**Tips to improve SQL Server database design and performance**

*Posted By : Shailendra Chauhan, 27 Aug 2012*

*Updated On : 27 Dec 2012*

*Version Support : SQL Server 2005,2008,2012*

*Keywords : sql server best practice, tips to improve sql query performance, tips for good database design, sql server optimization techniques, tips for sql query optimization*

Best performance is the main concern to develop a successful application. Like a coin database is the tail side (back-end) of an application. A good database design provides best performance during data manipulation which results into the best performance of an application.

During database designing and data manipulation we should consider the following key points:

1. **Choose Appropriate Data Type**

Choose appropriate SQL Data Type to store your data since it also helps in to improve the query performance. Example: To store strings use varchar in place of text data type since varchar performs better than text. Use text data type, whenever you required storing of large text data (more than 8000 characters). Up to 8000 characters data you can store in varchar.

1. **Avoid nchar and nvarchar**

Does practice to avoid nchar and nvarchar data type since both the data types takes just double memory as char and varchar. Use nchar and nvarchar when you required to store Unicode (16-bit characters) data like as Hindi, Chinese characters etc.

1. **Avoid NULL in fixed-length field**

Does practice to avoid the insertion of NULL values in the fixed-length (char) field. Since, NULL takes the same space as desired input value for that field. In case of requirement of NULL, use variable-length (varchar) field that takes less space for NULL.

1. **Avoid \* in SELECT statement**

Does practice to avoid \* in Select statement since SQL Server converts the \* to columns name before query execution. One more thing, instead of querying all columns by using \* in select statement, give the name of columns which you required.

* 1. ***-- Avoid***
  2. **SELECT \* FROM tblName**
  3. ***--Best practice***
  4. **SELECT col1,col2,col3 FROM tblName**

1. **Use EXISTS instead of IN**

Does practice to use EXISTS to check existence instead of IN since EXISTS is faster than IN.

* 1. ***-- Avoid***
  2. **SELECT Name,Price FROM tblProduct**
  3. **where ProductID IN (Select distinct ProductID from tblOrder)**
  4. ***--Best practice***
  5. **SELECT Name,Price FROM tblProduct**
  6. **where ProductID EXISTS (Select distinct ProductID from tblOrder)**

1. **Avoid Having Clause**

Does practice to avoid Having Clause since it acts as filter over selected rows. Having clause is required if you further wish to filter the result of an aggregations. Don't use HAVING clause for any other purpose.

1. **Create Clustered and Non-Clustered Indexes**

Does practice to create clustered and non clustered index since indexes helps in to access data fastly. But be careful, more indexes on a tables will slow the INSERT,UPDATE,DELETE operations. Hence try to keep small no of indexes on a table.

1. **Keep clustered index small**

Does practice to keep clustered index as much as possible since the fields used in clustered index may also used in nonclustered index and data in the database is also stored in the order of clustered index. Hence a large clustered index on a table with a large number of rows increase the size significantly. Please refer the article [Effective Clustered Indexes](http://www.simple-talk.com/sql/learn-sql-server/effective-clustered-indexes/)

1. **Avoid Cursors**

Does practice to avoid cursor since cursor are very slow in performance. Always try to use SQL Server cursor alternative. Please refer the article [Cursor Alternative](http://www.dotnet-tricks.com/Tutorial/sqlserver/IT5G180512-SQL-Server-Cursor-Alternatives.html).

1. **Use Table variable inplace of Temp table**

Does practice to use Table varible in place of Temp table since Temp table resides in the TempDb database. Hence use of Temp tables required interaction with TempDb database that is a little bit time taking task.

1. **Use UNION ALL inplace of UNION**

Does practice to use UNION ALL in place of UNION since it is faster than UNION as it doesn't sort the result set for distinguished values.

1. **Use Schema name before SQL objects name**

Does practice to use schema name before SQL object name followed by "." since it helps the SQL Server for finding that object in a specific schema. As a result performance is best.

* 1. ***--Here dbo is schema name***
  2. **SELECT col1,col2 from dbo.tblName**
  3. ***-- Avoid***
  4. **SELECT col1,col2 from tblName**

1. **Keep Transaction small**

Does practice to keep transaction as small as possible since transaction lock the processing tables data during its life. Some times long transaction may results into deadlocks. Please refer the article [SQL Server Transactions Management](http://www.dotnet-tricks.com/Tutorial/sqlserver/c2XF120412-SQL-Server-Transactions-Management.html)

1. **SET NOCOUNT ON**

Does practice to set NOCOUNT ON since SQL Server returns number of rows effected by SELECT,INSERT,UPDATE and DELETE statement. We can stop this by setting NOCOUNT ON like as:

When SET NOCOUNT is ON, the count 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 prevents the sending of DONE\_IN\_PROC messages to the client for each statement in a stored procedure. For stored procedures that contain several statements that do not return much actual data, or for procedures that contain Transact-SQL loops, setting SET NOCOUNT to ON can provide a significant performance boost, because network traffic is greatly reduced.

