SQL SERVER TRAINING

Gerhard Beck

1. General

This course was put together in 1997 for SQL Server version 6.5 and has not been updated since then. I put it together both for team training and to bring myself up to date from SQL Server version 4.2.

The purpose of this course is to give you a general familiarity with SQL Server. The course is not designed to teach you how to do anything with SQL Server. At the end of the course, you should have enough knowledge to know when you should look for a specific type of functionality with SQL Server (and maybe even have some idea where to look in the index). This course provides an overview of the SQL Server forest.

1.1. Databases and Devices

A SQL Server can support multiple databases per server. A SQL Server database can have many tables. You can do queries across multiple tables on multiple databases on a server. SQL Server databases are effectively a logical grouping of tables for administrative purposes. All tables in all databases are available to all users (with appropriate permissions) for doing cross table joins.

SQL Server does not create any DOS files for databases. Instead it uses “devices”. Devices are DOS files which SQL Server has created and formatted to hold databases. Think of a device as an actual hardware device. Ideally, a database’s data file and log files should reside on different devices (preferably in both the physical and SQL Server sense of the word). Devices can hold multiple databases and databases can span devices. Because of this, there is no direct connection between physical data storage and a database in SQL Server.

When you create a device you must specify its size. Devices can be made bigger, but they can not be made smaller. There are only a limited number of devices available. If you do not specify a device when you create a database, the database is created on the default device. Initially, MASTER is set as the default device. You should change the default device to some other device (after first creating other devices).

SQL Server divides devices into allocation units (0.5MB), extents (16K) and pages (2K). The pages can become fragmented. Use SHOWCONTIG to check the level of fragmentation and CREATE INDEX with SORTED_DATA_REOG to defragment the pages.

To summarize, before creating a database, you need a device with sufficient space for the database and its log file. Ideally, you will be able to put the database and log files on separate devices.

System Databases & Stored Procedures

SQL Server stores information about itself in four databases:

master	defines all user databases
model	copied when creating new user databases
tempdb	used for doing sorts and other processing
msdb	used to schedule administrative actions

These are regular SQL databases which you can use for your own purposes. For example, the DMT uses the master database to get database definitions. Adding data types to the model database will cause them to appear in all subsequently created databases. You can use tempdb to store temporary tables.

The system databases are on three devices: MASTER, MSDBdata and MSDBlog.

All system tables start with **sys**. There are 13 system tables to control all databases (the system catalogue in the master database) and 18 system tables stored in each database (the database catalog). The system catalogue (the 13 system tables) describe the SQL Server environment (*e.g.* its devices) and its databases. The database catalog (the 18 system table stored in each database) contains information which is specific to a particular database - - for example, its tables and columns. Treat all system tables as READ-ONLY.

A sample user database named pubs is loaded. You can try attaching to pubs to test connections and programming environments.

There are a set of system stored procedures which can provide useful information from a SQL command line. They all start with sp_. Some of the most useful are:

sp_help [objname] - - reports information about the object
sp_helpdb [dbname] - - reports information about the database
sp_helpindex [tablename] - - reports information about the indexes on a table
sp_spaceused [objname] - - figure this one out yourself
sp_columns [tablename]
sp_datatype_info

1.2. Miscellaneous

SQL Server has two types of administrators: the System Administrator and the Database Administrator. The System Administrator is responsible for allocating space among the databases and doing backups. The Database Administrator is responsible for creating the database definition (table and columns) and user permissions for a particular database. Note that the System Administrator can do anything the Database Administrator can do; in contrast, the Database Administrator can only do certain actions with respect to a particular database.

Databases contain the following objects:

Object	Description
Table	Where data is actually stored
Index	Speeds lookups; can enforce rules
View	See supersets or subsets of data
Stored Procedure	Program; Precompiled collection of SQL statements
Trigger	Automatic stored procedure
Constraint	Restricts values which can be entered to maintain referential integrity. Defined at the either the table or the column level.
Default	Value SQL Server puts in a column if the user does not enter a value
Rule	Controls what data can be entered into a table. Enforces business rules.
Data Type	Type of data the column holds

All databases have a transaction log. Transaction logs are where data is stored until the transaction is final. Transaction logs are used if the database needs to roll back the transaction. Transaction logs should be placed on a separate device from the database. If you do not specify where the transaction log is to be stored, it will be stored in the database itself! The size of the log file should be 10 to 25 percent of the size of the database. (You specify the size of the database and of the log file when you create the database.)

