestamp, and sql\_variant |
| table | A special data type used to store a result set for later processing |
| timestamp | A database-wide unique number that gets updated every time a row gets updated |
| uniqueidentifier | A globally unique identifier |

Some of these data types (bigint, sql\_variant, and table) are only

available in SQL Server 2000, while some were supported under the

previous SQL Server versions.

**User-defined data types**

SQL Server 2000 supports user-defined data types too. User-defined data types provide a mechanism for applying a name to a data type that is more descriptive of the types of values to be held in the object. Using user-defined data type can make it easier for a programmer

or database administrator to understand the intended use of any object defined with the data type. The user-defined data types are based on the system data types and can be used to predefine several attributes of a column, such as its data type, length, and whether it supports

NULL values. To create a user-defined data type, you can use the **sp\_addtype** system stored procedure or you could add one using the Enterprise Manager. When you create a user-defined data type, you should specify the following three properties:

* Data type's name.
* Built-in data type upon which the new data type is based.
* Whether it can contain NULL values.

The following example creates a user-defined data type based on money

data type named **cursale** that cannot be NULL:

**EXEC sp\_addtype cursale, money, 'NOT NULL'**

**GO**

Both system and user-defined data types are used to enforce data integrity. It is very important that we put a lot of effort while designing tables: the better you design your tables, the more time you can work without any performance problems. In an ideal case, you never will update the structure of your tables.

**Tips to choose the appropriate data types**

SQL Server 2000 stores data in a special structure called data pages that are 8Kb (8192 bytes) in size. Some space on the data pages is used to store system information, which leaves 8060 bytes to store user's data. So, if the table's row size is 4040 bytes, then only one row will be placed on each data page. If you can decrease the row size to 4030 bytes, you can store two rows within a single page because two rows can be placed into data page. The lesser the space used, the smaller the table and index, and lesser the I/O SQL Server has to perform when reading data pages from disk. So, you should design your tables in such a way as to maximize the number of rows that can fit into one data page. To maximize the number of rows that can fit into one data page, you should specify the narrowest columns you can. The narrower the columns are, the lesser the data that is stored, and the faster SQL Server is able to read and write data.

Try to use the following tips when choose the data types:

* **If you need to store integer data from 0 through 255, use *tinyint* data type.**

The columns with tinyint data type use only one byte to store their values, in comparison with two bytes, four bytes and eight bytes used to store the columns with smallint, int and bigint data types accordingly. For example, if you design tables for a small company with 5-7 departments, you can create the departments table with the DepartmentID tinyint column to store the unique number of each department.

* **If you need to store integer data from -32,768 through 32,767, use *smallint***

**data type.**

The columns with smallint data type use only two bytes to store their values, in comparison with four bytes and eight bytes used to store the columns with int and bigint data types accordingly. For example, if you design tables for a company with several hundred employees, you can create an employee table with the EmployeeID smallint column to store the unique number of each employee.

* **If you need to store integer data from -2,147,483,648 through 2,147,483,647,**

**use *int* data type.**

The columns with int data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with bigint data types. For example, to design tables for a library with more than 32,767 books, create a books table

with a BookID int column to store the unique number of each book.

* **Use *smallmoney* data type instead of money data type, if you need to store monetary data values from 214,748.3648 through 214,748.3647.**

The columns with smallmoney data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with money data types. For example, if you need to store the monthly employee payments, it might be possible to use a column with the smallmoney data type instead of money data type.

* **Use *smalldatetime* data type instead of datetime data type, if you need**

**to store the date and time data from January 1, 1900 through June 6, 2079,**

**with accuracy to the minute.**

The columns with smalldatetime data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with datetime data types. For example, if you need to store the employee's hire date, you can use column with the smalldatetime data type instead of datetime data type.

* **Use varchar/nvarchar columns instead of text/ntext columns whenever possible.**

Because SQL Server stores text/ntext columns on the Text/Image pages

separately from the other data, stored on the Data pages, it can

take more time to get the text/ntext values.

* **Use char/varchar columns instead of nchar/nvarchar if you do not need to store unicode data.**

The char/varchar value uses only one byte to store one character,

the nchar/nvarchar value uses two bytes to store one character,

so the char/varchar columns use two times less space to store data

in comparison with nchar/nvarchar columns.

**How do u trigger a COM from SQL?**

sp\_OACreate

Creates an instance of the OLE object on an instance of Microsoft® SQL Server™.

**Syntax**

**sp\_OACreate** *progid***,** | *clsid***,**     *objecttoken* **OUTPUT**  [ **,** *context* ]

**Arguments**

*progid*

Is the programmatic identifier (ProgID) of the OLE object to create. This character string describes the class of the OLE object and has the form:

**'***OLEComponent***.***Object***'**

*OLEComponent* is the component name of the OLE Automation server, and *Object* is the name of the OLE object. The specified OLE object must be valid and must support the **IDispatch** interface.

For example, SQLDMO.SQLServer is the ProgID of the SQL-DMO **SQLServer** object. SQL-DMO has a component name of SQLDMO, the SQLServer object is valid, and (like all SQL-DMO objects) the **SQLServer** object supports **IDispatch**.

*clsid*

Is the class identifier (CLSID) of the OLE object to create. This character string describes the class of the OLE object and has the form:

**'{***nnnnnnnn***-***nnnn***-***nnnn***-***nnnn***-***nnnnnnnnnnnn***}'**

The specified OLE object must be valid and must support the **IDispatch** interface.

For example, {00026BA1-0000-0000-C000-000000000046} is the CLSID of the SQL-DMO **SQLServer** object.

*objecttoken* **OUTPUT**

Is the returned object token, and must be a local variable of data type **int**. This object token identifies the created OLE object and is used in calls to the other OLE Automation stored procedures.

*context*

Specifies the execution context in which the newly created OLE object runs. If specified, this value must be one of the following:

**1** = In-process (.dll) OLE server only**4** = Local (.exe) OLE server only  
**5** = Both in-process and local OLE server allowed

If not specified, the default value is 5. This value is passed as the *dwClsContext* parameter of the call to **CoCreateInstance**.

If an in-process OLE server is allowed (by using a context value of **1** or **5** or by not specifying a context value), it has access to memory and other resources owned by SQL Server. An in-process OLE server may damage SQL Server memory or resources and cause unpredictable results, such as a SQL Server access violation.

When you specify a context value of **4**, a local OLE server does not have access to any SQL Server resources, and it cannot damage SQL Server memory or resources.

**Note**  The parameters for this stored procedure are specified by position, not by name.

**Return Code Values**

0 (success) or a nonzero number (failure) that is the integer value of the HRESULT returned by the OLE Automation object.

For more information about HRESULT Return Codes, see OLE Automation Return Codes and Error Information.

**Remarks**

The created OLE object is automatically destroyed at the end of the Transact-SQL statement batch.

**Permissions**

Only members of the **sysadmin** fixed server role can execute **sp\_OACreate**.

**What are optimizer hints**

How do you use optimizer hints with the following query

SELECT   
FROM myTable MTINNER JOIN MyOtherTable MOT ON MT.Col1 = MOT.Col1 AND MT.Col2 = MOT.Col2  
I want to force use of MyIndex

USE pubs

SELECT t1.au\_lname FROM authors

t1 (index=aunmind) INNER JOIN titleauthor t2 index=titleidind) ON t1.au\_id = t2.au\_id

It will force to use aunmind index on table authors and titleidind indexon table titleauthor.

**What is clustered index and a non-clustered index?**

## ***Clustered Indexes***

There can be only one clustered index per table because, while the upper parts of the clustered index B-tree structure are organized like the nonclustered index B-tree structures, the bottom level of the clustered index B-tree consists of the actual 8-KB data pages associated with the table. There are performance implications:

* Retrieval of SQL data based on key search with a clustered index requires no bookmark lookup (and a likely nonsequential change of location on the hard disk) to get to the associated data page, because the leaf level of the clustered index is already the associated data page.
* The leaf level of the clustered index is sorted by the columns that comprise the clustered index. Because the leaf level of the clustered index contains the actual 8-KB data pages of the table, the row data of the entire table is physically arranged on the disk drive in the order determined by the clustered index. This provides a potential I/O performance advantage when fetching a significant number of rows from tables greater than 64-KB based on the value of the clustered index, because sequential disk I/O is being used unless page splitting is occuring on this table. For more information about page-splitting, see “FILLFACTOR and PAD\_INDEX” later in this document. You should pick the clustered index on a table based on a column that is used to perform range scans to retrieve a large number of rows.

## **Nonclustered Indexes**

Nonclustered indexes are most useful for fetching few rows with good selectivity from large SQL Server tables based on a key value. Nonclustered indexes are B-trees formed out of 8-KB index pages. The bottom or leaf level of the B-tree of index pages contains all the data from the columns that comprised that index. When a nonclustered index is used to retrieve information from a table based on a match with the key value, the index B-tree is traversed until a key match is found at the leaf level of the index. A bookmark lookup is made if columns from the table are needed that did not form part of the index. This bookmark lookup will likely require a nonsequential I/O operation on the disk. It might even require the data to be read from another disk if the table and its accompanying index B-tree(s) are large. If multiple bookmark lookup lead to the same 8-KB data page, then there is less of an I/O performance penalty since it is only necessary to read the page into data cache once. For each row returned for a SQL query that involves searching with a nonclustered index, one bookmark lookup is required. These bookmark lookups are the reason that nonclustered indexes are better suited for SQL queries that return only one or a few rows from the table. Queries that require many rows to be returned are better served with a clustered index.

For more information, see this keyword in SQL Server 7.0 Books Online: nonclustered index.

## ***Covering Indexes***

A special situation that occurs with nonclustered indexes is called the covering index. A covering index is a nonclustered index built upon all of the columns required to satisfy a SQL query, both in the selection criteria and in the WHERE clause. Covering indexes can save I/O and improve query performance. But you must balance the costs of creating a new index (with its associated B-tree index structure maintenance) with the I/O performance gain the covering index will bring. If a covering index will benefit a query or a set of queries that run often on SQL Server, then creating the covering index may be worthwhile.

Example:

SELECT col1,col3 FROM table1 WHERE col2 = 'value'

CREATE INDEX indexname1 ON table1(col2,col1,col3)

or

From SQL Server Enterprise Manager, use the Create Index Wizard.

The **indexname1** index in this example is a covering index because it includes all columns from the SELECT statement and the WHERE clause. During the execution of this query, SQL Server does not need to access the data pages associated with **table1**. SQL Server can obtain all of the information required to satisfy the query by using the index called **indexname1**. When SQL Server has traversed the B-tree associated with **indexname1** and has found the range of index keys where **col2** is equal to value, SQL Server fetches all of required data (**col1**,**col2**,**col3**) from the leaf level of the covering index. This provides I/O performance in two ways:

* SQL Server obtains all required data from an index page, not a data page, so the data is more compressed and SQL Server saves disk I/O operations.
* The covering index organizes all of the required data by **col2** physically on the disk. The hard disk drives return all of the index rows associated with the WHERE clause (**col2** = value) in sequential order, which gives better I/O performance. From a disk I/O standpoint a covering index becomes a clustered index for this query and any other query that can be satisfied completely by the columns in the covering index.

If the number of bytes from all the columns in the index is small compared to the number of bytes in a single row of that table, and you are certain the query taking advantage of the covered index will be executed frequently, then it may make sense to use a covering index. But, before building a lot of covered indexes, consider how SQL Server 7.0 can effectively and automatically create covered indexes for queries on the fly.

**What is an indexed view and why do you need it?**

An indexed view allows indexes to be created on views, where the result set of the view is stored and indexed in the database.

Indexed views work best when the underlying data is infrequently updated. The maintenance of an indexed view can be higher than the cost of maintaining a table index. If the underlying data is updated frequently, then the cost of maintaining the indexed view data may outweigh the performance benefits of using the indexed view.

Indexed views improve the performance of these types of queries:

* Joins and aggregations that process many rows.
* Join and aggregation operations that are frequently performed by many queries.
* Note:
* Views can be used to partition data across multiple databases or instances of Microsoft® SQL Server™ 2000.
* Views in all versions of SQL Server are updatable (can be the target of UPDATE, DELETE, or INSERT statements), as long as the modification affects only one of the base tables referenced by the view.

**What is an execution plan?**

The execution plan determines the order in which a given query is executed , and the cost involved(CPU time and resources) at each step . The order involves , whether an Index(if available) is used to retrieve the rows or makes an FTS(Full Table Scan ). Looking at the E-plan, the developer can modify the query to ensure the rows are retrieved in faster way.

The e-plan for a query can be seen by typing the query in the query analyzer,select it , and choosing Query->Show Estimated Execution Plan(CTRL+M)

**How will you return results as xml from SQL server?**

***Using “FOR XML” Clause in the query***

**Select \* from Authors FOR XML Raw**

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

Arguments

XML *mode*

Specifies the XML mode. XML mode determines the shape of the resulting XML.   
*mode* can be RAW, AUTO, or EXPLICIT.

**RAW Mode**

RAW mode transforms each row in the query result set into an XML element with the generic identifier row. Each column value that is not NULL is mapped to an attribute of the XML element in which the attribute name is the same as the column name.

The BINARY BASE64 option must be specified in the query in order to return the binary data in base64-encoded format. In RAW mode, retrieving binary data without specifying the BINARY BASE64 option results in an error.

When an XML-Data schema is requested, the schema, declared as a namespace, appears at the beginning of the data. In the result, the schema namespace reference is repeated for every top-level element.

**AUTO Mode**

AUTO mode returns query results as nested XML elements. Each table in the FROM clause, from which at least one column is listed in the SELECT clause, is represented as an XML element. The columns listed in the SELECT clause are mapped to the appropriate attribute of the element. When the ELEMENTS option is specified, the table columns are mapped to subelements instead of attributes. By default, AUTO mode maps the table columns to XML attributes.

A table name (or the alias, if provided) maps to the XML element name. A column name (or the alias, if provided) maps to an attribute name or to a non-complex subelement name when the ELEMENTS option is specified in the query.

The hierarchy (nesting of the elements) in the result set is based on the order of tables identified by the columns that are specified in the SELECT clause; therefore, the order in which column names are specified in the SELECT clause is significant.

The tables are identified and nested in the order in which the column names are listed in the SELECT clause. The first, leftmost table identified forms the top element in the resulting XML document. The second leftmost table (identified by columns in the SELECT statement) forms a subelement within the top element (and so on).

If a column name listed in the SELECT clause is from a table that is already identified by a previously specified column in the SELECT clause, the column is added as an attribute (or as a subelement if the ELEMENTS option is specified) of the element already created, instead of opening a new hierarchy (adding a new subelement for that table).

**EXPLICIT Mode**

In EXPLICIT mode, the query writer controls the shape of the XML document returned by the execution of the query. The query must be written in a specific way so that the additional information about the expected nesting is explicitly specified as part of the query. You can also specify additional configurations at the column level by using the directives. When you specify EXPLICIT mode, you must assume the responsibility for ensuring that the generated XML is well-formed and valid (in the case of an XML-DATA schema).

**XMLDATA**

Specifies that an XML-Data schema should be returned. The schema is prepended to the document as an inline schema.

**ELEMENTS**

If the ELEMENTS option is specified, the columns are returned as subelements. Otherwise, they are mapped to XML attributes. This option is supported in AUTO mode only.

**BINARY BASE64**

If the BINARY Base64 option is specified, any binary data returned by the query is represented in base64-encoded format. To retrieve binary data using RAW and EXPLICIT mode, this option must be specified. In AUTO mode, binary data is returned as a reference by default.

**How do you design a database? - basics of DB Design, SPs, triggers?**

1. **Normalize your tables to the third normal form.**  
   A table is in third normal form (3NF) if it is in second normal form (2NF) and if it does not contain transitive dependencies. In most cases, you should normalize your tables to the third normal form. The normalization is used to reduce the total amount of redundant data in the database. The less data there is, the less work SQL Server has to perform, speeding its performance.
2. **Consider the denormalization of your tables from the forth or fifth normal forms to the third normal form.**  
   Normalization to the forth and fifth normal forms can result in some performance degradation, especially when you need to perform many joins against several tables. It may be necessary to denormalize your tables to prevent performance degradation.
3. **Consider horizontal partitioning of the very large tables into the current and the archives versions.**  
   The less space used, the smaller the table, the less work SQL Server has to perform to evaluate your queries. For example, if you need to query only data for the current year in your daily work, and you need all the data only once per month for the monthly report, you can create two tables: one with the current year's data and one with the old data.
4. **Create the table's columns as narrow as possible.**  
   This can reduce the table's size and improve performance of your queries as well as some maintenance tasks (such as backup, restore and so on).
5. **Try to reduce the number of columns in a table.**  
   The fewer the number of columns in a table, the less space the table will use, since more rows will fit on a single data page, and less I/O overhead will be required to access the table's data.
6. **Try to use constraints instead of triggers, rules, and defaults whenever possible.**  
   Constraints are much more efficient than triggers and can boost performance. Constraints are more consistent and reliable in comparison to triggers, rules and defaults, because you can make errors when you write your own code to perform the same actions as the constraints.
7. **If you need to store integer data from 0 through 255, use *tinyint* data type.**  
   The columns with tinyint data type use only one byte to store their values, in comparison with two bytes, four bytes and eight bytes used to store the columns with smallint, int and bigint data types accordingly. For example, if you design tables for a small company with 5-7 departments, you can create the departments table with the DepartmentID tinyint column to store the unique number of each department.
8. **If you need to store integer data from -32,768 through 32,767, use *smallint* data type.**  
   The columns with smallint data type use only two bytes to store their values, in comparison with four bytes and eight bytes used to store the columns with int and bigint data types respectively. For example, if you design tables for a company with several hundred employees, you can create an employee table with the EmployeeID smallint column to store the unique number of each employee.
9. **If you need to store integer data from -2,147,483,648 through 2,147,483,647, Use *int* data type.**  
   The columns with int data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with bigint data types. For example, to design tables for a library with more than 32,767 books, create a books table with a BookID int column to store the unique number of each book.
10. **Use *smallmoney* data type instead of money data type, if you need to store monetary data values from 214,748.3648 through 214,748.3647.**  
    The columns with smallmoney data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with money data types. For example, if you need to store the monthly employee payments, it might be possible to use a column with the smallmoney data type instead of money data type.
11. **Use *smalldatetime* data type instead of datetime data type, if you need to store the date and time data from January 1, 1900 through June 6, 2079, with accuracy to the minute.**  
    The columns with smalldatetime data type use only four bytes to store their values, in comparison with eight bytes used to store the columns with datetime data types. For example, if you need to store the employee's hire date, you can use column with the smalldatetime data type instead of datetime data type.
12. **Use varchar/nvarchar columns instead of text/ntext columns whenever possible.**  
    Because SQL Server stores text/ntext columns on the Text/Image pages separately from the other data, stored on the Data pages, it can take more time to get the text/ntext values.
13. **Use char/varchar columns instead of nchar/nvarchar if you do not need to store unicode data.**  
    The char/varchar value uses only one byte to store one character, the nchar/nvarchar value uses two bytes to store one character, so the char/varchar columns use two times less space to store data in comparison with nchar/nvarchar columns.
14. **Consider setting the 'text in row' SQL Server 2000 table's option.**  
    The text, ntext, and image values are stored on the Text/Image pages, by default. This option specifies that small text, ntext, and image values will be placed on the Data pages with other data values in a data row. This can increase the speed of read and write operations and reduce the amount of space used to store small text, ntext, and image data values. You can set the 'text in row' table option by using the sp\_tableoption stored procedure.
15. **If you work with SQL Server 2000, use cascading referential integrity constraints instead of triggers whenever possible.**  
    For example, if you need to make cascading deletes or updates, specify the ON DELETE or ON UPDATE clause in the REFERENCES clause of the CREATE TABLE or ALTER TABLE statements. The cascading referential integrity constraints are much more efficient than triggers and can boost performance.

**What are the normal forms and which is the best?**

Use this technique to improve system performance and accuracy

Database normalization, or data normalization, is a technique to organize the contents of the tables for transactional databases and data warehouses. Normalization is part of successful database design; without normalization, database systems can be inaccurate, slow, and inefficient, and they might not produce the data you expect.

Following a successful SQL Server installation (or any database management system installation), you'll have to create a database to hold the data (see SQL Server *Books Online*—*BOL*—for more information about how to create a database). After you've created the database framework, you must organize the files in such a way that you can easily manage them. The primary tool to help organize the data is the table, which looks like a two-dimensional structure that encompasses rows and columns.

A database table looks like a spreadsheet. One item of data, such as a first name, a last name, a phone number, or a street address, appears in each box, or cell, at each intersection of a row and column. Database designers use the term *atomicity* to describe this organization of data into one data item in each cell.

When you normalize a database, you have four goals: arranging data into logical groupings such that each group describes a small part of the whole; minimizing the amount of duplicate data stored in a database; organizing the data such that, when you modify it, you make the change in only one place; and building a database in which you can access and manipulate the data quickly and efficiently without compromising the integrity of the data in storage.

Data normalization helps you design new databases to meet these goals or to test databases to see whether they meet the goals. Sometimes database designers refer to these goals in terms such as *data integrity, referential integrity,* or *keyed data access*. Ideally, you normalize data before you create database tables. However, you can also use these techniques to test an existing database.

Data normalization is primarily important in the transactional, or online transactional processing (OLTP), database world, in which data modifications (e.g., inserts, updates, deletes) occur rapidly and randomly throughout the stored data. In contrast, a data warehouse contains a large amount of denormalized and summarized data—precalculated to avoid the performance penalty of ad hoc joins. In a data warehouse, updates happen periodically under extremely controlled circumstances. End users' updates to data in data warehouses are uncommon. This article addresses the normalization of OLTP data.

**Data and Information**

Data and information are terms people use interchangeably in everyday speech, but they mean different things. Data are raw facts, such as a name, a number, or a date. Information is organized, processed data. A data item (e.g., the date 7/15/99) means little. When you associate the data with other data items, such as a deadline and a subject, you can create information. For example, the deadline for your next project might be 7/15/99. You store data in a database; you retrieve information from the database.

One cornerstone of database design and data normalization is that data organization for storage differs from the information most people want to see. For example, a manager of a sporting goods supply company might want to see, for one sale, who the customer was, the destination of the order, the billing address, the contact phone number, the placement time of the order, the order's shipping destination, when and how delivery occurred, what articles the order included, and which of the company's employees was the key person in the sale. The organization of data in the company's database differs from the particular information the manager wants. [Screen 1](http://www.sqlmag.com/Files/09/4887/Screen_01.gif) (page 66) shows a diagram of the autogenerated Order Entry database from Microsoft Access 97. I'll use this database to illustrate the concepts in this article.

**Business Rules**

You need to conduct research at your company before you can begin to normalize a database. You need to perform a requirements analysis, which will identify policies and procedures and will list the business rules for them. You must have consensus on what the rules mean. By consensus, I mean that everyone who uses the database must agree on the definition and the use of these data items. Without consensus, if you ask three people in the company to define what *customer* means, you might get three different answers. To one person, a customer is the company that buys products and services. To a second person, the customer is the contact person for the company who buys product and services. To a third person, the customer is someone who might be interested in buying products and services. Some terms are standard, but under no circumstances can you assume the definition or meaning of a term. Confirm meanings of terms, and confirm how your company uses these terms.

You can use schemes, or methodologies, to guide you through the requirements-analysis phase of a database design. Think of schemes as playbooks or recipes for database design. If you are using a methodology and like it, continue to use it, no matter how small or insignificant the database design project. If you don't have a favorite methodology, you might want to explore techniques such as Bachman, IDEF1X (favored by the Department of Defense), or the new object-oriented Unified Modeling Language (UML—the scheme for the Microsoft Repository). If you're a Visual Basic (VB) programmer, you can download a free entry-level Visual Modeler from http://www.microsoft.com/vstudio. Using a scheme helps you analyze and design your database.

**Relationships**After you know what tables you need, the next step in data normalization is to understand relationships and how they pertain to the data you're working with. A database is an organized, integrated collection of data items. The integration is important; data items relate to other data items, and groups of related data items—called entities—relate to other entities. The relationships between entities can be one of three types, one-to-one (1:1), one-to-many (1:M), and many-to-many (M:N).

Binary relationships, those relationships that involve two entities, are the basis for all other relationship sets. [Figure 1](http://www.sqlmag.com/Files/09/4887/Figure_01.gif) displays these relationships through a technique called entity relationship modeling (ERM). Entities are of two types: noun-type entities and verb-type entities. Noun-type entities represent people, places, and things. Verb-type entities represent actions and interactions between the noun-type entities.

ERM is a way to graphically represent the architecture of a database and to model the informational requirements. You can choose among many entity-modeling schemes. Personal preference, flexibility, functionality of the scheme (what it will let you do and how it will limit you) and established corporate standards will help you choose a scheme. Figure 1 uses the crow's foot scheme, or methodology. Screen 1 uses Microsoft's proprietary ERM methodology.

So many different entity modeling methodologies exist that there is not a true standard. However, one commonly accepted rule for good modeling is that entity names are always expressed in the singular. You can identify the type of relationship between singular entity names more easily than you can determine the relationship between plural entity names. A second, less common, rule is that entity names are always capitalized.