The setting specified by SET NOCOUNT is in effect at execute or run time and not at parse time.

* 1. **CREATE PROCEDURE dbo.MyTestProc**
  2. **AS**
  3. **SET NOCOUNT ON**
  4. **BEGIN**
  5. **.**
  6. **.**
  7. **END**

1. **Use TRY-Catch**

Does practice to use TRY-CATCH for handling errors in T-SQL statements. Sometimes an error in a running transaction may cause deadlock if you have no handle error by using TRY-CATCH. Please refer the article[Exception Handling by TRY…CATCH](http://www.dotnet-tricks.com/Tutorial/sqlserver/O3P3120412-SQL-Server-Exception-Handling-by-TRY%E2%80%A6CATCH.html)

1. **Use Stored Procedure for frequently used data and more complex queries**

Does practice to create stored procedure for quaery that is required to access data frequently. We also created stored procedure for resolving more complex task.

1. **Avoid prefix "sp\_" with user defined stored procedure name**

Does practice to avoid prefix "sp\_" with user defined stored procedure name since system defined stored procedure name starts with prefix "sp\_". Hence SQL server first search the user defined procedure in the master database and after that in the current session database. This is time consuming and may give unexcepted result if system defined stored procedure have the same name as your defined procedure.

# SQL Server Different Types of Cursors

*Posted By : Shailendra Chauhan, 06 May 2012*

*Updated On : 25 Sep 2012*

*Version Support : SQL Server 2005,2008,2012*

*Keywords : Types of Sql Server Cursors, Dynamic Cursors, Static Cursors, Forward Only Cursors, Keyset Driven Cursors, different types of cursors in sql server with example pdf*

A Cursor allow us to retrieve data from a result set in singleton fashion means row by row. Cursor are required when we need to update records in a database table one row at a time. I have already explained the [basic of cursor](http://www.dotnet-tricks.com/Tutorial/sqlserver?id=4L7I050512&title=SQL%20Server%20-%20Basic%20of%20Cursor).

A Cursor impacts the performance of the SQL Server since it uses the SQL Server instances' memory, reduce concurrency, decrease network bandwidth and lock resources. Hence it is mandatory to understand the cursor types and its functions so that you can use suitable cursor according to your needs.

You should avoid the use of cursor. Basically you should use cursor alternatives like as WHILE loop, sub queries, Temporary tables and Table variables. We should use cursor in that case when there is no option except cursor.

## Types of Cursors

## Static Cursors

A static cursor populates the result set at the time of cursor creation and query result is cached for the lifetime of the cursor. A static cursor can move forward and backward direction. A static cursor is slower and use more memory in comparison to other cursor. Hence you should use it only if scrolling is required and other types of cursors are not suitable.

You can't update, delete data using static cursor. It is not sensitive to any changes to the original data source. By default static cursors are scrollable.

## Dynamic Cursors

A dynamic cursor allows you to see the data updation, deletion and insertion in the data source while the cursor is open. Hence a dynamic cursor is sensitive to any changes to the data source and supports update, delete operations. By default dynamic cursors are scrollable.

## Forward Only Cursors

A forward only cursor is the fastest cursor among the all cursors but it doesn't support backward scrolling. You can update, delete data using Forward Only cursor. It is sensitive to any changes to the original data source.

There are three more types of Forward Only Cursors.Forward\_Only KEYSET, FORWARD\_ONLY STATIC and FAST\_FORWARD.

A **FORWARD\_ONLY STATIC Cursor** is populated at the time of creation and cached the data to the cursor lifetime. It is not sensitive to any changes to the data source.

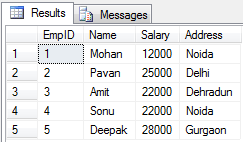
A **FAST\_FORWARD Cursor** is the fastest cursor and it is not sensitive to any changes to the data source.

## Keyset Driven Cursors

A keyset driven cursor is controlled by a set of unique identifiers as the keys in the keyset. The keyset depends on all the rows that qualified the SELECT statement at the time of cursor was opened. A keyset driven cursor is sensitive to any changes to the data source and supports update, delete operations. By default keyset driven cursors are scrollable.