Transaction logs can get full. If a log file gets full, you must clear it by dumping it. When you clear the log, you should also backup the database.

It is important to document the database creation sequence. This information will be needed to do restores. Restores assume the machine is configured the same way it was when the backup was made. Thus, if you have a media failure, you must set the machine up exactly as it was when the backup was made to do the restore. If you did not record the setup, it will be impossible to do a restore.

SQL Server Limits: 2 billion tables per database, 250 columns per table, 1962 bytes per row (not including image & text).

You can create user defined data types which can then be used as columns.

2. Syntax

SQL Server version 6.5 is now compliant with the ANSI SQL-92 standards.[1] As you might guess, it also means that SQL Server was not compliant prior to version 6.5. To maintain backwards compatibility, SQL Server provides for both the old and the new syntax in a variety of areas. Much to the chagrin of us old-timers, this course will only teach the ANSI SQL-92 syntax. Not only will the ANSI SQL-92 knowledge be more portable, it reads better and you can do more with it.[2]

2.1. General

When at the SQL command prompt, you can select the database to use with:
USE DBName

2.1.1. SELECT

SELECT can be used to return literals, system values and variables. For example SELECT 'ABC' and SELECT @@VERSION.

Abbreviated SELECT syntax:

SELECT [ALL|DISTINCT] select_list
FROM table_view_list
WHERE clause
ORDER BY clause

select_list is a comma separate lists of columns. If a column is in more than one table, use "table.column". "table.column" can be always used. To select all columns, use * instead of the list. Can use column_name 'lable1', column_name 'label2' or column_heading = column_name to change the heading for a column.

Tables can be either a table name or "database.owner.tablename".

Things that can be in a WHERE clause:

Comparison	= > < >= <= <> ()
Ranges	BETWEEN, NOT BETWEEN
List (sub-select)	IN, NOT IN
String matches (wild cards)	LIKE, NOT LIKE
Unknown values	IS NULL, IS NOT NULL
Combinations	AND, OR
Negations	NOT

[1] SQL Server version 6.5 has also been certified as compliant with FIPS 127-2, in case you were wondering.

[2] So out with the old and in with the new. Let's Party! (I know footnotes are supposed to contain references and illustrative material, but I like to do things just a little differently.)

Wild Card	Description
%	Any string of zero or more characters
_ (underscore)	Any single character
[]	Any character within the specified range
[^]	Any single character not within the specified range

Use the ESCAPE keyword if you need to search for a wild card.

Order by can use number of the column in the select list. (ORDER BY 1 instead of ORDER BY some_col)

Within a SELECT statement, you can do math and invoke math, string functions, date/time and system functions.

You can use the UNION operator to combine two separate select statements.

2.1.2. INSERT

Abbreviated INSERT syntax:

INSERT tablename (col1, col2, col3) VALUES (‘a’, ‘b’, ‘c’)

If the order of the values is identical to the order of the columns and there is a value for each column, you do not need the column list. However, if someone drops and recreates the table with the columns in a different order, you code will no longer work. Therefor, you should always code the column list.

Normally, when you do an INSERT statement you have to list the values to be inserted. Instead, you can substitute DEVAULT VALUES for the values list and create a new row with all default values (of course, all non-null columns will need to be defined with a default for this to work).

You can also get the values to insert from a SELECT statement. This is vary useful for moving data from table to table.

2.1.3. UPDATE

Abbreviated UPDATEsyntax:

```
UPDATE table_name
SET col1 = ‘a’, col2 = ‘b’, col3 = col2
WHERE key_col = ‘abc’
```

If you forget a WHERE clause on an UPDATE statement, ALL ROWS will be updated. I have known this to happen ON PRODUCTION DATA. So be careful! The safe thing to do is to do a select with the where clause to make sure you are getting the correct data before applying a WHERE clause.

