

Volume

6

FLEXCEL STUDIO FOR .NET

TMS Software



Performance Guide

Table of Contents

TABLE OF CONTENTS	1
INTRODUCTION	1
WARNING	1
MEMORY	1
VIRTUAL MODE.....	2
64 BITS.....	3
LOOPS	4
DON'T FEAR LOOPS.....	4
BEWARE OF THE ENDING CONDITION IN "FOR" LOOPS IN C#.....	4
AVOID CALLING COLCOUNT	5
LOOP ONLY OVER EXISTING CELLS	5
EVIL LOOP EXAMPLE.....	6
READING FILES	8
IGNORE FORMULA TEXTS IF POSSIBLE.....	8
REPORTS	9
AVOID TOO MANY MASTER DETAIL LEVELS	9
DECIDE WHETHER TO USE LINQ OR DATASETS AS DATA SOURCES	9
LOAD TABLES ON DEMAND WHEN USING DATASETS	10
PROFILE THE SQL SENT TO THE SERVER WHEN USING ENTITY FRAMEWORK.....	11
PROVIDE THE RECORD COUNT IF POSSIBLE WHEN USING LINQ DATASETS	12
AVOID MICROSOFT DISTRIBUTED TRANSACTION COORDINATOR (MSDTC) WHEN USING ENTITY FRAMEWORK	13
PREFER SNAPSHOT TRANSACTIONS TO SERIALIZABLE TRANSACTIONS	13

Introduction

Performance is a complex but very important characteristic of any application. One of our design goals with FlexCel is to be as fast as possible, and we are always finding ways to improve a little the performance from version to version. The idea is that for most of the cases, FlexCel should be “fast enough” and you shouldn't have to care about performance when using it, as we took care of it for you. But for cases where you need the absolute maximum of performance, you can help FlexCel perform better by writing code that uses it in an optimal way. To know how to code for maximum performance, you need to understand how FlexCel works from a performance viewpoint, and that's what this document is about.

Warning

Before doing anything else, let us make this very clear: **Don't over optimize**. In many cases code clarity and performance are not compatible goals and the more performing the code, the more difficult it is to maintain, change, or adapt. In most cases FlexCel is incredibly fast, and you shouldn't have to do anything to create or read files instantly.

Memory

If there is one thing that can make your applications slower, that is using too much memory and starting paging to disk. And sadly, because of the way Excel files are created, and also because of how a spreadsheet works, FlexCel needs to keep the full spreadsheet loaded in memory.

Even when in many cases we use Excel files as databases they aren't. It is impossible to randomly read or write cells from an xls/xlsx file, and due to the way formulas work, a single change in a cell might affect the full file. For example, imagine that we have a spreadsheet with cell A1 = 1, and in the second sheet, Sheet2!E4 has the formula “=Sheet1!A1 + 1” When we change the value at cell A1 from 1 to 2 we need to change the formula result in Sheet2!E4 from 2 to 3. Also, if we insert a row before A1, we will need to change the cell Sheet2!E4 from “= Sheet1!A1 + 1” to “=Sheet1!A2 + 1”. This makes it very difficult to work with just a part of the file in memory, and neither we nor Excel do it. We both load the full file into memory and do all of our work there.

The main issue with running out of memory is that performance degradation is not linear. That is, you might use 1 second to write a million cells, but 1 minute to write 2 million, and 1 hour to write 3 million. This is one of the reasons we don't normally quote silly benchmark numbers like “n cells written per second”; they are meaningless. If you take n seconds to

write m cells, it doesn't mean that to write $2*m$ cells you will need $2*n$ seconds. Performance degrades exponentially when you run out of memory.

So we make a lot of effort to try to not use too much memory, for example, repeated strings in cells will be stored only once in memory, or cell styles will be shared between many cells. But no matter what, we are a fully managed .NET library, and we can't escape the limitations of the platform. .NET applications do use a lot of memory, and the Garbage Collector might not release everything you need either. This is not necessarily a bad thing, using the memory you have is good not bad, but for huge files it can become problematic. There are 2 ways you can get around this problem: using FlexCel in "Virtual Mode", or going to 64 bits. We will expand on this in the sections below.

