How, when, and where to create MS SQL indexes in Relativity

Scott R. Ellis
Client Services - Infrastructure Engineering Architect

March 12, 2014
# Contents

1  The Database Tuning Advisor .................................................. 3
   1.1  Indexing Considerations .................................................. 4

2  Optimizing Queries ............................................................... 4
   2.1  Updating Statistics ........................................................... 4

3  Creating Indexes ..................................................................... 5
   3.1  Changing Configuration Values ........................................... 7
   3.2  Building and Optimizing a New Index ................................ 7
This document is a written copy of a webinar called “Managing and Creating Indexes in a Relativity Database” given in February of 2014. You can watch the recording of the original webinar here. Primarily, it is about how to add and maintain indexes in SQL.

This is a back-end task that is performed by database administrators (DBAs) or system admins who are seeking to improve the performance of a query. Please note that the following process is highly technical and can severely impact your Relativity environment if not performed properly. If you’d like some help proceeding, please contact support@kcura.com for assistance.

1 The Database Tuning Advisor

Often, people think of the database tuning advisor (DTA) as a “DBA in a can” to analyze performance issues with queries. While it can be helpful in some situations and when used properly, the DTA can introduce undesirable elements to your database. Here are a few examples:

1. Overly inclusive indexes
2. Duplicate indexes
3. Unwieldy indexes
4. Hypothetical indexes

If you have uncovered a problem resulting from the DTA, please contact our support team—we’d be happy to help resolve it for you.

To add some clarity to the DTA conversation, sometimes it may recommend placing an index on what is, essentially, a Relativity system table. Because some tables can be used very heavily during certain processes, our development team has tried to create indexes that will not cause contention or result in deadlocking. Adding indexes to tables other than the Document table may yield negative results. For example, if they contain the same columns as extant indexes but the sequence is different, they may create a deadlock between a select and an update query.

Duplicative DTA indexes are not uncommon, and can exist on system tables such as Field and Artifact. If you find duplicate indexes, it’s often best to disable them. If you do wish to continue to create indexes on these tables, my advice is to document the creation of them, and include the CREATE statement, along with the instigating query. It is important that, if the index causes trouble later on, you are able to understand the reason the index was created in the first place.
1.1 Indexing Considerations

Adding indexes to SQL requires a lot of caution, and, as a best practice, you should not be using the database tuning advisor to do this for you. It is notorious for making bad index recommendations and then giving the index a very generic and unhelpful name.

If you have created indexes in the past using the DTA, you should check for hypothetical indexes:

```
SELECT * FROM sys.indexes WHERE is_hypothetical = 1
```

2 Optimizing Queries

There are three basic query optimization methods you can perform to fix the three most common problems we see in Relativity queries:

1. Bookmark/key lookups
2. Non-clustered index scans v non-clustered index seeks
3. Clustered index scan v Clustered index Seek

First, there are a couple of additional things to keep in mind. Users make frequent changes to the data in the document table so, sometimes, a query plan can get stale.

2.1 Updating Statistics

A “stale” query plan simply means that it has old statistics. Statistics are quantitative histogram data that SQL keeps that help the query algebrizer understand where data is, how much of it is there, and how best to formulate an efficient retrieval plan, also known as a query plan.

Because stats can be the potential root cause of a problem and they are created when an index is built, a user may believe that his index actually helped the situation, when in fact he created a duplicate index. It just had newer statistics, so SQL generated a better query plan. Always check to see if the statistics are up to date. One way you can do that is by simply comparing the estimated query plan to the actual. If the estimated query plan believes that 1200 rows will be returned, but the reality is that there were 800,000 rows, then your statistics are definitely out of date.

Always check to see if your stats are updating properly. It’s a Relativity SQL best practice to update stats with a FULL SCAN on a weekly basis. This is a rather intensive process, but can yield large dividends in query execution time reductions. There is a good article on this here that explains the inner workings of index statistics.
SQL provides an AUTO_UPDATE_STATISTICS function that gets set to ON upon database creation. It can only be turned off using the ALTER DATABASE command—so, if it is off on your database, someone turned it off. kCura recommends that this be enabled. You can check the status of all the databases on a particular SQL server with this query:

```sql
SELECT name,
    is_auto_create_stats_on =
    CASE
        WHEN is_auto_create_stats_on = 1 THEN 'ON'
        ELSE 'OFF'
    END,
    is_auto_update_stats_on =
    CASE
        WHEN is_auto_update_stats_on = 1 THEN 'ON'
        ELSE 'OFF'
    END
FROM sys.databases
```

When auto-update statistics is on, statistics for an index will be automatically updated after a certain amount of modifications have been made to the table. This happens on a schedule based on the number of rows in the table, and it is not something that can be configured.

