empath-e
MAKING CRM MAKE SENSE

Tech Bulletin
19th March 2009
Updated: Nov 2009

www.empath-e.com/support

Issue

Indexes used within SalesLogix are not adequate for installations that have upgraded from earlier (pre v7.x) releases. SalesLogix performs sub-optimally and the database size is roughly 50% larger than it needs to be, due to compound and duplicated indexes.

Update: November 2009 - Sage have released index scripts for SalesLogix v7.5.2 and you should merge the changes from these scripts into your index handling strategy as enclosed.

Reason

In our experience, when taking on new customers that are already using SalesLogix, we have seen many complaints about general SalesLogix speed. After careful analysis, we have ascertained that this reduction in speed is due to the indexes (or lack of) on the primary entities and sub-entities. We also noticed that the removal of some of the older compound indexes have also reduced database size considerably. By removing all indexes and starting with a baseline of new indexes (and adding – or removing, where necessary, indexes) we can improve performance considerably.

An index is used by SQL Server (and Oracle) to ensure that data is retrieved in a manner that does not result in an entire data table being scanned for content. For example, if you require all accounts starting with "Con" then it would be wasteful to scan the entire table looking for those entries, especially if the number of accounts in the database is very large. By adding an index – SQL Server can locate only the rows matching this condition and bring those back very quickly – it does not have to search the entire table in order to find them. However, adding an index has a negative effect too – when a row is inserted, SQL Server has to account for the insertion of the row being added **and** modify all indexes that span this new row. Whilst this typically happens in milliseconds – it is still worthwhile bearing in mind when considering optimal index usage. Add indexes to aid read/search performance but not at the expense of write performance if possible.

The judicious analysis and maintenance of indexes can have a massive positive effect on your SalesLogix system! Enclosed in this document, you will find methods and tools to assist in this process but only effective analysis by you or your business partner will determine which indexes should be added or removed – there is, unfortunately, no magic bullet to this process!

Resolution

In order to work on index usage there are several tools required and these are:

- SalesLogix Architect
- Enterprise Manager/SQL Studio
- SalesLogix Profiler

All of these tools are provided with SalesLogix Admin and SQL Server.

NOTE: YOU SHOULD ENSURE YOU HAVE A WORKING & TESTED BACKUP PRIOR TO RUNNING ANY OF THESE PROCEDURES. EMPATH-E LIMITED CANNOT BE HELD RESPONSIBLE FOR ANY LOSS OF DATA HOWSOEVER CAUSED. THIS INFORMATION IS PROVIDED AS IS AND NO WARRANTY EXPRESS OR IMPLIED IS PROVIDED.

---

1) Check the size of existing tables/entries

The first step that we take is to ascertain the current index sizes. This gives us a base-line to concentrate our efforts. Tables that show a large number of rows and large index_size values should be the areas to focus on at the beginning of this process. You can then narrow your search to other tables later on.

This can be done via the following script with Query Analyser:

```
EXECUTE SP_MSFOREACHTABLE 'EXEC SP_SPACEUSED''?'''
```

The output from this will show the tables/indexes that are occupying the most space:

	name	rows	reserved	data	index_size	unused
1	ACCOUNT	4485	10160 KB	4552 KB	4296 KB	1312 KB

From here, you can ascertain the tables that contain the largest sizes as these will be good candidates for optimisation.

2) Establish tables with no indexes

On occasion, we have found databases with no indexes on primary/sub tables. You can check for this using this script:

```
SELECT sysobjects.name TableName,
    (SELECT rows FROM sysindexes
     WHERE id = sysobjects.id
       AND indid = 0) Rows
FROM sysobjects
WHERE type = 'U'
  AND OBJECTPROPERTY(sysobjects.id , 'TableHasIndex' ) = 0
```

This will report any tables that do not have any indexes at all. Make a note of these tables to address later.