The result of your entity modeling efforts is an entity-relationship diagram (ERD). Figure 1 shows three small ERDs, each of which represents a binary relationship between two entities. A rectangle represents an entity. A line with symbols at each end joins two entities. The symbols are cardinality indicators, each of which shows how many of one entity relates to how many of another entity. The symbols differ from one methodology to another.

The one-to-one (1:1) relationship means that each instance of one entity (CUSTOMER) relates to one instance of a second entity (CREDIT\_CHECK). Each CREDIT\_CHECK relates back to one CUSTOMER.

The one-to-many (1:M) relationship means that each instance of one entity (ORDER) relates to one or more instances of a second entity (PAYMENT). Each PAYMENT relates back to one ORDER.

The many-to-many (M:N) relationship means that many instances of one entity (ORDER) relate to many instances of a second entity (PRODUCT). Stated a different way, an ORDER can include many PRODUCTS, and a PRODUCT can be part of many ORDERS. The M:N relationship is an interaction between two noun-type entities, and it results in the creation of a verb-type entity (INCLUDES). This verb-type entity is a *gerund*. A gerund is a verb or action word that takes a noun form. A 1:M relationship exists between each of the noun-type entities and the verb-type entity. A M:N relationship is a pair of 1:M relationships, one in either direction. The verb-type entity (INCLUDES) captures the data that results from the two noun-type entities interacting with each other.

Entities in an ERD eventually become tables in a database. You use relationships between data items and groups of data items to test the level of normalization in a database design.

**Primary and Foreign Keys**

A primary key (pkey) is an attribute (column) in a table that serves a special purpose. The data items that make up this attribute are unique; no two data item values are the same. The pkey value serves to uniquely identify each row in the table. You can select a row by its pkey value for an operation. Although the SQL Server environment doesn't enforce the presence of a pkey, Microsoft strongly advises that each table have an explicitly designated pkey.

Each table has only one pkey, but the pkey can include more than one attribute. You can create a pkey for the table in [Screen 2](http://www.sqlmag.com/Files/09/4887/Screen_02.gif) by combining CustID, OrderID, and ProductName. We call this combination a concatenated pkey.

A foreign key (fkey) is an attribute that forms an implied link between two tables that are in a 1:M relationship. The fkey, which is a column in the table of the many, is usually a pkey in the table of the one. Fkeys represent a type of controlled redundancy.

**First Normal Form**

You now have a basic familiarity with data, relationships, information, and business rules. Now let's understand the first three levels of normalization.

Screen 2 shows a simulated, poorly normalized table, which I'll call AllData. AllData is data from the OrderEntry database that I have reorganized for this example. A fictitious manager of a sporting goods supply company requested one order, customer identity, the order's shipping destination, billing address, contact phone number, placement time of the order, how and when delivery occurred, what articles were in the order, and which of the company's employees was the key person in the sale. Screen 2 shows the data rows in the ORDER table in a poorly normalized database. The level of redundancy is high.

When you normalize, you start from the general and work toward the specific, applying certain tests along the way. Users call this process decomposition. Decomposition eliminates insertion, update, and deletion anomalies; guarantees functional dependencies; removes transitive dependencies, and reduces non-key data redundancy. We'll decompose the sample data in Screen 2 from First Normal Form (1NF) through Second Normal Form (2NF) into Third Normal Form (3NF).

For a table to be in 1NF you need to ensure that the data is *atomic,* having no repeating groups. A concatenated pkey characterizes a 1NF table.

*Atomic data* is a form of minimalism for data items. A data item is *atomic* if only one item is in each cell of a table. For example, in the AllData table in Screen 2, the attribute ShippedTo encompasses a street address, a city, a state, a postal code, and a country abbreviation. To render this data atomic, you separate this single attribute into several—ShipAddr, ShipCity, ShipState, ShipZip, and ShipCountry. You must do the same separation with the BilledTo attribute. Then, the data in the AllData table is atomic.

*Repeating groups* are cells that have more than one occurrence. In a programming language, this concept is an array. For instance, if this database supported repeating groups (which it does not because it is a relational database), you would see a single row for this order with a repeating group for ProductName and QuantityPurchased. The set of these two columns would occur five times for this one order, once for each product purchased, thereby minimizing the redundancy in the new version of table AllData. This minimalism might work for some nonrelational database architectures and file-processing schemes, but the relational model precludes having repeating groups.

After you designate a pkey, table AllData will be in 1NF. In the description of pkeys, I suggested that a pkey for the AllData table is CustID + OrderID + ProductName. Let's designate that combination as the concatenated pkey.

**Toward 2NF**

2NF is a condition of *full functional dependency* on the whole pkey; the pkey must determine each non-pkey attribute. 1NF requires that a table have a pkey, and we have designated the combination of CustID + OrderID + ProductName for that role. To test for functional dependency, let's see whether the pkey determines each non-pkey attribute.

For each non-key attribute, you proceed as follows. What determines the CompanyName? One of our business rules says that each company has a Customer ID (CustID), and the CustID represents the company and each of its related attributes (CompanyName, CustomerContact, ContactPhone). However, in table AllData, does CustID + OrderID + ProductName determine CompanyName? Does CompanyName depend on what it bought and when? No. Therefore, CompanyName is not fully functionally dependent on the whole pkey.

As you test each non-key attribute against the known business rules, you can see that CustID defines some non-key attributes, OrderID defines other non-pkey attributes, and ProductName defines still other non-pkey attributes. 2NF says that all non-pkey attributes must be fully functionally dependent on the whole pkey. You must modify table AllData to make it 2NF.

If you created three tables, each of which had as its pkey a single attribute of the AllData concatenated pkey, you would have at least part of AllData in 2NF. The solution would look like [Screen 3](http://www.sqlmag.com/Files/09/4887/Screen_03.gif).

The new Customer table has greatly reduced the redundant data present in table AllData. The new Order table still has a high level of redundancy, which we can correct by further decomposition. We have completely normalized the new Product table.

**Determining 3NF**

You achieve 3NF when you have resolved all *transitive dependencies*. Once again, you'll have to test the attributes in each table, but this time you test to see whether, within a table, any non-key attribute determines the value of another non-key attribute. Such a determination defines transitive dependency. A *transitive dependency* causes additional redundancy, which Screen 3 illustrates in the Order table.

Let's start with the Order table to analyze transitive dependencies. One business rule states that each order will have a unique order identifier. An order occurs when a customer purchases one or many products on a given day, at a given time. Therefore, attribute OrderDate is fully functionally dependent on OrderID. But what determines ShippingDate or ShippingMethod? Does OrderID determine the product and the shipping destination? The business rules will have to answer all these questions. For instance, OrderDate might affect ShippingDate. Having the ordered product (ProductName) in stock might also affect ShippingDate. A combination of OrderID and ProductName affect QuantityPurchased. OrderID and CustID affect the shipping address attributes (ShipAddr, ShipCity, ShipState, ShipZip, and ShipCountry).

The Customer table includes some transitive dependencies. The table recognizes a business rule that determines whether the customer is the company (CompanyName attribute). But, does CustID determine the CustomerContact? What if this company has more than one CustomerContact on file? If so, do you need to repeat all the billing address data for the second and third contact? Your company can institute a rule that allows only one contact person per customer, but from a salesperson's perspective, this rule would be restrictive. The salesperson and the retailer want to sell product and services. Why would they want a rule that would hamper this goal?

[Screen 4](http://www.sqlmag.com/Files/09/4887/Screen_04.gif), page 70, is a 3NF version of AllData, because each of the tables in Screen 4 meets the criteria for 3NF:

* Each table is a flat file, or spreadsheet format, with all-atomic data items, no repeating groups, and a designated pkey.
* Each table has all non-pkey attributes fully functionally dependent on the whole pkey.
* All transitive dependencies are removed from each table.

You still have to cross-reference the data from one table to the data in another table. Using cross-referencing, adding the second order to the Order table will let you know what that order included (OrderDetail table).

**Normalizing the Database**

Now that you have decomposed the AllData table into seven smaller tables, you need to cross-reference the seven tables. You have reduced the level of data redundancy and can now fit more rows of any one table on a single block for physical reads and writes. However, what good is this organization if you have no way of relating one table to another?

In the process of reorganizing the data into the new set of tables, you reviewed the business rules. The business rules define data interrelationships:

* A CUSTOMER has many associated CUSTOMER\_CONTACTs, but a CUSTOMER\_CONTACT works for only one CUSTOMER at a time (1:M, CUSTOMER:CUSTOMER\_CONTACT).
* A CUSTOMER can have as many ADDRESSes on file as necessary; each ADDRESS relates to one and only one CUSTOMER (1:M, CUSTOMER: ADDRESS).
* A CUSTOMER can place many ORDERs; each ORDER points back to one and only one CUSTOMER (1:M, CUSTOMER:ORDER).
* A SALESPERSON is responsible for many ORDERs; each ORDER is credited to one SALESPERSON (1:M, SALESPERSON:ORDER).
* Each ORDER can contain one or many ORDER\_DETAILs (items ordered); each ORDER\_ DETAIL relates back to one ORDER (1:M, ORDER: ORDER\_DETAIL).
* A PRODUCT can be a participant in many ORDER\_DETAILs; each ORDER\_DETAIL points back to one and only one PRODUCT (1:M, PRODUCT:ORDER\_DETAIL).

For each 1:M relationship, you take the pkey of the one and embed it as an fkey in the table of the many. [Screen 5](http://www.sqlmag.com/Files/09/4887/Screen_05.gif) shows the result. In some of these tables, I've concatenated the fkey to the existing pkey to enhance the design flexibility and functionality.

You can analyze each table independently of all others in the database, and then deduce a normal form for that table. However, the success of the database design and normalization hinges on what kind of relationship each table has to each other table, and on the correct expression of each relationship.

If you ensure that each table is in 3NF, you avoid problems that can arise when users update data. However, look at all data attributes across the database, and evaluate the normal form of the entire database. You must make sure that you've stored each non-pkey attribute only once to remove all redundant data and to remove the possibility of unsynchronized data, which can damage data recovery.

**Accuracy and Performance**

A poorly normalized database and poorly normalized tables can cause problems ranging from excessive disk I/O and subsequent poor system performance to inaccurate data. An improperly normalized condition can result in extensive data redundancy, which puts a burden on all programs that modify the data.

From a business perspective, the expense of bad normalization is poorly operating systems and inaccurate, incorrect, or missing data. Applying normalization techniques to OLTP database design helps create efficient systems that produce accurate data and reliable information.

Screen 1  
  
Screen 2  
  
Screen 3  
  
Screen 4  
  
Screen 5  
  
Figure 1  


**Normalize your tables to the third normal form.**  
A table is in third normal form (3NF) if it is in second normal form (2NF) and if it does not contain transitive dependencies. In most cases, you should normalize your tables to the third normal form. The normalization is used to reduce the total amount of redundant data in the database. The less data there is, the less work SQL Server has to perform, speeding its performance.

**If a database is in 3NF, how many minimum number of tables should it have?**

Atleast 2

**What is referential integrity & how is it achieved?**

Referential integrity is a system of rules that ensure relationships between rows in related tables are valid and that you do not accidentally delete or change related data.

**What is a Primary Key? Can it have >1 field?**

There are two kinds of keys.  A primary key is a set of columns from a table that are guaranteed to have unique values for each row of that table.  A primary key is also called a primary key constraint, because it effectively constrains the values you can add to the table: it prevents you from adding a row to the table whose primary key columns are all equal to the corresponding values of some other row in that table.

A foreign key is a correspondence between a set of columns in one table and the set of primary key columns in some other table.

Define a primary key to enforce uniqueness for values entered in specified columns that do not allow nulls.

If a primary key consists of more than one column, duplicate values are allowed in one column, but each combination of values from all the columns in the primary key must be unique.

**What are constraints?**

Constraints are business logic that your database server enforces for you.

The Database Designer accepts five types of constraints:

**Check Constraints**: A check constraint specifies the data values or formats that are acceptable in one or more columns in a table. For example, you can require the zip column of the authors table to allow only five-digit numeric entries

**Default Constraints** : A default constraint enables you to define the value that will be supplied for a column whenever a user fails to enter a value. For example, in a table with a column called payterms, you can instruct your database server to enter "???" or "fill in later" if the user leaves it blank.

**Unique Constraints**: A unique constraint ensures no duplicate values are entered into specified columns that are not a table's primary key. For example, in the employee table in which the emp\_id column is the primary key, you can define a unique constraint that requires entries in the Social Security number (ssn) column to be unique within the table.

**Primary Key Constraints:** A primary key constraint ensures no duplicate values are entered in particular columns and that NULL values are not entered in those columns. You can use primary key constraints to enforce uniqueness as well as referential integrity.

For example, the au\_id column uniquely identifies each author stored in the authors table.

**Foreign Key Constraints** : A foreign key constraint works in conjunction with primary key or unique constraints to enforce referential integrity among specified tables. For example, you can place a foreign key constraint on the title\_id column in the publishers

table to ensure that a value entered in that column matches an existing value in the title\_id column of the titles table

**Why should you create index? Types on indexes?**

An index is similar to the index found the books. That facilitates easy search mechanism.

An index in SQL Server assists the database engine with locating records, just like an index in a book helps you locate information quickly.

An index is structured by the SQL Server Index Manager as a Balanced tree (or B-tree).

Columns consisting of the **ntext**, **text**, or **image** data types cannot be specified as columns for an index.

Example CREATE INDEX au\_id\_ind

ON authors (au\_id)

Permissions for the views and indexes:

Default to the **sysadmin** fixed server role and the **db\_ddladmin** and **db\_owner** fixed database roles and the table owner, and are not transferable.

Only the table or view owner can create indexes on that table. The owner of a table or view can create an index at any time, whether or not there is data in the table. Indexes can be created on tables or views in another database by specifying a qualified database name.

**Index in detail:**

Syntax

CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX *index\_name*   
    ON { *table* | *view* } **(** *column* [ ASC | DESC ] [ **,**...*n* ] **)**[ WITH < index\_option > [ **,**...*n*] ]   
[ ON *filegroup* ]

< index\_option > :: =

    { PAD\_INDEX |     FILLFACTOR **=** *fillfactor* |     IGNORE\_DUP\_KEY | DROP\_EXISTING |  STATISTICS\_NORECOMPUTE |  SORT\_IN\_TEMPDB    
}

## ***UNIQUE:***

Microsoft® SQL Server™ checks for duplicate values when the index is created (if data already exists) and checks each time data is added with an INSERT or UPDATE statement. If duplicate key values exist, the CREATE INDEX statement is canceled and an error message giving the first duplicate is returned. Multiple NULL values are considered duplicates when UNIQUE index is created.

Examples :

CREATE Unique INDEX au\_id\_ind

ON authors (au\_id)

## **Clustered and NonClustered**

| **Clustered** | **Non-clustered** |
| --- | --- |
| The physical order of the rows in the table is the same as the logical indexed) order of the key values. | With a nonclustered index, the physical order of the rows is independent of their indexed order. The leaf level of a nonclustered index contains index rows. Each index row contains the nonclustered key value and one or more row locators that point to the row that contains the value.  **Note**: If the table does not have a clustered index, the row locator is the row's disk address. If the table does have a clustered index, the row locator is the clustered index key for the row. |
| A view with the clustered index is called the indexed view | For indexed views, nonclustered indexes can be created only on a view with a clustered index already defined |
| Create a clustered index before creating the non-clustered index, as it rebuilds the entire non-clustered index. |  |
| Must specify clustered to create an clustered index | If CLUSTERED is not specified, a nonclustered index is created. |
| A table can contain only one clustered index. | Each table can have as many as 249 nonclustered indexes. |

Examples

CREATE UNIQUE CLUSTERED INDEX employeeID\_ind

ON emp\_pay (employeeID)

Composite Index

CREATE INDEX emp\_order\_ind

ON order\_emp (orderID, employeeID)

Nonclustered using fillfactor.

CREATE NONCLUSTERED INDEX zip\_ind

ON authors (zip)

WITH FILLFACTOR = 100

### **PAD\_INDEX :**

Specifies the space to leave open on each page (node) in the intermediate levels of the index. The PAD\_INDEX option is useful only when FILLFACTOR is specified, because PAD\_INDEX uses the percentage specified by FILLFACTOR. The number of rows on an intermediate index page is never less than two, regardless of how low the value of FILLFACTOR.

CREATE INDEX au\_id\_ind

ON authors (au\_id)

WITH PAD\_INDEX, FILLFACTOR = 10

### **FILLFACTOR:**

Specifies a percentage that indicates how full SQL Server should make the leaf level of each index page during index creation. When FILLFACTOR is specified, SQL Server rounds up the number of rows to be placed on each page. For example, issuing CREATE CLUSTERED INDEX ... FILLFACTOR = 33 creates a clustered index with a FILLFACTOR of 33 percent. Assume that SQL Server calculates that 5.2 rows is 33 percent of the space on a page. SQL Server rounds so that six rows are placed on each page. User-specified FILLFACTOR values can be from 1 through 100. If no value is specified, the default is 0. When FILLFACTOR is set to 0, only the leaf pages are filled.

Use a FILLFACTOR of 100 only if no INSERT or UPDATE statements will occur, such as with a read-only table

### **IGNORE\_DUP\_KEY :**

Controls what happens when an attempt is made to insert a duplicate key value into a column that is part of a unique clustered index.

If IGNORE\_DUP\_KEY was not specified for the index, SQL Server issues an error message and rolls back the entire INSERT statement.

| **Index type** | **Options** |
| --- | --- |
| Clustered | Not allowed |
| Unique clustered | IGNORE\_DUP\_KEY allowed |
| Nonclustered | Not allowed |
| Unique nonclustered | IGNORE\_DUP\_KEY allowed |

CREATE UNIQUE CLUSTERED INDEX employeeID\_ind

ON emp\_pay(employeeID)

WITH IGNORE\_DUP\_KEY

### **DROP\_EXISTING :**

Specifies that the named, preexisting clustered or nonclustered index should be dropped and rebuilt. The index name specified must be the same as a currently existing index.

SQL SERVER 2000 supports Indexed views and creation on indexed view with “**schema binding**”.

**Schema binding** prohibits any table or column modification that would invalidate the view.  Any indexed view you create with the View Designer automatically gets schema binding, because SQL Server requires that indexed views have schema binding.

**Schema binding** does not mean you cannot modify the view; it means you cannot modify the underlying tables or views in ways that would change the view's result set

Restrictions on the Indexed Views:

The creator of the index must own the tables. All tables, the view, and the index, must be created in the same database.

The SELECT statement defining an indexed view must not have the TOP, DISTINCT, COMPUTE, HAVING, and UNION keywords. It cannot have a subquery.

The SELECT list may not include asterisks (\*), '*table*.\*' wildcard lists, DISTINCT, COUNT(\*), COUNT(<*expression*>), computed columns from the base tables, and scalar aggregates.

Nonaggregate SELECT lists cannot have expressions. Aggregate SELECT list (queries that contain GROUP BY) may include SUM and COUNT\_BIG(<expression>); it must contain COUNT\_BIG(\*). Other aggregate functions (MIN, MAX, STDEV,...) are not allowed.

Complex aggregation using AVG cannot participate in the SELECT list of the indexed view. However, if a query uses such aggregation, the optimizer is capable of using this indexed view to substitute AVG with a combination of simple aggregates SUM and COUNT\_BIG.

What is referential integrity & how is it achieved?

Ensure each value entered in a foreign key column matches an existing value in the related primary key column, it is done using foreign key constraint.

**What is a trigger and stored procedure?**

A stored procedure is a group of Transact-SQL statements that is compiled one time, and then can be executed many times. This increases performance when the stored procedure is executed because the Transact-SQL statements do not have to be recompiled.

A trigger is a special type of stored procedure that is not called directly by a user. When the trigger is created, it is defined to execute when a specific type of data modification is made against a specific table or column.

A CREATE PROCEDURE or CREATE TRIGGER statement cannot span batches. This means that a stored procedure or trigger is always created in a single batch and compiled into an execution plan.

\* Types of triggers?

A trigger is a database object that is *attached* to a table. In many aspects it is similar to a stored procedure. As a matter of fact, triggers are often referred to as a "special kind of stored procedure." The main difference between a trigger and a stored procedure is that the former is attached to a table and is only *fired* when an INSERT, UPDATE or DELETE occurs. You specify the modification action(s) that fire the trigger when it is created.

I only use them when I need to perform a certain action as a result of an INSERT, UPDATE or DELETE and ad hoc SQL (aka SQL Passthrough) is used. For example, let's say you want to send an email to the Sales Manager when an order is entered whose priority is high.

**Types of triggers?**

A trigger is a special kind of stored procedure that SQL Server offers. It might just get the job done quickly and efficiently—while you're on your canoe trip. A trigger is a predefined response to a specific data modification. In SQL Server terms, it refers to an UPDATE, INSERT, or DELETE action. Table 1 defines the two types of triggers.

| **Table 1: Types of SQL Server Triggers**  **Trigger** | **Description** | **Executed Against** | **Number Allowed** |
| --- | --- | --- | --- |
| FOR | Executes after the triggering statement is completed | Tables only | Multiple FOR (also known as AFTER) triggers are allowed, and you can control which trigger fires first and last using the sp\_settriggerorder. All other triggers fire in an undefined order, which you can't control. |
| INSTEAD OF | Executes in place of the triggering action | Tables and views | Only one per table or view |

The main benefit triggers offer is that they react automatically to a specific type of modification made to a specific table. Keep the following rules in mind when you're adding a trigger:

* Only the table owner has permission to create triggers, and permission can't be transferred.
* A trigger is considered a database object, so use object rules when naming and referencing a trigger.
* Triggers are restricted to the current database, although you can reference an object outside the database.
* A trigger can reference a temporary table but can't modify one.
* A trigger can't reference a system table.

**The Advantages of INSTEAD OF Triggers**

You can write a trigger for a view, but if the view is updateable it isn't necessary. Triggers on the underlying table fire automatically. (Of course, you may have your own reasons why you want triggers on such views.) Of all the advantages INSTEAD OF triggers offer, the main one is that they allow views that would normally not be updateable to support updates. A view that involves multiple tables must use an INSTEAD OF trigger to support inserts, updates, and deletes that reference data in more than one table. For example, you can write an INSTEAD OF trigger that inserts rows in multiple tables from a single view.

Another important advantage to INSTEAD OF triggers is that they allow you to write logic that accepts parts of a batch while rejecting other parts. Finally, INSTEAD OF triggers allow you to take some alternative action in the event of some particular condition that the application defines as an error.

**What are inserted and deleted tables?**

Triggers make use of two special tables called **inserted** and **deleted**. The inserted table contains the data referenced in an INSERT before it is actually committed to the database. The deleted table contains the data in the underlying table referenced in a DELETE before it is actually removed from the database. When an UPDATE is issued both tables are used. More specifically, the *new* data referenced in the UPDATE statement is contained in inserted and the data that is being updated is contained in deleted.

**What is a view? where will you use views?**

A view is simply a SELECT query saved in the database.  Thus, most operations you can perform on queries you can also perform on views.  However, there are some operations that apply only to one or the other.

| **View** | **Queries** |
| --- | --- |
| Views are stored as part of a database design | It is a specific request for retrival, Creation, Modification or deletion of data from the database. hey are part of DB design |
| They use SQL statements | They use SQL statements |
| Allows the administrator to disallow the access to the base table and permit access only to the views | No such option. |
| One cannot sort the view expect(TOP Clause) | One can sort the query result. |
| Can establish the query plan | No such option |
| Cannot create parameters | Can create parameters for a query |
| View can be encrypted | Query cannot be encrypted |

In addition, a view cannot include any **text**, **ntext**, or **image** columns, even if they are not referenced in the CREATE INDEX statement

Example

CREATE VIEW V1

WITH SCHEMABINDING

AS

SELECT SUM(UnitPrice\*Quantity\*(1.00-Discount)) AS Revenue, OrderDate, ProductID, COUNT\_BIG(\*) AS COUNT

FROM dbo.[Order Details] od, dbo.Orders o

WHERE od.OrderID=o.OrderID

GROUP BY OrderDate, ProductID

**What is fillfactor?**

**See in Indexes**

**What is SQL Trace?**

Microsoft supplied the SQL Trace utility with SQL Server 6.x and replaced it with SQL Profiler in SQL Server 7.0. You can use SQL Trace, an Open Data Services (ODS) sniffer program, to monitor and record SQL Server 6.x database activity and troubleshoot 6.x systems. For example, you can capture five activity types (connections, SQL statements, remote procedure calls, attentions, and disconnections) within SQL Server 6.x. You can save generated traces as a trace file or an SQL script and apply five filters (login name, application, hostname, remote procedure call filter, and SQL statement filter).

Because SQL Trace is external to the SQL Server architecture, it has limited reporting capability. SQL Trace is an intrusive monitoring program that can report on only high-level events. For example, SQL Trace can report that a user is calling a certain stored procedure. If stored procedures contain many conditional statements, the same stored procedure (depending on the system state at that time and parameters passed to it) can perform wildly different actions. You cannot use SQL Trace to determine what the stored procedures will do. In addition, if you run SQL Trace on a severely stressed server, you might bring the server down.

