

Multiples Table in DataReader

```
using (SqlConnection connection = new SqlConnection("connection string here"))
{
    using (SqlCommand command = new SqlCommand
        ("SELECT Column1 FROM Table1; SELECT Column2 FROM Table2", connection))
    {
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                MessageBox.Show(reader.GetString(0), "Table1.Column1");
            }

            if(reader.NextResult())
            {
                while (reader.Read())
                {
                    MessageBox.Show(reader.GetString(0), "Table2.Column2");
                }
            }
        }
    }
}
```



How do I change my table's identity column datatype without losing data

I'm assuming the IDENTITY column is your PRIMARY KEY, and it's probably clustered :) MY advice below is based on those assumptions.

If you've only got a few indexes on the table, and the PRIMARY KEY is only referenced by a few FOREIGN keys, you should be able to change the datatype by:

1. Dropping any nonclustered indexes which contain the IDENTITY value.
2. Dropping the FOREIGN KEY constraints which point to the PRIMARY KEY.
3. Drop the PRIMARY KEY
4. ALTER TABLE tablename ALTER COLUMN columnname INT;
5. REcreate the PRIMARY KEY
6. Re-enable the FOREIGN KEY constraints with CHECK.
7. Recreate your nonclustered indexes.

Create parameterized VIEW in SQL Server 2008

```
CREATE FUNCTION dbo.fxExample (@Parameter1
INTEGER)
RETURNS TABLE
AS
RETURN
```

No, you cannot. But you can create a user defined table function.

```
(  
    SELECT Field1, Field2  
    FROM SomeTable  
    WHERE Field3 = @Parameter1  
)  
  
-- Then call like this, just as if it's a  
table/view just with a parameter  
SELECT * FROM dbo.fxnExample(1)
```

Reorganize and Rebuild Indexes

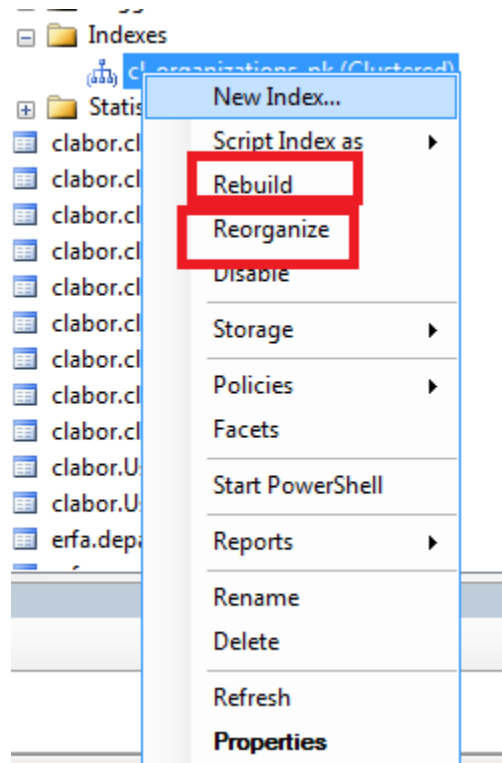
This topic describes how to reorganize or rebuild a fragmented index in SQL Server 2016 by using SQL Server Management Studio or Transact-SQL. The SQL Server Database Engine automatically maintains indexes whenever insert, update, or delete operations are made to the underlying data. Over time these modifications can cause the information in the index to become scattered in the database (fragmented). Fragmentation exists when indexes have pages in which the logical ordering, based on the key value, does not match the physical ordering inside the data file. Heavily fragmented indexes can degrade query performance and cause your application to respond slowly.