3) Remove all current indexes

If you have upgraded a database from v2/3/4/5 to v6/7 then your indexes are probably incorrect and no longer relevant to what is now going on within the system. Rather than analysing each one individually it is simply best to delete them all and re-start. Dropping all indexes should be done via Enterprise Manager/Studio rather than via a script just to err on the side of caution.

4) Re-Create the standard indexes

SalesLogix v7.x and above now contains a set of "good" indexes upon which to base your new indexes. We recommend you create the new indexes based on the 7.2/7.5 indexes. These can be found on the SalesLogix DVD in the **\Databases** folder, called **Sample MSSQL Indexes.sql**

5) Add back missed indexes

Despite the indexes above being much better, we have found the following to be beneficial and should be added to aid system performance:

```
CREATE NONCLUSTERED INDEX [ACCOUNT_ACCOUNTMANAGERID] ON [sysdba].[ACCOUNT]
(
      [ACCOUNTMANAGERID] ASC
) ON [PRIMARY]

CREATE NONCLUSTERED INDEX [CONTACT_ACCOUNTMANAGERID] ON [sysdba].[CONTACT]
(
      [ACCOUNTMANAGERID] ASC
) ON [PRIMARY]

CREATE  INDEX [IX_USEROPTIONS3] ON [SYSDBA].[USEROPTIONS]([NAME],
[CATEGORY], [USERID]) ON [PRIMARY]
CREATE  INDEX [IX_PROGRAM] ON [SYSDBA].[SECFUNCTIONS]([PROGRAM]) ON
[PRIMARY]
CREATE  INDEX [PICKLIST_ID] ON [sysdba].[PICKLIST]([ID] ASC) ON [PRIMARY]
```

6) Fix old issues

If you are upgrading old databases then the following will "tidy" up some old defects and remove orphaned rows/data

```
---Delete Old AdHocGroup
DELETE FROM ADHOCGROUP WHERE GROUPID LIKE 'Q%'


-- Delete non-used groups & templates
DELETE FROM sysdba.ADHOCGROUP WHERE GROUPID IN (
SELECT P.PluginID
FROM
  sysdba.Plugin P
  LEFT JOIN sysdba.Plugin C ON C.BasedOn = P.PluginID
WHERE
  P.BasedOn IS NULL AND
  P.Type in (8,23) AND
  C.PluginID IS NULL AND
  P.UserID IN
  (
    SELECT USERID FROM sysdba.UserSecurity WHERE Type = 'R'
  )
)

DELETE FROM sysdba.PLUGIN WHERE PLUGINID IN (
SELECT P.PluginID
FROM
  sysdba.Plugin P
  LEFT JOIN sysdba.Plugin C ON C.BasedOn = P.PluginID
WHERE
  P.BasedOn IS NULL AND
  P.Type in (8,23) AND
  C.PluginID IS NULL AND
  P.UserID IN
  (
    SELECT USERID FROM sysdba.UserSecurity WHERE Type = 'R'
  )
 )
```

7) Not using SpeedSearch ?

We often find companies start out using SalesLogix SpeedSearch but soon forget about it or don't use it at all. If this is the case, and S/Search was originally configured then you will find that the INDEXUPDATES table is potentially huge. This is because S/Search is not running and periodically clearing this table.

In situations like this, it is better to switch off this functionality and put it back later if required. By doing so, the internal triggers are also removed and SalesLogix benefits from a considerable speed advantage too.

Disable the triggers like so:

```
-- Disabling Triggers
UPDATE SLXTRIGGERS SET ENABLED = 'F' WHERE PATH = 'SLXSEARCHTRIGGER.DLL'
```

You can then remote the data (which will **never** be used and will just continue to accumulate) by truncating the table:

```
TRUNCATE TABLE INDEXUPDATES
```

This then stops speed search and clears out the space used. You can always re-configure it later if required.