SQL Server stores trace definitions in the Registry. If the user has appropriate permissions, SQL Server stores those permissions in the Registry of the SQL Server being traced. If the user doesn't have appropriate permissions, SQL Server stores user permissions in the Registry of the machine performing the trace. The location is not negotiable. If many developers or database administrators (DBAs) define traces, one server (typically a development server) could hold many trace definitions.

Although you can still use the old stored procedure xp\_trace (a stored procedure for controlling trace activity on a server), SQL Server *Books Online* (*BOL*) states that this stored procedure is "for backward compatibility only and may not be supported in future versions." In a future article, I'll tell you about a richer mechanism you can use for profiling that involves about 65 stored procedures, but this method of controlling profiles is beyond the scope of this article.

SQL Trace is the most powerful tool in my tuning bag of tricks. It lets me see the big picture of application-level interaction with SQL Server without knowing anything about the application or looking at the source code.

**What is a join and what are different types of joins?**

The SQL join command is used to join two or more tables. At times you may need to join two or more tables to retrieve the information you need from your SQL database

Syntax:

Select *\*/tablename.fieldname<mask> ...*

from *tablename <alias>, tablename <alias>, tablename <alias>*

where *tablename.fieldname = tablename.fieldname*

*and*

*tablename.fieldname = tablename.fieldname*

*and*

*tablename.fieldname = tablename.fieldname*

order by *fieldname <desc>...*

**Explanations:**

When you join two or more tables, a good idea is to precede the field names with the table names. This is not mandatory unless the same field name is found in more than one table.

If you precede the field name with a table name, place a period between the two names. For example, tablename.fieldname.

You must specify which fields are being joined.

If you do not specify which fields are being joined, the result is what is commonly referred to as a "Cartesian join" in which all rows in the first table are joined with all rows in the second table.

You can give each table name an *alias*, or alternative table name. When you assign an alias, you can then refer to the table by using its alias.

**Inner joins**

A general characteristic of inner joins is to find matching values and find the rows that have these matching values.

SELECT customer\_info.firstname, customer\_info.lastname, purchases.item

FROM customer\_info INNER JOIN purchases

ON customer\_info.customer\_number = purchases.customer\_number;

**Outer Join**

1. LEFT Outer Join
2. Right outer Join
3. FULL Outer Join

Outer joins retain unmatched rows from one or both the tables, depending on the keyword – LEFT, RIGHT or FULL – used.

**Difference between union and join**

Combines the results of two or more queries into a single result set consisting of all the rows belonging to all queries in the union

Two basic rules for combining the result sets of two queries with UNION are:

The number and the order of the columns must be identical in all queries.

The data types must be compatible.

**Difference between Group by and Having clause**

You can use a WHERE clause in a query containing a GROUP BY clause. Rows not meeting the conditions in the WHERE clause are eliminated before any grouping is done

Both the group by and having clause can be applied only on the aggregate fields

Specifies a search condition for a group or an aggregate. HAVING is usually used with the GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause.

**Implementation of Stored procedures**

***Creating, compiling, and using stored procedures***

A stored procedure is compiled code that you can call from within T-SQL statements or from client applications. SQL Server runs the code in the procedure and then returns the results to the calling application. Using stored procedures is efficient for several reasons. First, SQL Server has already parsed, optimized, and compiled stored procedures, so they run quickly without needing to repeat these steps each time. Also, stored procedures run on the SQL Server, using the power of the server and reducing the load on the client (which might be a much less powerful computer). Finally, using stored procedures reduces network traffic. Instead of the SQL Server sending all the data to the client and having the client run a query, the client sends a request to the server to run the procedure. The server returns only the result data set, which is usually a lot smaller than the full data set.

Once a stored procedure has run, it remains in memory, so the next user can run it without incurring the overhead of loading it into memory. SQL Server 6.5 and earlier releases require multiple copies of the procedure in memory if more than one user will be running the procedure at the same time. SQL Server 7.0 improved on this situation by letting multiple users, each with a different execution context (including parameters and variables), share one copy of the procedure in memory.

You can use stored procedures to enforce a level of consistency in your client applications. If all the client applications use the same stored procedures to update the database, the code base is smaller and easier to maintain, and you run less risk of deadlocks because everyone is updating tables in the same order.

Stored procedures enhance security, too, because you can give the users only EXECUTE permission on the stored procedures, while restricting access to the tables and not allowing the users any direct update privileges. When other users run a stored procedure, they run it as if they had the permissions of the user who created the query.

You can do almost anything in a stored procedure, including CREATE DEFAULT, CREATE PROCEDURE, CREATE RULE, CREATE TRIGGER, and CREATE VIEW. You can create tables—both permanent and temporary—but any temporary objects you create within a stored procedure vanish when the stored procedure is complete. And you can nest stored procedures up to 32 levels deep, so that one procedure can call another, which calls a third, and so on.

**Definitions**   
In *SQL Server Books Online (BOL)* and elsewhere, you might see references to several types of stored procedures, but there are only two primary types. A stored procedure is T-SQL code that SQL Server has parsed, optimized, and compiled. An extended stored procedure is a DLL (typically written in C or C++) that leverages the integration of SQL Server and Windows 2000/NT to make OS-level calls and perform functions that are beyond T-SQL's scope. Both types can accept input parameters and return output values, error codes, and status messages. Within the two primary types of stored procedure, you'll find several others.

* *Local stored procedure:* a standard stored procedure, executed on the local computer. You define stored procedures within a database and they become local objects within that database. You can call local stored procedures from another database by giving the full name, including the database name, owner name, and procedure name. You can also call them from other applications, including common client applications such as Microsoft Access and Visual Basic (VB) applications. Typically, you use the Query Analyzer to create local stored procedures. (You could use the Enterprise Manager, but it’s not very helpful—you still have to know how to write the code, and it’s easier to test from the Query Analyzer.)
* *Temporary stored procedure:* just like a local stored procedure, except that the name starts with a # symbol for a private temporary stored procedure and a ## for a global temporary stored procedure. A private temporary procedure is good only within the session it was created in; a global procedure can be seen and run from other sessions. Note the difference in terminology between temporary tables and temporary procedures. Temporary tables can be local or global, but a "local" stored procedure is any in the current database, so we use the word private to denote stored procedures that are limited to the current session.
* *Remote stored procedures:* a standard stored procedure, run on a remote computer. As Einstein might have pointed out, to an observer on the remote computer, this is actually a local stored procedure that has been activated from another computer. The stored procedure exists on the remote computer; you just send across the network the set of instructions to run it. In SQL Server 7.0, distributed queries officially replaced remote stored procedures, but the ideas are the same—one server asks another to run a stored procedure and return the results to the calling server.
* *System stored procedure:* a stored procedure that you create in the Master database and that is available to any other database. System stored procedure names start with sp\_; the term "system stored procedures" is usually understood to mean the stored procedures Microsoft supplies with SQL Server. You can create your own system stored procedure simply by creating a stored procedure in Master and prefixing it with sp\_. However, Microsoft recommends that you don't use sp\_ when you name a stored procedure. If you do, regardless of where you created it, SQL Server looks first in Master, then in the database you specified in your calling string, and finally (if you didn't specify a database) in the local database with the Database Owner (DBO) as the owner. Yes, SQL Server checks Master first, even if you supply the database name. Not only is this inefficient if the procedure is local, but a procedure with the same name might exist in Master, leading to confusion.

An *extended stored procedure* is a DLL that is coded in a programming language other than T-SQL. An extended stored procedure's name begins with xp\_. Microsoft supplies with SQL Server some extended stored procedures, such as xp\_readmail, which handles reading email messages on behalf of the sp\_processmail stored procedure. One especially versatile extended stored procedure is xp\_cmdshell, which executes a command string as an OS command shell. (Essentially, you can run from xp\_cmdshell any command that you can run from the Win2K/NT command line.)

**Creating a Stored Procedure**

To create a stored procedure, you must be the DBO or systems administrator (sa), or be a member of the db\_ddladmin role. Users with the permission to grant permissions can grant other users permission to create procedures using T-SQL commands. You can't grant this permission to specific users from the Enterprise Manager interface; you need to place them in the db\_ddladmin role instead. When you create the procedure, you must have permission to perform all the steps listed in the query. These steps might include accessing other tables or performing inserts, updates, and deletes.

When you create a procedure, you can create it only in the current database. So you need to specify only the procedure name, not the database name or even the owner name. The exception to this rule is temporary stored procedures, which are created in tempdb. Procedure names must be unique within a database, but you can use the same procedure name in different databases. If you want a procedure to be available in every database but don't want to make it a system stored procedure in Master, add it to the Model database. It will appear in every new database you create thereafter.

You can't combine the CREATE PROCEDURE statement with any other statement in the same batch. That restriction sounds minor—you might think that you can just put in a GO statement and move on to the next part of the script. However, you can't put a GO statement in the middle of a procedure: As soon as it reaches a GO, the parser treats that as the end of the procedure and compiles everything up to that point.

The syntax for creating a stored procedure is simple:

CREATE PROC procedure\_name

[@parameter datatype] [= default] [OUPUT],

[@parameter datatype] [= default] [OUPUT],

….

WITH {RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION}

AS

T-SQL statement……….

GO

You specify input parameters as @parameter\_name, and you must define a data type for each parameter. Optionally, you can specify a default value for any parameters; the default must be a constant or NULL. Procedures can contain up to 1024 parameters; these are local to the procedure so you can use the same parameter names for other procedures without risk of interference. If you want a parameter to return a value, you must specify the parameter as an OUTPUT parameter when you create the procedure. You don't necessarily always have to ask for the parameter as a returned parameter, but unless you have defined it as a potential output parameter, you can't ask for a parameter to return a value when you execute the procedure. Parameters can act as both input and output parameters, so a simple procedure could have only one input parameter, with the value being modified and passed back as an output parameter.

The WITH ENCRYPTION option prevents others from reverse engineering your procedures by looking at the code in the syscomments table. It also prevents you from looking at the code because there is no way to ask for the code to be unencrypted. SQL Server can unencrypt it, because it can recompile the code when necessary, including when you upgrade to a new release of SQL Server. But you can't supply a key or password to unencrypt the code, so keep a copy of your source code somewhere secure.

If you need to change the definition of a stored procedure, you can do so by rerunning the CREATE PROCEDURE statement, changing CREATE PROCEDURE to ALTER PROCEDURE. SQL Server 7.0 introduced the ability to ALTER database objects, so you can change objects without having to drop and recreate them as you had to do in previous releases. Dropping the object also removes all permissions on that object, so when you recreate a dropped object, you also have to rebuild the permissions. (Scripting the permissions on an object before you drop it is always a good idea.)

**The Creation Process**

When you create a stored procedure, SQL Server places an entry in the sysobjects table for the database, listing the new object. SQL Server parses the T-SQL code and checks it for syntax errors, then stores the procedure code in the syscomments table (in its encrypted form, if you chose that option). A process called Delayed Name Resolution lets you create stored procedures and refer to objects, such as tables, that don't yet exist. The parser doesn't give you an error because it assumes that the object referenced will exist by the time you execute the query.

**Running a Procedure for the First Time**

When you execute stored procedure for the first time, the query optimizer builds an execution plan for the procedure, then compiles the plan and uses it to run the procedure. SQL Server 2000 and 7.0 don't store this execution plan permanently. The plan remains in memory unless your recompile options specify otherwise. The procedure cache is an area of memory where SQL Server keeps stored procedures and cached queries. SQL Server 6.5 and earlier releases required you to configure how much of the available memory to allocate to data cache and how much to procedure cache. SQL Server 2000 and 7.0 allocate the memory dynamically to each cache as SQL Server requires.

**Running the Procedure Again**

Once the procedure is in memory, other client applications can use it without any action on their part—they simply run the procedure, and if it is found in memory, they use it. If the procedure isn't in memory, SQL Server must reoptimize and compile it. Most SQL Servers with adequate memory keep frequently run procedures in the cache. But if memory use becomes a concern, SQL Server can drop some procedures from memory. It uses a sophisticated algorithm to decide which to drop and which to keep, giving preference to the most frequently used procedures, but taking into account the effort necessary to recompile a procedure if it were flushed from the cache. In other words, the algorithm calculates whether to drop one large procedure that hasn't been used for a long time, or several small procedures that are used occasionally but not frequently.

Users can share a copy of the procedure in memory as long as their environment—the server, database, and connection settings—is identical. Different connection settings require different copies of the procedure in memory, so try to standardize connection settings (SET options such as ANSI PADDING, for example). And here's a heads-up for database designers: Users can't share a copy of a procedure if a referenced object requires name resolution. Such a situation can occur when two objects have the same name but different owners: for example, the sales and engineering departments might each have added a table called "budget," with different owners. Or a programmer might have made a copy of the employees table, and own the copy but not the original. Having rules in place about object naming and ownership—ideally all objects have unique names and are owned by the DBO—will prevent this problem.

**Recompile Options**

One benefit of stored procedures is that they remain in memory and can be reused until the server is restarted, so they could be in memory for months. In some situations, you'll want to recompile a procedure. For example, you might have a procedure that produces radically differing results every time you run it, such as a procedure to return a list of customer names and addresses for a range of ZIP codes. If you run this procedure on a few ZIP codes in New York, you might get back several thousand names. But if you run it on a larger range of ZIP codes in Wyoming, you might get only two or three names. For such a stored procedure, consider putting the keywords WITH RECOMPILE in the CREATE PROC code. SQL Server will then discard the compiled execution plan every time the procedure is run, and recompile it again for the next user.

You might usually run a stored procedure with certain parameters—for example, if most of your business is in New York, you expect a large number of names per ZIP code. When you need to send a mailing to Wyoming, you can run the query again, using the EXECUTE procname WITH RECOMPILE option. SQL Server discards the plan in the procedure cache and compiles a new one with the atypical parameters, so your procedure runs with the query optimized for this mailing list. As a courtesy to the next user, you should run the query again, using a typical data set and the WITH RECOMPILE option. If you don't, your atypical plan will sit in memory, and the next time someone runs the procedure, SQL Server will use it.

You might also consider recompiling your procedure if you add an index to the table. Recompiling gives the query optimizer a chance to calculate whether the new index might be useful. The command to use here is sp\_recompile. You can specify a procedure name for the recompile, or you can supply a table name or view name. If you specify a table or view, SQL Server will recompile all procedures that reference that table or view the next time they run.

**Automatic Recompile**

*BOL* says that "SQL Server automatically recompiles stored procedures and triggers when it is advantageous to do so." Actually, all SQL Server does is kick the procedure out of the cache, so it has to recompile the procedure the next time it runs. SQL Server recomputes the statistics on an index when there have been a significant number of changes to the data. With new statistics, recompiling (and therefore reoptimizing) the query makes sense. If you drop an index that a procedure used, the procedure will need recompiling. Any time you rebuild indexes to reclaim space in the database, SQL Server recompiles the procedure with the new index statistics. So it's unlikely that any procedure will remain in memory indefinitely. In fact, by not storing optimized query plans, SQL Server 2000 and 7.0 look for opportunities to recompile with the latest statistics.

**Temporary Stored Procedures vs. Cached Queries**

As I mentioned, you can create a temporary stored procedure by starting the procedure name with # or ##, and these procedures are built in tempdb. A private procedure can be used only in the session where it was created; a global procedure can be used by anyone with the right permissions, but it vanishes when the session it was created in is disconnected. Fortunately, SQL Server lets anyone running the procedure at the time complete the procedure. Also, if you create stored procedures in tempdb directly, you don't need to start the name with # or ##. You just run the following command from your database

EXECUTE tempdb.dbo.procname

and the stored procedure remains in tempdb until the server restarts, even if you log off. If you don't want to create temporary stored procedures in tempdb, you can create the procedure in your database, use it, then drop it.

SQL Server 2000 and 7.0 can cache queries, so SQL Server can spot a query being repeated. It can also detect a query being run over and over with just one different parameter—say someone in sales is calling up customer data and the only difference in each iteration is the customerID. SQL Server can take this query, treat customerID as a parameter, and use the same execution plan with different customerID values. That way, SQL Server avoids having to parse, optimize, and compile every time. (This process is sometimes called autoparameterization.) You also have the option of using the sp\_executesql system stored procedure to tell SQL Server that you're going to keep sending it the same query with different values. This option is great if you have a loop in your code and use it to step through a list of updates to a table one at a time.

So if SQL Server can cache queries and discard them when it is done with them, what's the difference between a temporary stored procedure and a cached query? Not a lot, really. Both are parsed, optimized, compiled, used, and dropped. The biggest difference is that the cached query uses space in your database, and the temporary procedure is built in tempdb. That distinction might be not be important unless you're working on a server with many other users who are competing for space in tempdb. Microsoft recommends the sp\_executesql approach, and I agree because of the reduced possibility of contention in tempdb.

**Automatic Stored Procedure Execution**

SQL Server has a little-known feature called automatic stored procedure execution, which lets you specify stored procedures to run when the server starts—it's like an autoexec.bat file for SQL Server. To make a procedure run at startup, run the sp\_procoption procedure with the syntax

sp\_procoption procname startup true

You can use true or on to make the procedure run at startup and false or off to remove it from the list of procedures to run at startup. (In SQL Server 6.5, you run sp\_makestartup and sp\_unmakestartup to add procedures to or remove them from the autostart list. These procedures don't exist in later releases.) Be careful about adding multiple procedures because all the procedures will start at once. If you need the procedures to run in a sequence, build one procedure that calls all the others in the right order, then make that one an autostart procedure.

**Stored Procedure Guidelines**

You might prefer to use a prefix of usp\_ for user stored procedures, but there's no accepted industry-wide naming standard. Microsoft recommends limiting each stored procedure to executing one task. If you need to perform multiple tasks, you can create one procedure that calls all the others in the correct sequence. If you've used the tasks in the sequence elsewhere, modularizing the code makes sense. But there's no reason not to have a sequence of tasks in one procedure. You might encounter problems, though, if you build branching logic into the procedure or if you have a procedure that performs one of several different tasks depending on the input parameters. If you run the procedure the first time with a set of parameters that cause it to activate branch B, SQL Server can't optimize the code for branches A and C. And if you use branching logic, you load into memory code that you might not use very often. So for a straight-through sequence that SQL Server processes from start to finish, you might be better off writing it with more steps in fewer procedures. For branching logic, having each branch as a separate procedure and calling them as needed might make more sense.

**Grouping Stored Procedures**

Grouping stored procedures is one of those features that nobody seems to use, which makes it a likely exam topic. If you have several associated procedures, perhaps for the branching logic I discussed, you can group them by naming them with the same name and individual identification numbers, with a semicolon between the name and the number. The procedure names would look like procname;1, procname;2, and so on. The benefit of using this naming convention, apart from consistency, is that you can use a DROP PROCEDURE procname command to drop all the procedures at once. However, after you group the procedures, you can't drop them individually.

Stored procedures are a powerful part of SQL Server. They offer benefits in both security and performance. Any client application you use will almost certainly run better if you add some stored procedures running on the server.

**SQL performance tuning**

* Fine tuning the queries by avoiding the subquery
* Creating indexes
* Writing stored procedures and triggers
* Creating a unique clustered index on a view improves query performance because the view is stored in the database in the same way a table with a clustered index is stored.

**What Is an Indexed View?**

For many years, Microsoft® SQL Server™ has supported the ability to create virtual tables known as views. Historically, these views served two main purposes:

1. To provide a security mechanism that restricts users to a certain subset of data in one or more base tables.
2. To provide a mechanism that allows developers to customize how users can logically view the data stored in base tables.

With SQL Server 2000, the functionality of SQL Server views has been expanded to provide system performance benefits. It is possible to create a unique clustered index on a view, as well as nonclustered indexes, to improve data access performance on the most complex queries. In SQL Server 2000, a view that has a unique clustered index is referred to as an indexed view.

**Note**Indexed views can be created in any edition of SQL Server 2000. In SQL Server 2000 Enterprise Edition, the indexed view will be automatically considered by the query optimizer. To use an indexed view in all other editions, the NOEXPAND hint must be used.

**---------------------------------------------END---------------------------------------------**

| How do you implement one-to-one, one-to-many and many-to-many relationships while designing tables? |
| --- |

One-to-One relationship can be implemented as a single table and rarely as two tables with primary and foreign key relationships.  
One-to-Many relationships are implemented by splitting the data into two tables with primary key and foreign key relationships.  
Many-to-Many relationships are implemented using a junction table with the keys from both the tables forming the composite primary key of the junction table.  
  
It will be a good idea to read up a database designing fundamentals text book.

| What's the difference between a primary key and a unique key? |
| --- |

Both primary key and unique enforce uniqueness of the column on which they are defined. But by default primary key creates a clustered index on the column, where are unique creates a nonclustered index by default. Another major difference is that, primary key doesn't allow NULLs, but unique key allows one NULL only.

| What are user defined datatypes and when you should go for them? |
| --- |

User defined datatypes let you extend the base SQL Server datatypes by providing a descriptive name, and format to the database. Take for example, in your database, there is a column called Flight\_Num which appears in many tables. In all these tables it should be varchar(8). In this case you could create a user defined datatype called Flight\_num\_type of varchar(8) and use it across all your tables.   
  
See sp\_addtype, sp\_droptype in books online.

| What is bit datatype and what's the information that can be stored inside a bit column? |
| --- |

Bit datatype is used to store boolean information like 1 or 0 (true or false). Untill SQL Server 6.5 bit datatype could hold either a 1 or 0 and there was no support for NULL. But from SQL Server 7.0 onwards, bit datatype can represent a third state, which is NULL.

| Define candidate key, alternate key, composite key. |
| --- |

A candidate key is one that can identify each row of a table uniquely. Generally a candidate key becomes the primary key of the table. If the table has more than one candidate key, one of them will become the primary key, and the rest are called alternate keys.   
  
A key formed by combining at least two or more columns is called composite key.

| What are defaults? Is there a column to which a default can't be bound? |
| --- |

A default is a value that will be used by a column, if no value is supplied to that column while inserting data. IDENTITY columns and timestamp columns can't have defaults bound to them. See CREATE DEFUALT in books online.

| What is a transaction and what are ACID properties? |
| --- |

A transaction is a logical unit of work in which, all the steps must be performed or none. ACID stands for Atomicity, Consistency, Isolation, Durability. These are the properties of a transaction. For more information and explanation of these properties, see SQL Server books online or any RDBMS fundamentals text book.

| Explain different isolation levels |
| --- |

An isolation level determines the degree of isolation of data between concurrent transactions. The default SQL Server isolation level is Read Committed. Here are the other isolation levels (in the ascending order of isolation): Read Uncommitted, Read Committed, Repeatable Read, Serializable. See SQL Server books online for an explanation of the isolation levels. Be sure to read about SET TRANSACTION ISOLATION LEVEL, which lets you customize the isolation level at the connection level.

| CREATE INDEX myIndex ON myTable(myColumn)  What type of Index will get created after executing the above statement? |
| --- |

Non-clustered index. Important thing to note: By default a clustered index gets created on the primary key, unless specified otherwise.

| What's the maximum size of a row? |
| --- |

8060 bytes. Don't be surprised with questions like 'what is the maximum number of columns per table'. Check out SQL Server books online for the page titled: "Maximum Capacity Specifications".

| Explain Active/Active and Active/Passive cluster configurations |
| --- |

Hopefully you have experience setting up cluster servers. But if you don't, at least be familiar with the way clustering works and the two clusterning configurations Active/Active and Active/Passive. SQL Server books online has enough information on this topic and there is a good white paper available on Microsoft site.

| Explain the architecture of SQL Server |
| --- |

This is a very important question and you better be able to answer it if consider yourself a DBA. SQL Server books online is the best place to read about SQL Server architecture. Read up the chapter dedicated to SQL Server Architecture.

| What is lock escalation? |
| --- |

Lock escalation is the process of converting a lot of low level locks (like row locks, page locks) into higher level locks (like table locks). Every lock is a memory structure too many locks would mean, more memory being occupied by locks. To prevent this from happening, SQL Server escalates the many fine-grain locks to fewer coarse-grain locks. Lock escalation threshold was definable in SQL Server 6.5, but from SQL Server 7.0 onwards it's dynamically managed by SQL Server.

| What's the difference between DELETE TABLE and TRUNCATE TABLE commands? |
| --- |

DELETE TABLE is a logged operation, so the deletion of each row gets logged in the transaction log, which makes it slow. TRUNCATE TABLE also deletes all the rows in a table, but it won't log the deletion of each row, instead it logs the deallocation of the data pages of the table, which makes it faster. Of course, TRUNCATE TABLE can be rolled back.

| Explain the storage models of OLAP |
| --- |

Check out MOLAP, ROLAP and HOLAP in SQL Server books online for more infomation.

| What are the new features introduced in SQL Server 2000 (or the latest release of SQL Server at the time of your interview)? What changed between the previous version of SQL Server and the current version? |
| --- |

This question is generally asked to see how current is your knowledge. Generally there is a section in the beginning of the books online titled "What's New", which has all such information. Of course, reading just that is not enough, you should have tried those things to better answer the questions. Also check out the section titled "Backward Compatibility" in books online which talks about the changes that have taken place in the new version.

| What are constraints? Explain different types of constraints. |
| --- |

Constraints enable the RDBMS enforce the integrity of the database automatically, without needing you to create triggers, rule or defaults.   
  