2.1.4. DELETE

Abbreviated DELETE syntax:

```
DELETE table_name
WHERE key_col = 'abc'
```

If you forget a WHERE clause on an DELETE statement, ALL ROWS will be deleted. If you really want to delete all rows, use: TRUNCATE TABLE tablename. Its faster.

2.2. Advanced Selection

You can use the following aggregate functions:
AVG(ALL/DISTINCT col)
COUNT(ALL/DISTINCT col)
COUNT(*)
MAX(col)
MIN(col)
SUM(ALL/DISTINCT col)

You can group the summary information using GROUP BY and HAVING clauses.

Example:

```
SELECT title_id, copies_sold = SUM(qty)
FROM sales
GROUP BY title_id
HAVING SUM(qty) > 30
```

title_id	copies_sold
BU2075	35
MC3021	40
PC8888	50
PS2091	108
TC3218	40

You can produce break point summary reports using COMPUTE BY. For example:

```
SELECT type, price
FROM titles
WHERE type LIKE '%cook'
ORDER BY type, price
COMPUTE AVG(price) BY type
```

type	price
mod_cook	2.99
mod_cook	19.99
	avg
	==========
	11.49

type	price
trad_cook	11.95
trad_cook	14.99
trad_cook	20.95
	avg
	==========
	15.96

Microsoft recommends that if a column allows nulls and will be used for financial totals, use ISNULL(col_name,0) to make sure a 0 is substituted for null. This will be especially important for average processing. ***If you need to do counts and averages on a column and do not need to know if it has nulls, use a not null column with 0 as a default.***

For even more complicated summary reports, try ROLLUP and CUBE.

2.3. Joins

As you discuss normalization and other crucial database design topics with users (especially managers), you will eventually be confronted with "I just want my @#*% data!" Normalized tables may be great for doing data entry, they are terrible for doing data analysis. Users just want a big spreadsheet with all of their data to analyze. To give users what the want, you have to de-normalize the data! Well, we could de-normalize the database design and ruin data entry. We could create query optimized (*i.e.* de-normalized) data warehouse tables every night. Or we could do the obvious thing and just join the tables before presenting a result set to the user. The point is that having to do joins is the penalty we pay for having a normalized database. (Being able to do joins which were not planned before the database was implemented is a major benefit of a normalized database.)

There are lots of types of joins. The major joins are:

INNER JOIN Include only rows from both tables which satisfy the join condition. That is, if you are joining the tables on name, only the names which are in both tables will appear.

OUTER JOIN Include all rows from one (or both) tables and rows from the other table which match the join condition. If you are joining the tables on name, print all the names from one table with information from the other table where available.

CROSS JOIN Return every possible combination of rows between the two tables. Your result set will be the product of the rows in the two tables. That is, if one table has 50

rows and the other table has 100 rows, you will get 5,000 rows back! Cross joins are also called “unrestricted joins”.[3]

An INNER JOIN can be either an “equijoin” or a “natural join”.[4] If you do a SELECT *, you are doing an equijoin and the join column will appear twice in the result set. If you use a list of columns in the SELECT, you are doing a natural join and the join column will appear only once (assuming you only list it once).

OUTER JOINs come in three flavors: LEFT OUTER JOIN, RIGHT OUTER JOIN and FULL OUTER JOIN. (You actually have to specify LEFT, RIGHT or FULL.) LEFT, RIGHT or FULL relates to which table you select all the rows from even if the join conditions are not met. LEFT uses all rows from the table on the left, RIGHT (predictably) uses all rows from the table on the right and FULL (get this) uses all rows from both tables. Note that you can only do an OUTER JOIN between two tables.