8) Re-Index All Indexes

Just to ensure everything is in order – it is now best to execute a full re-index on all indexes in the database. This can be performed using the following method:

```
-- SQL 2000
EXEC SP_MSFOREACHTABLE @COMMAND1="PRINT '?' DBCC DBREINDEX('?',
'',0,SORTED_DATA_REORG)"

-- SQL 2005
EXEC SP_MSFOREACHTABLE @COMMAND1="PRINT '?' DBCC DBREINDEX('?', '',0)"
```

You are now clear of any issues and able to shrink the database, you should find a considerable difference in database size and general performance!

The remainder of this document will discuss how to further optimise your SalesLogix database by using other tools to identify potential issues and how to resolve them.

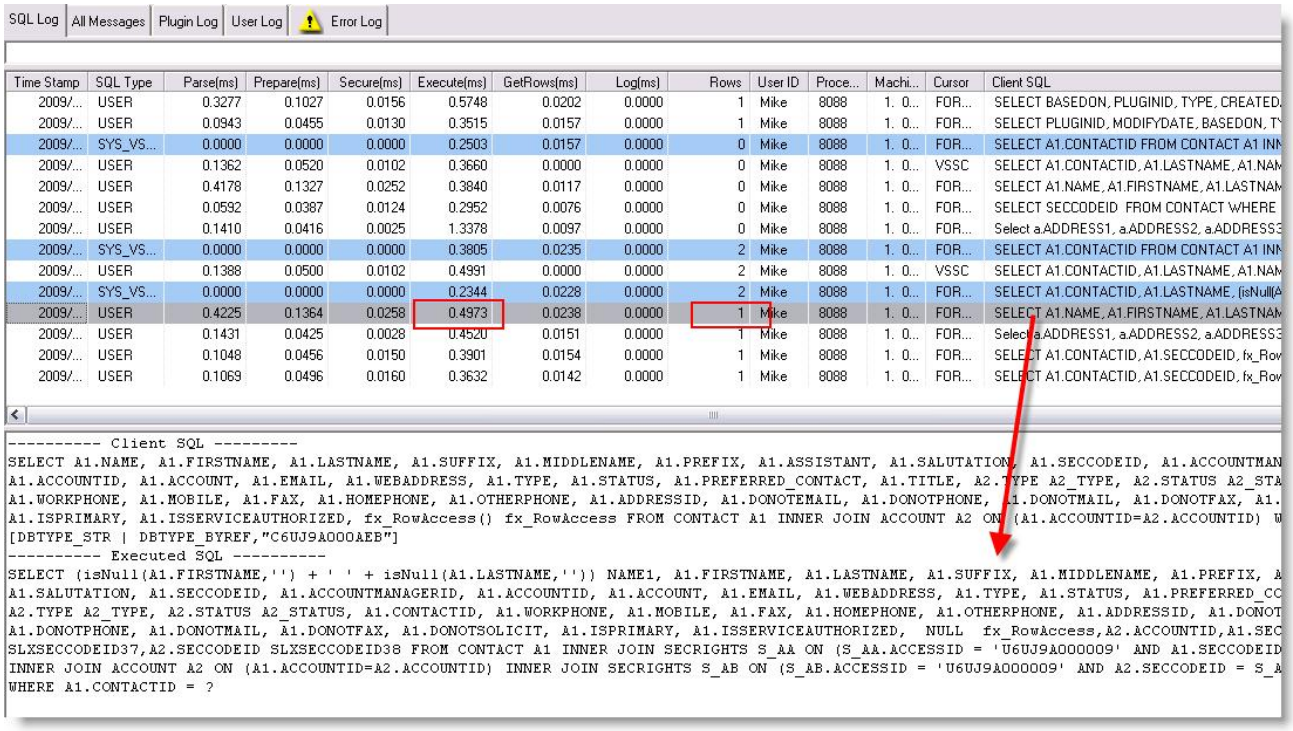
If you are using remotes/remote offices – these will need to be re-generated at the next available maintenance period to also take advantages of the changes made. All the changes we have made are not "sync-aware" so your remotes will still be operating inefficiently until they are re-built.