Types of constraints: NOT NULL, CHECK, UNIQUE, PRIMARY KEY, FOREIGN KEY  
  
For an explanation of these constraints see books online for the pages titled: "Constraints" and "CREATE TABLE", "ALTER TABLE"

| Whar is an index? What are the types of indexes? How many clustered indexes can be created on a table? I create a separate index on each column of a table. what are the advantages and disadvantages of this approach? |
| --- |

Indexes in SQL Server are similar to the indexes in books. They help SQL Server retrieve the data quicker.  
  
Indexes are of two types. Clustered indexes and non-clustered indexes. When you craete a clustered index on a table, all the rows in the table are stored in the order of the clustered index key. So, there can be only one clustered index per table. Non-clustered indexes have their own storage separate from the table data storage. Non-clustered indexes are stored as B-tree structures (so do clustered indexes), with the leaf level nodes having the index key and it's row locater. The row located could be the RID or the Clustered index key, depending up on the absence or presence of clustered index on the table.  
  
If you create an index on each column of a table, it improves the query performance, as the query optimizer can choose from all the existing indexes to come up with an efficient execution plan. At the same t ime, data modification operations (such as INSERT, UPDATE, DELETE) will become slow, as every time data changes in the table, all the indexes need to be updated. Another disadvantage is that, indexes need disk space, the more indexes you have, more disk space is used.

[~~Back to top~~](http://vyaskn.tripod.com/iq.htm#top)

| Database administration | [~~(top)~~](http://vyaskn.tripod.com/iq.htm#top) |
| --- | --- |

| What is RAID and what are different types of RAID configurations? |
| --- |

RAID stands for Redundant Array of Inexpensive Disks, used to provide fault tolerance to database servers. There are six RAID levels 0 through 5 offering different levels of performance, fault tolerance. MSDN has some information about RAID levels and for detailed information, check out the [~~RAID advisory board's homepage~~](http://www.raid-advisory.com/)

| What are the steps you will take to improve performance of a poor performing query? |
| --- |

This is a very open ended question and there could be a lot of reasons behind the poor performance of a query. But some general issues that you could talk about would be: No indexes, table scans, missing or out of date statistics, blocking, excess recompilations of stored procedures, procedures and triggers without SET NOCOUNT ON, poorly written query with unnecessarily complicated joins, too much normalization, excess usage of cursors and temporary tables.  
  
Some of the tools/ways that help you troubleshooting performance problems are: SET SHOWPLAN\_ALL ON, SET SHOWPLAN\_TEXT ON, SET STATISTICS IO ON, SQL Server Profiler, Windows NT /2000 Performance monitor, Graphical execution plan in Query Analyzer.  
  
Download the white paper on performance tuning SQL Server from Microsoft web site. Don't forget to check out [~~sql-server-performance.com~~](http://www.sql-server-performance.com/)

| What are the steps you will take, if you are tasked with securing an SQL Server? |
| --- |

Again this is another open ended question. Here are some things you could talk about: Preferring NT authentication, using server, databse and application roles to control access to the data, securing the physical database files using NTFS permissions, using an unguessable SA password, restricting physical access to the SQL Server, renaming the Administrator account on the SQL Server computer, disabling the Guest account, enabling auditing, using multiprotocol encryption, setting up SSL, setting up firewalls, isolating SQL Server from the web server etc.  
  
Read the white paper on SQL Server security from Microsoft website. Also check out [~~My SQL Server security best practices~~](http://vyaskn.tripod.com/sql_server_security_best_practices.htm)

| What is a deadlock and what is a live lock? How will you go about resolving deadlocks? |
| --- |

Deadlock is a situation when two processes, each having a lock on one piece of data, attempt to acquire a lock on the other's piece. Each process  would wait indefinitely for the other to release the lock, unless one of the user processes is terminated. SQL Server detects deadlocks and terminates one user's process.  
  
A livelock is one, where a  request for an exclusive lock is repeatedly denied because a series of overlapping shared locks keeps interfering. SQL Server detects the situation after four denials and refuses further shared locks. A livelock also occurs when read transactions monopolize a table or page, forcing a write transaction to wait indefinitely.  
  
Check out SET DEADLOCK\_PRIORITY and "Minimizing Deadlocks"  in SQL Server books online. Also check out the article Q169960 from Microsoft knowledge base.

| What is blocking and how would you troubleshoot it? |
| --- |

Blocking happens when one connection from an application holds a lock and a second connection requires a conflicting lock type. This forces the second connection to wait, blocked on the first.   
  
Read up the following topics in SQL Server books online: Understanding and avoiding blocking, Coding efficient transactions.

| Explain CREATE DATABASE syntax |
| --- |

Many of us are used to craeting databases from the Enterprise Manager or by just issuing the command: CREATE DATABAE MyDB. But what if you have to create a database with two filegroups, one on drive C and the other on drive D with log on drive E with an initial size of 600 MB and with a growth factor of 15%? That's why being a DBA you should be familiar with the CREATE DATABASE syntax. Check out SQL Server books online for more information.

| How to restart SQL Server in single user mode? How to start SQL Server in minimal configuration mode? |
| --- |

SQL Server can be started from command line, using the SQLSERVR.EXE. This EXE has some very important parameters with which a DBA should be familiar with. -m is used for starting SQL Server in single user mode and -f is used to start the SQL Server in minimal confuguration mode. Check out SQL Server books online for more parameters and their explanations.

| As a part of your job, what are the DBCC commands that you commonly use for database maintenance? |
| --- |

DBCC CHECKDB, DBCC CHECKTABLE, DBCC CHECKCATALOG, DBCC CHECKALLOC, DBCC SHOWCONTIG, DBCC SHRINKDATABASE, DBCC SHRINKFILE etc. But there are a whole load of DBCC commands which are very useful for DBAs. Check out SQL Server books online for more information.

| What are statistics, under what circumstances they go out of date, how do you update them? |
| --- |

Statistics determine the selectivity of the indexes. If an indexed column has unique values then the selectivity of that index is more, as opposed to an index with non-unique values. Query optimizer uses these indexes in determining whether to choose an index or not while executing a query.   
  
Some situations under which you should update statistics:  
1) If there is significant change in the key values in the index  
2) If a large amount of data in an indexed column has been added, changed, or removed (that is, if the distribution of key values has changed), or the table has been truncated using the TRUNCATE TABLE statement and then repopulated  
3) Database is upgraded from a previous version  
  
Look up SQL Server books online for the following commands: UPDATE STATISTICS, STATS\_DATE, DBCC SHOW\_STATISTICS, CREATE STATISTICS, DROP STATISTICS, sp\_autostats, sp\_createstats, sp\_updatestats

| What are the different ways of moving data/databases between servers and databases in SQL Server? |
| --- |

There are lots of options available, you have to choose your option depending upon your requirements. Some of the options you have are: BACKUP/RESTORE, dettaching and attaching databases, replication, DTS, BCP, logshipping, INSERT...SELECT, SELECT...INTO, creating INSERT scripts to generate data.

| Explian different types of BACKUPs avaialabe in SQL Server? Given a particular scenario, how would you go about choosing a backup plan? |
| --- |

Types of backups you can create in SQL Sever 7.0+ are Full database backup, differential database backup, transaction log backup, filegroup backup. Check out the BACKUP and RESTORE commands in SQL Server books online. Be prepared to write the commands in your interview. Books online also has information on detailed backup/restore architecture and when one should go for a particular kind of backup.

| What is database replicaion? What are the different types of replication you can set up in SQL Server? |
| --- |

Replication is the process of copying/moving data between databases on the same or different servers. SQL Server supports the following types of replication scenarios:

* Snapshot replication
* Transactional replication (with immediate updating subscribers, with queued updating subscribers)
* Merge replication

See SQL Server books online for indepth coverage on replication. Be prepared to explain how different replication agents function, what are the main system tables used in replication etc.

| How to determine the service pack currently installed on SQL Server? |
| --- |

The global variable @@Version stores the build number of the sqlservr.exe, which is used to determine the service pack installed. To know more about this process visit [~~SQL Server service packs and versions.~~](http://vyaskn.tripod.com/sqlsps.htm)

[~~Back to top~~](http://vyaskn.tripod.com/iq.htm#top)

| Database programming | [~~(top)~~](http://vyaskn.tripod.com/iq.htm#top) |
| --- | --- |

| What are cursors? Explain different types of cursors. What are the disadvantages of cursors? How can you avoid cursors? |
| --- |

Cursors allow row-by-row prcessing of the resultsets.  
  
Types of cursors: Static, Dynamic, Forward-only, Keyset-driven. See books online for more information.  
  
Disadvantages of cursors: Each time you fetch a row from the cursor, it results in a network roundtrip, where as a normal SELECT query makes only one rowundtrip, however large the resultset is. Cursors are also costly because they require more resources and temporary storage (results in more IO operations). Furthere, there are restrictions on the SELECT statements that can be used with some types of cursors.  
  
Most of the times, set based operations can be used instead of cursors. Here is an example:  
  
If you have to give a flat hike to your employees using the following criteria:  
  
Salary between 30000 and 40000 -- 5000 hike  
Salary between 40000 and 55000 -- 7000 hike  
Salary between 55000 and 65000 -- 9000 hike  
  
In this situation many developers tend to use a cursor, determine each employee's salary and update his salary according to the above formula. But the same can be achieved by multiple update statements or can be combined in a single UPDATE statement as shown below:  
  
UPDATE tbl\_emp SET salary =   
CASE WHEN salary BETWEEN 30000 AND 40000 THEN salary + 5000  
WHEN salary BETWEEN 40000 AND 55000 THEN salary + 7000  
WHEN salary BETWEEN 55000 AND 65000 THEN salary + 10000  
END  
  
Another situation in which developers tend to use cursors: You need to call a stored procedure when a column in a particular row meets certain condition. You don't have to use cursors for this. This can be achieved using WHILE loop, as long as there is a unique key to identify each row. For examples of using WHILE loop for row by row processing, check out the '[~~My code library~~](http://vyaskn.tripod.com/code.htm)' section of my site or [~~search~~](http://vyaskn.tripod.com/search.htm) for WHILE.

| Write down the general syntax for a SELECT statements covering all the options. |
| --- |

Here's the basic syntax: (Also checkout SELECT in books online for advanced syntax).  
  
SELECT select\_list  
[INTO new\_table\_]  
FROM table\_source  
[WHERE search\_condition]  
[GROUP BY group\_by\_expression]  
[HAVING search\_condition]  
[ORDER BY order\_expression [ASC | DESC] ]

| What is a join and explain different types of joins. |
| --- |

Joins are used in queries to explain how different tables are related. Joins also let you select data from a table depending upon data from another table.   
  
Types of joins: INNER JOINs, OUTER JOINs, CROSS JOINs. OUTER JOINs are further classified as LEFT OUTER JOINS, RIGHT OUTER JOINS and FULL OUTER JOINS.  
  
For more information see pages from books online titled: "Join Fundamentals" and "Using Joins".

| Can you have a nested transaction? |
| --- |

Yes, very much. Check out BEGIN TRAN, COMMIT, ROLLBACK, SAVE TRAN and @@TRANCOUNT

| What is an extended stored procedure? Can you instantiate a COM object by using T-SQL? |
| --- |

An extended stored procedure is a function within a DLL (written in a programming language like C, C++ using Open Data Services (ODS) API) that can be called from T-SQL, just the way we call normal stored procedures using the EXEC statement. See books online to learn how to create extended stored procedures and how to add them to SQL Server.   
  
Yes, you can instantiate a COM (written in languages like VB, VC++) object from T-SQL by using sp\_OACreate stored procedure. Also see books online for sp\_OAMethod, sp\_OAGetProperty, sp\_OASetProperty, sp\_OADestroy. For an example of creating a COM object in VB and calling it from T-SQL, see '[~~My code library~~](http://vyaskn.tripod.com/code.htm)' section of this site.

| What is the system function to get the current user's user id? |
| --- |

USER\_ID(). Also check out other system functions like USER\_NAME(), SYSTEM\_USER, SESSION\_USER, CURRENT\_USER, USER, SUSER\_SID(), HOST\_NAME().

| What are triggers? How many triggers you can have on a table? How to invoke a trigger on demand? |
| --- |

Triggers are special kind of stored procedures that get executed automatically when an INSERT, UPDATE or DELETE operation takes place on a table.   
  
In SQL Server 6.5 you could define only 3 triggers per table, one for INSERT, one for UPDATE and one for DELETE. From SQL Server 7.0 onwards, this restriction is gone, and you could create multiple triggers per each action. But in 7.0 there's no way to control the order in which the triggers fire. In SQL Server 2000 you could specify which trigger fires first or fires last using sp\_settriggerorder  
  
Triggers can't be invoked on demand. They get triggered only when an associated action (INSERT, UPDATE, DELETE) happens on the table on which they are defined.  
  
Triggers are generally used to implement business rules, auditing. Triggers can also be used to extend the referential integrity checks, but wherever possible, use constraints for this purpose, instead of triggers, as constraints are much faster.  
  
Till SQL Server 7.0, triggers fire only after the data modification operation happens. So in a way, they are called post triggers. But in SQL Server 2000 you could create pre triggers also. Search SQL Server 2000 books online for INSTEAD OF triggers.   
  
Also check out books online for 'inserted table', 'deleted table' and COLUMNS\_UPDATED()

| There is a trigger defined for INSERT operations on a table, in an OLTP system. The trigger is written to instantiate a COM object and pass the newly insterted rows to it for some custom processing. What do you think of this implementation? Can this be implemented better? |
| --- |

Instantiating COM objects is a time consuming process and since you are doing it from within a trigger, it slows down the data insertion process. Same is the case with sending emails from triggers. This scenario can be better implemented by logging all the necessary data into a separate table, and have a job which periodically checks this table and does the needful.

| What is a self join? Explain it with an example. |
| --- |

Self join is just like any other join, except that two instances of the same table will be joined in the query. Here is an example: Employees table which contains rows for normal employees as well as managers. So, to find out the managers of all the employees, you need a self join.  
  
CREATE TABLE emp   
(  
empid int,  
mgrid int,  
empname char(10)  
)  
  
INSERT emp SELECT 1,2,'Vyas'  
INSERT emp SELECT 2,3,'Mohan'  
INSERT emp SELECT 3,NULL,'Shobha'  
INSERT emp SELECT 4,2,'Shridhar'  
INSERT emp SELECT 5,2,'Sourabh'  
  
SELECT t1.empname [Employee], t2.empname [Manager]  
FROM emp t1, emp t2  
WHERE t1.mgrid = t2.empid

Here's an advanced query using a LEFT OUTER JOIN that even returns the employees without managers (super bosses)  
  
SELECT t1.empname [Employee], COALESCE(t2.empname, 'No manager') [Manager]  
FROM emp t1   
LEFT OUTER JOIN  
emp t2  
ON   
t1.mgrid = t2.empid

| Given an employee table, how would you find out the second highest salary? |
| --- |

[~~Why are my insert, update statements failing with the following error?~~  
  
~~Server: Msg 8152, Level 16, State 9, Line 1~~  
~~String or binary data would be truncated.~~  
~~The statement has been terminated.~~](http://vyaskn.tripod.com/programming_faq.htm#q1)  
  
[~~What is the T-SQL equivalent of IIF (immediate if/ternary operator) function of other programming languages?~~](http://vyaskn.tripod.com/programming_faq.htm#q2)  
  
[~~How to programmatically find out when the SQL Server service started?~~](http://vyaskn.tripod.com/programming_faq.htm#q3)  
  
[~~How to get rid of the time part from the date returned by GETDATE function?~~](http://vyaskn.tripod.com/programming_faq.htm#q4)  
  
[~~How to upload images or binary files into SQL Server tables?~~](http://vyaskn.tripod.com/programming_faq.htm#q5)  
  
[~~How to run an SQL script file that is located on the disk, using T-SQL?~~](http://vyaskn.tripod.com/programming_faq.htm#q6)  
  
[~~How to get the complete error message from T-SQL while error handling?~~](http://vyaskn.tripod.com/programming_faq.htm#q7)  
  
[~~How to get the first day of the week, last day of the week and last day of the month using T-SQL date functions?~~](http://vyaskn.tripod.com/programming_faq.htm#q8)  
  
[~~How to pass a table name, column name etc. to the stored procedure so that I can dynamically select from a table?~~](http://vyaskn.tripod.com/programming_faq.htm#q9)  
  
[~~Error inside a stored procedure is not being raised to my front-end applications using ADO. But I get the error when I run the procedure from Query Analyzer~~](http://vyaskn.tripod.com/programming_faq.htm#q10)[~~How to suppress error messages in stored procedures/triggers etc. using T-SQL?~~](http://vyaskn.tripod.com/programming_faq.htm#q11)[~~How to save the output of a query/stored procedure to a text file?~~](http://vyaskn.tripod.com/programming_faq.htm#q12)[~~How to join tables from different databases?~~](http://vyaskn.tripod.com/programming_faq.htm#q13)[~~How to join tables from different servers?~~](http://vyaskn.tripod.com/programming_faq.htm#q14)  
  
[~~How to convert timestamp data to date data (datetime datatype)?~~](http://vyaskn.tripod.com/programming_faq.htm#q15)  
  
[~~Can I invoke/instantiate COM objects from within stored procedures or triggers using T-SQL?~~](http://vyaskn.tripod.com/programming_faq.htm#q16)[~~Oracle has a rownum to access rows of a table using row number or row id. Is there any equivalent for that in SQL Server? Or how to generate output with row number in SQL Server?~~](http://vyaskn.tripod.com/programming_faq.htm#q17)[~~How to specify a network library like TCP/IP using ADO connect string?~~](http://vyaskn.tripod.com/programming_faq.htm#q18)  
  
[~~How to generate scripts for repetitive tasks like truncating all the tables in a database, changing owner of all the database objects, disabling constraints on all tables etc?~~](http://vyaskn.tripod.com/generate_scripts_repetitive_sql_tasks.htm)[~~Is there a way to find out when a stored procedure was last updated?~~](http://vyaskn.tripod.com/programming_faq.htm#q19)[~~How to find out all the IDENTITY columns of all the tables in a given database?~~](http://vyaskn.tripod.com/sql_server_programming_faq_2.htm#q1)[~~How to search the code of stored procedures?~~](http://vyaskn.tripod.com/sql_server_search_stored_procedure_code.htm)[~~How to retrieve the generated GUID value of a newly inserted row? Is there an @@GUID, just like @@IDENTITY ?~~](http://vyaskn.tripod.com/retrieve_guid_value_like_identity.htm)

| Why are my insert, update statements failing with the following error?  Server: Msg 8152, Level 16, State 9, Line 1 String or binary data would be truncated. The statement has been terminated. [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

This error occurs, when the length of the value entered by you into a char, varchar, nchar, nvarchar column is longer than the maximum length of the column. For example, inserting 'FAQ' into a char(2) column would result in this error.  
  
Profiler is handy in troubleshooting this error. If data truncation is okay with you and you don't want to see this error, then turn off ANSI WARNINGS by using the following SET command: SET ANSI\_WARNINGS OFF.   
  
Steps to reproduce the problem:  
  
CREATE TABLE MyTable  
(  
Pkey int PRIMARY KEY,  
Col1 char(10)  
)  
GO  
INSERT INTO MyTable (Pkey, Col1) VALUES (1, 'SQL Server Clustering FAQ')  
GO  
  
Make sure, you restrict the length of input, in your front-end applications. For example, you could use the MAXLENGTH property of the text boxes in HTML forms. E.g:  
  
<INPUT NAME = "Name" TYPE= TEXTBOX MAXLENGTH=20>

| [~~What is the T-SQL equivalent of IIF (immediate if/ternary operator) function of other programming languages?~~](http://vyaskn.tripod.com/programming_faq.htm#q2) [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

CASE is the equivalent of IIF function. See SQL Server Books Online for more information. Here's a quick example:  
  
CREATE TABLE People  
(  
[ID] int PRIMARY KEY,  
[Name] varchar(25) NOT NULL,  
Sex bit NULL  
)  
  
INSERT INTO People ([ID],[Name], Sex) VALUES (1,'John Dykes', 1)  
INSERT INTO People ([ID],[Name], Sex) VALUES (2,'Deborah Crook', 0)  
INSERT INTO People ([ID],[Name], Sex) VALUES (3,'P S Subramanyam', NULL)  
  
SELECT [ID], [Name],   
    CASE Sex   
        WHEN 1   
            THEN 'Male'   
        WHEN 0   
            THEN 'Female'   
        ELSE 'Not specified'   
    END AS Sex  
FROM People

| How to programmatically find out when the SQL Server service started? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

Everytime SQL Server starts, it recreates the tempdb database. So, the creation date and time of the tempdb database tells us the date and time at which SQL Server service started. This information is stored in the crdate column of the sysdatabases table in master database. Here's the query to find that out:  
  
SELECT crdate AS 'SQL Server service started approximately at:'  
FROM master.dbo.sysdatabases  
WHERE name = 'tempdb'  
  
SQL Server error log also has this information (This is more accurate) and the error log can be queried using xp\_readerrorlog

| How to get rid of the time part from the date returned by GETDATE function? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

We have to use the CONVERT function to strip the time off the date. Any of the following commands will do this:  
  
SELECT CONVERT(char,GETDATE(),101)  
SELECT CONVERT(char,GETDATE(),102)  
SELECT CONVERT(char,GETDATE(),103)  
SELECT CONVERT(char,GETDATE(),1)  
  
See SQL Server Books Online for more information on CONVERT function.

| How to upload images or binary files into SQL Server tables? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

First of all, if possible, try not to stored images and other binary files in the SQL Server tables, as they slow things down. Instead, store a link (file path) to the file in the tables and let your applications directly access the files. But if you must store these files within SQL Server, use the text/ntext or image datatype columns and consider the following options:

* SQL Server 7.0 and 2000 come with a utility called textcopy.exe. You can locate this file in the Binn folder under your SQL Server installation folder. Run this file from command prompt, and it will prompt you for required input
* Use the GetChunk and AppendChunk methods of ADO Field object. MSDN has examples
* Use the ADO Stream object
* Use the Bulk Insert Image utility (BII) that ships with SQL Server 2000 (Can be found at \Program Files\Microsoft SQL  
  Server\80\Tools\Devtools\Samples\Utils)

| How to run an SQL script file that is located on the disk, using T-SQL? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

There's no direct command to read a script file and execute it. But the isql.exe and osql.exe come in handy when you have to execute a script file from within T-SQL. Just call any of these exes using xp\_cmdshell and pass the script file name as parameter to it. See SQL Server Books Online for more information about the input parameters of these exes. Here are some quick examples:  
  
EXEC master..xp\_cmdshell 'osql -Svaio -Usa -Pzaassds1 -ic:\MySQl.sql -n'  
EXEC master..xp\_cmdshell 'isql -Svaio -Usa -Pzaassds1 -ic:\MySQl.sql -n'  
  
See xp\_cmdshell in SQL Server Books Online if you are having permissions problems in getting this technique to work.

| How to get the complete error message from T-SQL while error handling? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

Unfortunately, the error handling capabilities of SQL Server are limited. When an error occurs, all you can get is the error number, using the @@ERROR global variable. There is no @@ERROR\_MESSAGE global variable to get the error description.   
  
For a complete error message, you can always query the master..sysmessages table using the error number, but most of these messages have place holders (like %s, %l etc.), and hence we can't get the complete error message.  
  
However, the client applications using an object model such as RDO, ADO have access to the complete error message.

| How to get the first day of the week, last day of the week and last day of the month using T-SQL date functions? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

Here's the code:  
  
DECLARE @Date datetime  
SET @Date = '2001/08/31'  
SELECT DATEADD(dd,-(DATEPART(dw, @Date) - 1),@Date) AS 'First day of the week'  
SELECT DATEADD(dd,-(DATEPART(dw, @Date) - 7),@Date) AS 'Last day of the week'  
SELECT DAY(DATEADD(d, -DAY(DATEADD(m,1,@Date)),DATEADD(m,1,@Date))) AS 'Last day of the month'

| How to pass a table name, column name etc. to the stored procedure so that I can dynamically select from a table? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

Basically, SELECT and other commands like DROP TABLE won't let you use a variable instead of a hardcoded table name. To overcome this problem, you have to use dynamic sql. But dynamic SQL has some disadvantages. It's slow, as the dynamic SQL statement needs to be parsed everytime it's executed. Further, the user who is executing the dynamic SQL string needs direct permissions on the tables, which defeats the purpose of having stored procedures to mask the underlying tables. Having said that, here are some examples of dynamic SQL: (Also see sp\_executesql in SQL Server Books Online)  
  
CREATE PROC DropTable  
@Table sysname  
AS  
EXEC ('DROP TABLE ' + @Table)  
GO  
  
EXEC DropTable 'MyTable'  
GO  
  
CREATE PROC SelectTable  
@Table sysname  
AS  
EXEC ('SELECT \* FROM ' + @Table)  
GO  
  
EXEC SelectTable 'MyTable'  
  
For a complete discussion on the pros and cons of dynamic SQL check out Erland's article:   
[~~The curse and blessings of dynamic SQL~~](http://www.algonet.se/~sommar/dynamic_sql.html)

| Error inside a stored procedure is not being raised to my front-end applications using ADO. But I get the error when I run the procedure from Query Analyzer [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

This typically happens when your stored procedure is returning multiple resultsets and the offending SQL statement is executed after returning one or more resultsets. ADO will not return an error untill it processes all the recordsets returned before the offending SQL statement got executed. So, to get to the error message returned by your procedure. You have to loop through all the recordsets returned. ADO Recordset object has a method called NextRecordset, which lets you loop through the recordsets.   
  
