# MDM Architecture

## General



## Network Traffic Considerations

Customer installation may include multiple locations



Multiple Application Servers



# Performance Considerations

When developing on your own machine, we are making local calls to local IIS as well as in many cases local database. Our databases do not contain as much data as our customers. Therefore, some of the coding decisions we make seem to not carry any performance issues. Almost 100% of the performance issues are found by the customer, in some of these issues it brought customer’s systems to a halt. If not corrected quickly, customers start looking for a different vendor.

## Main application vs. supporting application

MDM is always used side-by-side with another Merge product. We are used to assist with the main functionality. If RIS as an example got all the data and user has to wait for MDM to perform a task, we are no longer helping/assisting, we have become an interference for their current process.

So MDM must especially pay attention to Seamless integration

* User-friendly design
* Intuitive workflows/easy to learn/easy to use
* **PERFORMANCE**

## Performance Breakdown

#### Database

The full list of database tuning is quite large. And the purpose of this document is not to provide generic database tuning training. Books are written on database tuning but this is the first level of performance considerations:

* Have you design your tables to handle large set of data
* Have you used the most optimal data type for your columns
* Are you operating (comparing) matching data types
  + Bigint and varchar can be compared without error but for each comparison a conversion will occur to bring both variables into the common data type first
* Using JOINS vs Subquery
  + SQL is more likely to construct more efficient execution plan for table JOINS
* Avoid using #temp tables (if you must use temporary tables, please make sure to obtain an authorization from team lead, explaining the pros/cons of the situation) I have not seen a reason to use temp tables in MDM.
* Table variables can be used for small data sets as it cannot be indexed
  + Define PK to be the field that you will use for searches and joins
  + It is not recommended to manipulate data or do non-key searches on the variable table
* ABSOLUTELY NO CURSOR QUERIES
* Learn to use CTE (Common Table Expression) to your advantage
* Operations on numeric values are faster than on strings
* Design your transactional databases with ability to always search and join on primary keys.
* INNER JOIN can be used to minimize the row count you need to operate on
* When designing indexes using multiple columns use MOST COMMON -> LEAST COMMON approach.

ETC.. ETC… ETC

#### Code

With fast machines, emphasis is less and less on code optimization, but compact, wisely designed code will execute faster.

Optimizers and compiles are doing an excellent job at translating our code into the most efficient machine code. However, convoluted statements might not translate to the most efficient code. Help compiler to help you! (did you know that it can also make your project to compile faster ☺ )

In general if your code is easy to read, it will be properly translated.

I will only mention a few items that I (usually) correct if I catch them

Example1

|  |
| --- |
| public void MakeItHappen(int i)  {  if (i==0)  {  //do lots of work here    }  return;  }  public void MakeItHappenWithoutExtraBraketsToDigInto(int i)  {  if (i ! =0 ) return;    //do lots of work here  } |

Why It matters? To compiler it doesn’t as it will do it automatically…

Readability:

* I have to scroll all of the way to the end of the method (potentially making sure I matched up to the correct bracket) to answer the question, what will happen if I pass a non-zero value? And the answer is.. NOTHING! This method only interested in the case where zero is passed.
* I have an extra bracket to match up

Example 2

|  |
| --- |
| foreach (DataRow dr in col.Records.Tables[0].Rows)  {  if (level == DocumentTypeRecord.eLevel.PATIENT)  {  if (dr["TagTypeID"].ToString() == "1" && dr["Value"].ToString() == accession)  {  Int64.TryParse(dr["ID"].ToString(), out id);  return id;  }  }  else if (dr["TagTypeID"].ToString() == "2" && dr["Value"].ToString() == accession)  {  Int64.TryParse(dr["ID"].ToString(), out id);  return id;  }  else if (dr["TagTypeID"].ToString() == "3" && dr["Value"].ToString() == accession)  {  Int64.TryParse(dr["ID"].ToString(), out id);  return id;  }  }  int tagtype;  int requestedLevel = (int) level;  foreach (DataRow dr in col.Records.Tables[0].Rows)  {  if (dr["Value"].ToString() != accession) continue;  if (!int.TryParse(dr["TagTypeID"].ToString(), out tagtype )) continue;  if (tagtype > 3) continue;  if (level == DocumentTypeRecord.eLevel.PATIENT && tagtype != 1) continue;  Int64.TryParse(dr["ID"].ToString(), out id);  return id;    } |

Think of the worst case scenario, how many times have we performed an operation of getting a “Value” column PER one ROW.