9) Establish other indexes and benefits

By following steps 1-8 you should now be in a position to start to analyse the database properly for index utilisation. You should start by running SalesLogix and performing normal operations whilst **SLXProfiler** is running. Then, take the SQL generated and pass this through Query Analyser and press CTRL-L to establish the query plan. You can then check for non-index usage and table-scans to see whether adding an index would be beneficial.

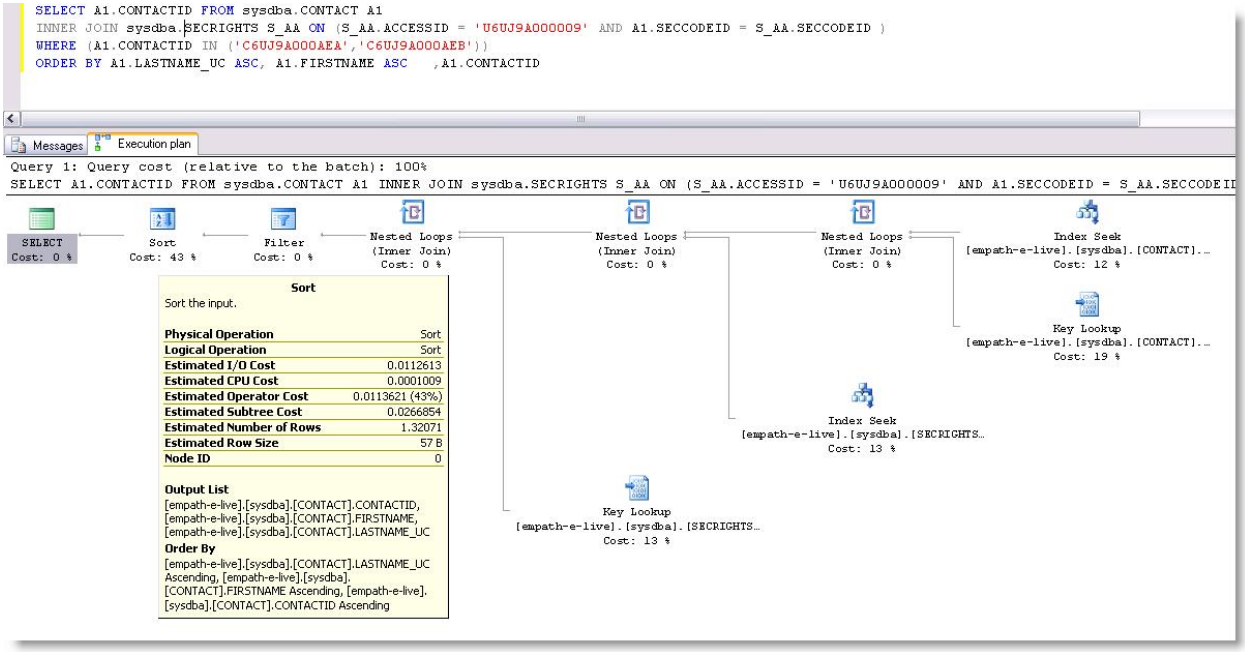
9a) Using SLXProfiler

- Run SalesLogix and pause at the login prompt
- Locate SLXProfiler.exe in the \Program Files\SalesLogix folder.
- Press the SalesLogix Logo Ball and it should show you a list of running processes (that use the SalesLogix Provider)
- Select the SalesLogix.exe process
- Press Pause
- Now, immediately prior to the operation you wish to test – Unpause and you will see the SQL requested **and** the SQL sent to SQL Server – you can run the executed code in Query Analyzer.
- Also, whilst in here you should check the Execute(ms) column and the Rows column as these tell you whether a particular query is taking a long time to execute and returning more rows than expected.