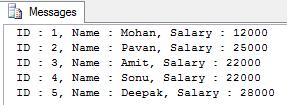
## SQL SERVER – Examples of Cursors

1. **CREATE TABLE Employee**
2. **(**
3. **EmpID int PRIMARY KEY,**
4. **EmpName varchar (50) NOT NULL,**
5. **Salary int NOT NULL,**
6. **Address varchar (200) NOT NULL,**
7. **)**
8. **GO**
9. **INSERT INTO Employee(EmpID,EmpName,Salary,Address) VALUES(1,'Mohan',12000,'Noida')**
10. **INSERT INTO Employee(EmpID,EmpName,Salary,Address) VALUES(2,'Pavan',25000,'Delhi')**
11. **INSERT INTO Employee(EmpID,EmpName,Salary,Address) VALUES(3,'Amit',22000,'Dehradun')**
12. **INSERT INTO Employee(EmpID,EmpName,Salary,Address) VALUES(4,'Sonu',22000,'Noida')**
13. **INSERT INTO Employee(EmpID,EmpName,Salary,Address) VALUES(5,'Deepak',28000,'Gurgaon')**
14. **GO**
15. **SELECT \* FROM Employee**



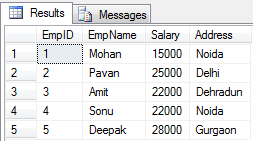
## Static Cursor - Example

1. **SET NOCOUNT ON**
2. **DECLARE @Id int**
3. **DECLARE @name varchar(50)**
4. **DECLARE @salary int**
5. **DECLARE cur\_emp CURSOR**
6. **STATIC FOR**
7. **SELECT EmpID,EmpName,Salary from Employee**
8. **OPEN cur\_emp**
9. **IF @@CURSOR\_ROWS > 0**
10. **BEGIN**
11. **FETCH NEXT FROM cur\_emp INTO @Id,@name,@salary**
12. **WHILE @@Fetch\_status = 0**
13. **BEGIN**
14. **PRINT 'ID : '+ convert(varchar(20),@Id)+', Name : '+@name+ ', Salary : '+convert(varchar(20),@salary)**
15. **FETCH NEXT FROM cur\_emp INTO @Id,@name,@salary**
16. **END**
17. **END**
18. **CLOSE cur\_emp**
19. **DEALLOCATE cur\_emp**
20. **SET NOCOUNT OFF**

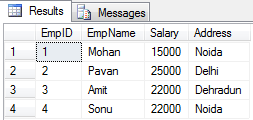


## Dynamic Cursor - Example

1. ***--Dynamic Cursor for Update***
2. **SET NOCOUNT ON**
3. **DECLARE @Id int**
4. **DECLARE @name varchar(50)**
5. **DECLARE Dynamic\_cur\_empupdate CURSOR**
6. **DYNAMIC**
7. **FOR**
8. **SELECT EmpID,EmpName from Employee ORDER BY EmpName**
9. **OPEN Dynamic\_cur\_empupdate**
10. **IF @@CURSOR\_ROWS > 0**
11. **BEGIN**
12. **FETCH NEXT FROM Dynamic\_cur\_empupdate INTO @Id,@name**
13. **WHILE @@Fetch\_status = 0**
14. **BEGIN**
15. **IF @name='Mohan'**
16. **Update Employee SET Salary=15000 WHERE CURRENT OF Dynamic\_cur\_empupdate**
17. **FETCH NEXT FROM Dynamic\_cur\_empupdate INTO @Id,@name**
18. **END**
19. **END**
20. **CLOSE Dynamic\_cur\_empupdate**
21. **DEALLOCATE Dynamic\_cur\_empupdate**
22. **SET NOCOUNT OFF**
23. **Go**
24. **Select \* from Employee**

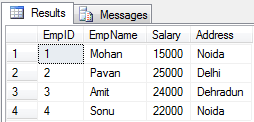


1. ***-- Dynamic Cursor for DELETE***
2. **SET NOCOUNT ON**
3. **DECLARE @Id int**
4. **DECLARE @name varchar(50)**
5. **DECLARE Dynamic\_cur\_empdelete CURSOR**
6. **DYNAMIC**
7. **FOR**
8. **SELECT EmpID,EmpName from Employee ORDER BY EmpName**
9. **OPEN Dynamic\_cur\_empdelete**
10. **IF @@CURSOR\_ROWS > 0**
11. **BEGIN**
12. **FETCH NEXT FROM Dynamic\_cur\_empdelete INTO @Id,@name**
13. **WHILE @@Fetch\_status = 0**
14. **BEGIN**
15. **IF @name='Deepak'**
16. **DELETE Employee WHERE CURRENT OF Dynamic\_cur\_empdelete**
17. **FETCH NEXT FROM Dynamic\_cur\_empdelete INTO @Id,@name**
18. **END**
19. **END**
20. **CLOSE Dynamic\_cur\_empdelete**
21. **DEALLOCATE Dynamic\_cur\_empdelete**
22. **SET NOCOUNT OFF**
23. **Go**
24. **Select \* from Employee**