Note that that two consecutive “else if” statements perform almost the same function. Just in case I threw in a check for values greater than 3.

#### Network traffic

This is really the reason that I started this document. Because while we work on our development machines, I think we forget what it means to call Proxy.GetThis() 4 times.

Each time you want to request something from the server/database:

* A connection is being requested to the server.
* Connection is being opened
* Request is send to the application server
* Client then is “waiting for response”
* Application server, does similar actions but when connecting to the database server (Open Connection, send request, process response, close connection)
* Application server than wraps the response and sends it to client
* Client receives the response
* Closes the connection
* Returns to the calling method

While some might argue that this is still happens in the fraction of the second.. if you do it enough times, that turns into seconds, then… slowly into 10 seconds… etc

From one release to another, we are slowing down our system.

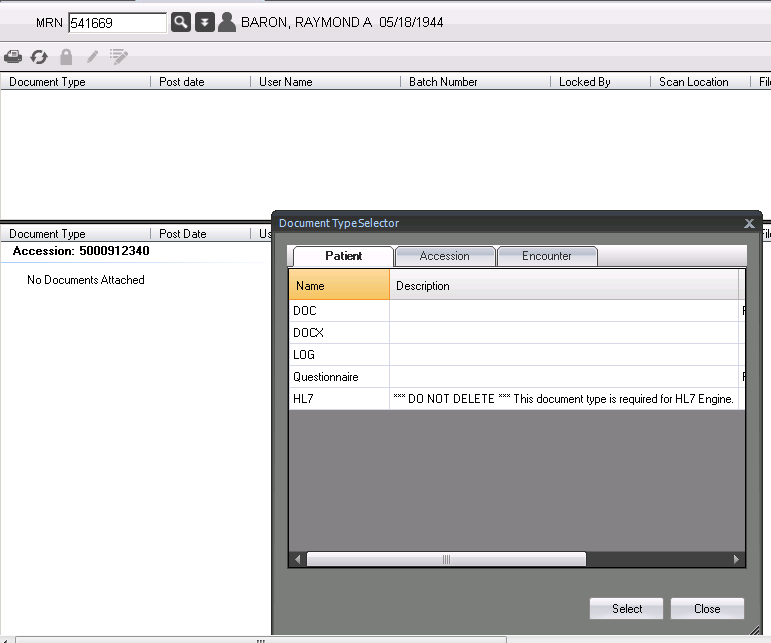
Always think network traffic, client-server connection handling, and efficiency of the communication.

USUALLY:

If you are using a method within the class that ends with “… Proxy”, you are most likely making a call to a server.

**EXAMPLE 1**

Document Type Selector form pops up every time user needs to import a document.

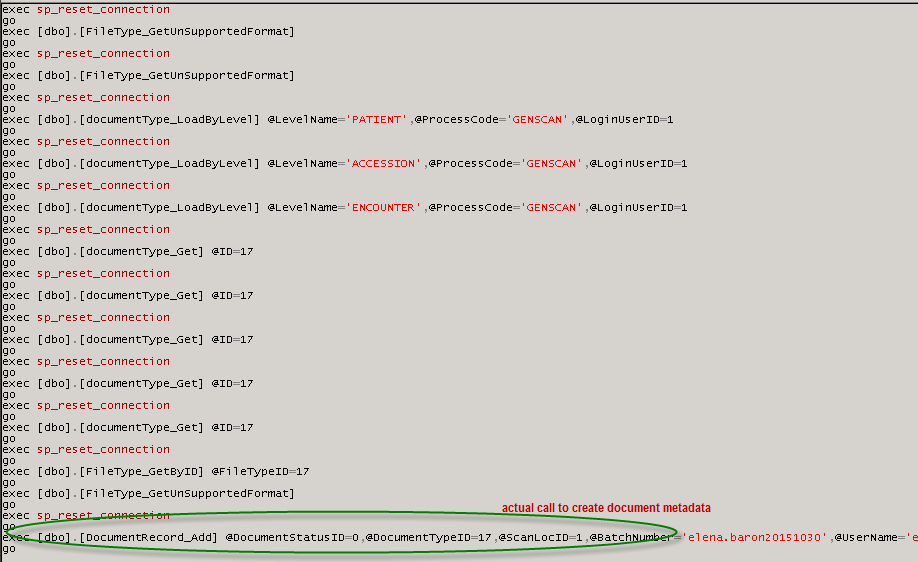


Document Import is one of the key functionalities within MDM (Being a document management system).

So this form will be popping up on user’s workstation thousand times per day.