To force an update of statistics against an entire table, you can run this statement:

```sql
UPDATE STATISTICS SchemaName.TableName
```

Execute this statement to update the statistics for a specific index. All indexes will have statistics; they are created automatically.

```sql
UPDATE STATISTICS SchemaName.TableName IndexName
```

3 Creating Indexes

Once you have verified that out-of-date statistics are not the problem, you’ll want to know if an index can actually help you improve the performance of your query. Let’s take a look at a search query, and walk through what the best approach will be.
When creating indexes on the Document table, be aware of whether or not the index will actually help solve the problem you’re trying to address. You should document the indexes you create as you go, and record any improvements in speed. Create a log for each database. You could even create a custom application to help you track this. That way, later on, if someone wants to remove an index, you can justify it. VARSCAT can be used to maintain a sort of inventory of searches—that is, how many times it was run, who runs it the most, and when was it last run.

**Note:** To help you identify poorly performing searches that may be negatively impacting your environment, visit the Code Exchange in the [Relativity Customer Portal](https://customerrelativity.com) and download VARSCAT.

Let’s assume you’ve captured the offending query. Maybe you grabbed it from the history tab, from a profiler trace, or you saw it turn up in VARSCAT results as a high-frequency, long-running query.

Let’s use this `SELECT TOP 1000` query as an example of a need to create an index. We’ll assume that this is a search or a view that is run often throughout the day.

```sql
SELECT TOP 1000
    [Document].[ArtifactID]
FROM
    eddsdbo.[Document] (NOLOCK)
WHERE
    [Document].[AccessControllistID_D] IN (1)
AND
    ((([Document].[EmailFrom] IS NULL OR NOT
         ([Document].[EmailFrom] LIKE N'% [msmith@kcura.com]%'))
    AND
    ([Document].[EmailTo] LIKE N'%Lbaker %'))
ORDER BY
    [Document].[DateSent] DESC,
    [Document].[EmailFrom],
    [Document].[ArtifactID]
-- CPU time = 499 ms, elapsed time = 24755 ms.
```
Every time a search in Relativity runs, three queries will run. We are mainly concerned with the SELECT TOP 1000 and the SELECT COUNT queries. We’re looking at the SELECT TOP query. You can get the COUNT query to appear in audits by changing the AuditCountQueries value in the EDDS configuration table to TRUE. By default it is set to FALSE. Sometimes, a SELECT COUNT query can have a very different query plan than a SELECT TOP 1000 plan. Here is some SQL that you may use in your test environment to make a change to this configuration.

3.1 Changing Configuration Values

**WARNING**: If you accidentally update all of the rows in your configuration table, you will bring down Relativity. Ensure that you follow these steps exactly.

A script is available in the Code Exchange on the Customer Portal that can help safeguard against accidental destruction of this critical table. Look for the The Reconfigurator in the exchange to run the script and perform this action.

After running The Reconfigurator, you suspect, based on all of the ORDER BY conditions, that your primary problem may be the sort. But you also see that there are a number of WHERE conditions. What then, will be the best approach to indexing this query?

3.2 Building and Optimizing a New Index

Slow down, and think through the problem. Here are some questions that need to be answered before building an index:

1. What does system load look like right now? Can I build it now or do I need to build it later? The index may take as long to build as the original query took to run. It may also put as much load on the system while it is building as the original, un-optimized query had.
2. How many users are affected?
3. How often is the query used?
4. How long does it take to complete?
5. What level of pain does it inflict on the server when it runs?
6. Does the index already exist and is it in use?