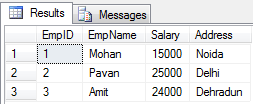


## Forward Only Cursor - Example

1. ***--Forward Only Cursor for Update***
2. **SET NOCOUNT ON**
3. **DECLARE @Id int**
4. **DECLARE @name varchar(50)**
5. **DECLARE Forward\_cur\_empupdate CURSOR**
6. **FORWARD\_ONLY**
7. **FOR**
8. **SELECT EmpID,EmpName from Employee ORDER BY EmpName**
9. **OPEN Forward\_cur\_empupdate**
10. **IF @@CURSOR\_ROWS > 0**
11. **BEGIN**
12. **FETCH NEXT FROM Forward\_cur\_empupdate INTO @Id,@name**
13. **WHILE @@Fetch\_status = 0**
14. **BEGIN**
15. **IF @name='Amit'**
16. **Update Employee SET Salary=24000 WHERE CURRENT OF Forward\_cur\_empupdate**
17. **FETCH NEXT FROM Forward\_cur\_empupdate INTO @Id,@name**
18. **END**
19. **END**
20. **CLOSE Forward\_cur\_empupdate**
21. **DEALLOCATE Forward\_cur\_empupdate**
22. **SET NOCOUNT OFF**
23. **Go**
24. **Select \* from Employee**

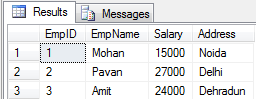


1. ***-- Forward Only Cursor for Delete***
2. **SET NOCOUNT ON**
3. **DECLARE @Id int**
4. **DECLARE @name varchar(50)**
5. **DECLARE Forward\_cur\_empdelete CURSOR**
6. **FORWARD\_ONLY**
7. **FOR**
8. **SELECT EmpID,EmpName from Employee ORDER BY EmpName**
9. **OPEN Forward\_cur\_empdelete**
10. **IF @@CURSOR\_ROWS > 0**
11. **BEGIN**
12. **FETCH NEXT FROM Forward\_cur\_empdelete INTO @Id,@name**
13. **WHILE @@Fetch\_status = 0**
14. **BEGIN**
15. **IF @name='Sonu'**
16. **DELETE Employee WHERE CURRENT OF Forward\_cur\_empdelete**
17. **FETCH NEXT FROM Forward\_cur\_empdelete INTO @Id,@name**
18. **END**
19. **END**
20. **CLOSE Forward\_cur\_empdelete**
21. **DEALLOCATE Forward\_cur\_empdelete**
22. **SET NOCOUNT OFF**
23. **Go**
24. **Select \* from Employee**

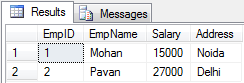


## Keyset Driven Cursor - Example

1. ***-- Keyset driven Cursor for Update***
2. **SET NOCOUNT ON**
3. **DECLARE @Id int**
4. **DECLARE @name varchar(50)**
5. **DECLARE Keyset\_cur\_empupdate CURSOR**
6. **KEYSET**
7. **FOR**
8. **SELECT EmpID,EmpName from Employee ORDER BY EmpName**
9. **OPEN Keyset\_cur\_empupdate**
10. **IF @@CURSOR\_ROWS > 0**
11. **BEGIN**
12. **FETCH NEXT FROM Keyset\_cur\_empupdate INTO @Id,@name**
13. **WHILE @@Fetch\_status = 0**
14. **BEGIN**
15. **IF @name='Pavan'**
16. **Update Employee SET Salary=27000 WHERE CURRENT OF Keyset\_cur\_empupdate**
17. **FETCH NEXT FROM Keyset\_cur\_empupdate INTO @Id,@name**
18. **END**
19. **END**
20. **CLOSE Keyset\_cur\_empupdate**
21. **DEALLOCATE Keyset\_cur\_empupdate**
22. **SET NOCOUNT OFF**
23. **Go**
24. **Select \* from Employee**