It would make sense for us to make sure that it doesn’t create unnecessary traffic each time.

Turning on SQL Server Profiler shows the following



Calls to get unsupported formats x3

Call to populate 3 tabs with document types x3

Document Type Get (for selected type ID=17) x5

Now do that a thousand times with 250 users online.

Just the fact that we needed to call the same document type (17) so many times, suggest that we do not pay attention to surrounding code. Have we already received that information?

The whole purpose of the Document Type Selector is return you a DocumentType Record to be used for file import. Why did we need to load it 5 times?

Unsupported formats

From a different area in the code

|  |
| --- |
| Private Function GetOpenFileFilter() As String  If (FileTypeProxy.GetUnSupportedFileFormatCollection().Records.Tables(0).Rows.Count > 0) Then  Return FileAccessProxy.GetSupportedFileFormatFilter() + "|" + FileAccessProxy.GetUnSupportedFileFormatFilter()  Else  Return FileAccessProxy.GetSupportedFileFormatFilter()  End If  End Function |

The code below does the following:

* Establishes network connection and retrieves unsupported format collection
* Check if we have records
* Throws it away
* Establishes network connection and retrieves supported format collection
* Establishes network connection and retrieves unsupported format collection

This is an example of performance defect that will not be caught by an optimizer.

## Caching Data Decision Tree

I would divide data into 3 categories:

* Transactional (frequent changes)
* Support/Master records (in-frequent changes)
* Configurations (rarely changes)

Unsupported Formats are actually an example of rarely changing data. At this time it can be changed only via direct access to the database. Should we make 3 calls to retrieve almost-static-data?



## Dataset with multiple tables

How often the same queries are made as a set?

How large is the typical result?

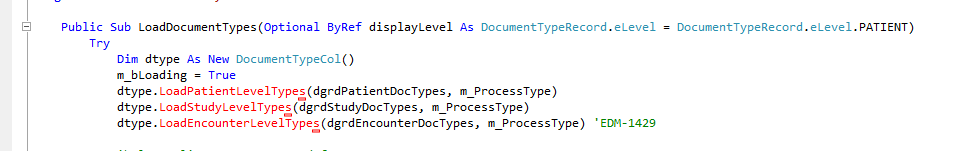
Are these queries utilize the same objects in the database?

Consider making one connection to retrieve a dataset with multiple tables inside vs. 3 connections to retrieve each table separately.

In case of Load Document Types by Level, it looks like we have one generic action calling these operations, and they are always called in a row. So it will never request a Patient level, without calling Accession & Encounter.

To investigate I have commented out the calls to find all of the affected area.





Modifying to load all levels at once (included into one stored procedure)

|  |
| --- |
| Public Function LoadByLevel(ByRef grdListPatient As DevComponents.DotNetBar.Controls.DataGridViewX, \_  ByRef grdListAccession As DevComponents.DotNetBar.Controls.DataGridViewX, \_  ByRef grdListEncounter As DevComponents.DotNetBar.Controls.DataGridViewX, \_  Optional ByVal processType As ProcessRecord.ProcessType = ProcessRecord.ProcessType.GENSCAN) As Boolean  Me.Clear()  Try  Dim col As DocumentTypeCollection = New DocumentTypeCollection()  col.ProcessCode = processType  col.DataRequest = DocumentTypeCollection.DatabaseRequests.GetAllLoadByLevel  col = DocumentTypeProxy.GetDocumentTypeCollection(col)  If col Is Nothing OrElse col.Records Is Nothing OrElse col.Records.Tables.Count <= 0 Then  Exit Function  End If  For counter As Integer = 0 To 2 'Patient= is the first table, Accession = 1 & Encounter = 3  If col.Records.Tables.Count < counter Then Exit For  Select Case counter  Case 0  PopulateData(grdListPatient, col.Records.Tables(counter))  Case 1  PopulateData(grdListAccession, col.Records.Tables(counter))  Case 2  PopulateData(grdListEncounter, col.Records.Tables(counter))  End Select  Next  Return True  Catch ex As Exception  Call PostError(m\_CONST\_PROJECTFILENAME, ex.Message, Err.Number, ex.StackTrace, ex.TargetSite.Name)  End Try  End Function |