Here’s some samples:

```
SELECT *
FROM a_table
INNER JOIN b_table ON a_table.a_key = b_table.a_key
WHERE <some where clause goes here>
```

```
SELECT a_table.a_key, a_name, b_info
FROM a_table
INNER JOIN b_table ON a_table.a_key = b_table.a_key
WHERE <some where clause goes here>
```

```
SELECT a_table.a_key, a_name, b_info
FROM a_table
CROSS JOIN b_table
WHERE <some where clause goes here>
```

```
SELECT a_table.a_key, a_name, b_info
FROM a_table
LEFT OUTER JOIN b_table ON a_table.a_key = b_table.a_key
WHERE <some where clause goes here>
```

```
SELECT a_table.a_key, a_name, b_info
FROM a_table
RIGHT OUTER JOIN b_table ON a_table.a_key = b_table.a_key
WHERE <some where clause goes here>
```

[3] No, cross joins are not related to cross dressing, but they appear to be about as useful. If anyone has found a need to use a cross join, let me know.

[4] These terms are not used in the SQL syntax and are used mainly to impress weak-minded friends and bore dates.

```
SELECT a_table.a_key, a_name, b_info
FROM a_table
FULL OUTER JOIN b_table ON a_table.a_key = b_table.a_key
WHERE <some where clause goes here>
```

When creating selects, you can use aliases for table names (usually to reduce typing). For example:

```
SELECT a.a_key, a.a_name, b.b_info
FROM a_table a
INNER JOIN b_table b ON a.a_key = b.a_key
WHERE <some where clause goes here>
```

With aliases, you can join a table to itself.[5]

2.4. Subquery (aka Sub-Selects)

A subquery is a select statement within a select statement, typically in the where clause. Skipping all of the fine points, to create a subquery put a query in parenthesis. For example:

```
SELECT a_name,
FROM a_table
WHERE a_key IN (SELECT a_key FROM another_table WHERE a = b)
```

Ttypes of subqueries:
WHERE a_column = <> > >= < <= !> !< (if you just return one)
WHERE a_column IN or NOT IN (if you have a list)
WHERE a_column = … ANY/ALL (if you have a list)
WHERE EXISTS or NOT EXISTS (SELECT * FROM x WHERE …)

Subqueries can be nested (infinately). Subqueries can be alliased to create the equivalent of temporary tables. For example, table b is the results of a subquery:

```
SELECT a.col_1, b.col_2
FROM a_table a, (SELECT AVG(price) AS col_2 FROM b_table) b
WHERE …
```

A correlated (repeating) subquery uses the outer query for its values and recomputes each row. For example (look at the subquery where clause):

```
SELECT x, y FROM table ta WHERE y =
(SELECT MAX(y) FROM table WHERE table.x = ta.x)
```

[5] I know what your are thinking and you can just keep that your self.

2.5. Update & Delete Based on Other Tables

```
UPDATE titles
SET ytd_sales = 0

UPDATE title
SET ytd_sales =
        (SELECT SUM(qty)
        FROM sales
        WHERE sales.title_id = titles.title_id)
```

The sub-select will be performed for each row thereby updating the year to date sales for each title. The sub-select must return only one value. You can delete rows with the same technique.

3. Data Integrity

There are lots of integrity types:

Entity Integrity All rows must have a unique identifier - the primary key.

Domain Integrity Values that are valid for a given column.

Referential Integrity Preserves defined relationships between tables as records are modified. Based on a table with a primary key (referenced table) and a table with a foreign key (referencing table). Rows (& keys) in the referenced table can not be deleted/changed if there are any rows using that foreign key in the referencing table. Rows can not added to the referencing table unless the foreign key is either null or in the referenced table.

User-Defined Integrity Business rules.

Integrity can be implemented by either declarative referential integrity (defined in the data structure) or by procedural referential integrity (defined in code such as in triggered stored procedures).

SQL Server provides the following to implement integrity rules:
Constraints
Identity
Default
Indexes
Datatypes
Triggers
Stored Procedures

3.1. Constraints

All constraints are created with tables (CREATE TABLE or ALTER TABLE). A column can have multiple constraints and a constraint can span multiple columns. You can drop, defer (make the constraint apply only for new data), disable and enable constraints.

Constraint Type	Enforces	Description
PRIMARY KEY	Entity	Creates a unique index and enforce not-null.
UNIQUE	Entity	Creates a unique index. One null is allowed.
FOREIGN KEY/ REFERENCES	Referential	Defines column(s) whose values must match the primary key of another table
CHECK	Domain	Specifies allowable data values
DEFAULT	Domain	Value to be used on insert if none is provided

Column constraints can include one UNIQUE or FOREIGN KEY constraint, one DEFAULT constraint and any number of CHECK constraints.