To reorganize or rebuild an index

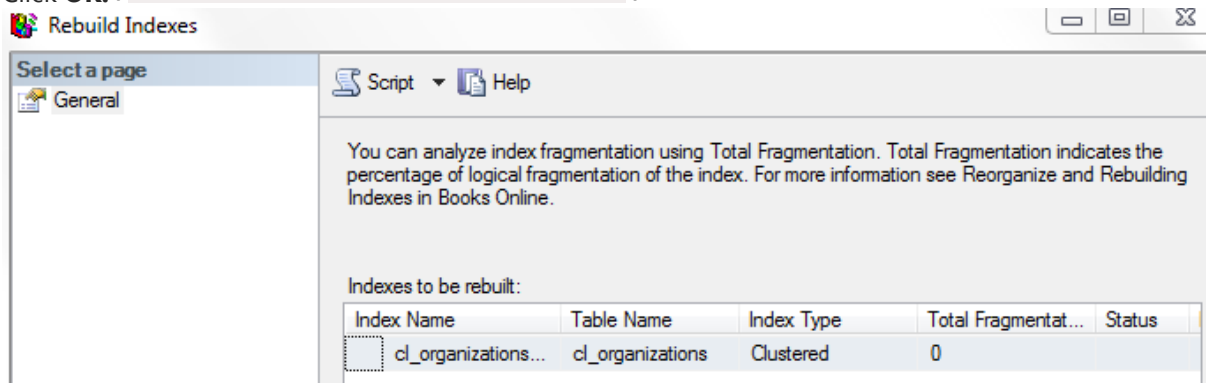
1. In Object Explorer, Expand the database that contains the table on which you want to reorganize an index.
2. Expand the **Tables** folder.
3. Expand the table on which you want to reorganize an index.
4. Expand the **Indexes** folder.
5. Right-click the index you want to reorganize and select **Reorganize**.
6. In the **Reorganize Indexes** dialog box, verify that the correct index is in the **Indexes to be reorganized** grid and click **OK**.
7. Select the **Compact large object column data** check box to specify that all pages that contain large object (LOB) data are also compacted.
8. Click **OK**.