# Final result

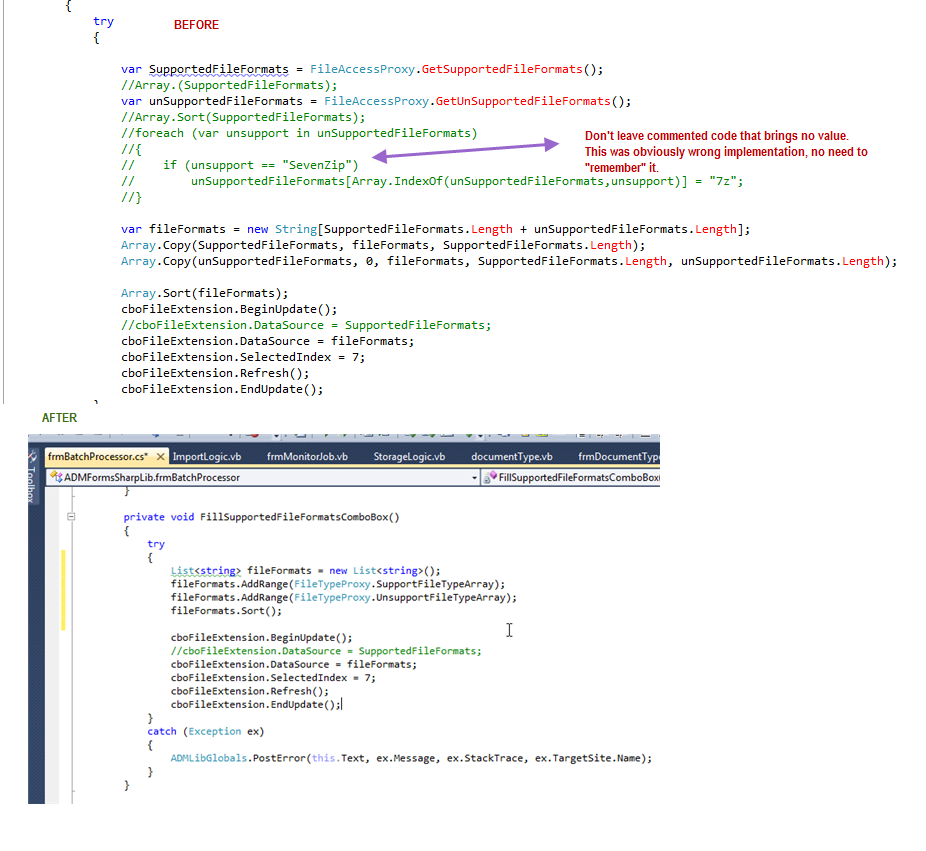
Selector does 3 calls (down from 12)

# 

# Other code preferences

## Use List<type> vs. array

Be wise about using “array”, they also can be expensive



## Make comments useful

Don’t leave commented out code… If you think it might be “needed”, write a comment why this code is “kept”

If you have to re-design your previous implementation because it was wrong, don’t “treasure” incorrect code.. delete it.

## Var –don’t use it

There’s no benefits to using “var”, besides problems reading code. (I found myself scrolling down the screen to figure out what will be variable type.. I have to then.. track down the assigning method return value to determine it).

Compile will evaluate the variable type at the time of compilation. It is not “variable” type, as you are not able to switch the type at a later time. It will assume the data type based on the first line of code where it is being set, and it will enforce that type for the remainder of the code.

If you need “variable” type (which I doubt you will ever need), use “object”. Otherwise, don’t hide from me the type of the variable we are using.. I WILL find out!

Knowing the type, allows clearly evaluate performance pros/cons. Example, comparing to vars .. doesn’t tell me the efficiency..

But when I see (string == string) or (int == int) or (object == object).. I know what it costs me and what it really means (value vs. reference comparison).

## Don’t Repeat Code

Calculation on supported/unsupported formats vs. file extensions.. will be the same in all of the import routines.

If you finding yourself copy/pasting or repeating code, think of the single place you can stored it as a function…

If you need to make a change.. you will have only one place to worry about..

The more code you re-use.. the smaller your executable size, the faster your program.

The less time you spend next time when logic changes to track down ALL of the places you need to modify.

DO NOT EVER repeat your code!

(P.S. it was too labor intensive for me to re-write this .. but I noticed that Batch Processor does its own file format validation, as is document type selector, etc)..

## String Manipulations

These are string operations, and they do carry a cost.. (see “string is immutable” in C#)

Every time you request a manipulation to a string you creating a new string.

All operations that seem to "change" a string instead produce a new one.

There’s nothing wrong with string manipulations.. we NEED to do it.. but if you have done it once.. re-use it



Above code performs the same changes 3 times.. creates 3 more strings.. that are then thrown away.

In some cases StringBuilder is a good choice for “building” a string.

If you comparing items.. use string.compare to specify “IgnoreCase = true”