Having SET NOCOUNT ON at the beginning of the procedure also helps avoid this problem. SET NOCOUNT ON also helps in improving the stored procedure performance. Here's a sample procedure to simulate the problem:  
  
CREATE PROC TestProc  
AS  
SELECT MAX(Col1) FROM TestTable  
SELECT MIN(Col1) FROM TestTable  
INSERT INTO TestTable (Col1, Col2) VALUES (1,'Oracle and SQL Server comparison')  
INSERT INTO TestTable (Col1, Col2) VALUES (1,'How to configure SQL Server?') -- Dupplicate key error occurs  
GO

| How to suppress error messages in stored procedures/triggers etc. using T-SQL? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

It's not possible to suppress error messages from within T-SQL. Error messages are always returned to the client. If you don't want your users to see these raw error messages, you should handle them in your front-end applications.  For example, if you are using ADO from ASP to connect to SQL Server, you would do something like the following:  
  
On Error Resume Next  
Set Rs = Conn.Execute ("INSERT INTO MyTable (1,'How to migrate from Oracle to SQL Server','Book'")  
If Err.Number <> 0 Then Response.Write ("Error occurred while inserting new data")  
On Error GoTo 0

| How to save the output of a query/stored procedure to a text file using T-SQL? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

T-SQL by itself has no support for saving the output of queries/stored procedures to text files. But you could achieve this using the command line utilities like isql.exe and osql.exe.  You could either invoke these exe files directly from command prompt/batch files or from T-SQL using the xp\_cmdshell command. Here are the examples:  
  
From command prompt:  
osql.exe -S YourServerName -U sa -P secretcode -Q "EXEC sp\_who2" -o "E:\output.txt"  
  
From T-SQL:  
EXEC master..xp\_cmdshell 'osql.exe -S YourServerName -U sa -P secretcode -Q "EXEC sp\_who2" -o "E:\output.txt"'  
  
Query Analyzer lets you save the query output to text files manually. The output of stored procedures that are run as a part of a scheduled job, can also be saved to a text file.  
  
BCP and Data Transformation Services (DTS) let you export table data to text files.

| How to join tables from different databases? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

You just have to qualify the table names in your SELECT queries with database name, followed by table owner name. In  the following example, Table1 from pubs database and Table2 from northwind database are being joined on the column i. Both tables are owned by dbo.  
  
SELECT a.i, a.j  
FROM pubs.dbo.Table1 a   
INNER JOIN   
northwind.dbo.Table2 b  
ON a.i = b.i  
GO

| How to join tables from different servers? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

To be able to join tables between two SQL Servers, first you have to link them. After the linked servers are setup, you just have to prefix your tables names with server name, database name, table owner name in your SELECT queries. The following example links SERVER\_01 to SERVER\_02. Execute the following commands in SERVER\_02:  
  
EXEC sp\_addlinkedserver SERVER\_01  
GO  
  
/\* The following command links 'sa' login on SERVER\_02 with the 'sa' login of SERVER\_01 \*/  
EXEC sp\_addlinkedsrvlogin @rmtsrvname = 'SERVER\_01', @useself = 'false', @locallogin = 'sa', @rmtuser = 'sa', @rmtpassword = 'sa password of SERVER\_01'  
GO  
  
SELECT a.title\_id  
FROM SERVER\_01.pubs.dbo.titles a  
INNER JOIN SERVER\_02.pubs.dbo.titles b  
ON a.title\_id = b.title\_id  
GO

| How to convert timestamp data to date data (datetime datatype)? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

The name timestamp is a little misleading. Timestamp data has nothing to do with dates and times and can not be converted to date data. A timestamp is a unique number within the database and is equivalent to a binary(8)/varbinary(8) datatype. A table can have only one timestamp column. Timestamp value of a row changes with every update of the row. To avoid the confusion, SQL Server 2000 introduced a synonym to timestamp, called rowversion.

| Can I invoke/instantiate COM objects from within stored procedures or triggers using T-SQL? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

Yes. SQL Server provides system stored procedures that let you instantiate COM objects using T-SQL from stored procedures, triggers and SQL batches. Search SQL Server Books Online for sp\_OACreate and sp\_OA\* for documentation and examples. Also check out my [~~code library~~](http://vyaskn.tripod.com/code.htm) for an example.

| Oracle has a rownum to access rows of a table using row number or row id. Is there any equivalent for that in SQL Server? Or how to generate output with row number in SQL Server? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

There is no direct equivalent to Oracle's rownum or row id in SQL Server. Strictly speaking, in a relational database, rows within a table are not ordered and a row id won't really make sense. But if you need that functionality, consider the following three alternatives:

* Add an IDENTITY column to your table. See Books Online for more information
* Use the following query to generate a row number for each row. The following query generates a row number for each row in the authors table of pubs database. For this query to work, the table must have a unique key.  
    
  SELECT        (SELECT COUNT(i.au\_id)   
                 FROM pubs..authors i   
                 WHERE i.au\_id >= o.au\_id ) AS RowID,   
                au\_fname + ' ' + au\_lname AS 'Author name'  
  FROM          pubs..authors o  
  ORDER BY      RowID
* Use a temporary table approach, to store the entire resultset into a temporary table, along with a row id generated by the IDENTITY() function. Creating a temporary table will be costly, especially when you are working with large tables. Go for this approach, if you don't have a unique key in your table. Search for IDENTITY (Function) in SQL Server Books Online.

For more ideas on this topic, [~~click here~~](http://support.microsoft.com/support/kb/articles/q186/1/33.asp) to read an informative article from Microsoft Knowledgebase.

| How to specify a network library like TCP/IP using ADO connect string? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

To specify TCP/IP net library, append the following to your ADO connect string:  
  
Network=dbmssocn  
  
For more information on specifying other net libraries in ADO connect strings, [~~click here~~](http://support.microsoft.com/support/kb/articles/Q238/9/49.ASP) to read the article from Microsoft Knowledgebase.

| Is there a way to find out when a stored procedure was last updated? [~~<top>~~](http://vyaskn.tripod.com/programming_faq.htm#top) |
| --- |

Simple answer is 'No'. The crdate column in the sysobjects table always contains the stored procedure create date, not the last updated date. You can use Profiler to trace ALTER PROC calls to the database, but you can't really afford to run a trace for ever, as it's resource intensive. Here is a simple idea! Whenever you have to alter your stored procedure, first drop it, then recreate it with the updated code. This resets the crdate column of sysobjects table. If you can make sure your developers always follow this plan, then the crdate column of sysobjects will always reflect the last updated date of the stored procedure. For example, if I have to modify a procedure named MyProc, instead of doing "ALTER PROC MyProc", here's what I would do:  
  
- Use sp\_helptext to get the current code of MyProc.   
- Change the code as needed.  
- Run the following code to drop the existing version of MyProc:  
  
IF EXISTS(SELECT 1 FROM sysobjects WHERE name = 'MyProc' AND type = 'P' AND USER\_NAME(uid) = 'dbo')  
BEGIN  
DROP PROC dbo.MyProc  
END  
  
- Run the updated code to recreate MyProc  
  
There is a much more powerful way out, if you can use Visual Source Safe (VSS). VSS is a version control software, that lets you manage your code. With VSS in place, you will have to maintain all your object creation scripts as script files and check them into VSS. When you have to modify a particular stored procedure, check out that script from VSS, modify it, test it, create the stored procedure, and check the script back into VSS. VSS can show you when a script got modified, by who and a whole lot of other information.  
  
Advantages of using VSS  
- You can version control your software, as VSS maintains all your changes as different versions  
- You can go back to a previous known good version of your stored procedure, if a developer makes a mistake  
- Using the labelling feature, you can revert back to an entire set of scripts at a particular point in time  
- You can control access to your source code by configuring permissions to your developers  
- By maintaining backups of VSS database, you can secure all your code centrally, instead of worrying about individual script files

# **What are views ?**

A view is simply a SELECT query saved in the database.  Thus, most operations you can perform on queries you can also perform on views.  However, there are some operations that apply only to one or the other.

| **View** | **Queries** |
| --- | --- |
| Views are stored as part of a database design | It is a specific request for retrival, Creation, Modification or deletion of data from the database. hey are part of DB design |
| They use SQL statements | They use SQL statements |
| Allows the administrator to disallow the access to the base table and permit access only to the views | No such option. |
| One cannot sort the view expect(TOP Clause) | One can sort the query result. |
| Can establish the query plan | No such option |
| Cannot create parameters | Can create parameters for a query |
| View can be encrypted | Query cannot be encrypted |

In addition, a view cannot include any **text**, **ntext**, or **image** columns, even if they are not referenced in the CREATE INDEX statement

Example

CREATE VIEW V1

WITH SCHEMABINDING

AS

SELECT SUM(UnitPrice\*Quantity\*(1.00-Discount)) AS Revenue, OrderDate, ProductID, COUNT\_BIG(\*) AS COUNT

FROM dbo.[Order Details] od, dbo.Orders o

WHERE od.OrderID=o.OrderID

GROUP BY OrderDate, ProductID

# **What are indexes?**

An index is similar to the index found the books. That facilitates easy search mechanism.

**An index in SQL Server assists the database engine with locating records, just like an index in a book helps you locate information quickly**.

**An index is structured by the SQL Server Index Manager as a Balanced tree (or B-tree).**

Columns consisting of the **ntext**, **text**, or **image** data types cannot be specified as columns for an index.

Example CREATE INDEX au\_id\_ind

ON authors (au\_id)

Permissions for the views and indexes:

Default to the **sysadmin** fixed server role and the **db\_ddladmin** and **db\_owner** fixed database roles and the table owner, and are not transferable.

Only the table or view owner can create indexes on that table. The owner of a table or view can create an index at any time, whether or not there is data in the table. Indexes can be created on tables or views in another database by specifying a qualified database name.

**Index in detail:**

Syntax

CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX *index\_name*   
    ON { *table* | *view* } **(** *column* [ ASC | DESC ] [ **,**...*n* ] **)**[ WITH < index\_option > [ **,**...*n*] ]   
[ ON *filegroup* ]

< index\_option > :: =   
    { PAD\_INDEX |   
    FILLFACTOR **=** *fillfactor* |   
    IGNORE\_DUP\_KEY |   
    DROP\_EXISTING |   
    STATISTICS\_NORECOMPUTE |   
    SORT\_IN\_TEMPDB    
}

## ***UNIQUE:***

Microsoft® SQL Server™ checks for duplicate values when the index is created (if data already exists) and checks each time data is added with an INSERT or UPDATE statement. If duplicate key values exist, the CREATE INDEX statement is canceled and an error message giving the first duplicate is returned. Multiple NULL values are considered duplicates when UNIQUE index is created.

Examples :

CREATE Unique INDEX au\_id\_ind

ON authors (au\_id)

## ***Clustered and NonClustered***

| **Clustered** | **Non-clustered** |
| --- | --- |
| The physical order of the rows in the table is the same as the logical indexed) order of the key values. | With a nonclustered index, the physical order of the rows is independent of their indexed order. The leaf level of a nonclustered index contains index rows. Each index row contains the nonclustered key value and one or more row locators that point to the row that contains the value.  **Note**: If the table does not have a clustered index, the row locator is the row's disk address. If the table does have a clustered index, the row locator is the clustered index key for the row. |
| A view with the clustered index is called the indexed view | For indexed views, nonclustered indexes can be created only on a view with a clustered index already defined |
| Create a clustered index before creating the non-clustered index, as it rebuilds the entire non-clustered index. |  |
| Must specify clustered to create an clustered index | If CLUSTERED is not specified, a nonclustered index is created. |
| A table can contain only one clustered index. | Each table can have as many as 249 nonclustered indexes. |

Examples

CREATE UNIQUE CLUSTERED INDEX employeeID\_ind

ON emp\_pay (employeeID)

Composite Index

CREATE INDEX emp\_order\_ind

ON order\_emp (orderID, employeeID)

Nonclustered using fillfactor.

CREATE NONCLUSTERED INDEX zip\_ind

ON authors (zip)

WITH FILLFACTOR = 100

### **PAD\_INDEX :**

Specifies the space to leave open on each page (node) in the intermediate levels of the index. The PAD\_INDEX option is useful only when FILLFACTOR is specified, because PAD\_INDEX uses the percentage specified by FILLFACTOR. The number of rows on an intermediate index page is never less than two, regardless of how low the value of FILLFACTOR.

CREATE INDEX au\_id\_ind

ON authors (au\_id)

WITH PAD\_INDEX, FILLFACTOR = 10

### **FILLFACTOR:**

Specifies a percentage that indicates how full SQL Server should make the leaf level of each index page during index creation. When FILLFACTOR is specified, SQL Server rounds up the number of rows to be placed on each page. For example, issuing CREATE CLUSTERED INDEX ... FILLFACTOR = 33 creates a clustered index with a FILLFACTOR of 33 percent. Assume that SQL Server calculates that 5.2 rows is 33 percent of the space on a page. SQL Server rounds so that six rows are placed on each page. User-specified FILLFACTOR values can be from 1 through 100. If no value is specified, the default is 0. When FILLFACTOR is set to 0, only the leaf pages are filled.

Use a FILLFACTOR of 100 only if no INSERT or UPDATE statements will occur, such as with a read-only table

### **IGNORE\_DUP\_KEY :**

Controls what happens when an attempt is made to insert a duplicate key value into a column that is part of a unique clustered index.

If IGNORE\_DUP\_KEY was not specified for the index, SQL Server issues an error message and rolls back the entire INSERT statement.

| **Index type** | **Options** |
| --- | --- |
| Clustered | Not allowed |
| Unique clustered | IGNORE\_DUP\_KEY allowed |
| Nonclustered | Not allowed |
| Unique nonclustered | IGNORE\_DUP\_KEY allowed |

CREATE UNIQUE CLUSTERED INDEX employeeID\_ind

ON emp\_pay(employeeID)

WITH IGNORE\_DUP\_KEY

### **DROP\_EXISTING :**

Specifies that the named, preexisting clustered or nonclustered index should be dropped and rebuilt. The index name specified must be the same as a currently existing index.

SQL SERVER 2000 supports Indexed views and creation on indexed view with “**schema binding**”.

**Schema binding** prohibits any table or column modification that would invalidate the view.  Any indexed view you create with the View Designer automatically gets schema binding, because SQL Server requires that indexed views have schema binding.

**Schema binding** does not mean you cannot modify the view; it means you cannot modify the underlying tables or views in ways that would change the view's result set

Restrictions on the Indexed Views:

The creator of the index must own the tables. All tables, the view, and the index, must be created in the same database.

The SELECT statement defining an indexed view must not have the TOP, DISTINCT, COMPUTE, HAVING, and UNION keywords. It cannot have a subquery.

The SELECT list may not include asterisks (\*), '*table*.\*' wildcard lists, DISTINCT, COUNT(\*), COUNT(<*expression*>), computed columns from the base tables, and scalar aggregates.

Nonaggregate SELECT lists cannot have expressions. Aggregate SELECT list (queries that contain GROUP BY) may include SUM and COUNT\_BIG(<expression>); it must contain COUNT\_BIG(\*). Other aggregate functions (MIN, MAX, STDEV,...) are not allowed.

Complex aggregation using AVG cannot participate in the SELECT list of the indexed view. However, if a query uses such aggregation, the optimizer is capable of using this indexed view to substitute AVG with a combination of simple aggregates SUM and COUNT\_BIG.

What is referential integrity & how is it achieved?

Ensure each value entered in a foreign key column matches an existing value in the related primary key column, it is done using foreign key constraint.

# **What is a Primary Key? Can it have >1 field?**

There are two kinds of keys.  A primary key is a set of columns from a table that are guaranteed to have unique values for each row of that table.  A primary key is also called a primary key constraint, because it effectively constrains the values you can add to the table: it prevents you from adding a row to the table whose primary key columns are all equal to the corresponding values of some other row in that table.

A foreign key is a correspondence between a set of columns in one table and the set of primary key columns in some other table.

Define a primary key to enforce uniqueness for values entered in specified columns that do not allow nulls.

If a primary key consists of more than one column, duplicate values are allowed in one column, but each combination of values from all the columns in the primary key must be unique.

# **What are constraints?**

Constraints are business logic that your database server enforces for you.

The Database Designer accepts five types of constraints:

**Check Constraints**: A check constraint specifies the data values or formats that are acceptable in one or more columns in a table. For example, you can require the zip column of the authors table to allow only five-digit numeric entries

**Default Constraints** : A default constraint enables you to define the value that will be supplied for a column whenever a user fails to enter a value. For example, in a table with a column called payterms, you can instruct your database server to enter "???" or "fill in later" if the user leaves it blank.

**Unique Constraints**: A unique constraint ensures no duplicate values are entered into specified columns that are not a table's primary key. For example, in the employee table in which the emp\_id column is the primary key, you can define a unique constraint that requires entries in the Social Security number (ssn) column to be unique within the table.

**Primary Key Constraints:** A primary key constraint ensures no duplicate values are entered in particular columns and that NULL values are not entered in those columns. You can use primary key constraints to enforce uniqueness as well as referential integrity.

For example, the au\_id column uniquely identifies each author stored in the authors table.

**Foreign Key Constraints** : A foreign key constraint works in conjunction with primary key or unique constraints to enforce referential integrity among specified tables. For example, you can place a foreign key constraint on the title\_id column in the publishers

table to ensure that a value entered in that column matches an existing value in the title\_id column of the titles table

# **What is a trigger and stored procedure?**

A stored procedure is a group of Transact-SQL statements that is compiled one time, and then can be executed many times. This increases performance when the stored procedure is executed because the Transact-SQL statements do not have to be recompiled.

A trigger is a special type of stored procedure that is not called directly by a user. When the trigger is created, it is defined to execute when a specific type of data modification is made against a specific table or column.

A CREATE PROCEDURE or CREATE TRIGGER statement cannot span batches. This means that a stored procedure or trigger is always created in a single batch and compiled into an execution plan.

\* Types of triggers?

A trigger is a database object that is *attached* to a table. In many aspects it is similar to a stored procedure. As a matter of fact, triggers are often referred to as a "special kind of stored procedure." The main difference between a trigger and a stored procedure is that the former is attached to a table and is only *fired* when an INSERT, UPDATE or DELETE occurs. You specify the modification action(s) that fire the trigger when it is created.

I only use them when I need to perform a certain action as a result of an INSERT, UPDATE or DELETE and ad hoc SQL (aka SQL Passthrough) is used. For example, let's say you want to send an email to the Sales Manager when an order is entered whose priority is high.

# **What are inserted and deleted tables?**

Triggers make use of two special tables called **inserted** and **deleted**. The inserted table contains the data referenced in an INSERT before it is actually committed to the database. The deleted table contains the data in the underlying table referenced in a DELETE before it is actually removed from the database. When an UPDATE is issued both tables are used. More specifically, the *new* data referenced in the UPDATE statement is contained in inserted and the data that is being updated is contained in deleted.

# **What is SQL Trace?**

Microsoft supplied the SQL Trace utility with SQL Server 6.x and replaced it with SQL Profiler in SQL Server 7.0. You can use SQL Trace, an Open Data Services (ODS) sniffer program, to monitor and record SQL Server 6.x database activity and troubleshoot 6.x systems. For example, you can capture five activity types (connections, SQL statements, remote procedure calls, attentions, and disconnections) within SQL Server 6.x. You can save generated traces as a trace file or an SQL script and apply five filters (login name, application, hostname, remote procedure call filter, and SQL statement filter).

Because SQL Trace is external to the SQL Server architecture, it has limited reporting capability. SQL Trace is an intrusive monitoring program that can report on only high-level events. For example, SQL Trace can report that a user is calling a certain stored procedure. If stored procedures contain many conditional statements, the same stored procedure (depending on the system state at that time and parameters passed to it) can perform wildly different actions. You cannot use SQL Trace to determine what the stored procedures will do. In addition, if you run SQL Trace on a severely stressed server, you might bring the server down.

SQL Server stores trace definitions in the Registry. If the user has appropriate permissions, SQL Server stores those permissions in the Registry of the SQL Server being traced. If the user doesn't have appropriate permissions, SQL Server stores user permissions in the Registry of the machine performing the trace. The location is not negotiable. If many developers or database administrators (DBAs) define traces, one server (typically a development server) could hold many trace definitions.

Although you can still use the old stored procedure xp\_trace (a stored procedure for controlling trace activity on a server), SQL Server *Books Online* (*BOL*) states that this stored procedure is "for backward compatibility only and may not be supported in future versions." In a future article, I'll tell you about a richer mechanism you can use for profiling that involves about 65 stored procedures, but this method of controlling profiles is beyond the scope of this article.

SQL Trace is the most powerful tool in my tuning bag of tricks. It lets me see the big picture of application-level interaction with SQL Server without knowing anything about the application or looking at the source code.

# **What is a join and what are different types of joins?**

The SQL join command is used to join two or more tables. At times you may need to join two or more tables to retrieve the information you need from your SQL database

Syntax:

Select *\*/tablename.fieldname<mask> ...*   
from *tablename <alias>, tablename <alias>, tablename <alias>*   
where *tablename.fieldname = tablename.fieldname*   
*and*   
*tablename.fieldname = tablename.fieldname*   
*and*   
*tablename.fieldname = tablename.fieldname*   
order by *fieldname <desc>...*

Explanations:

When you join two or more tables, a good idea is to precede the field names with the table names. This is not mandatory unless the same field name is found in more than one table.

If you precede the field name with a table name, place a period between the two names. For example, tablename.fieldname.

You must specify which fields are being joined.

If you do not specify which fields are being joined, the result is what is commonly referred to as a "Cartesian join" in which all rows in the first table are joined with all rows in the second table.

You can give each table name an *alias*, or alternative table name. When you assign an alias, you can then refer to the table by using its alias.

**Inner joins**

A general characteristic of inner joins is to find matching values and find the rows that have these matching values.

SELECT customer\_info.firstname, customer\_info.lastname, purchases.item

FROM customer\_info INNER JOIN purchases

ON customer\_info.customer\_number = purchases.customer\_number;

**Outer Join**

1. LEFT Outer Join
2. Right outer Join
3. FULL Outer Join

Outer joins retain unmatched rows from one or both the tables, depending on the keyword – LEFT, RIGHT or FULL – used.

# **Difference between union and join**

Combines the results of two or more queries into a single result set consisting of all the rows belonging to all queries in the union

Two basic rules for combining the result sets of two queries with UNION are:

The number and the order of the columns must be identical in all queries.

The data types must be compatible.

# **Difference between Group by and Having clause**

You can use a WHERE clause in a query containing a GROUP BY clause. Rows not meeting the conditions in the WHERE clause are eliminated before any grouping is done

Both the group by and having clause can be applied only on the aggregate fields

Specifies a search condition for a group or an aggregate. HAVING is usually used with the GROUP BY clause. When GROUP BY is not used, HAVING behaves like a WHERE clause.

# **SQL performance tuning**

Fine tuning the queries by avoiding the subquery

Creating indexes

Writing stored procedures and triggers

Creating a unique clustered index on a view improves query performance because the view is stored in the database in the same way a table with a clustered index is stored.

# **What is an execution plan?**

The execution plan determines the order in which a given query is executed , and the cost involved(CPU time and resources) at each step . The order involves , whether an Index(if available) is used to retrieve the rows or makes an FTS(Full Table Scan ). Looking at the E-plan, the developer can modify the query to ensure the rows are retrieved in faster way.

The e-plan for a query can be seen by typing the query in the query analyzer,select it , and choosing Query->Show Estimated Execution Plan(CTRL+M)

# **How will you return results as xml from SQL server?**

***Using “FOR XML” Clause in the query***

**Select \* from Authors FOR XML Raw**

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

Arguments

XML *mode*

Specifies the XML mode. XML mode determines the shape of the resulting XML.   
*mode* can be RAW, AUTO, or EXPLICIT.

RAW Mode

RAW mode transforms each row in the query result set into an XML element with the generic identifier row. Each column value that is not NULL is mapped to an attribute of the XML element in which the attribute name is the same as the column name.

The BINARY BASE64 option must be specified in the query in order to return the binary data in base64-encoded format. In RAW mode, retrieving binary data without specifying the BINARY BASE64 option results in an error.

When an XML-Data schema is requested, the schema, declared as a namespace, appears at the beginning of the data. In the result, the schema namespace reference is repeated for every top-level element.