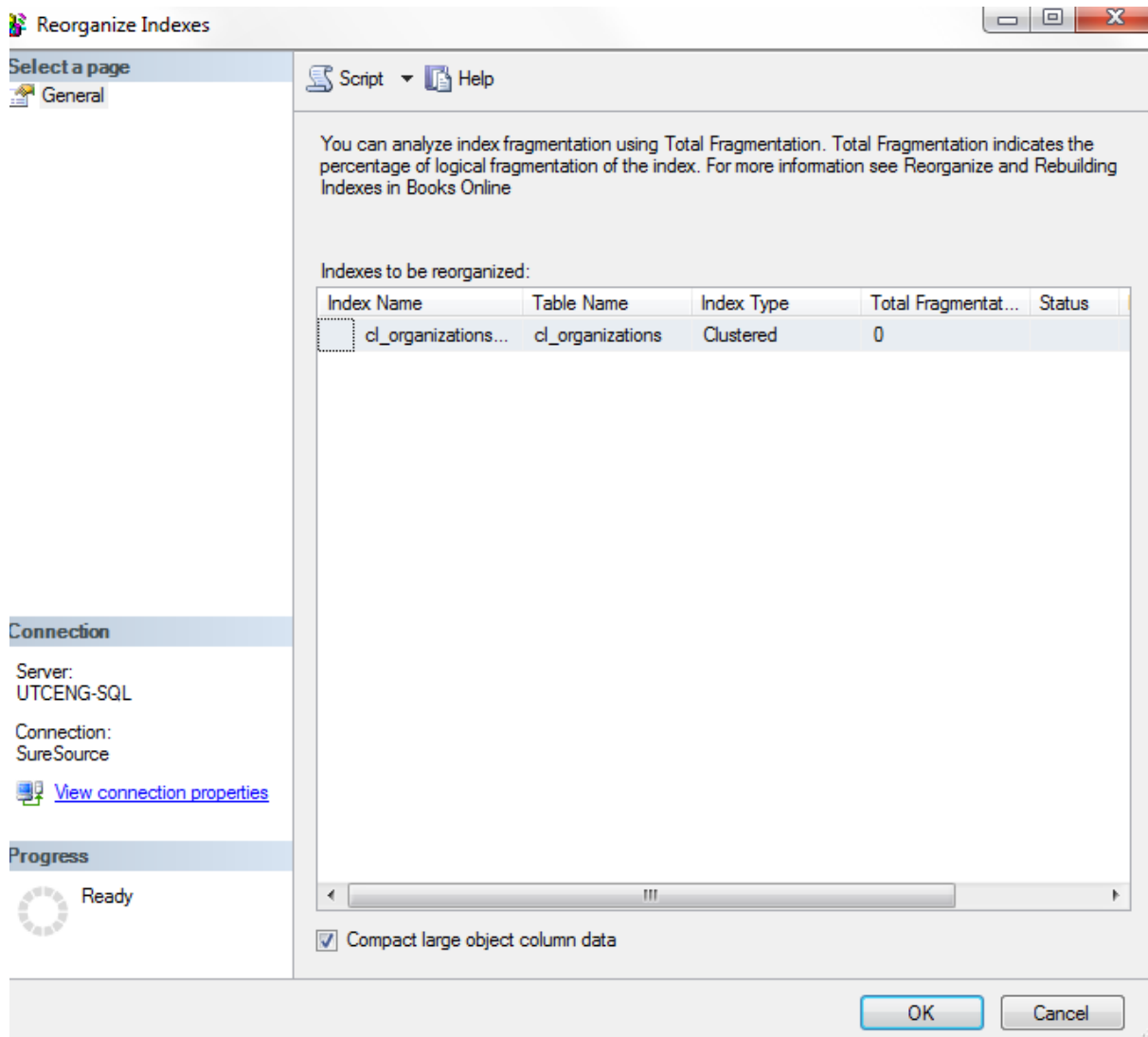
To reorganize all indexes in a table

1. In Object Explorer, Expand the database that contains the table on which you want to reorganize the indexes.
2. Expand the **Tables** folder.
3. Expand the table on which you want to reorganize the indexes.
4. Right-click the **Indexes** folder and select **Reorganize All**.
5. In the **Reorganize Indexes** dialog box, verify that the correct indexes are in the **Indexes to be reorganized**. To remove an index from the **Indexes to be reorganized** grid, select the index and then press the Delete key.
6. Select the **Compact large object column data** check box to specify that all pages that contain large object (LOB) data are also compacted.



7. Click **OK.**





To rebuild an index

1. In Object Explorer, Expand the database that contains the table on which you want to reorganize an index.
2. Expand the **Tables** folder.
3. Expand the table on which you want to reorganize an index.
4. Expand the **Indexes** folder.
5. Right-click the index you want to reorganize and select **Reorganize**.
6. In the **Rebuild Indexes** dialog box, verify that the correct index is in the **Indexes to be rebuilt** grid and click **OK**.
7. Select the **Compact large object column data** check box to specify that all pages that contain large object (LOB) data are also compacted.
8. Click **OK**.

Difference Between Index Rebuild and Index Reorganize Explained with T-SQL Script

Index Rebuild : This process drops the existing Index and Recreates the index.

```
USE AdventureWorks;
```

```
GO
```

```
ALTER INDEX ALL ON Production.Product REBUILD
```

```
GO
```

Index Reorganize : This process physically reorganizes the leaf nodes of the index.

```
USE AdventureWorks;
```

```
GO
```

```
ALTER INDEX ALL ON Production.Product REORGANIZE
```

```
GO
```

Recommendation: Index should **be rebuild** when **index fragmentation is great than 40%**. Index should be **reorganized** when index fragmentation is **between 10% to 40%**.

Index rebuilding process **uses more CPU** and it locks the database resources. SQL Server development version and Enterprise version has option **ONLINE**, which can be turned on when Index is rebuilt. **ONLINE** option will keep index available during the rebuilding.

sys.dm_db_index_physical_stats function includes the following columns.

Column	Description
avg_fragmentation_in_percent	The percent of logical fragmentation (out-of-order pages in the index).
fragment_count	The number of fragments (physically consecutive leaf pages) in the index.
avg_fragment_size_in_pages	Average number of pages in one fragment in an index.

avg_fragmentation_in_percent value	Corrective statement
> 5% and < = 30%	ALTER INDEX REORGANIZE
> 30%	ALTER INDEX REBUILD WITH (ONLINE = ON)*

Paging a Query with SQL Server

Introduction

Sometimes it is necessary to **optimize the data returned by a query**, removing unnecessary fields from a SELECT statement, and including conditions in your WHERE clause so that the user can retrieve only the data that really needed.

It is very important in the critical conditions for small companies, with limited hardware or software. The places **where Internet access is limited and slow we also have the need to reduce network traffic data.**

These are factors that require a query to retrieve only the data that will be displayed in the User Interface screen, excluding the extra that **maybe used** in another page, displaying of a "Grid View" or "Report".

One of the features that can be used to meet these requirements is the **Paging in the SQL Server.**

In the end of this article, we will evaluate the query performance under the following conditions:

- 1 - No pagination query in the SQL Server (normally used by programmers, leaving the data pagination task to a component as a "GridView" or other similar);
- 2 - Pagination Query, using [ROW NUMBER](#) (traditional model of data pagination in SQL Server);
- 3 - Pagination Query, using the [OFFSET and FETCH](#) clauses (new model, used from the SQL Server 2012+);

```
--CREATING A TABLE FOR DEMO
CREATE TABLE dbo.TB_EXAMPLE(
    ID_EXAMPLE int NOT NULL IDENTITY(1,1),
    NM_EXAMPLE varchar(25) NOT NULL,
    DT_CREATE datetime NULL DEFAULT(GETDATE())
);
GO

-- INSERTING 1,000,000 DIFFERENT ROWS IN THE TABLE
INSERT INTO TB_EXAMPLE (NM_EXAMPLE) VALUES ('Item de paginação
' + CONVERT(VARCHAR, ISNULL(@@IDENTITY, 0)))
GO 1000000

--QUERYING 1,000,000 ROWS
SELECT * FROM TB_EXAMPLE
GO
```

Paging rows with Limit

In order to understand the pagination concepts in T-SQL, with [ROW NUMBER](#) and with [OFFSET / FETCH](#), let's define a result pattern in the T-SQL script, for an evaluation of the above queries.