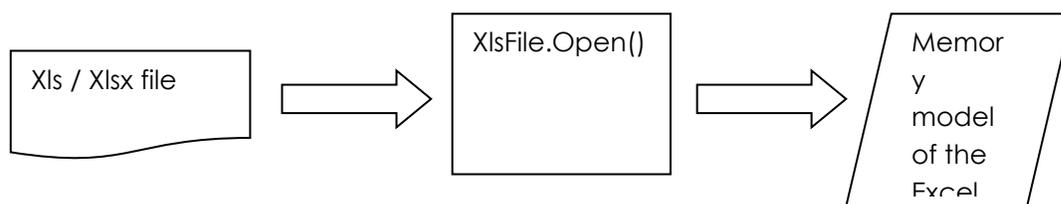
But before continuing, we would like to give one last warning about memory. Measuring memory consumption can be very complex, especially in .NET. You need to measure how many objects you have in Generations 0, 1 and 2, you need to distinguish between private and public bytes, and you need a memory measurement tool. Task manager isn't such tool.

Virtual Mode

Sometimes you don't really need all the complexity that a spreadsheet allows; you just want to read values from cells or dump a big database into a file. In those cases, it might not make sense to have the full file into memory, you could read a cell value, load it in your application, and discard the value as the file is being loaded. There is no need to wait until the entire file has been read to discard all the cells.

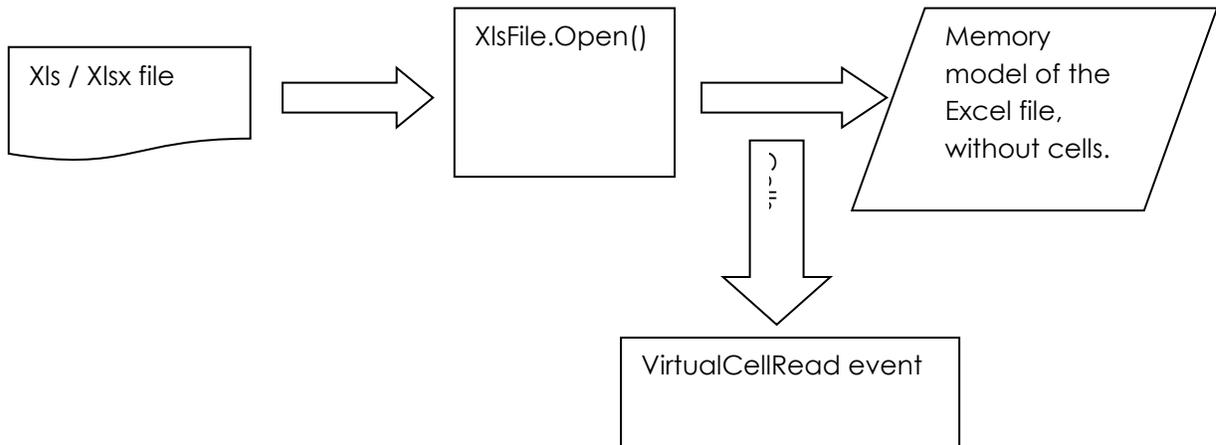
For this, FlexCel provides what we know as "Virtual Mode", a special mode where cells are read on demand. Not all features are supported in this mode, but normally for the huge files where Virtual mode makes sense, those features aren't needed either.

The normal way you open a file is as follows:



You call `XlsFile.Open`, and FlexCel reads the file and loads it into memory.

In virtual mode, it works as follows:



You call `XlsFile.Open`, and the file is read. But for every cell read, the `VirtualCellRead` event is called, and the cell is not loaded into the memory model. You will end up with an `XlsFile` object that has no cells (but has charts, drawings, comments, etc.). Cells can be used by your application as they are being read. You can get the rest of the things (comments, merged cells, etc.) from the memory model after the file has been opened.

Please look at the "Virtual Mode" api demo for an example on how to implement virtual mode.

64 bits

Sometimes, if you have a memory problem and the application runs server-side, the cheapest solution might be to install more memory in the server. But in 32 bit mode, no matter how much memory you have, FlexCel will never use more than 2 Gb.

If you want to use more memory than that you should use FlexCel in 64 bit mode. Note that FlexCel.dll is compiled as "Any CPU", and that means that it will work in 32 bit mode if your application is compiled as a 32 bit application and 64 bit mode if it is compiled as 64 bit. You don't need a different dll.