Table constraints can include one PRIMARY KEY constraint per table, one DEFAULT constraint per column, and any number of FOREIGN KEY, UNIQUE or CHECK constraints on any column or columns.

PRIMARY KEY and UNIQUE constraints can use CLUSTERED, NONCLUSTERED and FILLFACTOR.

You can use the WITH NOCHECK option to prevent FOREIGN KEY and CHECK constraints for attempting to enforce their rules at creation.

You can use the following "niladic" functions as defaults: USER, CURRENT_USER, SESSION_USER (all of which give the username), SYSTEM_USER (login ID) and CURRENT_TIMESTAMP (same as GETDATE())

Numeric columns can have the IDENTITY constraint. This makes the column a one-up column for each row added. IDENTITY columns are ideal for primary keys. An IDENTITY column can not be updated, can not have nulls and can not have a default. You can set a starting value (for merging separate databases) and the increment (usually one is a good increment). There can be only one IDENTITY column per table and you can refer to it as IDENTITYCOL in WHERE clauses. You can get back the most recent identity value this session with @@IDENTITY. You can get the increment and seed the identity column if needed. If necessary, it is possible to manually add a row with a specific identity (usually to recover a deleted row). IDENTITY does ***not*** guarantee uniqueness; it is best used with the PRIMARY KEY attribute. You can use DBCC CHECKIDENT to check to make sure the system is using the correct maximum identity values.

3.2. Defaults and Rules

Default and check *constraints* do the same thing as *defaults* and *rules*. Microsoft suggests using default and check constraints (which are stored with the table definition) over defaults and rules (which must be bound to the table/column). If you define a default constraint and bind a default to a column, the default constraint wins over the default.[6]

So why would you ever use a default or rule? If you want the same default or rule to apply to multiple columns. Defaults and rules can be attached to user defined data types (which can then be used to define multiple columns). ***Defaults and rules should be attached to data types which can then be used in table definitions.***

3.3. Transactions

Transactions are blocks of SQL code which will either all succeed or fail together. If a transaction fails, SQL Server will restore the database to the state it had before the transaction started.

Every single SQL statement results in an implicit transaction. That is, if a single SQL statement fails part way through, SQL Server will roll back the results.

Explicit transactions are defined by the programmer/user. Explicit transactions go across more than one SQL statement. They begin with a BEGIN TRAN and end with either a COMMIT

[6] Yes, the terminology is very confusing. I guest its de fault of Microsoft.

TRAN or a ROLLBACK TRAN. Note that transactions have names - use the same name with the BEGIN, COMMIT and ROLLBACK. Transactions should be completed within a single batch (*i.e.* there should not be a GO statement in the middle of a transaction). It is possible to implement partial rollbacks with save points (SAVE TRAN).

3.4. Locks

SQL Server has some provision for adjusting lock levels. Note that exclusive locks for add, edit and delete can not be gotten until all shared locks have been removed. So don't be idling with open cursors.

You can now specify row level locking for inserts on a per table basis. This is useful where all inserts go to the end of the table (like a table with a clustered index on an identity column). In general, it looks like a good idea to turn on the insert row level locking on for all tables, unless the table has a clustered index which is not on an identity column.[7]

3.5. Concurrency

Use timestamp columns to implement optimistic concurrency control. Copy the row to the front-end and release the lock. When the user is done, check to timestamp to see if another user has changed the row. If not, save the update. Otherwise, take the appropriate action (which could be to over write the other user or to notify the current user).

[7] Sounds like the gobbledy-gook a tax lawyer would come up with!

4. Indexes

Indexes provide ordered access to data. According to Microsoft, to have a high performance system, you spend about as much time implementing the indexes as you spend initially designing the system!