AUTO Mode

AUTO mode returns query results as nested XML elements. Each table in the FROM clause, from which at least one column is listed in the SELECT clause, is represented as an XML element. The columns listed in the SELECT clause are mapped to the appropriate attribute of the element. When the ELEMENTS option is specified, the table columns are mapped to subelements instead of attributes. By default, AUTO mode maps the table columns to XML attributes.

A table name (or the alias, if provided) maps to the XML element name. A column name (or the alias, if provided) maps to an attribute name or to a non-complex subelement name when the ELEMENTS option is specified in the query.

The hierarchy (nesting of the elements) in the result set is based on the order of tables identified by the columns that are specified in the SELECT clause; therefore, the order in which column names are specified in the SELECT clause is significant.

The tables are identified and nested in the order in which the column names are listed in the SELECT clause. The first, leftmost table identified forms the top element in the resulting XML document. The second leftmost table (identified by columns in the SELECT statement) forms a subelement within the top element (and so on).

If a column name listed in the SELECT clause is from a table that is already identified by a previously specified column in the SELECT clause, the column is added as an attribute (or as a subelement if the ELEMENTS option is specified) of the element already created, instead of opening a new hierarchy (adding a new subelement for that table).

EXPLICIT Mode

In EXPLICIT mode, the query writer controls the shape of the XML document returned by the execution of the query. The query must be written in a specific way so that the additional information about the expected nesting is explicitly specified as part of the query. You can also specify additional configurations at the column level by using the directives. When you specify EXPLICIT mode, you must assume the responsibility for ensuring that the generated XML is well-formed and valid (in the case of an XML-DATA schema).

XMLDATA

Specifies that an XML-Data schema should be returned. The schema is prepended to the document as an inline schema.

ELEMENTS

If the ELEMENTS option is specified, the columns are returned as subelements. Otherwise, they are mapped to XML attributes. This option is supported in AUTO mode only.

BINARY BASE64

If the BINARY Base64 option is specified, any binary data returned by the query is represented in base64-encoded format. To retrieve binary data using RAW and EXPLICIT mode, this option must be specified. In AUTO mode, binary data is returned as a reference by default.

**Microsoft® SQL Server™ 2000 introduces several server improvements and new features:**

### **XML Support**

* The relational database engine can return data as Extensible Markup Language (XML) documents.
* Additionally, XML can also be used to insert, update, and delete values in the database.
* Support for XML-Data schemas and the ability to specify XPath queries against these schemas.
* The ability to retrieve and write XML data:
  + Retrieve XML data using the SELECT statement and the FOR XML clause.
  + Write XML data using the OpenXML rowset provider.

You can expose the data from an XML document as a relational rowset using the new OPENXML rowset function. OPENXML can be used everywhere a rowset function can be used in a Transact-SQL statement, such as in place of a table or view reference in a FROM clause. This allows you to use the data in XML documents to insert, update, or delete data in the tables of the database, including modifying multiple rows in multiple tables in a single operation.

* Enhancements to the Microsoft SQL Server 2000 OLE DB provider (SQLOLEDB) that allow XML documents to be set as command text and to return result sets as a stream.

### **Federated Database Servers**

SQL Server 2000 supports enhancements to distributed partitioned views that allow you to partition tables horizontally across multiple servers. This allows you to scale out one database server to a group of database servers that cooperate to provide the same performance levels as a cluster of database servers. This group, or federation, of database servers can support the data storage requirements of the largest Web sites and enterprise data processing systems.

SQL Server 2000 introduces Net-Library support for Virtual Interface Architecture (VIA) system-area networks that provide high-speed connectivity between servers, such as between application servers and database servers.

### **User-Defined Functions**

Allow you to define your own Transact-SQL functions using the CREATE FUNCTION statement.

User-defined functions take zero or more input parameters, and return a single value. Some user-defined functions return a single, scalar data value, such as an **int**, **char**, or **decimal** value. They can also return a **table** data type

A function can declare an internal **table** variable, insert rows into the variable, and then return the variable as its return value.

A class of user-defined functions known as in-line functions, return the result set of a SELECT statement as a variable of type **table**.

### **Indexed Views**

An indexed view allows indexes to be created on views, where the result set of the view is stored and indexed in the database.

Indexed views work best when the underlying data is infrequently updated. The maintenance of an indexed view can be higher than the cost of maintaining a table index. If the underlying data is updated frequently, then the cost of maintaining the indexed view data may outweigh the performance benefits of using the indexed view.

Indexed views improve the performance of these types of queries:

* Joins and aggregations that process many rows.
* Join and aggregation operations that are frequently performed by many queries.

Note:

* Views can be used to partition data across multiple databases or instances of Microsoft® SQL Server™ 2000.
* Views in all versions of SQL Server are updatable (can be the target of UPDATE, DELETE, or INSERT statements), as long as the modification affects only one of the base tables referenced by the view.

### **New Data Types**

SQL Server 2000 introduces three new data types.

**bigint** is an 8-byte integer type.

**sql\_variant** is a type that allows the storage of data values of different data types.

**table** is a type that allows applications to store results temporarily for later use. It is supported for variables, and as the return type for user-defined functions.

### **INSTEAD OF and AFTER Triggers**

INSTEAD OF triggers are executed instead of the triggering action (for example, INSERT, UPDATE, DELETE). They can also be defined on views, in which case they greatly extend the types of updates a view can support. AFTER triggers fire after the triggering action. SQL Server 2000 introduces the ability to specify which AFTER triggers fire first and last.

AFTER is the default in SQL Server 2000.

### **Cascading Referential Integrity Constraints**

You can control the actions SQL Server 2000 takes when you attempt to update or delete a key to which existing foreign keys point. This is controlled by the new ON DELETE and ON UPDATE clauses in the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements.

CREATE TABLE order\_part

(order\_nmbr int,

part\_nmbr int

FOREIGN KEY REFERENCES part\_sample(part\_nmbr)

**ON DELETE NO ACTION**,

qty\_ordered int)

* NO ACTION specifies that the deletion fails with an error.
* CASCADE specifies that all the rows with foreign keys pointing to the deleted row are also deleted.

### **Collation Enhancements**

SQL Server 2000 replaces code pages and sort orders with collations. SQL Server 2000 includes support for most collations supported in earlier versions of SQL Server, and introduces a new set of collations based on Windows collations. You can now specify collations at the database level or at the column level. Previously, code pages and sort orders could be specified only at the server level and applied to all databases on a server.

Collations support code page translations. Operations with **char** and **varchar** operands having different code pages are now supported. Code page translations are not supported for **text** operands. You can use ALTER DATABASE to change the default collation of a database.

### **Full-Text Search Enhancements**

Full-text search now includes change tracking and image filtering. Change tracking maintains a log of all changes to the full-text indexed data. You can update the full-text index with these changes by flushing the log manually, on a schedule, or as they occur, using the background update index option. Image filtering allows you to index and query documents stored in **image** columns. The user provides the document type in a column that contains the file name extension that the document would have had if it were stored as a file in the file system. Using this information, full-text search is able to load the appropriate document filter to extract textual information for indexing.

### **Multiple Instances of SQL Server**

SQL Server 2000 supports running multiple instances of the relational database engine on the same computer. Each computer can run **one** instance of the relational database engine from **SQL Server version 6.5 or 7.0**, along with **one or more** instances of the database engine from **SQL Server 2000**. Each instance has its own set of system and user databases. Applications can connect to each instance on a computer similar to the way they connect to instances of SQL Servers running on different computers. The SQL Server 2000 utilities and administration tools have been enhanced to work with multiple instances.

### **Index Enhancements**

You can now create indexes on computed columns. You can specify whether indexes are built in ascending or descending order, and if the database engine should use parallel scanning and sorting during index creation.

The CREATE INDEX statement can now use the **tempdb** database as a work area for the sorts required to build an index. This results in improved disk read and write patterns for the index creation step, and makes it more likely that index pages will be allocated in contiguous strips. In addition, the complete process of creating an index is eligible for parallel operations, not only the initial table scan.

### **Failover Clustering Enhancements**

The administration of failover clusters has been greatly improved to make it very easy to install, configure, and maintain a Microsoft SQL Server 2000 failover cluster. Additional enhancements include the ability to failover and failback to or from any node in a SQL Server 2000 cluster, the ability to add or remove a node from the cluster through SQL Server 2000 Setup, and the ability to reinstall or rebuild a cluster instance on any node in the cluster without affecting the other cluster node instances. The SQL Server 2000 utilities and administration tools have been enhanced to work with failover clusters.

### **Net-Library Enhancements**

The SQL Server 2000 Net-Libraries have been rewritten to virtually eliminate the need to administer Net-Library configurations on client computers when connecting SQL Server 2000 clients to instances of SQL Server 2000. The new Net-Libraries also support connections to multiple instances of SQL Server on the same computer, and support Secure Sockets Layer encryption over all Net-Libraries. SQL Server 2000 introduces Net-Library support for Virtual Interface Architecture (VIA) system-area networks that provide high-speed connectivity between servers, such as between application servers and database servers.

### **64-GB Memory Support**

Microsoft SQL Server 2000 Enterprise Edition can use the Microsoft Windows 2000 Advanced Windows Extension (AWE) API to support up to 64 GB of physical memory (RAM) on a computer.

### **Distributed Query Enhancements**

SQL Server 2000 introduces a new OPENDATASOURCE function, which you can use to specify ad hoc connection information in a distributed query. SQL Server 2000 also specifies methods that OLE DB providers can use to report the level of SQL syntax supported by the provider and statistics on the distribution of key values in the data source. The distributed query optimizer can then use this information to reduce the amount of data that has to be sent from the OLE DB data source. SQL Server 2000 delegates more SQL operations to OLE DB data sources than earlier versions of SQL Server. Distributed queries also support the other functions introduced in SQL Server 2000, such as multiple instances, mixing columns with different collations in result sets, and the new **bigint** and **sql\_variant** data types.

SQL Server 2000 distributed queries add support for the OLE DB Provider for Exchange and the Microsoft OLE DB Provider for Microsoft Directory Services.

### **Updatable Distributed Partitioned Views**

SQL Server 2000 introduces enhancements to distributed partitioned views. You can partition tables horizontally across several servers, and define a distributed partitioned view on each member server that makes it appear as if a full copy of the original table is stored on each server. Groups of servers running SQL Server that cooperate in this type of partitioning are called federations of servers. A database federation built using SQL Server 2000 databases is capable of supporting the processing requirements of the largest Web sites or enterprise-level databases.

### **Kerberos and security delegation**

SQL Server 2000 uses Kerberos to support mutual authentication between the client and the server, as well as the ability to pass the security credentials of a client between computers, so that work on a remote server can proceed using the credentials of the impersonated client. With Microsoft Windows® 2000, SQL Server 2000 uses Kerberos and delegation to support both integrated authentication as well as SQL Server logins.

### **Backup and Restore Enhancements**

SQL Server 2000 introduces a new, more easily understood model for specifying backup and restore options. The new model makes it clearer that you are balancing increased or decreased exposure to losing work against the performance and log space requirements of different plans. SQL Server 2000 introduces support for recovery to specific points of work using named log marks in the transaction log, and the ability to do partial database restores.

Users can define passwords for backup sets and media sets that prevent unauthorized users from accessing SQL Server backups.

### **Scalability Enhancements for Utility Operations**

SQL Server 2000 enhancements for utility operations include faster differential backups, parallel Database Console Command (DBCC) checking, and parallel scanning. Differential backups can now be completed in a time that is proportional to the amount of data changed since the last full backup. DBCC can be run without taking shared table locks while scanning tables, thereby enabling them to be run concurrently with update activity on tables. Additionally, DBCC now takes advantage of multiple processors, thus enabling near-linear gain in performance in relation to the number of CPUs (provided that I/O is not a bottleneck).

### **Text in Row Data**

SQL Server 2000 supports a new **text in row** table option that specifies that small **text**, **ntext**, and **image** values be placed directly in the data row instead of in a separate page. This reduces the amount of space used to store small **text**, **ntext**, and **image** data values, and reduces the amount of disk I/O needed to process these values.

The administrator uses the text in row option in sp\_tableoption to specify whether small text, ntext, or image values are stored directly in a row:

* When text in row is OFF, SQL Server 2000 has the same ntext, text, and image behavior as SQL Server version 7.0. For each text, ntext, or image value, all that is stored in the data row is a 16-byte pointer.
* When text in row is ON, SQL Server 2000 stores small text, ntext, and image values in the data row. Only text, ntext, or image values that cannot fit in the row are stored in a separate collection of pages.

### **SQL Query Analyzer Enhancements**

SQL Query Analyzer now includes Object Browser, which allows you to navigate through and get information (such as parameters and dependencies) about database objects, including user and system tables, views, stored procedures, extended stored procedures, and functions. The Object Browser also supports generating scripts to either execute or create objects. Other enhancements include server tracing and client statistics that show information about the server-side and client-side impact of a given query.

### **Saving DTS Packages to Visual Basic files**

DTS packages now can be saved to a Microsoft Visual Basic file. This allows a package created by the DTS Import/Export Wizard or DTS Designer to be incorporated into Visual Basic programs or to be used as prototypes by Visual Basic developers who need to reference the components of the DTS object model.

### **General Information**

* SQL Server also supports up to 249 nonclustered indexes on each table or indexed view.
* Objects in a Microsoft® SQL Server™ 2000 database are stored as a collection of 8-KB pages.

***Performance tuning in SQL server 2000 Tm***

The goal of performance tuning is to minimize the response time for each query and to maximize the throughput of the entire database server by reducing network traffic, disk I/O, and CPU time. This goal is achieved through understanding application requirements, the logical and physical structure of the data, and tradeoffs between conflicting uses of the database, such as online transaction processing (OLTP) versus decision support.

To most effectively optimize the performance of Microsoft® SQL Server™ 2000, you must identify the areas that will yield the largest performance increases over the widest variety of situations and focus analysis on those areas.

**Designing Federated Database Servers:**

To achieve the high levels of performance required by the largest Web sites, a multi tier system typically balances the processing load for each tier across multiple servers. Microsoft® SQL Server™ 2000 shares the database processing load across a group of servers by horizontally partitioning the SQL Server data. These servers are managed independently, but cooperate to process the database requests from the applications; such a cooperative group of servers is called a federation.

A federated database tier can achieve extremely high levels of performance only if the application sends each SQL statement to the member server that has most of the data required by the statement. This is called collocating the SQL statement with the data required by the statement. Collocating SQL statements with the required data is not a requirement unique to federated servers. It is also required in clustered systems

**Database Design:**

There are two components to designing a database: logical and physical. Logical database design involves modeling your business requirements and data using database components, such as tables and constraints, without regard for how or where the data will be physically stored. Physical database design involves mapping the logical design onto physical media, taking advantage of the hardware and software features available, which allows the data to be physically accessed and maintained as quickly as possible, and indexing.

It is important to correctly design the database to model your business requirements, and to take advantage of hardware and software features early in the development cycle of a database application, because it is difficult to make changes to these components later.

**Query Tuning:**

Most of the performance problem can be resolved by analyzing the by analyzing the application, queries, and updates that the application is submitting to the database, and how these queries and updates interact with the database schema.

Unexpected long-lasting queries and updates can be caused by:

* Slow network communication.
* Inadequate memory in the server computer or not enough memory available for Microsoft® SQL Server™ 2000.
* Lack of useful statistics.
* Out-of-date statistics.
* Lack of useful indexes.
* Lack of useful data striping.

When a query or update takes longer than expected, use the following checklist to improve performance.

1. Is the performance problem related to a component other than queries? For example, is the problem slow network performance? Are there any other components that might be causing or contributing to performance degradation? Windows NT Performance Monitor can be used to monitor the performance of SQL Server and non-SQL Server related components
2. If the performance issue is related to queries, which query or set of queries is involved? Use SQL Profiler to help identify the slow query or queries.
3. The performance of a database query can be determined by using the SET statement to enable the SHOWPLAN, STATISTICS IO, STATISTICS TIME, and STATISTICS PROFILE options.
   * SHOWPLAN describes the method chosen by the SQL Server query optimizer to retrieve data.
   * STATISTICS IO reports information about the number of scans, logical reads (pages accessed in cache), and physical reads (number of times the disk was accessed) for each table referenced in the statement.
   * STATISTICS TIME displays the amount of time (in milliseconds) required to parse, compile, and execute a query.
   * STATISTICS PROFILE displays a result set after each executed query representing a profile of the execution of the query.

In SQL Query Analyzer, you can also turn on the **graphical execution plan** option to view a graphical representation of how SQL Server retrieves data.

The information gathered by these tools allows you to determine how a query is executed by the SQL Server query optimizer and which indexes are being used. Using this information, you can determine if performance improvements can be made by rewriting the query, changing the indexes on the tables, or perhaps modifying the database design.

1. Was the query optimized with useful statistics?

Statistics on the distribution of values in a column are automatically created on indexed columns by SQL Server. They can also be created on nonindexed columns either manually, using SQL Query Analyzer or the CREATE STATISTICS statement, or automatically, if the **auto create statistics** database option is set to **true**. These statistics can be used by the query processor to determine the optimal strategy for evaluating a query. Maintaining additional statistics on nonindexed columns involved in join operations can improve query performance.

Monitor the query using SQL Profiler or the graphical execution plan in SQL Query Analyzer to determine if the query has enough statistics.

1. Are the query statistics up-to-date? Are the statistics automatically updated?

SQL Server automatically creates and updates query statistics on indexed columns (as long as automatic query statistic updating is not disabled). Additionally, statistics can be updated on nonindexed columns either manually, using SQL Query Analyzer or the UPDATE STATISTICS statement, or automatically, if the **auto update statistics** database option is set to **true**. Up-to-date statistics are not dependent upon date or time data. If no UPDATE operations have taken place, then the query statistics are still up-to-date.

If statistics are not set to update automatically, then set them to do so.

1. Are suitable indexes available? Would adding one or more indexes improve query performance?
2. Are there any data or index hot spots? Consider using disk striping.
3. Is the query optimizer provided with the best opportunity to optimize a complex query?.

**Application Design**

Application design plays a pivotal role in determining the performance of a system using Microsoft® SQL Server™ 2000. Consider the client the controlling entity rather than the database server. The client determines the type of queries, when they are submitted, and how the results are processed. This in turn has a major effect on the type and duration of locks, amount of I/O, and processing (CPU) load on the server, and hence on whether performance is generally good or bad.

For this reason, it is important to make the correct decisions during the application design phase. However, even if a performance problem occurs using a turn-key application, where changes to the client application seem impossible, this does not change the fundamental factors that affect performance: The client plays a dominant role and many performance problems cannot be resolved without making client changes.

A well-designed application allows SQL Server to support thousands of concurrent users. Conversely, a poorly designed application prevents even the most powerful server platform from handling more than a few users.

**Guidelines for client-application design include:**

* Eliminate excessive network traffic.
* Use small result sets.
* Allow cancellation of a query in progress when the user needs to regain control of the application.
* Always implement a query or lock time-out.

Do not use application development tools that do not allow explicit control over the SQL statements sent to SQL Server.

* Do not intermix decision support and online transaction processing (OLTP) queries.

Do not use cursors more than necessary.

* Keep transactions as short as possible.
* Use stored procedures.
* Use prepared execution to execute a parameterized SQL statement.
* Always process all results to completion.
* Do not design an application or use an application that stops processing result rows without canceling the query. Doing so will usually lead to blocking and slow performance.
* Ensure that your application is designed to avoid deadlocks.
* Ensure that all the appropriate options for optimizing the performance of distributed queries have been set.

**Optimizing Utility and Tool Performance**

Three operations performed on a production database that can benefit from optimal performance include:

* Backup and restore operations.
* Bulk copying data into a table.
* Performing database console command (DBCC) operations.

**Optimizing Server Performance**

Microsoft® SQL Server™ 2000 automatically tunes many of the server configuration options, therefore requiring little, if any, tuning by a system administrator. Although these configuration options can be modified by the system administrator, it is generally recommended that these options be left at their default values, allowing SQL Server to automatically tune itself based on run-time conditions.

However, if necessary, the following components can be configured to optimize server performance:

* SQL Server Memory
* I/O subsystem
* Microsoft Windows NT® options

1. **Types of Joins**

Joins as used to combine the contents of two or more tables and produce a result set that incorporates rows and columns from each table. Tables are typically joined using data that they have in common

Join conditions can be specified in either the FROM or WHERE clauses; specifying them in the FROM clause is recommended. WHERE and HAVING clauses can also contain search conditions to further filter the rows selected by the join conditions.

**Joins can be categorized as:**

**Inner joins** (the typical join operation, which uses some comparison operator like = or <>). An inner join is a join in which the values in the columns being joined are compared using a comparison operator

Inner joins use a comparison operator to match rows from two tables based on the values in common columns from each table.

# **Equi Join**

It returns all the columns in both tables, and returns only the rows for which there is an equal value in the join column

( SELECT **\*** FROM authors AS a INNER JOIN publishers AS p ON a.city = p.city ORDER BY a.au\_lname DESC )

# **Self Join**

A table can be joined to itself in a self-join

**Outer joins**.

Outer joins can be a left, right, or full outer join.

Outer joins are specified with one of the following sets of keywords when they are specified in the FROM clause:

# **LEFT JOIN or LEFT OUTER JOIN**

The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause, not just the ones in which the joined columns match. When a row in the left table has no matching rows in the right table, the associated result set row contains null values for all select list columns coming from the right table.

**RIGHT JOIN or RIGHT OUTER JOIN.**

A right outer join is the reverse of a left outer join. All rows from the right table are returned. Null values are returned for the left table any time a right table row has no matching row in the left table.

**FULL JOIN or FULL OUTER JOIN.**

A full outer join returns all rows in both the left and right tables. Any time a row has no match in the other table, the select list columns from the other table contain null values. When there is a match between the tables, the entire result set row contains data values from the base tables.

**Cross joins.**

Cross joins return all rows from the left table, each row from the left table is combined with all rows from the right table. Cross joins are also called Cartesian products.

1. **Diff. between left and Right outer Join.**

The result set of a left outer join includes all the rows from the left table specified in the LEFT OUTER clause , while in case of Right outer join all the rows from the right table are returned in the result set.

1. **Unicode In Sql server**

Using Unicode data types, a column can store any character that is defined by the Unicode Standard, which includes all of the characters that are defined in the various character sets. Unicode data types take twice as much storage space as non-Unicode data types.

Unicode data is stored using the **nchar**, **nvarchar**, and **ntext** data types in SQL Server. Use these data types for columns that store characters from more than one character set.

The SQL Server Unicode data types are based on the National Character data types in the SQL-92 standard

1. **Fill Factor and its value when the index is created.**

An option used when creating an index to reserve free space on each page of the index. FILLFACTOR accommodates future expansion of table data and reduces the potential for page splits. FILLFACTOR is a value from 1 through 100 that specifies the percentage of the index page to be left empty.

Default is **0**

A **fill factor** value of 0 does not mean that pages are 0 percent full. It is treated similarly to a **fill factor** value of 100 in that SQL Server creates clustered indexes with full data pages and nonclustered indexes with full leaf pages.

If you set **fill factor** to 100, SQL Server creates both clustered and nonclustered indexes with each page 100 percent full. Setting **fill factor** to 100 is suitable only for read-only tables, to which additional data is never added.

1. **Types of Filebackup options**

All data and objects in the database, such as tables, stored procedures, triggers, and views, are stored only within the following operating system files:

# **Primary**

This file contains the startup information for the database and is used to store data. Every database has one primary data file.

# **Secondary**

These files hold all of the data that does not fit in the primary data file. If the primary file can hold all of the data in the database, databases do not need to have secondary data files. Some databases may be large enough to need multiple secondary data files or to use secondary files on separate disk drives to spread data across multiple disks.

# **Transaction Log**

These files hold the log information used to recover the database. There must be at least one log file for each database.

**File groups** allow files to be grouped together for administrative and data allocation/placement purposes

# **Rules of Files and Filegroups**

Rules for designing files and filegroups include:

* A file or filegroup cannot be used by more than one database. For example, file sales.mdf and sales.ndf, which contain data and objects from the **sales** database, cannot be used by any other database.
* A file can be a member of only one filegroup.
* Data and transaction log information cannot be part of the same file or filegroup.
* Transaction log files are never part of any filegroups.

## ***Types of BackUP***

Database

Transaction log

Differential

File and filegroup

1. **Differential File group backup**

Differential database backup records only those data changes made to the database after the last full database backup. A differential database backup is smaller and takes less time to complete than a database backup. By creating differential database backups more frequently than database backups, you can decrease the amount of data you risk losing.

1. **What is Collate**

A clause that can be applied to a database definition or a column definition to define the collation, or to a character string expression to apply a collation cast.

The COLLATE clause can be applied only for the **char**, **varchar**, **text**, **nchar**, **nvarchar**, and **ntext** data types.

The physical storage of character strings in Microsoft® SQL Server™ 2000 is controlled by **collations**. A collation specifies the bit patterns that represent each character and the rules by which characters are sorted and compared

1. **What is Code Page**

A character set that a computer uses to interpret and display characters, often to handle international characters. Essentially, it is a table of characters and corresponding numbers in memory that the computer uses to display data properly. Different languages and locales may use different code pages.