9b) Use Query Analyser

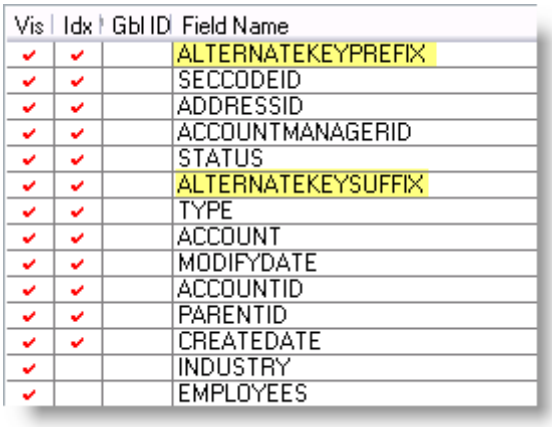
Paste the query found from the profiler into Query Analyzer/Studio and press **CTRL-L** – this executes the query and shows how the SQL Server Optimiser has decided to execute that particular query:



Here, you can see the selection used indexes to locate the data and the SORT operation was the most costly.

By carefully looking at the SQL SalesLogix generates and the use of indexes used it is fully possible to add/remove indexes to ensure that the SQL Query Optimiser returns data in the fastest possible method. In particular, any queries that are not using indexes correctly may show a table-scan being performed – this is a very slow method of data retrieval. However, you should note that table-scans will still be performed on tables with very few rows - as using an index would not assist in terms of speed.

9c) Remove indexes not in-use



By default, there are many indexes added to a standard SalesLogix system and we have re-introduced some of them in step 4. You should use Architect to identify indexes (sort by index column) and ascertain whether you can remove indexes that are potentially not in use.

For example, if you are not using the campaign/ticketing systems – you may wish to remove the indexes on the main Account/Contact and Opportunity tables that pertain to them. This has the advantage in that it removes space, ensures that SQL does not have to maintain those indexes and will also speed up any insert operations. Here, you can see the account table indexes – the highlighted items show indexes will be maintained for 2 columns used for the Support module. If you are not using this, then these are ideal candidates to be removed. However, if you do start to use Support then you should remember to re-establish these indexes in the future.

10) Index Types & Attributes

Whilst the definition of all indexes, their usage and implementation is outside the scope of this bulletin the following should assist when determining types of indexes available and how they are used.

In essence, a table that is not indexed is populated on a LIFO basis – the content is simply appended to the end of the table (for the purists, this is not strictly true obviously!). As data is requested – it is searched for until found. This is known as a **table-scan** which, as you can imagine, has a performance penalty on caching, disk I/O etc. but is, conversely, the fastest way to write the data – the row is written and that's it.

However, in 90% of the operations in SalesLogix, the data is being **read** via SQL Server and SalesLogix is normally very specific about what it needs (a WHERE or JOIN is pretty much always present). Therefore, by adding an index, a table-scan is avoided and an index is used to locate only the data requested.

SIMPLE INDEX

DATA

ROWID	COL1
1	ZZA
2	BBB
3	BBA
4	CCC
5	CCE
6	DDD
7	EEEE

INDEX

ROWID	COL1
3	BBA
2	BBB
4	CCC
5	CCE
6	DDD
7	EEEE
1	ZZA

Here we can see that the data (COL1) is written in a random format. However, the index on COL1 is now representing the sort order of the data (alpha/ascending) as well as maintaining the location of the row.

Whilst RowID is shown in this example, it is not part of the data set.

If we now performed a `SELECT * FROM TAB ORDER BY COL1` the index can be used (internally, it would return 3,2,4,5,6,7,1). In other words, without needing to re-sort the data – it's already in order (via the index). The data is then looked up from here.

Similarly, if we performed `SELECT * FROM TAB ORDER WHERE COL1 = 'BBB'` then the index is scanned – rowid 2 is identified and the pointer back to the row in the table is ascertained. Even a query such as `WHERE COL1 IN ('ZZA','CCC')` would still only need to scan the index – it can "jump" to the beginning of the C's, locate the rowid and, as soon as it hits a D, it knows the data has been found (and can "escape" the index search at that point). Where this index would fail is if we did `WHERE COL1 LIKE '%d%'` – as the index cannot work on the double-wildcard. A table-scan is performed in this instance.