We created two variables to facilitate the data manipulation:

@PageNumber - Sets the number of the page to display

@RowspPage - Sets the rows number of each page

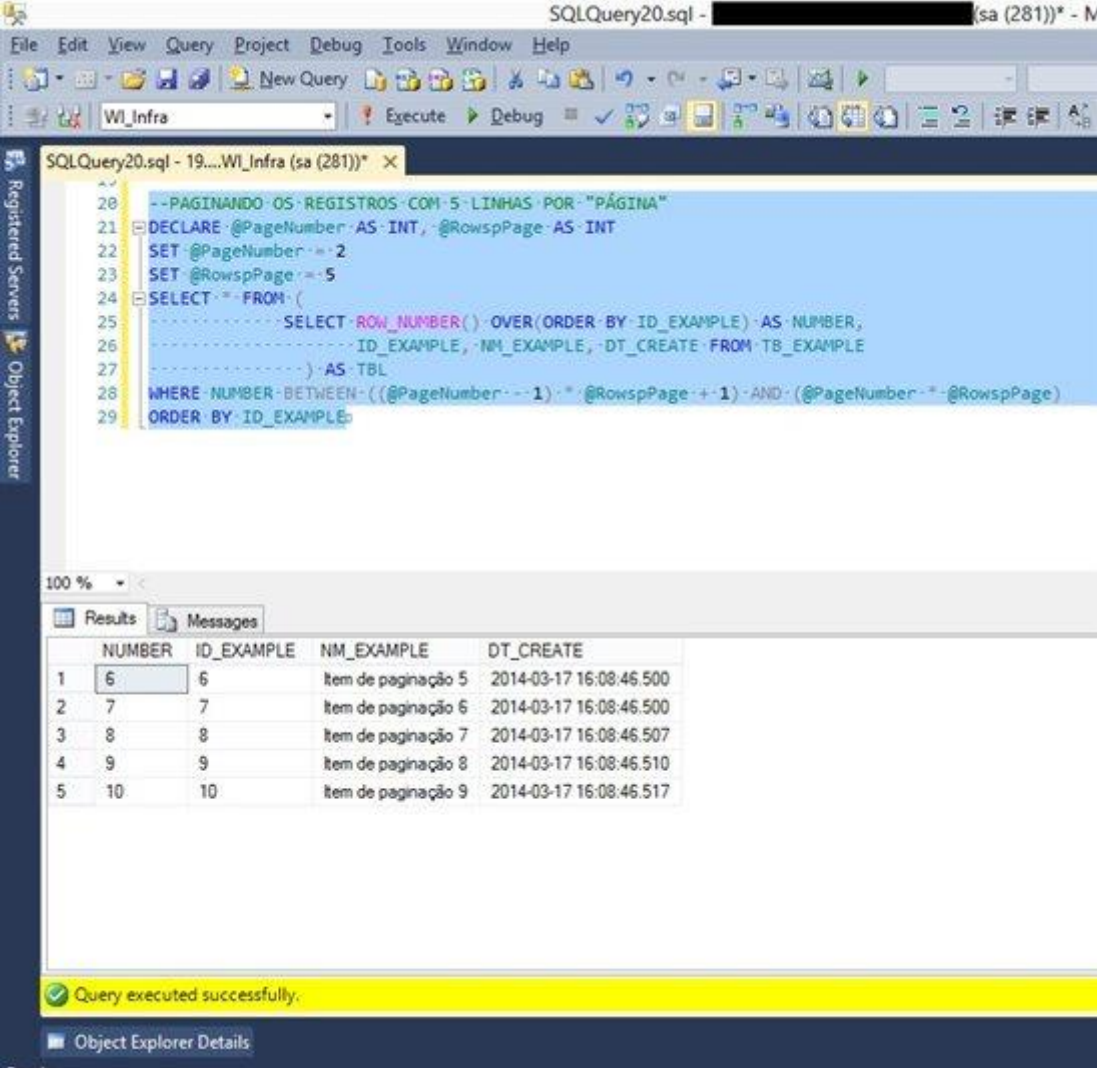
After setting these variables, let's start using the function [ROW_NUMBER](#), sorting the data by ID_EXAMPLE field.