When to use an index:

- On a column you want to search frequently
- Logical primary and foreign keys
- A column to be retrieved in a sorted order
- Columns used regularly in joins
- Columns often searched for a range of values
- Columns used frequently in a WHERE clause

When NOT to use an index:

- If the index is never used by the optimizer
- If more than 10-20 percent of the rows are returned
- If the column contains only one, two or three unique values
- If the column to be indexed is more than 20 bytes
- If the overhead of maintaining the index is greater than the benefit[8]

The primary costs of indexes are the time it takes to update them as information is inserted and edited (especially inserted) and disk space.

Some indexes are created automatically as a result of constraints. Other indexes are created specifically by the database designer.

You can not create indexes on bit, text or image fields. You can not alter indexes. To change an index, you must first drop the index and then recreate the index.

When an index is created, it requires a temporary space of twice the size of the indexed table in the database where the index is being created (not in tempdb). Once built, indexes are about 5% of the size of the original table. The "Manage Indexes" dialog box in SQL Enterprise Manager will show the distribution of data in the table.

Microsoft suggest the following naming conventions:

PRIMARY KEY INDEX:	tablename_IDENT
FOREIGN KEY INDEX:	fktablename_pktablename_LINK

Most indexes are non-clustered indexes. Each table may have one clustered index. A clustered index actually arranges the data in the order of the index. Carefully review the documentation before creating clustered indexes. Two good candidates for a clustered index are the primary key and a column where many rows with contiguous key values are being retrieved (*i.e.* the column is searched for a range of values).

[8] Yeh, mom & apple pie too. This is an obvious test; it would have been nicer if Microsoft had actually described how to determine this.

For clustered index, you can indicate the FILLFACTOR. The FILLFACTOR is the number of data pages left open for later data inserts. Use a fill factor of 100 for read-only tables. Adjust the fill factor based on estimated frequency of updates.

Indexes created with the UNIQUE keyword enforce uniqueness on the columns.

To find duplicate values in a column, use the following query:

```
SELECT indx_col, COUNT(indx_col)
        FROM tablename
        GROUP BY indx_col
        HAVING COUNT(indx_col) > 1
```

To show the number of unique values in a column:

```
SELECT COUNT (DISTINCT indx_col)
        FROM tablename
```

Composite indexes allow you to specify up to 16 columns (up to 900 bytes). Composite indexes are preferred to multiple single column indexes because they:

- Require less overhead during data manipulation
- Can reduce overall number of indexes on a table
- Can be useful for a greater number of queries
- Can be a better choice for any query that accesses multiple columns in a single table

Note that the order in which columns are specified in the WHERE clause does not affect how composite indexes are used. It only matters that the leftmost column in the index is in the WHERE clause.

List the most unique column first when creating composite indexes.

If the data is already sorted, creating an index with SORTED_DATA prevents the data from being sorted again (which speeds the creation of the index). You can use STORED_DATA_REORG to move the data to new data pages, compacting the data in the process. This is useful when a table becomes fragmented. You can do the same thing for existing indexes with DBCC DBREINDEX, which reindexes the data.

When significant changes occur to a table, you should UPDATE STATISTICS on the indexes for that table. The SQL query optimizer uses information from the statistics to decide on what (if any) index to use.

You can determine fragmentation with DBCC SHOWCONTIG (table_id). This should be done before an sp_spaceused. To find out if an index is being used for a query, use SHOWPLAN.

5. Views

When doing queries, views act just like tables. However, views have no data. Views retrieve data from the tables each time they are displayed. When you change the data in a view, it modifies the data in the underlying table.

You can use views to provide users access to a restricted set of data (restricted as to columns or rows or both).

The basic syntax to create a view is:
CREATE VIEW viewname AS SELECT <some select statement>

Creating a view WITH CHECK OPTION ensures that any data manipulations made by the user must be within the data set the user is allowed to see. Thus, the user will continue to be able to see the changed data.

If you need to prevent users from seeing the view definition, create it WITH ENCRYPTION.

NEVER USE AN OUTER JOIN IN THE DEFINITION OF A VIEW.

Name views to clearly distinguish them from tables (*e.g.* stick _VW after it).

Updates in view can only affect one table of a multi-table view. You can not update computed or aggregate columns. You can not insert into a view if the table contains not null columns which are not in the view and which do not have a default.