1. **When we should go for with Recompile option in Stored Procedures**

When stored procedures take parameters whose values differ widely between executions of the stored procedure, resulting in different execution plans to be created each time. If the parameter you are supplying is atypical or if the data has significantly changed since the stored procedure was created.

Use of this option is uncommon, and causes the stored procedure to execute more slowly because the stored procedure must be recompiled each time it is executed.

Creating a stored procedure that specifies the WITH RECOMPILE option in its definition indicates that SQL Server does not cache a plan for this stored procedure; the stored procedure is recompiled each time it is executed.

1. **Diff. between Having and Group by Clause**

The HAVING clause sets conditions on the GROUP BY clause similar to the way WHERE interacts with SELECT. The WHERE search condition is applied before the grouping operation occurs; the HAVING search condition is applied after the grouping operation occurs. The HAVING syntax is exactly like the WHERE syntax, except HAVING can contain aggregate functions. HAVING clauses can reference any of the items that appear in the select list.

The WHERE clause is used to filter the rows that result from the operations specified in the FROM clause.

The GROUP BY clause is used to group the output of the WHERE clause.

The HAVING clause is used to filter rows from the grouped result

Where – Groupby – Having – Order By

1. **Isolation levels**

An isolation level determines the degree to which data is isolated for use by one process and guarded against interference from other processes.

# **Read Committed**

SQL Server acquires a share lock while reading a row into a cursor but frees the lock immediately after reading the row. Because a shared lock request is blocked by an exclusive lock, a cursor is prevented from reading a row that another task has updated but not yet committed. Read committed is the default isolation level setting for both SQL Server and ODBC.

# **Read Uncommitted**

SQL Server requests no locks while reading a row into a cursor and honors no exclusive locks. Cursors can be populated with values that have already been updated but not yet committed. The user is bypassing all of SQL Server’s locking transaction control mechanisms.

# **Repeatable Read or Serializable**

SQL Server requests a shared lock on each row as it is read into the cursor as in READ COMMITTED, but if the cursor is opened within a transaction, the shared locks are held until the end of the transaction instead of being freed after the row is read. This has the same effect as specifying HOLDLOCK on a SELECT statement.

1. **When we should go for NOLOCK hint**

A range of table-level locking hints can be specified using the SELECT, INSERT, UPDATE, and DELETE statements to direct Microsoft® SQL Server™ 2000 to the type of locks to be used. Table-level locking hints can be used when a finer control of the types of locks acquired on an object is required. These locking hints override the current transaction isolation level for the session.

**NOLOCK**

Do not issue shared locks and do not honor exclusive locks. When this option is in effect, it is possible to read an uncommitted transaction or a set of pages that are rolled back in the middle of a read. Dirty reads are possible. Only applies to the SELECT statement.

1. **Types of Database import.**

BCP, DTS, BulkCopy Command

1. **Diff. between BCP and DTS, which one is advantageous**

**bcp** is a command prompt utility. **bcp** provides for running bulk copies in .bat and .cmd scripts. **bcp** is used to bulk copy large files into tables or views in SQL Server databases. BCP is non logged operation.

**DTS** allows the user to program transformations through two different interfaces.

BCP is extremely fast and has a minimal overhead, but it also has a very rigid (and sometimes unforgiving) syntax. Then along came Data Transformation Services (DTS), an improved method for importing and exporting data between heterogeneous data sources. Whether you want to move data from a legacy system on a onetime basis or continually move data back and forth for data warehousing, DTS should be your first choice. With DTS you don't need to struggle with BCP anymore. DTS is extremely flexible and surprisingly fast, and you can use the technology to copy and transform data to or from any OLE DB or ODBC data source.

BCP is designed to work with flat files, while DTS can work with "any" database. using BCP, you have a much more mature tool, and clearer error messages than with DTS.

1. **What are DBCC COMMANDS?**

The Transact-SQL programming language provides DBCC statements that act as the “database consistency checker” for Microsoft® SQL Server™. These statements check the physical and logical consistency of a database. Many DBCC statements can fix detected problems.

These database consistency-checking statements are grouped into these categories.

**Maintenance Statements:**

Maintenance tasks on a database, index, or filegroup.

DBCC DBREPAIR

DBCC SHRINKFILE

DBCC DBREINDEX

DBCC UPDATEUSAGE

DBCC SHRINKDATABASE

**Miscellaneous Statements:**

Miscellaneous tasks such as enabling row-level locking or removing a dynamic-link library (DLL) from memory

DBCC dllname (FREE)

DBCC TRACEOFF

DBCC HELP

DBCC TRACEON

DBCC PINTABLE

DBCC UNPINTABLE

DBCC ROWLOCK

**Status Statements:**

Status Checks

DBCC INPUTBUFFER

DBCC SHOW\_STATISTICS

DBCC OPENTRAN

DBCC SQLPERF

DBCC OUTPUTBUFFER

DBCC TRACESTATUS

DBCC PROCCACHE

DBCC USEROPTIONS

DBCC SHOWCONTIG

**Validation Statements:**

Validation operations on a database, table, index, catalog, filegroup, system tables, or allocation of database pages.

DBCC CHECKALLOC

DBCC CHECKTABLE

DBCC CHECKCATALOG

DBCC NEWALLOC

DBCC CHECKDB

DBCC TEXTALL

DBCC CHECKFILEGROUP

DBCC TEXTALLOC

DBCC CHECKIDENT

1. **System Stored Procedures.**

SQL Server-supplied, precompiled collection of Transact-SQL statements. System stored procedures are provided as shortcuts for retrieving information from system tables or mechanisms for accomplishing database administration and other tasks that involve updating system tables. The names of all system stored procedures begin with **sp\_**. System stored procedures are located in the **master** database and are owned by the system administrator, but many of them can be run from any database. If a system stored procedure is executed in a database other than **master**, it operates on the system tables in the database from which it is executed.

1. **Can we prefix user defined SP's with sp\_, if so what will happen if u call that SP.**

SQL Server always looks for stored procedures beginning with **sp\_** in this order:

* + 1. Look for the stored procedure in the **master** database first.
    2. Look for the stored procedure based on any qualifiers provided (database name or owner).
    3. Look for the stored procedure using **dbo** as the owner, if one is not specified.

Therefore, although the user-created stored procedure prefixed with **sp\_** may exist in the current database, the **master** database is always checked first, even if the stored procedure is qualified with the database name

**Important** : If any user-created stored procedure has the same name as a system stored procedure, the user-created stored procedure will never be executed

1. **Indexes.**

Microsoft SQL Server index is a structure associated with a table that speeds retrieval of the rows in the table. An index contains keys built from one or more columns in the table. These keys are stored in a structure that allows SQL Server to find the row or rows associated with the key values quickly and efficiently.

If a table is created with no indexes, the data rows are not stored in any particular order. This structure is called a heap.

The two types of SQL Server indexes are:

* Clustered

Clustered indexes sort and store the data rows in the table based on their key values. Because the data rows are stored in sorted order on the clustered index key, clustered indexes are efficient for finding rows. There can only be one clustered index per table, because the data rows themselves can only be sorted in one order. The data rows themselves form the lowest level of the clustered index.

The only time the data rows in a table are stored in sorted order is when the table contains a clustered index. If a table has no clustered index, its data rows are stored in a heap.

* Nonclustered

Nonclustered indexes have a structure that is completely separate from the data rows. The lowest rows of a nonclustered index contain the nonclustered index key values and each key value entry has pointers to the data rows containing the key value. The data rows are not stored in order based on the nonclustered key.

The pointer from an index row in a nonclustered index to a data row is called a row locator. The structure of the row locator depends on whether the data pages are stored in a heap or are clustered. For a heap, a row locator is a pointer to the row. For a table with a clustered index, the row locator is the clustered index key

1. **When we will go for clustered and Non-clustered Indexes.**

**Indexes assist when a query:**

* Searches for rows that match a specific search key value (an exact match query). An exactmatch comparison is one in which the query uses the WHERE statement to specify a column entry with a given value. For example:

WHERE emp\_id = 'VPA30890F'

* Searches for rows with search key values in a range of values (a range query). A range query is one in which the query specifies any entry whose value is between two values. For example:

WHERE job\_lvl BETWEEN 9 and 12

or,

WHERE job\_lvl >= 9 and job\_lvl <= 12

* Searches for rows in a table **T1** that match, based on a join predicate, a row in another table **T2** (an index nested loops join).
* Produces sorted query output without an explicit sort operation, in particular for sorted dynamic cursors.
* Scans rows in a sorted order to permit an order-based operation, such as merge join and stream aggregation, without an explicit sort operation.
* Scans all rows in a table with better performance than a table scan, due to the reduced column set and overall data volume to be scanned (a covering index for the query at hand).
* Searches for duplicates of new search key values in insert and update operations, to enforce PRIMARY KEY and UNIQUE constraints.
* Searches for matching rows between two tables for which a FOREIGN KEY constraint is defined.

Queries using LIKE comparisons can benefit from an index if the pattern starts with a specific character string, for example 'abc%', but not if the pattern starts with a wildcard search, for example '%xyz'.

**Consider using nonclustered indexes for:**

* Columns that contain a high number of distinct values, such as a combination of last name and first name (if a clustered index is used for other columns). If there are very few distinct values, such as only 1 and 0, no index should be created.
* Queries that do not return large result sets.
* Columns frequently involved in search conditions of a query (WHERE clause) that return exact matches.
* Decision Support System applications for which joins and grouping are frequently required. Create multiple nonclustered indexes on columns involved in join and grouping operations, and a clustered index on any foreign key columns.
* Covered queries

**Consider using a clustered index for:**

* Columns that contain a limited number of distinct values, such as a state column that contains only 50 distinct state codes. However, if there are very few distinct values, such as only 1 and 0, no index should be created.
* Queries that return a range of values using operators such as BETWEEN, >, >=, <, and <=.
* Columns that are accessed sequentially.
* Queries that return large result sets.
* Columns that are frequently accessed by queries involving join or GROUP BY clauses; typically these are foreign key columns. An index on the column(s) specified in the ORDER BY or GROUP BY clause eliminates the need for SQL Server to sort the data because the rows are already sorted. This improves query performance.
* OLTP-type applications where very fast single row lookup is required, typically by means of the primary key. Create a clustered index on the primary key.

**Clustered indexes are not a good choice for:**

* Columns that undergo frequent changes because this results in the entire row moving (because SQL Server must keep the row’s data values in physical order). This is an important consideration in high-volume transaction processing systems where data tends to be volatile.
* Covered queries. The more columns within the search key, the greater the chance for the data in the indexed column to change, resulting in additional I/O.

#### **Referential Integrity**

Referential integrity preserves the defined relationships between tables when records are entered or deleted. In SQL Server, referential integrity is based on relationships between foreign keys and primary keys or between foreign keys and unique keys. Referential integrity ensures that key values are consistent across tables. Such consistency requires that there be no references to nonexistent values and that if a key value changes, all references to it change consistently throughout the database.

When you enforce referential integrity, SQL Server prevents users from:

Adding records to a related table if there is no associated record in the primary table.

Changing values in a primary table that result in orphaned records in a related table.

Deleting records from a primary table if there are matching related records.

#### **Features of Normalized and De-normalized Databases.**

#### **Normalizing** a logical database design involves using formal methods to separate the data into multiple, related tables. A greater number of narrow tables (with fewer columns) is characteristic of a normalized database.

#### A few wide tables (with more columns) is characteristic of an **unnormalized** database

Some of the benefits of normalization include:

* Faster sorting and index creation.
* A larger number of clustered indexes.
* Narrower and more compact indexes.
* Fewer indexes per table, which improves the performance of INSERT, UPDATE, and DELETE statements.
* Fewer NULL values and less opportunity for inconsistency, which increase database compactness.

#### **22. When u will go for De-normalization**

To introduce redundancy into a table in order to incorporate data from a related table. The related table can then be eliminated. Denormalization can improve efficiency and performance by reducing complexity in a data warehouse schema.

**23. Union and Union ALL which will give better performance and why?**

The UNION operator allows you to combine the results of two or more SELECT statements into a single result set. The result sets combined using UNION must all have the same structure. They must have the same number of columns, and the corresponding result set columns must have compatible data types.

The result set column names of a UNION are the same as the column names in the result set of the first SELECT statement in the UNION. The result set column names of the other SELECT statements are ignored.

By default, the UNION operator removes duplicate rows from the result set. If you use ALL, all rows are included in the results and duplicates are not removed

UNION ALL is faster because it doesn't try to eliminate the duplicates. In fact, as a "duplicate" is only determined by comparing the whole row (all the columns returned), you should use UNION ALL all the time, except if you WANT to eliminate the duplicates by using UNION

**24. Diff. between drop and detach database.**

**Drop Database:** Removes one or more databases from SQL Server. Removing a database deletes the database and the disk files used by the database.

A database that has been dropped can be re-created only by restoring a backup. You cannot drop a database currently in use (open for reading or writing by any user). Whenever a database is dropped, the **master** database should be backed up .

**Detach Database**

**Detaches a database** from a server and, optionally, runs UPDATE STATISTICS on all tables before detaching.

Detaching a database removes the database from SQL Server, but leaves the database intact within the data and transaction log files that compose the database. These data and transaction log files can then be used to attach the database to any computer running SQL Server, including the server from which the database was detached. This makes the database available in exactly the same state it was in when it was detached.

**25. Instead of Triggers.**

INSTEAD OF triggers are executed instead of the triggering action (for example, INSERT, UPDATE, DELETE). They can also be defined on views, in which case they greatly extend the types of updates a view can support.

The trigger executes in place of the triggering action. INSTEAD OF triggers can be specified on both tables and views. You can define only one INSTEAD OF trigger for each triggering action (INSERT, UPDATE, and DELETE). INSTEAD OF triggers can be used to perform enhance integrity checks on the data values supplied in INSERT and UPDATE statements. INSTEAD OF triggers also let you specify actions that allow views, which would normally not support updates, to be updatable.

**26. Customer Table**  
**Name Phoneno**Abc 123  
Pqr 234  
Rst 235  
Xyz  999

**a. Write a query to find out customers and count of phno with more  
than one phnumber**

Select name, Count(Phoneno) from customer having count(Phoneno)>1

**b.    Add a row pqr  phno 234. Now write a query to get the count of  
phno and name with no duplicate values** Select [Name],phoneno,count(phoneno) as countofPNo from customer group by [Name],phoneno

**27. Explain about DTS**

Data Transformation Services (DTS) provides the functionality to import, export, and transform data between SQL Server and any OLE DB, ODBC, or text file format.

**Using DTS, it is possible to:**

Build data warehouses and data marts in Microsoft SQL Server by importing and transferring data from multiple heterogeneous sources interactively or automatically on a regularly scheduled basis.

Create custom transformation objects that can be integrated into third-party products.

Access applications using third-party OLE DB providers. This allows applications, for which an OLE DB provider exists, to be used as sources and destinations of data.

**DTS also provides support for:**

* High-speed nonlogged inserts (**bcp**) into SQL Server version 7.0.
* Creating customized transformations.
* Transferring complete database objects between source and destination SQL Server 7.0 data sources. The Transfer SQL Server Objects task can be used to transfer all of the metadata and data for some or all of the objects in one SQL Server 7.0 database to another SQL Server 7.0 database. For example, the Transfer SQL Server Objects task can be used to move a table with all of its associated index, constraint, rule, default, and trigger definitions and the existing rows in the table. The Transfer SQL Server Objects task also can be used to transfer the definitions of objects such as views and stored procedures.

**What is OpenRowset**

It includes all connection information necessary to access remote data from an OLE DB data source. This method is an alternative to accessing tables in a linked server and is a one-time, ad hoc method of connecting and accessing remote data using OLE DB. The OPENROWSET function can be referenced in the FROM clause of a query as though it is a table name. The OPENROWSET function can also be referenced as the target table of an INSERT, UPDATE, or DELETE statement, subject to the capabilities of the OLE DB provider. Although the query may return multiple result sets, OPENROWSET returns only the first one.

**Ex:**

SELECT a.\* FROM OPENROWSET('SQLOLEDB','seattle1';'sa';'MyPass',

    'SELECT \* FROM pubs.dbo.authors ORDER BY au\_lname, au\_fname') AS a

### **29. What are stored procedures**

When you use Transact-SQL programs, two methods are available for storing and executing the programs. You can store the programs locally and create applications that send the commands to SQL Server and process the results, or you can store the programs as stored procedures in SQL Server and create applications that execute the stored procedures and process the results.

**Stored procedures in SQL Server are similar to procedures in other programming languages in that they can:**

Accept input parameters and return multiple values in the form of output parameters to the calling procedure or batch.

Contain programming statements that perform operations in the database, including calling other procedures.

Return a status value to a calling procedure or batch to indicate success or failure (and the reason for failure).

You can use the Transact-SQL EXECUTE statement to run a stored procedure. Stored procedures are different from functions in that they do not return values in place of their names and they cannot be used directly in an expression.

**The benefits** of using stored procedures in SQL Server rather than Transact-SQL programs stored locally on client computers are:

**They allow modular programming.**

You can create the procedure once, store it in the database, and call it any number of times in your program. Stored procedures can be created by a person who specializes in database programming, and they can be modified independently of the program source code.

**They allow faster execution.**

If the operation requires a large amount of Transact-SQL code or is performed repetitively, stored procedures can be faster than batches of Transact-SQL code. They are parsed and optimized when they are created, and an in-memory version of the procedure can be used after the procedure is executed the first time. Transact-SQL statements repeatedly sent from the client each time they run are compiled and optimized every time they are executed by SQL Server.

**They can reduce network traffic.**

An operation requiring hundreds of lines of Transact-SQL code can be performed through a single statement that executes the code in a procedure, rather than by sending hundreds of lines of code over the network.

**They can be used as a security mechanism.**

Users can be granted permission to execute a stored procedure even if they do not have permission to execute the procedure’s statements directly.

A SQL Server stored procedure is created with the Transact-SQL CREATE PROCEDURE statement and can be modified with the ALTER PROCEDURE statement. The stored procedure definition contains two primary components: the specification of the procedure name and its parameters, and the body of the procedure, which contains Transact-SQL statements that perform the procedure’s operations

## ***Extended Stored Procedures***

Extended stored procedures allow you to create your own external routines in a programming language such as C. The extended stored procedures appear to users as normal stored procedures and are executed in the same way. Parameters can be passed to extended stored procedures, and they can return results and return status. Extended stored procedures can be used to extend the capabilities of Microsoft® SQL Server™.

Extended stored procedures are dynamic link libraries (DLLs) that SQL Server can dynamically load and execute. Extended stored procedures run directly in the address space of SQL Server and are programmed using the SQL Server Open Data Services API.

After an extended stored procedure has been written, members of the s**ysadmin** fixed server role can register the extended stored procedure with SQL Server and then grant permission to other users to execute the procedure. Extended stored procedures can be added only to the **master** database

**Remote Stored Procedure**

A collection of SQL statements and optional control-of-flow statements stored under a name on a remote server. Remote stored procedures can be called by clients or SQL Server

**Distributed Queries**

Distributed queries access data from multiple heterogeneous data sources, which can be stored in either the same or different computers. SQL Server supports distributed queries by using OLE DB, the Microsoft specification of an application programming interface (API) for universal data access.

Distributed queries provide SQL Server users with access to:

* Distributed data stored in multiple computers that are running SQL Server.
* Heterogeneous data stored in various relational and non-relational data sources that can be accessed using an OLE DB provider.

OLE DB providers expose their data in tabular objects called rowsets. SQL Server version 7.0 allows rowsets from OLE DB providers to be referenced in Transact-SQL statements as if they were a SQL Server table.

### **30.   Name         Tel**

### ABC 100

### DEF 200

### HIG 300

### ABC 100

### DEF 400

### PQR  500

### **Write a query to list all the customers and the no. of telephones they have , only if they**

### **have more than one unique telephone no**

Select Name,count(phoneno) as noofPhoneno from customers group by Name having noofPhoneno>1

### **31. Consider the following table:**

### **Sernum Day Temp.**         1         Mon         10         2         Tue         12         3         Wed         9         4         Thurs         15    Write a query to list a new column with the difference in temp of the days Mon and Tue

### , Tue and Wed and soon. (Don't use cursors)

### Select a.srno,b.srno,a.[day],b.[day],a.[temp],b.[temp], (a.[temp]-b.[temp])AS DiffinTemp from diff a inner join diff b on a.srno=b.srno-1

### **32. Consider the tables**

### **Team Table**

### **TeamId    TeamName**

### 1 Team1

### 2         Team2

### **PlayerTable**

### **PlayId  Name** 1  Name1

### 2  Name2

### **TeamPlayer**

### **TeamId  PlayerId** 1  1 2  1 1  2

### **Write a query to get Team name and Player name**

### Select c.teamname,d.playername from (Select a.teamid,a.teamname,b.playerid from team a inner join teamplayer b on a.teamid=b.teamid) as c inner join player d on c.playerid=d.playerid

### **33. Different ways of getting count of rows from a table in queryanalyser \*\*\*\*\*\*\***

Select distinct count(\*)

### **34. What is sp\_addlinkedserver**

Creates a linked server, which allows access to distributed, heterogeneous queries against OLE DB data sources. After creating a linked server with **sp\_addlinkedserver**, this server can then execute distributed queries. If the linked server is defined as SQL Server, remote stored procedures can be executed.

A linked server configuration allows SQL Server to execute commands against OLE DB data sources on different servers. Linked servers offer these advantages:

Remote server access.

The ability to issue distributed queries, updates, commands, and transactions on heterogeneous data sources across the enterprise.

Freedom from the need to address diverse data sources differently.

### **35 . Types of Locks**

There are three main types of locks that SQL Server 6.5 uses:

Shared locks

Update locks

Exclusive locks

**Shared locks** are used for operations that do not change or update data, such as a SELECT statement.

**Update locks** are used when SQL Server intends to modify a page, and later promotes the update page lock to an exclusive page lock before actually making the changes.

**Exclusive locks** are used for the data modification operations, such as UPDATE, INSERT, or DELETE.

Shared locks are compatible with other Shared locks or Update locks.

Update locks are compatible with Shared locks only.

Exclusive locks are not compatible with other lock types.

Let me to describe it on the real example. There are four processes, which attempt to lock the same page of the same table. These processes start one after another, so Process1 is the first process, Process2 is the second process and so on.

Process1 : SELECT

Process2 : SELECT

Process3 : UPDATE

Process4 : SELECT

Process1 sets the Shared lock on the page, because there are no another locks on this page.

Process2 sets the Shared lock on the page, because Shared locks are compatible with other Shared locks.

Process3 wants to modify data and wants to set Exclusive lock, but it cannot make it before Process1 and Process2 will be finished, because Exclusive lock is not compatible with other lock types. So, Process3 sets Update lock.

Process4 cannot set Shared lock on the page before Process3 will be finished. So, there is no Lock starvation. Lock starvation occurs when read transactions can monopolize a table or page, forcing a write transaction to wait indefinitely. So, Process4 waits before Process3 will be finished.

After Process1 and Process2 were finished, Process3 transfer Update lock into Exclusive lock to modify data. After Process3 was finished, Process4 sets the Shared lock on the page to select data.

**Locking optimizer hints**

There are six Locking optimizer hints in SQL Server 7.0:

NOLOCK

HOLDLOCK

UPDLOCK

TABLOCK

PAGLOCK

TABLOCKX

READCOMMITTED

READUNCOMMITTED

REPEATABLEREAD

SERIALIZABLE

READPAST

ROWLOCK

NOLOCK is also known as "dirty reads". This option directs SQL Server not to issue shared locks and not to honor exclusive locks. So, if this option is specified, it is possible to read an uncommitted transaction. This results in higher concurrency and in lower consistency.

HOLDLOCK directs SQL Server to hold a shared lock until completion of the transaction in which HOLDLOCK is used. You cannot use HOLDLOCK in a SELECT statement that includes the FOR BROWSE option. HOLDLOCK is equivalent to SERIALIZABLE.

UPDLOCK instructs SQL Server to use update locks instead of shared locks while reading a table and holds them until the end of the command or transaction.

TABLOCK takes a shared lock on the table that is held until the end of the command. If you also specify HOLDLOCK, the lock is held until the end of the transaction.

PAGLOCK is used by default. Directs SQL Server to use shared page locks.

TABLOCKX takes an exclusive lock on the table that is held until the end of the command or transaction.

READCOMMITTED

Perform a scan with the same locking semantics as a transaction running at the READ COMMITTED isolation level. By default, SQL Server operates at this isolation level.

READUNCOMMITTED

Equivalent to NOLOCK.

REPEATABLEREAD

Perform a scan with the same locking semantics as a transaction running at the REPEATABLE READ isolation level.

SERIALIZABLE

Perform a scan with the same locking semantics as a transaction running at the SERIALIZABLE isolation level. Equivalent to HOLDLOCK.

READPAST

Skip locked rows. This option causes a transaction to skip over rows locked by other transactions that would ordinarily appear in the result set, rather than block the transaction waiting for the other transactions to release their locks on these rows. The READPAST lock

hint applies only to transactions operating at READ COMMITTED isolation and will read only past row-level locks. Applies only to the SELECT statement.