1. ***-- Keyse Driven Cursor for Delete***
2. **SET NOCOUNT ON**
3. **DECLARE @Id int**
4. **DECLARE @name varchar(50)**
5. **DECLARE Keyset\_cur\_empdelete CURSOR**
6. **KEYSET**
7. **FOR**
8. **SELECT EmpID,EmpName from Employee ORDER BY EmpName**
9. **OPEN Keyset\_cur\_empdelete**
10. **IF @@CURSOR\_ROWS > 0**
11. **BEGIN**
12. **FETCH NEXT FROM Keyset\_cur\_empdelete INTO @Id,@name**
13. **WHILE @@Fetch\_status = 0**
14. **BEGIN**
15. **IF @name='Amit'**
16. **DELETE Employee WHERE CURRENT OF Keyset\_cur\_empdelete**
17. **FETCH NEXT FROM Keyset\_cur\_empdelete INTO @Id,@name**
18. **END**
19. **END**
20. **CLOSE Keyset\_cur\_empdelete**
21. **DEALLOCATE Keyset\_cur\_empdelete**
22. **SET NOCOUNT OFF**
23. **Go Select \* from Employee**



**SQL Server Cursor Alternatives**

*Posted By : Shailendra Chauhan, 18 May 2012*

*Updated On : 26 Nov 2012*

*Version Support : SQL Server 2005,2008,2012*

*Keywords : cursor alternative in sql server,cursor alternative, cursor alternative with example pdf*

As we know, the cursors are required when we need to update records in a database table in singleton fashion means row by row. A Cursor also impacts the performance of the SQL Server since it uses the SQL Server instance’s memory, reduce concurrency, decrease network bandwidth and lock resources.

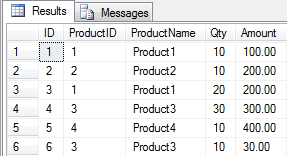
You should avoid the use of cursor. In this article, I am explaining how you can use cursor alternatives like as WHILE loop, Temporary tables and Table variables. We should use cursor in that case when there is no option except cursor.

**Example of Cursor Alternative**

Suppose we have table "ProductSales" that stores the information about each product sales. Now we want to calculate the Total Sales Quantity and Amount of each and every product.

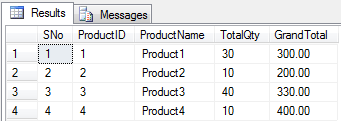
We can solve this problem by following three methods.

1. **CREATE TABLE ProductsSales**
2. **(**
3. **ID int IDENTITY(1,1) NOT NULL,**
4. **ProductID int NOT NULL,**
5. **ProductName varchar(50) NOT NULL,**
6. **Qty int NOT NULL,**
7. **Amount decimal(10, 2) NOT NULL )**
8. **GO**
9. **SELECT \* FROM ProductsSales**
10. ***--We have the table with below data***

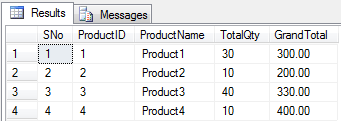


**Problem solution methods**

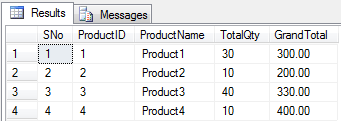
1. **Using Cursor**
   1. **SET NOCOUNT ON**
   2. **DECLARE @ProductID INT**
   3. **DECLARE @ProductName VARCHAR(100)**
   4. **DECLARE @TotalQty INT**
   5. **DECLARE @Total INT**
   6. **DECLARE @TProductSales TABLE**
   7. **(**
   8. **SNo INT IDENTITY(1,1),**
   9. **ProductID INT,**
   10. **ProductName VARCHAR(100),**
   11. **TotalQty INT,**
   12. **GrandTotal INT**
   13. **)**
   14. ***--Declare Cursor***
   15. **DECLARE Cur\_Product CURSOR FOR SELECT DISTINCT ProductID FROM ProductsSales**
   16. ***--Open Cursor***
   17. **OPEN Cur\_Product**
   18. ***--Fetch Cursor***
   19. **FETCH NEXT FROM Cur\_Product INTO @ProductID**
   20. **WHILE @@FETCH\_STATUS = 0**
   21. **BEGIN**
   22. **SELECT @ProductName = ProductName FROM ProductsSales WHERE ProductID = @ProductID**
   23. **SELECT @TotalQty = SUM(Qty),@Total = SUM(Amount) FROM ProductsSales WHERE ProductID = @ProductID**
   24. **INSERT INTO @TProductSales(ProductID,ProductName,TotalQty,GrandTotal) VALUES(@ProductID,@ProductName,@TotalQty,@Total)**
   25. **FETCH NEXT FROM Cur\_Product INTO @ProductID END**
   26. ***--Close and Deallocate Cursor***
   27. **CLOSE Cur\_Product**
   28. **DEALLOCATE Cur\_Product**
   29. ***--See Calculated data***
   30. **SELECT \* FROM @TProductSales**