See the script below, where we define to be displayed the "page 2" with "5 rows per page":

```
--VIEWING THE PAGE "2" WITH 5 ROWS
DECLARE @PageNumber AS INT, @RowspPage AS INT
SET @PageNumber = 2
SET @RowspPage = 5

SELECT * FROM (
    SELECT ROW_NUMBER() OVER(ORDER BY ID_EXAMPLE) AS NUMBER,
           ID_EXAMPLE, NM_EXAMPLE, DT_CREATE FROM TB_EXAMPLE
    ) AS TBL
WHERE NUMBER BETWEEN ((@PageNumber - 1) * @RowspPage + 1) AND (@PageNumber * @RowspPage)
ORDER BY ID_EXAMPLE
```

See this output SQL script in the image below.



The screenshot shows the SQL Server Enterprise Manager interface. The top pane displays a SQL query script named 'SQLQuery20.sql'. The script is as follows:

```
--PAGINANDO OS REGISTROS COM 5 LINHAS POR "PÁGINA"
DECLARE @PageNumber AS INT, @RowspPage AS INT
SET @PageNumber = 2
SET @RowspPage = 5
SELECT * FROM (
    SELECT ROW_NUMBER() OVER(ORDER BY ID_EXAMPLE) AS NUMBER,
           ID_EXAMPLE, NM_EXAMPLE, DT_CREATE FROM TB_EXAMPLE
    ) AS TBL
WHERE NUMBER BETWEEN ((@PageNumber - 1) * @RowspPage + 1) AND (@PageNumber * @RowspPage)
ORDER BY ID_EXAMPLE
```

The bottom pane shows the results of the query. The results are displayed in a table with the following columns: NUMBER, ID_EXAMPLE, NM_EXAMPLE, and DT_CREATE. The table contains 5 rows of data, representing the second page of results with 5 rows per page.

	NUMBER	ID_EXAMPLE	NM_EXAMPLE	DT_CREATE
1	6	6	Item de paginação 5	2014-03-17 16:08:46.500
2	7	7	Item de paginação 6	2014-03-17 16:08:46.500
3	8	8	Item de paginação 7	2014-03-17 16:08:46.507
4	9	9	Item de paginação 8	2014-03-17 16:08:46.510
5	10	10	Item de paginação 9	2014-03-17 16:08:46.517

At the bottom of the interface, a status bar indicates 'Query executed successfully.' and 'Object Explorer Details' is visible.

Paging in SQL Server 2012

In the SQL Server 2012 a new feature was added in the ORDER BY clause, to query optimization of a set data, making work easier with data paging for anyone who writes in T-SQL as well for the entire Execution Plan in SQL Server.

Below the T-SQL script with the same logic used in the [previous example](#).

```
--CREATING A PAGING WITH OFFSET and FETCH clauses IN "SQL SERVER 2012"
```

```
DECLARE @PageNumber AS INT, @RowspPage AS INT
```

```
SET @PageNumber = 2
```

```
SET @RowspPage = 10
```

```
SELECT ID_EXAMPLE, NM_EXAMPLE, DT_CREATE
```

```
FROM TB_EXAMPLE
```

```
ORDER BY ID_EXAMPLE
```

```
OFFSET ((@PageNumber - 1) * @RowspPage) ROWS
```

```
FETCH NEXT @RowspPage ROWS ONLY;
```

Now we will evaluate the performance of three different queries under the following conditions:

- 1 - Simple query without pagination in SQL Server (responsibility to maintain the pagination is of an Application [ASP.Net](#), [WinForms](#) or [other](#)).
- 2 - Query with pagination, using [ROW NUMBER](#) (traditional model of pagination in T-SQL);
- 3 - Query with pagination, using [OFFSET e FETCH](#) clauses (new model, using SQL Server 2012 or higher);

Conclusion

We can see that the pagination of a SELECT statement is simple to set up and can be considered an excellent resource for large amounts of data.

In the data optimization query by T-SQL has a considerable gain both using the **ROW_NUMBER** as use of **OFFSET / FETCH**, but taking into account the analysis of these data we can see that the "script 3" **query (using OFFSET / FETCH) may be the best option** for large volumes of rows, if you use a **SQL server 2012 version**.