But we should go into warning mode again. **Going 64 bits might not only not improve performance, it might decrease it.** In our tests in a 4Gb machine, 64bit version is consistently slower than the 32bit one (both versions running in 64 bit OS). It kind of makes sense, since by going 64 bits now every pointer is 8 bytes instead of 4, so there is a lot of memory used in those bigger pointers, that are mostly 0 anyway. For 64 bit to improve things, you need a lot

of memory installed. By the way, .NET 4.0 is much better than 3.5 in our tests for 64 bits, getting almost the speed of 32 bits, while in 3.5 64bit is really bad. So if you are thinking in going 64bit, consider at least .NET 4.0.

Loops

Don't fear loops

From time to time we get emails from people migrating from OLE Automation solutions, asking us for a way to load all values into an array, and then write the full array into FlexCel in one single method call.

This is because in OLE Automation, one of the “performance tips” is to do exactly that. Set the cell values into an array, and then copy that array into Excel. But this is because **in OLE Automation method calls are very expensive**. And this concept is worth expanding: Excel itself isn't slow; it is coded in optimized C/C++ by some of the best coders in the world. So how is it possible that third party libraries written in managed languages like FlexCel can be so much faster? This is because Excel is optimized for interactive use, but more important, because while Excel itself is fast, calling Excel from OLE Automation is not.

So any good OLE Automation programmer knows that to maximize the speed, he needs to minimize the Excel calls. The more you can do in a single call the better, and if you can set a thousand cells in one call that is much better than doing a thousand calls setting the values individually.

But **FlexCel is not OLE Automation and the rules that apply are different**. In our case, if you were to fill an array with values so you can pass them to FlexCel, it would actually be slower. You would use the double of memory since you need to have the cells both in the array and in FlexCel, you would lose time filling the array, and when you finally call “XlsFile.SetRangeOfValues(array)” it would loop over all the array and cells to copy them from the Array to FlexCel since that is the only way to do it. So just lopping and filling FlexCel directly is way faster, the array only would add overhead.

Beware of the ending condition in “for” loops in C#

When you are coding in C#, you need to be aware that the ending condition of the loops is called for every iteration. So in the code:

```
for (int i = 0; i < VeryExpensiveFunction(); i++)
{
    DoSomething();
}
```

VeryExpensiveFunction() is called for every time DoSomething() is called. This can be from a non-issue when VeryExpensiveFunction isn't too expensive (for example it is List.Count) to a disaster if VeryExpensiveFunction is actually expensive and the iteration runs for millions of times (as it can be the case when using "XlsFile.ColCount").

The solutions to this problem normally can be:

1) Cache the value of VeryExpensiveFunction before entering the loop:

```
int Count = VeryExpensiveFunction();
for (int i = 0; i < Count; i++)
{
    DoSomething();
}
```

2) If the order in which the loop executes doesn't matter, you might simply reverse the loop:

```
for (int i = VeryExpensiveFunction() - 1; i >= 0; i--)
{
    DoSomething();
}
```

This way VeryExpensiveFunction will be called only once at the start, and the comparison is against "i >= 0" which is very fast.

Avoid calling ColCount

XlsFile.ColCount is a very slow method in FlexCel, and it is mostly useless. To find out how many columns there are in the whole sheet, FlexCel needs to loop over all rows and find which row has the biggest column. You normally just need to know the columns in the row you are working in, not the maximum column count in the whole sheet. **Note that RowCount is very fast, the issue is with ColCount, because FlexCel stores cells in row collections.**

Loop only over existing cells

A spreadsheet can have many empty cells in it, and normally you don't care about them. FlexCel offers methods to retrieve only the cells with some kind of data in them (be it actual values or formats), and ignore cells that have never have been modified. Use those methods whenever possible. Look at the example below to see how this can be done.

Evil loop Example

Here we will show an innocent looking example of all the problems studied in this “Loops” section. Even when it might look harmless, it can bring the performance of your application to its knees.

```
XlsFile xls = new XlsFile("somefile.xls")
for (int row = 1; row <= xls.RowCount; row++)
{
    for (int col = 1; col <= xls.ColCount; col++)
    {
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

Let's study this by our rules. First of all, let's see the ending conditions of the loops (marked in red in the example). As explained, `xls.RowCount` and `xls.ColCount` are called for every iteration in the loop. This is not too bad for `RowCount` since `RowCount` is fast and it is the outer loop, but it is a huge problem for `Xls.ColCount`, which is not only slow as mentioned earlier, but also runs in the inner loop.

So if the file has 50,000 rows and the maximum used column is column 30, you are iterating $50,000 \times 30 = 1,500,000$ times. `xls.RowCount` is called 50,000 times (since it is in the outer loop) and `xls.ColCount` is called 1,500,000 times. And every time `ColCount` is called, `FlexCel` needs to loop over all the 50,000 rows to find out which row has the biggest column. **Using this example is the surest way to kill performance.**

So, how do we improve it? The first way could be, as explained, to cache the results of `RowCount` and `ColCount`:

```
XlsFile xls = new XlsFile("somefile.xls")
int RowCount = xls.RowCount;
for (int row = 1; row <= RowCount; row++)
{
    int ColCount = xls.ColCount;
    for (int col = 1; col <= ColCount; col++)
    {
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

But while this was a huge improvement, it is not enough. By our second rule, `xls.ColCount` is slow, and we are calling it 50,000 times anyway. This is better than calling it 1,500,000 times, but still 50,000 times worse than what it could be. The column count is not going to change while we loop, so we can cache it outside the row loop:

```
XlsFile xls = new XlsFile("somefile.xls")
int RowCount = xls.RowCount;
int ColCount = xls.ColCount;
for (int row = 1; row <= RowCount; row++)
{
    for (int col = 1; col <= ColCount; col++)
    {
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

So, for now we have fixed this code to rules 1) and 2). We cache the end conditions in the loop, and we avoid calling `ColCount` much. (A single call to `ColCount` isn't a problem). But still, this code can be incredibly inefficient, and this is where rule 3) comes to play.

In this example, where our spreadsheet has 50,000 rows and 30 columns, we are looping 1,500,000 times. But do we really have 1,500,000 cells? In most cases, we don't. Remember that **ColCount returns the maximum used column in the sheet**. So, if we have 10 columns, but we also have some helper cell at row 1 column 30, `ColCount` will return 30. But the loop will run as if every row had 30 columns, looping through millions of empty cells. If we only have 10 columns except for row 1, then we have $10 * 50,000 + 1$ cells = 500,001 cells. But we are looping 1,500,000 times. Whenever you use `ColCount` for the limits of the loop, you are doing a square with all the rows multiplied by all the used columns, and this will include a lot of empty cells.

We can do much better:

```
XlsFile xls = new XlsFile("somefile.xls")
int RowCount = xls.RowCount;
for (int row = 1; row <= RowCount; row++)
{
    int ColCountInRow = xls.ColCountInRow(r);
    for (int cIndex = 1; cIndex <= ColCountInRow; cIndex++)
    {
        int col = xls.ColFromIndex(cIndex);
        DoSomething(row, col, xls.GetCellValue(row, col));
    }
}
```

Note that even when `ColCountInRow(r)` is fast, we still cache it in the `ColCountInRowVariable`, as it doesn't make sense to call it in every column loop just because. And as the used columns in every row will be different, we need to call it inside the row loop.

This last version will run millions of ways faster than the first, naïve version.

Reading Files

Ignore formula texts if possible

By default, when you are reading a cell with `xlsFile.GetCellValue` and the cell contains a formula, `FlexCel` returns a `TFormula` class that includes the formula value and text. But returning the formula text isn't too fast, because `FlexCel` needs to convert the internal representation into text for each cell.

If you don't care about the formula texts (`"=A1 + 5"`), but only about the formula results (`"7"`), and you are reading huge files with thousands of formulas, setting:

```
xlsFile.IgnoreFormulaText = true
```

before starting to read the file can improve the performance a little.

Note: This particular tip won't improve performance a lot, and it can make the things more complex if you ever start needing the formula texts. Don't use it unless you really need to.

Reports

Reports are by design a little slower than the API, because they must read a template before, parse it and compile before filling the cells. So if you need ultimate performance, don't use reports, use the API directly. But remember; this advice is similar to "If you need absolute performance use assembly and not .NET". Yes, you will get the most performance in assembly or the API than in .NET or Reports. But it most likely won't be worth the flexibility and ease of use you will get by using the later options.

Avoid too many master detail levels

In most cases, the FlexCelReport overhead is virtually zero, as the code in FlexCel was designed from the start to be very efficient when running reports. But there is a case where you can see visible performance degradation and that is when you have many nested master-detail reports.

The problem here is that FlexCel works by inserting ranges of cells. If you have a template with a single range, it will insert the rows needed for every record in the database at the beginning, and then fill those rows. A single insert of n rows, this is very fast.

But if you have a master detail report, it will first insert all the rows needed for the master, and then, for every record in the master, it will insert all the rows needed for the corresponding detail. If that detail itself has other detail then the inserts grow a lot.

Normally, for more than 3 levels of master-detail, you shouldn't have very big reports. Most of the times this is ok, since when you have huge reports, normally they are simple dumps of a database, that will be used for analyzing the data because nobody is going to print or read a million row report anyway. Complex reports designed to be printed or read normally are smaller, and for small reports you don't really need to care about performance.

Decide whether to use LINQ or Datasets as data sources

FlexCel provides two different ways to access the data: `IQueryable<T>` objects and datasets. In theory, `IQueryable` objects can be a little faster because they don't load all the data in memory, fetching it as it is needed. But in practice, there is normally not a real performance difference unless you are really memory bounded. In fact, datasets can be faster than `IQueryable` objects because all the data is just fetched once from the database, and once that is done you don't hit the database anymore. And datasets are extremely efficient at fetching tables from databases.

As always, the best way to know what is faster in your cases might be testing. You might expect LINQ datatables to be a little faster with huge datasets with no master details, and datasets to be faster in small reports with lots of nested master detail levels.

Load tables on demand when using Datasets

When using DataSets in the common way, you need to know in advance what tables are needed for the report, and load them before the report is run. Something like this:

```
FlexCelReport.AddTable(LoadTable(table1));  
FlexCelReport.AddTable(LoadTable(table2));  
FlexCelReport.Run();
```

This is ok if you are going to need both table1 and table2 in the report, but in more complicated cases where you don't know in advance which tables might be used it can be a waste of resources. You might be loading table2 from the database, and only using table1 because the user checked a checkbox in the application for a "short" report that doesn't need table2.

If you are using LINQ this isn't a real problem; you can AddTable() as many tables as you think you might need, and the data will be fetched from the database only when that data is actually requested. But with datasets, all data is preloaded before running, so all tables will be queried from the database.

To solve this problem, FlexCel offers "on demand" loading of datasets through the **LoadTable** event. Instead of using AddTable, you can assign the "LoadTable" event in the report and only load the tables when they are actually requested by the report. For example you could call:

```
FlexCelReport fr = NewFlexCelReport();  
fr.LoadTable += new LoadTableEventHandler(fr_LoadTable);  
fr.Run(...);
```

and define the event "fr_LoadTable" as:

```
void fr_LoadTable(object sender, LoadTableEventArgs e)  
{  
    ((FlexCelReport) sender).AddTable(e.TableName,  
        GetTable(e.TableName), TDisposeMode.DisposeAfterRun);  
}
```

You could also use DirectSQL or User Tables to avoid preloading the datasets. And we will repeat: This only applies to datasets. When using LINQ, data is always fetched from the server when (and if) it is needed.

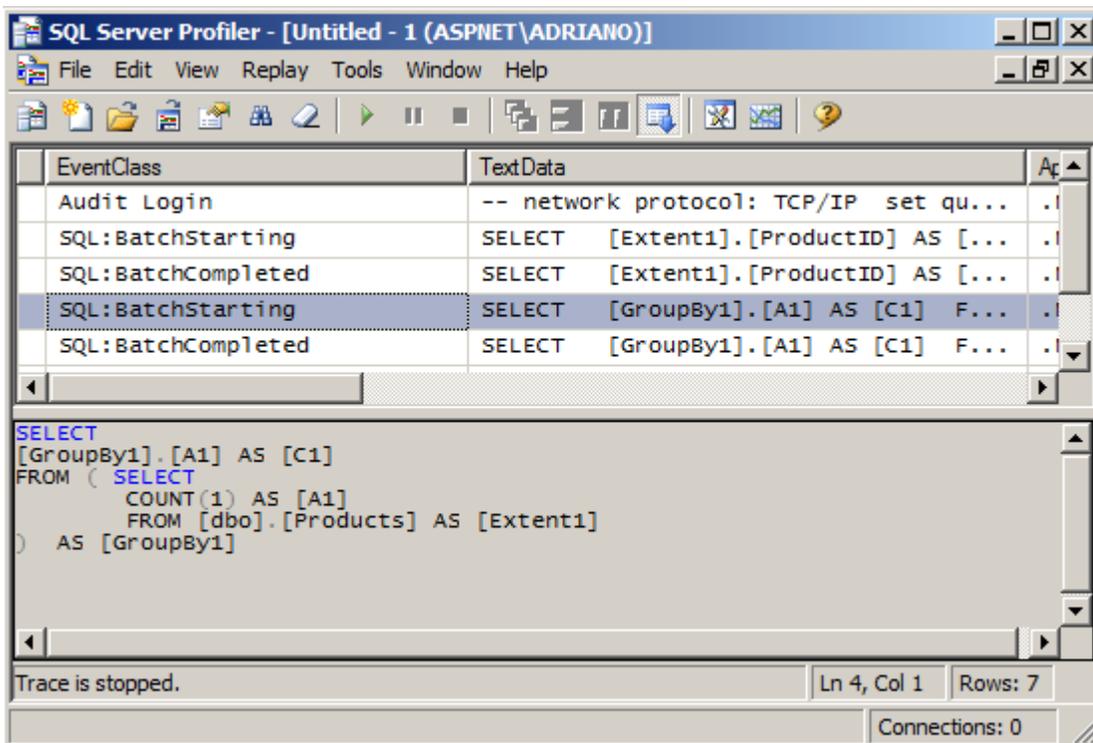
Profile the SQL sent to the server when using Entity Framework

When using IQueryable<T> datasets as datasources for the reports, SQL is generated on the fly and sent to the database. While this is good because you are not loading the datasets fully in memory, if you don't have the correct indexes in the database or if the queries are not correctly made, you might have a big performance impact. Sometimes the full table might be loaded from the database and then filtered locally. Make sure the SQLs sent have the correct "Where" part.

If you are using SQL Server, a simple way to check this is to run the SQL profiler in the server while running the report.

You should see two queries for every band, the first to query for the number of records in the band, and the second to retrieve the values. FlexCel needs the first query to know how many rows to insert in advance, since inserting rows in Excel is a costly operation.

The following example shows a typical log from a single table:



You can see the "Select count" statement highlighted in the screenshot above.

Make sure that the SQL sent to the server make sense. For some tables that have the same data repeated over and over again it might make sense to cache the results.

Provide the Record Count if possible when using LINQ Datasets

FlexCel does its reports by inserting rows and columns. As inserting a row or column isn't a cheap operation, it tries to minimize the number of inserts. So, if you have a table with 1000 rows, FlexCel won't do 1000 inserts of one row each; it will ask for the row count to the database, and insert 1000 rows in a single pass.

For datasets this isn't an issue, as the dataset has all data in memory, the row count is a very fast operation, just returning the length of the internal array that holds the data in memory. But when using LINQ datasets, this is not always the case. It is ok for `List<...>` collections since again "Count" will be called and that is fast. But in Entity Framework an SQL "Select count * from..." will be sent to the server, and in the worst case LINQ might need to iterate through all values in the `IEnumerable` to get the count.

If the collection that holds your objects is slow to compute the count, and you know the count anyway, you can tell FlexCel the record count by having a field "`__FlexCelCount`" in your business object. For example:

```
Class MyBusinessObject
{
    ...
    public int __FlexCelCount{get;}
}
```

If FlexCel finds a column named "`__FlexCelCount`" in your collection, it will assume this is the value of the row count, and it won't call "`Count()`" in the collection.

Please note that this is a last resort optimization. Count should be fast anyway for most cases, and by adding a "`__FlexCelCount`" field to your business objects, you will probably just making them more complex without any real gain. We provide this for really special situations, but this optimization isn't something that should normally be done.

Avoid Microsoft Distributed Transaction Coordinator (MSDTC) when using Entity Framework

When using entity Framework as data sources, you normally need to run the report inside a “**Snapshot**” transaction to avoid data being modified while the report is running. If you don't open the database connection before running the report, transactions might be promoted to distributed transactions, as the connection will be opened and closed many times while running the report. When the connection is opened a second time inside the transaction, it will be promoted to distributed.

See

<http://www.digitallycreated.net/Blog/48/entity-framework-transactionscope-and-msdtc>

for more information.

Prefer snapshot transactions to serializable transactions

When reading the data for a report, there are two transaction levels that can be used to avoid inconsistent data: **snapshot** or **serializable**. While both will provide consistent data, “serializable” provides it by locking the rows that are being used, therefore blocking other applications to modify the data while the report is running. “Snapshot” transactions on the other side operate in a copy of the data, allowing the other applications to continue working in the data while the report is running.

If your database supports snapshot transactions, and your reports take a while to run, and other users might want to modify the data while the report is running, snapshot transactions are a better choice.