****

1. **Using Table Variable**
   1. **SET NOCOUNT ON**
   2. **DECLARE @ProductID INT**
   3. **DECLARE @ProductName VARCHAR(100)**
   4. **DECLARE @TotalQty INT**
   5. **DECLARE @Total INT**
   6. **DECLARE @i INT =1**
   7. **DECLARE @count INT**
   8. ***--Declare Table variables for storing data***
   9. **DECLARE @TProduct TABLE ( SNo INT IDENTITY(1,1),**
   10. **ProductID INT**
   11. **)**
   12. **DECLARE @TProductSales TABLE**
   13. **(**
   14. **SNo INT IDENTITY(1,1),**
   15. **ProductID INT,**
   16. **ProductName VARCHAR(100),**
   17. **TotalQty INT,**
   18. **GrandTotal INT**
   19. **)**
   20. ***--Insert data to Table variable @Product***
   21. **INSERT INTO @TProduct(ProductID)**
   22. **SELECT DISTINCT ProductID FROM ProductsSales ORDER BY ProductID ASC**
   23. ***-- Count number of rows***
   24. **SELECT @count = COUNT(SNo) FROM @TProduct WHILE (@i <= @count)**
   25. **BEGIN**
   26. **SELECT @ProductID = ProductID FROM @TProduct WHERE SNo = @i**
   27. **SELECT @ProductName = ProductName FROM ProductsSales WHERE ProductID = @ProductID**
   28. **SELECT @TotalQty = SUM(Qty),@Total = SUM(Amount) FROM ProductsSales WHERE ProductID = @ProductID**
   29. **INSERT INTO @TProductSales(ProductID,ProductName,TotalQty,GrandTotal) VALUES(@ProductID,@ProductName,@TotalQty,@Total)**
   30. **SELECT @i = @i + 1**
   31. **END**
   32. ***--See Calculated data***
   33. **SELECT \* FROM @TProductSales**

****

1. **Using Temporary Table**
   1. **SET NOCOUNT ON**
   2. **DECLARE @ProductID INT**
   3. **DECLARE @ProductName VARCHAR(100)**
   4. **DECLARE @TotalQty INT**
   5. **DECLARE @Total INT**
   6. **DECLARE @i INT =1**
   7. **DECLARE @count INT**
   8. ***--Create Temporary Tables for storing data***
   9. **CREATE TABLE #TProduct ( SNo INT IDENTITY(1,1),**
   10. **ProductID INT**
   11. **)**
   12. **CREATE TABLE #TProductSales**
   13. **(**
   14. **SNo INT IDENTITY(1,1),**
   15. **ProductID INT, ProductName VARCHAR(100), TotalQty INT, GrandTotal INT )**
   16. ***--Insert data to temporary table #Product***
   17. **INSERT INTO #TProduct(ProductID) SELECT DISTINCT ProductID FROM ProductsSales ORDER BY ProductID ASC**
   18. **SELECT @count = COUNT(SNo) FROM #TProduct**
   19. **WHILE (@i <= @count)**
   20. **BEGIN**
   21. **SELECT @ProductID = ProductID FROM #TProduct WHERE SNo = @i**
   22. **SELECT @ProductName = ProductName FROM ProductsSales WHERE ProductID = @ProductID**
   23. **SELECT @TotalQty = SUM(Qty),@Total = SUM(Amount) FROM ProductsSales WHERE ProductID = @ProductID**
   24. **INSERT INTO #TProductSales(ProductID,ProductName,TotalQty,GrandTotal) VALUES(@ProductID,@ProductName,@TotalQty,@Total)**
   25. **SELECT @i = @i + 1**
   26. **END**
   27. ***--See Calculated data***
   28. **SELECT \* FROM #TProductSales**
   29. ***--Now Drop Temporary Tables***
   30. **DROP TABLE #TProduct**
   31. **DROP TABLE #TProductSales**

****

**Create a comma separated list from column using select statement**

*Posted By : Shailendra Chauhan, 23 Aug 2012*

*Updated On : 25 Sep 2012*

*Version Support : SQL Server 2005,2008,2012*

*Keywords : using coalesce to build comma delimited string, make or generate a comma separated list with select statement, creating a comma delimited list from a column*

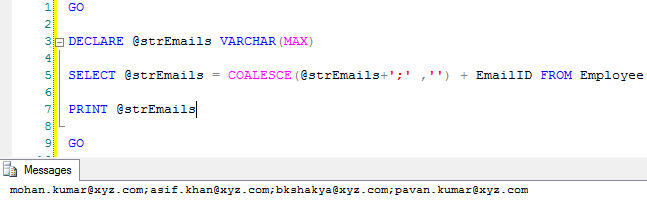
Sometimes we required to generate a comma separated list of columns values like a list of EmailIDs to send mail. In SQL Server, we can make a comma separated list by using COALESCE as shown in below.