The above shows a normal index (non-clustered). A **clustered index** is different in that the data is **physically ordered** on the disk in the required index order. Taking the same example above this would be the result:

DATA

ROWID	COL1
3	BBA
2	BBB
4	CCC
5	CCE
6	DDD
7	EEEE
1	ZZA

INDEX

ROWID	COL1
3	BBA
2	BBB
4	CCC
5	CCE
6	DDD
7	EEEE
1	ZZA

In this instance, as the data is written **ALL** rows are re-ordered on the disk to ensure the data is kept in order. The rows are then held together and if we inserted a new row (BBZ for example) then the table is "split" at row 3 and the new row is inserted. As you can imagine, this has a huge impact on write performance and index maintenance. Its main advantage is very fast row retrieval on similar data rows. All clustered indexes in SalesLogix are based upon the ID – as this is unique and does not require the insertion of new rows into the middle of an existing "page" of rows. Typically, they would be inserted at the end of the table. There are arguments also as to why this is inefficient use of a clustered index but that's beyond the scope of this document!

UNIQUE/NON-UNIQUE – when creating an index you can specify where it should contain unique or non-unique values. A SalesLogix ID is (or should) always be unique so this is a good candidate for switching on this attribute when creating the index. If, somehow, a duplicate value is created then the query (INSERT/UPDATE) will fail and be reported to the user.

NULL, NOT NULL – this ascertains whether this column should contain data (or is allowed to be null). In the example of the SalesLogix ID – again, NOT NULL should be chosen to avoid rows being created where the primary key is blank (through bad coding, errors for example)

COMPOUND INDEXES – this is an index that contains several columns. Earlier, we created one of these:

```
CREATE  INDEX [IX_USEROPTIONS3] ON [SYSDBA].[USEROPTIONS]([NAME],
[CATEGORY], [USERID]) ON [PRIMARY]
```

In this instance – the index IX_USEROPTIONS3 is made up of the columns NAME, CATEGORY and the USERID. The reason for this is that there are many instances in SalesLogix where it needs to find a particular user option and does so via a query like:

```
SELECT COL1 FROM USEROPTIONS WHERE NAME = 'MyOption' AND CATEGORY =
'MyCategory' AND USERID = 'UXXXXXXXXXX'
```

By creating IX_USEROPTION3 we have ensured that the SQL Server Query Optimiser only needs to traverse **one** index to find that information. Ordinarily, it would have potentially reverted to a table-scan or used three separate index searches to find the relevant row.

You cannot create Compound indexes with Architect – only simple indexes can be created. You need to add them manually and, if necessary, send them to remotes as well.

NOTE: Do **not** be tempted to add too many columns to an index! Not only will this have an adverse affect on performance it is unlikely you will be able to find a "covering" index that satisfies all queries. Additionally, this index has to be maintained each time a row changes (as every column needs to be checked and the index updated accordingly). The **index_size** is also increased to account for this scenario – leading to space usage that has a negative effect to database size and performance is only slightly increased (due to the infrequency of index hits).

When checking queries – you should look for these icons. A **seek** is much better than a **scan**. You can hover over the query plan and it will describe the action and its relevant cost to the whole query.

SUMMARY

As can be seen – it is very possible to make changes to indexes that have very positive effects on the running of SalesLogix. By careful examination, continual "spot" checking and the use of the tools provided you should be able to create well performing indexes, but without the expense of sacrificing write-performance.

It is unlikely you will get this right first time – use the tools provided with SalesLogix and SQL Server to monitor the performance of your system over time and do not be afraid to reverse out changes you make initially – there is no "ideal" set of indexes – only those that are ideal at the time!