Check to see if you already have an index. This query will let you check the Document table of any workspace and see if the field you want to index already exists in an index. Sometimes, indexes get created and, while it may seem like the index you want to create is needed, the field already exists in a different index but, for one reason or another, the search query you are looking at isn’t using the index.
Here is a script that will let you see inside. It won’t tell you why the index isn’t being used, but it will at least tell you if the fields exist in an index.

```sql
SELECT DISTINCT sc.name, si.name, si.type_desc FROM sys.indexes si (nolock)
INNER JOIN sys.index_columns ic (nolock)
  ON si.object_id = ic.object_id
  AND si.index_id = ic.index_id
INNER JOIN sys.columns sc (nolock)
  ON ic.object_id = sc.object_id
  AND ic.column_id = sc.column_id
WHERE si.object_id = object_id('EDDSDBO.document')
ORDER by si.name
```

If you run across a situation where you feel like the query should be using an index you created but it is not and you can’t identify the reason, don’t hesitate to contact our support team for help working through it.

7. And finally, and most important, did you check statistics? Is this a problem that updating statistics will fix?

Assuming your answers to those questions still tell you that you need a new index, let’s talk about how to make it in a way that will best serve this query. The query from the beginning of Section 3 has three criteria searches, and three sort orders.

The criteria, or predicates, are

- [Document].[AccessControllistID_D] IN (1)
- [Document].[EmailFrom] LIKE N'% [msmith@kcura.com]%')
- [Document].[EmailTo] LIKE N'\%Lbaker %'

The sort orders are

- [Document].[DateSent] DESC,
- [Document].[EmailFrom]
- [Document].[ArtifactID]

This is an OR search, which means that regardless of the outcome of each search, each one of these could cause a full index scan. Without any indexing, this query triggers a full clustered index scan. What does that mean and why is it bad?

Since the clustered index is the physical layout, on disk, of the entire Document table, this means that this query just read the entire
document table from disk, into RAM. For very large Document tables that do not fit into RAM, this can take a very, very long time to accomplish.

How do we know if it triggers a clustered index scan? By running the query in Management Studio, and right clicking on it, we select Include Actual Execution Plan. This shows us the culprit.

You review this, and you decide to use the GUI to create the following index. In SQL Server Management Studio you can right-click on the index folder under your Document table and click Create New index to do this.

```
CREATE NONCLUSTERED INDEX [emailIndex_kIE_SRE] ON [EDDSDBO].[Document]
(
    [EmailFrom] DESC
    ,[DateSent] ASC
)
WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF,
SORT_IN_TEMPDB = OFF, DROP_EXISTING = OFF, ONLINE = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
GO
```

Once the index is created, you find that it improves performance, but not as much as you would like. Let’s say this query is the number one query in your environment, and it is run almost every 10 seconds. Now, you look at the plan and you see that there is an index scan and a Key Lookup. Let’s get rid of that Key Lookup on the clustered index by creating a covering index for it. You also suspect that the query is lighter weight now, fits into RAM, and has cached there. This is good, and better than what it was doing before, but you want this thing to be lightspeed fast.
Now, you find that even after this change, the query actually takes longer to execute. You roll your mouse over the SELECT box on the left-hand side of the query plan, and see that the cost is less than 50. This cost is actually directly related to the Cost Threshold for Parallelization (CTP), which on my system is set to 50. Now, I have learned that, on my system, a query with a CTP of 20 takes 4 seconds to run. Please note that this is not a perfect benchmark and all queries with a cost of 20 may not take exactly 4 seconds to run—they will vary from system to system. There is no universal translator. This is just a number that tells you what the cost threshold is for that particular query. It will likely be a good approximation for most queries.

Take a close look at this plan, and you will see that the little symbol for parallelization is missing.

Right-clicking on the Instance name and navigating to Properties and Advanced will allow you to lower your CTP. Many plans that previously did not parallelize will now begin to recalculate their plans. This shouldn’t matter for many queries that already are cached, but you could trigger a lot of plan recompiles by setting this. Unfortunately, it is a cost that must be paid—preferably not during peak usage but in the morning when the users are logging in and are more used to a warm up period. Letting your users know that some queries may take longer to run as the cache warms up, but should be faster in the future, will be the ideal way to do this.

Now, after making this setting, your plan looks like this, and the query finishes in less than a second:
You have now optimized your first query. Ideally, you should expand your horizons by making minor adjustments to this query. Change the sort order from ascending to descending, create different indexes, and explore all of the different icons that appear in your query plan.

If you have any questions about the above process or would like assistance getting started, please don’t hesitate to contact support@kcura.com.