**Use of COALESCE to create comma separated list**

Suppose we have following data in Employee table and we need to make a semicolon separated list of EmailIDs to send mail, then we can use COALESCE as shown in below fig.



Here I am creating a semicolon(;) separated list. You can use comma(,) in place of semicolon to make comma separated list.



For SQL Server database mail setup and configuration and more over how to send mail from SQL Server database you can refer this [article](http://www.dotnet-tricks.com/Tutorial/sqlserver/aPJa030512-SQL-Server-Database-Mail-Setup.html).

# SQL Server Exception Handling by TRY…CATCH

*Posted By : Shailendra Chauhan, 17 Apr 2011*

*Updated On : 25 Sep 2012*

*Version Support : SQL Server 2005,2008,2012*

*Keywords : Sql Server Exception Handling, Sql Server TRY..CATCH, Sql Server Error Functions pdf*

Like C#, SQL Server also has an exception model to handle exceptions and errors that occurs in T-SQL statements. To handle exception in Sql Server we have TRY..CATCH blocks. We put T-SQL statements in TRY block and to handle exception we write code in CATCH block. If there is an error in code within TRY block then the control will automatically jump to the corresponding CATCH blocks. In Sql Server, against a Try block we can have only one CATCH block.

**TRY..CATCH Syntax**

1. **BEGIN TRY**
2. ***--T-SQL statements***
3. ***--or T-SQL statement blocks***
4. **END TRY**
5. **BEGIN CATCH**
6. ***--T-SQL statements***
7. ***--or T-SQL statement blocks***
8. **END CATCH**

## Error Functions used within CATCH block

## ERROR\_NUMBER()

This returns the error number and its value is same as for @@ERROR function.

## ERROR\_LINE()

This returns the line number of T-SQL statement that caused error.

## ERROR\_SEVERITY()

This returns the severity level of the error.

## ERROR\_STATE()

This returns the state number of the error.

## ERROR\_PROCEDURE()

This returns the name of the stored procedure or trigger where the error occurred.

## ERROR\_MESSAGE()

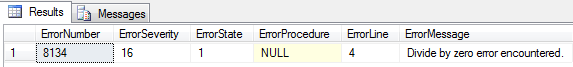
This returns the full text of error message. The text includes the values supplied for any substitutable parameters, such as lengths, object names, or times.

## Exception handling example

1. **BEGIN TRY**
2. **DECLARE @num INT, @msg varchar(200)**
3. ***---- Divide by zero to generate Error***
4. **SET @num = 5/0**
5. **PRINT 'This will not execute'**
6. **END TRY**
7. **BEGIN CATCH**
8. **PRINT 'Error occured that is'**
9. **set @msg=(SELECT ERROR\_MESSAGE())**
10. **print @msg;**
11. **END CATCH**
12. **GO**

http://www.dotnet-tricks.com/Content/images/sqlserver/dividebyzero1.png

1. **BEGIN TRY**
2. **DECLARE @num INT**
3. ***---- Divide by zero to generate Error***
4. **SET @num = 5/0**
5. **PRINT 'This will not execute'**
6. **END TRY**
7. **BEGIN CATCH**
8. **SELECT ERROR\_NUMBER() AS ErrorNumber, ERROR\_SEVERITY() AS ErrorSeverity, ERROR\_STATE() AS ErrorState, ERROR\_PROCEDURE() AS ErrorProcedure, ERROR\_LINE() AS ErrorLine, ERROR\_MESSAGE() AS ErrorMessage;**
9. **END CATCH;**
10. **GO**



#### Note

1. A TRY..CATCH block combination catches all the errors that have a severity between 11 and 19.
2. The CATCH block is executed only if there is an error occurs in T-SQL statements within TRY block otherwise the CATCH block is ignored.
3. Each TRY block is associated with only one CATCH block and vice versa
4. TRY and CATCH blocks can’t be separated with the GO statement. We need to put both TRY and CATCH blocks within the same batch.
5. TRY..CATCH blocks can be used with transactions. We check the number of open transactions by using @@TRANCOUNT function in Sql Server.
6. XACT\_STATE function within the TRY..CATCH block can be used to check whether a open transaction is committed or not. It will return -1 if transaction is not committed else returns 1.