You can only specify the READPAST lock in the READ COMMITTED or REPEATABLE READ isolation levels.

ROWLOCK

Use row-level locks rather than use the coarser-grained page- and table-level locks.

You can specify one of these locking options in a SELECT statement.

This is the example:

SELECT au\_fname FROM pubs..authors (holdlock)

**Lock Escalation**

You can customize locking by setting Lock Escalation level. The Lock Escalation level determines, when SQL Server applies table locks instead of page locks, and it affects all users of SQL Server. So it's escalation from the page to the table level locking.

There are three Lock Escalation options:

LE threshold maximum

LE threshold minimum

LE threshold percent

**LE threshold maximum** is the maximum number of page locks to hold before escalating to a table lock. The default value is 200.

**LE threshold minimum** is the minimum number of page locks required before escalating to a table lock. The default value is 20.

**LE threshold percent** is the percentage of page locks needed on a table before escalating to a table lock. The default value is 0, it means that a table lock will be occur only when the LE threshold maximum will be exceeded.

You can configure Lock Escalation levels by using the sp\_configure system stored procedure.

This is the example to set LE threshold maximum to 250:

EXEC sp\_configure 'LE threshold maximum', 250

RECONFIGURE WITH OVERRIDE

**Deadlocks**

Deadlock occurs when two users have locks on separate objects and each user wants a lock on the other's object.

For example, User1 has a lock on object "A" and wants a lock on object "B" and User2 has a lock on object "B" and wants a lock on object "A".

You can decide which connection will be the candidate for deadlock victim by using SET DEADLOCK\_PRIORITY. In other case, SQL Server selects the deadlock victim by choosing the process that completes the circular chain of locks.

So, in a multiuser situation, your application should check the message 1205 to indicate that the transaction was rolled back, and if it's so, restart the transaction.

Note. To reduce the chance of a deadlock, you should minimize the size of transactions and transaction times.

### **36 . Why we will denormalize tables for reports\*\*\*\*\*\*\*\*\***

### **37. Instancing of Sqlservers in 2000\*\*\*\*\*\*\*\*\***

### **38. What's a Primary Key?**

### The column or combination of columns that uniquely identifies one row from any other row in a table. A primary key (PK) must be nonnull and must have a unique index. A primary key is commonly used for joins with foreign keys (matching nonprimary keys) in other tables.

Primary Key creates Cluster Index.

### **39. What's a Transaction? Syntax**

A group of database operations combined into a logical unit of work that is either wholly committed or rolled back. A transaction is atomic, consistent, isolated, and durable.

Begin Transaction

Save Transaction

RollBack Transaction

Commit Transaction

### **SaveTransation**

### Sets a savepoint within a transaction.

A user can set a savepoint, or marker, within a transaction. The savepoint defines a location to which a transaction can return if part of the transaction is conditionally canceled. If a transaction is rolled back to a savepoint, it must proceed to completion (with more Transact-SQL statements if needed and a COMMIT TRANSACTION statement), or it must be canceled altogether (by rolling the transaction back to its beginning). To cancel an entire transaction, use the form ROLLBACK TRANSACTION *transaction\_name*. All the statements or procedures of the transaction are undone.

SAVE TRANSACTION is not supported in distributed transactions started either explicitly with BEGIN DISTRIBUTED TRANSACTION or escalated from a local transaction.

**Important** When a transaction begins, resources used during the transaction are held until the completion of the transaction (namely locks). When part of a transaction is rolled back to a savepoint, resources continue to be held until the completion of the transaction (or a rollback of the complete transaction).

**Distributed Transation**

Distributed transactions span two or more servers known as resource managers. The management of the transaction must be coordinated between the resource managers by a server component called a transaction manager. Microsoft® SQL Server™ can operate as a resource manager in distributed transactions coordinated by transaction managers such as the Microsoft Distributed Transaction Coordinator (MS DTC), or other transaction managers that support the X/Open XA specification for Distributed Transaction Processing. For more information, see your Microsoft Distributed Transaction Coordinator documentation.

A transaction within a single SQL Server that spans two or more databases is actually a distributed transaction. SQL Server, however, manages the distributed transaction internally; to the user it operates as a local transaction.

At the application, a distributed transaction is managed much the same as a local transaction. At the end of the transaction, the application requests the transaction to be either committed or rolled back. A distributed commit must be managed differently by the transaction manager to minimize the risk that a network failure may result in some resource managers successfully committing while others roll back the transaction. This is achieved by managing the commit process in two phases (the prepare phase and the commit phase), which is known as a two-phase commit (2PC).

Prepare phase

When the transaction manager receives a commit request, it sends a prepare command to all the resource managers involved in the transaction. Each resource manager then does everything required to make the transaction durable and all buffers holding log images for the transaction are flushed to disk. As each resource manager completes the prepare phase, it returns success or failure of the prepare to the transaction manager.

Commit phase

If the transaction manager receives successful prepares from all the resource managers, it sends commit commands to each resource manager. The resource managers can then complete the commit. If all the resource managers report a successful commit, the transaction manager then sends a success notification to the application. If any resource manager reported a failure to prepare, the transaction manager sends a rollback command to each resource manager and indicates the failure of the commit to the application.

### **Identity and Row Count?**

Identity : Creates an identity column in a table

@@Identity : - Returns the last-inserted identity value

The @@IDENTITYvalue does not revert to a previous setting if theINSERT or SELECT INTO statementor bulk copy fails, or if the transaction is rolled back.

### **RowCount:**

### Returns the number of rows affected by the last statement.

### This variable is set to 0 by any statement that does not return rows, such as an IF statement

### **41.    Error Number when no Error?**

**@@Error**

Returns the error number for the last Transact-SQL statement executed

When SQL Server completes the execution of a Transact-SQL statement, @@ERROR is set to the value 0 if the statement executed successfully. If an error occurs, an error message is returned. @@ERROR returns the number of the error message until another Transact-SQL statement is executed. You can view the text associated with an @@ERROR error number in the **sysmessages** system table.

Because @@ERROR is cleared and reset on each statement executed, check it immediately following the statement being validated or save it to a local variable that can be checked later

### **42.    How to get Error Number and Row Count together?**

### SELECT @rowcount = @@ROWCOUNT,@error = @@ERROR

### **43. Executing a SQL Statement with Table Name coming in as Parameter?**

### declare @sql varchar(5000)

### Set @sql = 'Select \* from ' + @name

### execute( @sql)

### **44.    Different Tables in a Trigger?**

Two special tables are used in trigger statements: the **deleted** table and the **inserted** table. You can use these temporary tables to test the effects of certain data modifications and to set conditions for trigger actions. You cannot alter the data in the trigger test tables directly, but you can use the tables in SELECT statements to determine whether the trigger was fired by an INSERT, UPDATE, or DELETE statement.

The **deleted** table stores copies of the affected rows during DELETE and UPDATE statements. During the execution of a DELETE or UPDATE statement, rows are deleted from the trigger table and transferred to the **deleted** table. The **deleted** table and the trigger table ordinarily have no rows in common.

The **inserted** table stores copies of the affected rows during INSERT and UPDATE statements. During an INSERT or UPDATE transaction, new rows are added simultaneously to both the **inserted** table and the trigger table. The rows in the **inserted** table are copies of the new rows in the trigger table.

An UPDATE transaction is like a delete followed by an insert; the old rows are copied to the **deleted** table first, and then the new rows are copied to the trigger table and to the **inserted** table.

When you set trigger conditions, use the **inserted** and **deleted** tables appropriately for the action that fired the trigger. Although referencing **deleted** while testing an INSERT, or **inserted** while testing a DELETE does not cause any errors, these trigger test tables will not contain any rows in these cases.

**Note** A given trigger fires only once per statement. If trigger actions depend on the number of rows a data modification effects, use tests (such as an examination of @@ROWCOUNT) for multirow data modifications (an INSERT, DELETE, or UPDATE based on a SELECT statement), and take appropriate actions.

### **45.    What are Temporary Tables?**

Temporary tables are similar to permanent tables, except temporary tables are stored in **tempdb** and are deleted automatically when no longer in use.

The two types of temporary tables, local and global, differ from each other in their names, their visibility, and their lifetimes. Local temporary tables have a single number sign (#) as the first character of their names; they are visible only to the current connection for the user; and they are deleted when the user disconnects from computers running SQL Server. Global temporary tables have two number signs (##) as the first characters of their names; they are visible to any user after they are created; and they are deleted when all users referencing the table disconnect from SQL Server.

### **46. There is a Table With 3 Columns. 1. EmpID 2. EmpName 3.ManagerID Write a Query to Get all the Employee for each Managers.**

select b.empname,a.empname from emp a inner join emp b on a.manid=b.empid order by a.manid

### **47.    A table with 2 Primary Key Fields(P1,P2).**

### **P1    P2**

### 1    1

### 1    2

### 1    3

### 2    1

### 2    2

### 2    3

### **Write a query to select Max of P2 for each P1.**

### select p1,max(p2) from p1p2 group by p1

### **48.    Performance Tuning of a SQL SERVER?**

### **49. Explain Third normalization form with an example?**

### **50. Why there is a performance difference between two similar queries that uses UNION and UNION ALL?**

### The UNION operator allows you to combine the results of two or more SELECT statements into a single result set. The result sets combined using UNION must all have the same structure. They must have the same number of columns, and the corresponding result set columns must have compatible data types

### **Guidelines for Using UNION**

Follow these guidelines when using UNION operators:

* All select lists in the statements being combined with UNION must have the same number of expressions (column names, arithmetic expressions, aggregate functions, and so on).
* Corresponding columns in the result sets being combined with UNION, or any subset of columns used in individual queries, must be of the same data type, have an implicit data conversion possible between the two data types, or have an explicit conversion supplied. For example, UNION is not possible between a column of **datetime** data type and one of **binary** data type unless an explicit conversion is supplied, while UNION is possible between a column of **money** data type and one of **int** data type because they can be implicitly converted.
* Corresponding result set columns in the individual statements being combined with UNION must occur in the same order, because UNION compares the columns one-to-one in the order given in the individual queries.

By default, the UNION operator removes duplicate rows from the result set. If you use ALL, all rows are included in the results and duplicates are not removed

### **51 . Write a SQL Query to find first day of month?**

Select Cast('1-' + cast(DatePart(mm, getdate()) as varchar) + '-' + cast(DatePart(yy, getdate()) as varchar) as smalldatetime)

Select Dateadd(d,1,Dateadd(dd,-(DatePart(dd,GetDate())),Getdate()))

### **52. A user is a member of Public role and Sales role. Public role has the permission to**

### **select on all the table, and Sales role, which doesnt have a select permission on some of**

### **the tables. Will that user be able to select from all tables ? \*\*\*\*\*\*\*\***

### **NO 53. If a user doesnt have a permission on a table, but he has permission to a view created on it, will he be able to view the data in table? \*\*\*\*\*\*\*\***

### **YES 54. Describe Application Role and explain a scenario when you will use it?**

You may want users to be restricted to accessing data only through a specific application without the ability to access data directly, for example using SQL Server Query Analyzer or Microsoft Excel. This prevents a user from connecting to SQL Server using an application such as SQL Server Query Analyzer and executing a poorly written query, which affects the performance of the whole server.

SQL Server accommodates this situation through the use of application roles. The fundamental differences between standard and application roles are:

* Application roles contain no members. Users, Microsoft Windows NT® groups, and roles cannot be added to application roles; the permissions of the application role are gained when the application role is activated for the user’s connection through a specific application(s). A user’s association with an application role is due to being capable of running an application that activates the role, rather than being a member of the role.
* Application roles are inactive by default and require a password to be activated by using the **sp\_setapprole** system stored procedure. The password can be provided by the user, for example, through an application prompt, but it is more likely that the password is incorporated within the application. The password can be encrypted as it is sent to SQL Server.
* When an application role is activated for a connection by the application, the connection permanently loses all permissions applied to the login, user account, or other groups or database roles in all databases for the duration of the connection. The connection gains the permissions associated with the application role for the database in which the application role exists. Because application roles are applicable only to the database in which they exist, the connection can gain access to another database only by virtue of permissions granted to the **guest** user account in the other database. Therefore, if the **guest** user account does not exist in a database, the connection cannot gain access to that database. If the guest user account does exist in the database but permissions to access an object are not explicitly granted to **guest**, the connection cannot access that object regardless of who created the object. The permissions the user gained from the application role remain in effect until the connection logs out of SQL Server.

**Important** It is necessary for a connection to lose default permissions applied to the login/user account or other groups or database roles in all databases for the duration of the connection and gain the permissions associated with the application role to ensure that all the functions of the application can be performed. For example, if a user is usually denied access to a table that the application must access, then the denied access should be revoked for the user to successfully use the application. Application roles overcome any conflicts with user’s default permissions by temporarily suspending the user’s default permissions and assigning them only the permissions of the application role.

Application roles allow the application to take over the responsibility of user authentication, rather than SQL Server. However, because SQL Server still needs to authenticate the application when it accesses databases, the application must provide a password because there is no other way to authenticate an application.

If ad hoc access to a database is not required, users and Windows NT groups do not need to be granted any permissions because all permissions can be assigned by the applications they use to access the database. In such an environment, assuming access to the applications is secure, standardizing on one system-wide password assigned to an application role is possible.

There are several options for managing application role passwords without hard-coding them into applications. For example, an encrypted key stored in the registry (or the SQL Server database), for which only the application has the decryption code, can be used. The application reads the key, decrypts it, and uses the value to set the application role. Using the Multiprotocol Net-Library, the network packet containing the password can also be encrypted. Additionally, the password can be encrypted, before being sent to SQL Server, when the role is activated.

When an application user connects to SQL Server using Windows NT Authentication Mode, an application role is a good way to set the permissions the Windows NT user has in a database when using the application. This allows Windows NT auditing of the user account and control over user permissions, while they use the application, to be easily maintained.

If SQL Server Authentication is used and auditing user access in the database is not required, it can be easier for the application to connect to SQL Server using a predefined SQL Server login. For example, an order entry application authenticates users running the application itself, and then connects to SQL Server using the same **OrderEntry** login. All connections use the same login, and relevant permissions are granted to this login.

**Note** Application roles work with both authentication modes.

### **55. Both a UNIQUE constraint and a PRIMARY KEY constraint enforce uniqueness, so when you should use UNIQUE Constraint?**

Use a UNIQUE constraint instead of a PRIMARY KEY constraint when you want to enforce the uniqueness of:

* A column, or combination of columns, that is not the primary key.

Multiple UNIQUE constraints can be defined on a table, whereas only one PRIMARY KEY constraint can be defined on a table.

* A column that allows null values.

UNIQUE constraints can be defined on columns that allow null values, whereas PRIMARY KEY constraints can be defined only on columns that do not allow null values.

A UNIQUE constraint can also be referenced by a FOREIGN KEY constraint.

### **56. What is the difference between the REPEATABLE READ and SERIALIZE isolation levels?**

### **57. You have several tables, and they are joined together for querying. They have clustered indexes and non clustered indexes. To optimize the performance how you will**

### **distribute the tables and their indexes on different file Groups?**

For VLDB, tables and their related indexes should be separated onto separate files and physical disks for optimum performance running queries, but not separate filegroups. If they are in separate filegroups, then you cannot back up and restore them as a single unit.

If one of your join queries is used much more often than others, place the tables used in this query in different filegroups on different physical disk arrays.  
  
**58. Which event (Check constraints, Foreign Key, Rule, trigger, Primary   
key check) will be performed last for integrity check?**

**59. After removing a table from database, what other related objects have to be dropped explicitly?**

Removes a table definition and all data, indexes, triggers, constraints, and permission specifications for that table. Any view or stored procedure that references the dropped table must be explicitly dropped by using the DROP VIEW or DROP PROCEDURE  
  
**60. How can you get an identity value inside a trigger, there are different ways to get identity value, explain why you will prefer one on another.**

### **62. How to find out which stored procedure is recompiling?**

### **63. How to stop stored procedures from recompiling?**

### **64. When should one use instead of Trigger?**

### **65. What is a derived table?**

### A derived table is a SELECT statement used as a table. It is always enclosed by parenthesis and follows a FROM or JOIN keyword

### **66. When you should use low fill factor?**

### If the table is going to have many updates and inserts

### **67. How you can minimize the deadlock situation?**

* Access objects in the same order.
* Avoid user interaction in transactions.
* Keep transactions short and in one batch.
* Use as low an isolation level as possible.
* Use bound connections

### **69. What are User Defined Types? and how do you use them?**

### **70. How do u trigger a COM from SQL?**

### Using extended stored Procedures

### **75. What is an execution plan?**

### **What is SQL Trace?**

### **Difference between union and join**

### **8.    How do u find if there was error when SQL statement was executed?**

### **9.    How do find number of records fetched when SQL statement was executed?**

### **14.    What is SET NO COUNT?**

### **16.    When a single update statement is executed, do we need to put it in transaction?**

### **18.    What is maximum number of columns can we create in a table?**

### **22.    Write a query to update all the names that has spaces in front of their names (above table).**

### **24.    You have the following table**

### **Table: Staff**

### **Columns     Id (this reference to the empid column in the EMP table) LastName     FirstName**

### **Write a query to get all the empnames that are not there in Staff table.**

### **27.    what is the difference between DELETE and TRUNCATE?**

# **Typical Queries**

**1) For Sequence Number**

select col2 , (Select count(\*) from table a where a.col2<=b.col2) as rownumber from tablea b

**SQL Server**

1. **What was the size of the database you worked on in terms of MB’s?**

**5 GB**

1. **What is Primary Key? Can we have primary key on multiple columns?**

A column or set of columns that uniquely identify all the rows in a table. Primary keys do not allow null values. No two rows can have the same primary key value; therefore, a primary key value always uniquely identifies a single row. More than one key can uniquely identify rows in a table, each of these keys is called a candidate key. Only one candidate can be chosen as the primary key of a table; all other candidate keys are known as alternate keys.

1. **What is foreign key?**

The column or combination of columns whose values match the primary key (PK) or unique key in the same or another table. Also called the referencing key.

1. **What is identity column?**

A column in a table that has been assigned the identity property. The identity property generates unique numbers.

1. **What is a join? What is outer join?**
2. **What is Unicode in SQL Server?**

Unicode defines a set of letters, numbers, and symbols that SQL Server recognizes in the **nchar** , **nvarchar** , and **ntext** data types. Unicode has more than 65,000 possible values compared to a character set's 256, and takes twice as much space to store. Unicode includes characters for most languages

1. **What is page size?**

8 KB

1. **How does u find if there was error when SQL statement was executed?**

When Microsoft® SQL Server™ completes the execution of a Transact-SQL statement, @@ERROR is set to 0 if the statement executed successfully. If an error occurs, an error message is returned. @@ERROR returns the number of the error message until another Transact-SQL statement is executed. You can view the text associated with an @@ERROR error number in the **sysmessages** system table.Because @@ERROR is cleared and reset on each statement executed, check it immediately following the statement validated, or save it to a local variable that can be checked later.

1. **How do find number of records fetched when SQL statement was executed?**
2. **What are the virtual tables did u use in the triggers?**
3. **What is temp table and how do u create it?**

###### **Temporary Tables**

A table placed in the temporary database, **tempdb** , and erased at the end of the session.

You can create local and global temporary tables. Local temporary tables are visible only in the current session; global temporary tables are visible to all sessions.

Prefix local temporary table names with single number sign (#table\_name), and prefix global temporary table names with a double number sign (##table\_name).

SQL statements reference the temporary table using the value specified for table\_name in the CREATE TABLE statement:

CREATE TABLE #MyTempTable (cola INT PRIMARY KEY)

INSERT INTO #MyTempTable VALUES (1)

If a local temporary table is created in a stored procedure or application that can be executed at the same time by several users, SQL Server has to be able to distinguish the tables created by the different users. SQL Server does this by internally appending a numeric suffix to each local temporary table name. The full name of a temporary table as stored in the sysobjects table in tempdb consists of table name specified in the CREATE TABLE statement and the system-generated numeric suffix. To allow for the suffix, table\_name specified for a local temporary name cannot exceed 116 characters.

Temporary tables are automatically dropped when they go out of scope, unless explicitly dropped using DROP TABLE:

* A local temporary table created in a stored procedure is dropped automatically when the stored procedure completes. The table can be referenced by any nested stored procedures executed by the stored procedure that created the table. The table cannot be referenced by the process which called the stored procedure that created the table.
* All other local temporary tables are dropped automatically at the end of the current session.
* Global temporary tables are automatically dropped when the session that created the table ends and all other tasks have stopped referencing them. The association between a task and a table is maintained only for the life of a single Transact-SQL statement. This means that a global temporary table is dropped at the completion of the last Transact-SQL statement that was actively referencing the table when the creating session ended.

A local temporary table created within a stored procedure or trigger is distinct from a temporary table with the same name created before the stored procedure or trigger is called. If a query references a temporary table, and two temporary tables with the same name exist at that time, it is not defined which table the query is resolved against. Nested stored procedures can also create temporary tables with the same name as a temporary table created by the stored procedure that called it. All references to the table name in the nested stored procedure are resolved to the table created in the nested procedure, for example:

CREATE PROCEDURE Test2

AS

CREATE TABLE #t(x INT PRIMARY KEY)

INSERT INTO #t VALUES (2)

SELECT Test2Col = x FROM #t

GO

CREATE PROCEDURE Test1

AS

CREATE TABLE #t(x INT PRIMARY KEY)

INSERT INTO #t VALUES (1)

SELECT Test1Col = x FROM #t

EXEC Test2

GO

CREATE TABLE #t(x INT PRIMARY KEY)

INSERT INTO #t VALUES (99)

GO

EXEC Test1

GO

Here is the result set:

(1 row(s) affected)

Test1Col

-----------

1

(1 row(s) affected)

Test2Col

-----------

2

When you create local or global temporary tables, the CREATE TABLE syntax supports constraint definitions with the exception of FOREIGN KEY constraints. If a FOREIGN KEY constraint is specified in a temporary table, the statement returns a warning message indicating that the constraint was skipped, and the table is still created without the FOREIGN KEY constraints. Temporary tables cannot be referenced in FOREIGN KEY constraints.

Consider using table variables instead of temporary tables. Temporary tables are useful in cases when indexes need to be created explicitly on them, or when the table values need to be visible across multiple stored procedures or functions. In general, table variables contribute to more efficient query processing.

1. **What is clustered index? How many clustered index can be created in a table? Can we create clustered index on the column other than primary key column?**
2. **How can u execute SQL statement where table name is passed as variable?**
3. **What is SET NO COUNT?**

When SET NOCOUNT is ON, the count (indicating the number of rows affected by a Transact-SQL statement) is not returned. When SET NOCOUNT is OFF, the count is returned.

The @@ROWCOUNT function is updated even when SET NOCOUNT is ON.

SET NOCOUNT ON eliminates the sending of DONE\_IN\_PROC messages to the client for each statement in a stored procedure. When using the utilities provided with Microsoft® SQL Server™ to execute queries, the results prevent "nn rows affected" from being displayed at the end Transact-SQL statements such as SELECT, INSERT, UPDATE, and DELETE.

1. **What are transactions?**

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the ACID (Atomicity, Consistency, Isolation, and Durability) properties, to qualify as a transaction:

Atomicity

A transaction must be an atomic unit of work; either all of its data modifications are performed, or none of them is performed.

Consistency

When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.

Isolation

Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either sees data in the state it was in before another concurrent transaction modified it, or it sees the data after the second transaction has completed, but it does not see an intermediate state. This is referred to as serializability because it results in the ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

Durability

After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

1. When a single update statement is executed, do we need to put it in transaction?
2. **What is maximum size of varchar data type?**

Variable-length Unicode character data of *n* characters. *n* must be a value from 1 through 4,000. Storage size, in bytes, is two times the number of characters entered. The data entered can be 0 characters in length.

1. **What is maximum number of columns can we create in a table?**
   1. 1024
2. **What is BCP?**
3. **What is OSQL?**
4. **Write a query to fetch all the managers names with the employee names in the order of manager name.**

**Table : EMP**

**Columns**

**EmpId (Primary Key)**

**EmpName (varchar)**

**MgrId**

1. **Write a query to update all the names that has spaces in front of their names (above table).**
2. **Write a query to delete all the names that starts with J.**
3. **You have the following table**

**Table: Staff**

**Columns**

**Id (this reference to the empid column in the EMP table)**

**LastName**

**FirstName**

**Write a query to get all the empnames that are not there in Staff table.**

1. **You have the following table**

**Table : Test**

**Columns**

**Pk1**

**Pk2**

**Write a query to get all the pk1 with maximum value of corresponding pk2 value.**

1. **What is code review in database? How will do the code review for lots of Stored Procedure in an hour?**
2. **What is the difference between DELETE and TRUNCATE?**

DELETE TABLE is a logged operation, so the deletion of each row gets logged in the transaction log, which makes it slow. TRUNCATE TABLE also deletes all the rows in a table, but it won't log the deletion of each row, instead it logs the deallocation of the data pages of the table, which makes it faster. Of course, TRUNCATE TABLE can be rolled back.

1. **How do you increase the performance of your database?**
2. **What will u do if the query is taking too long to execute?**