6. Transact-SQL

Transact-SQL is a *programming language*! It has variables, branching and looping!

You can execute Transact-SQL at the SQL command line or compile it into a *stored procedure* (which can then be executed by a *trigger*). Thus, all a trigger does is call a program! You can execute Transact-SQL code from any interface which will allow you to type in a custom SELECT statement. Thus, ***your front-end can create and execute back-end programs at run-time!***

The source code for Transact-SQL is stored in ASCII files. A single program/file is called a script. Within a script, there may be several batches. (Thus, a script is a series of batches run one after another.) A batch is a series of statements which are compiled and executed all at once. Batches are separated by the keyword GO. The maximum size of a batch is 128k. It pays to liberally sprinkle GO in your scripts.

-- or /* comment */ are used for comments.

Global variables are all predefined and are maintained by the system. Global variables start with two @@.

System Monitoring Global Variables:

@@CONNECTIONS	@@MAX_PRECISION	@@SERVICENAME
@@CPU_BUSY	@@MICROSOFTVERSION	@@TIMETICKS
@@DTBS	@@PACK_RECEIVED	@@TOTAL_ERRORS
@@IDLE	@@PACK_SENT	@@TOTAL_READ
@@IO_BUSY	@@PACKET_ERRORS	@@TOTAL_WRITE
@@MAX_CONNECTIONS	@@SERVERNAME	@@VERSION

Connection Specific Global Variables:

@@CURSOR_ROWS	@@LANGUAGE	@@SPID
@@ERROR	@@NESTLEVEL	@@TEXTSIZE
@@FETCH_STATUS	@@PROCID	@@TRANCOUNT
@@IDENTITY	@@REMSERVER	
@@LANGID	@@ROWCOUNT	

Local variables are created within a batch or procedure and are deallocated at the end of the batch or procedure.

```
DECLARE @var_name int
SELECT @var_name = 10 + @var_name
```

You can use the PRINT statement to return information to the user. RAISERROR can be used to create and manage error messages.

You can dynamically create a TRANSACT-SQL statement within a TRANSACT-SQL program and run it with EXECUTE.

RETURN returns optionally with a return code.

Flow of control:
BEGIN & END - like curly braces { } in C.
IF & ELSE
WHILE, BREAK, CONTINUE
CASE WHEN…THEN…ELSE…END

TRANSACT-SQL provides for “ANSI-SQL cursors”. You use ANSI-SQL cursors to do normal cursor things like DECLARE, OPEN, FETCH, CLOSE and DEALLOCATE.

You can create “extended stored procedures” which call server DLLs. That is not what we are calling stored procedures. Stored procedures are stored Transact-SQL programs.

Parsing is done when the stored procedure is stored so stored procedures execute quickly.

Stored procedures can accept parameters, call other stored procedures, return a status variable and even return values in parameters (using the OUTPUT option).

You can create temporary stored procedures (# in front of name for local and ## for global) and stored procedures that can be run from any database (if they start with sp_ and are created in the master database).

You can execute a stored procedure to insert records into a table (usually a temporary table).

If the stored procedure has parameters which will cause widely varying results sets, you can either compile or execute the stored procedure WITH RECOMPILE. That will cause the stored procedure to recompile its access plan. Use the procedure sp_reompile tablename to recompile all stored procedures for a table (for example, after adding a lot of data to the table).

Triggers are stored procedures which are called on INSERT, UPDATE or DELETE. Triggers can be nested to cascade change throughout a database.

Triggers are slower than using declarative referential integrity. Therefor, only use triggers to deal with matters not dealt with in the declarative referential integrity rules. Nested triggers can use considerable resources. The reason for the extra overhead is that SQL Server actually makes the table changes before executing the trigger. The data before the change is placed in a special storage area which the trigger can access. If the trigger fails, the whole transaction is then rolled back. In the ODCSOPS architecture, triggers are not used and stored procedures are directly called to make changes. The changes are made only *after* the checks have been completed avoiding this extra overhead.

Triggers are executed only after the constraints on a table have been passed.

You can grant users access to stored procedures without giving them insert, upate or delete access to the underlying table. Microsoft recommends using stored procedures if a key must be changed.