**SQL Server Transactions Management**

*Posted By : Shailendra Chauhan, 23 Apr 2011*

*Updated On : 25 Sep 2012*

*Version Support : SQL Server 2005,2008,2012*

*Keywords : Types of transaction, Transaction management, Explicit transaction, Implicit Transaction, Transaction scope pdf*

A transaction is a set of T-SQL statements that are executed together as a unit like as a single T-SQL statement. If all of these T-SQL statements executed successfully, then a transaction is committed and the changes made by T-SQL statements permanently saved to database. If any of these T-SQL statements within a transaction fail, then the complete transaction is cancelled/ rolled back.

We use transaction in that case, when we try to modify more than one tables/views that are related to one another. Transactions affect SQL Server performance greatly. Since When a transaction is initiated then it locks all the tables data that are used in the transaction. Hence during transaction life cycle no one can modify these tables’ data that are used by the transaction. The reason behind the locking of the data is to maintain Data Integrity.

**Types of Transactions**

1. **Implicit Transaction**

Implicit transactions are maintained by SQL Server for each and every DDL (CREATE, ALTER, DROP, TRUNCATE), DML (INSERT, UPDATE, DELETE) statements. All these T-SQL statements runs under the implicit transaction. If there is an error occurs within these statements individually, SQL Server will roll back the complete statement.

1. **Explicit Transaction**

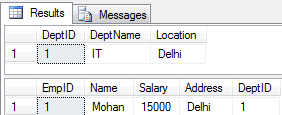
Explicit transactions are defined by programmers. In Explicit transaction we include the DML statements that need to be execute as a unit. Since SELECT statements doesn’t modify data. Hence generally we don’t include Select statement in a transaction.

**Transactions Example**

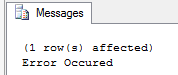
1. **CREATE TABLE Department**
2. **(**
3. **DeptID int PRIMARY KEY,**
4. **DeptName varchar(50) NULL,**
5. **Location varchar(100) NULL,**
6. **)**
7. **GO**
8. **CREATE TABLE Employee**
9. **(**
10. **EmpID int PRIMARY KEY,**
11. **Name varchar(50) NULL,**
12. **Salary int NULL,**
13. **Address varchar(100) NULL,**
14. **DeptID int foreign Key references Department(DeptID)**
15. **)**

http://www.dotnet-tricks.com/Content/images/sqlserver/success.png

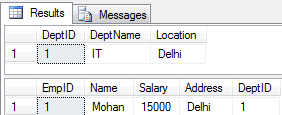
1. ***--Now Insert data***
2. **INSERT INTO Department(DeptID,DeptName,Location)VALUES(1,'IT','Delhi')**
3. **GO**
4. **INSERT INTO Employee(EmpID,Name,Salary,Address,DeptID)VALUES(1,'Mohan',15000,'Delhi',1)**
5. **SELECT \* FROM Department**
6. **SELECT \* FROM Employee**



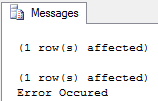
1. **BEGIN TRANSACTION trans**
2. **BEGIN TRY**
3. **INSERT INTO Department(DeptID,DeptName,Location)VALUES(2,'HR','Delhi')**
4. **INSERT INTO Employee(EmpID,Name,Salary,Address,DeptID)VALUES(1,'Mohan',18000,'Delhi',1)**
5. **IF @@TRANCOUNT > 0**
6. **BEGIN COMMIT TRANSACTION trans**
7. **END**
8. **END TRY**
9. **BEGIN CATCH**
10. **print 'Error Occured'**
11. **IF @@TRANCOUNT > 0**
12. **BEGIN ROLLBACK TRANSACTION trans**
13. **END**
14. **END CATCH**



1. ***--Now Select data to see transaction affects***
2. **SELECT \* FROM Employee**
3. **SELECT \* FROM Department**



1. ***--Transaction with Save Point BEGIN TRANSACTION trans***
2. **BEGIN TRY**
3. **INSERT INTO Department(DeptID,DeptName,Location)VALUES(2,'HR','Delhi')**
4. **IF @@TRANCOUNT > 0**
5. **BEGIN SAVE TRANSACTION trans;**
6. **END**
7. **INSERT INTO Department(DeptID,DeptName,Location)VALUES(3,'Admin','Delhi')**
8. **INSERT INTO Employee(EmpID,Name,Salary,Address,DeptID)VALUES(1,'Mohan',18000,'Delhi',1)**
9. **IF @@TRANCOUNT > 0**
10. **BEGIN COMMIT TRANSACTION trans**
11. **END**
12. **END TRY**
13. **BEGIN CATCH**
14. **print 'Error Occured'**
15. **IF @@TRANCOUNT > 0**
16. **BEGIN ROLLBACK TRANSACTION trans**
17. **END**
18. **END CATCH**



1. ***--Now Select data to see transaction affects***
2. **SELECT \* FROM Employee**
3. **SELECT \* FROM Department**

