SQL Server Quick Guide: Data Compression

An Introduction

Compression, Can You Dig It?

Compression has hoops, JUMP!

By now you should know whether or not you can benefit from compression. If you do, it's time to decide what kind of compression is going to work best and lastly what kind of downtime you're facing or any performance impacts you'll see.

But first, there are some additional hoops we need to jump through first before you start into your databases guns blazing. The hoops you need to get through are:

Enterprise Edition

Table compression is an enterprise only feature, boys and girls. If you're on standard, you can't use it. In fact, if you're on SQL Server Standard and below, you can't even use the compression estimation stored procedures to see if it warrants the upgrade to Enterprise or not (I know, great marketing Microsoft – buy it, then figure out if you need it!). You'll be greeted with this lovely error if you attempt to use the sp_estimate_data_compression_savings stored procedure:

```
Msg 7738, Level 16, State 2, Line 1
Cannot enable compression for object '#sam-
ple_tableDBA05385A6FF40F888204D05C7D56D2B_____________________________________
__________________________________000000000015'. Only SQL Server Enterprise
Edition supports compression.
```

In order to find out the edition of your SQL Server environment, you can use:

```
select @@version;
```

CPU Impact

Do you have the spare cycles on your CPU to handle the extra workload introduced by compression?

In order to determine if you have the spare room in the CPU department to handle the extra workload introduced by compression, head back to Performance Monitor to get some information. One counter is all you'll need for this: Processor: % Processor Time. You'll want to get a relatively large sample size (over a few days) in order to ensure that you are seeing CPU utilization from all of your processing/database access patterns. Eyeball this to make sure average utilization is not exceeding 90% for all processors. Ideally you'll be around the 60-80% level, which will leave all compression options available to you.

Types of Compression—CPU Impact

Since we're on the topic, we might as well go over the types of compression and the impact they have on CPU:

- Row compression
  Stores fixed data type columns in variable-length format
- Page compression
  The following operations are performed (and it's even done in this order!):
  - **Row compression** - you can't have one without the other!
  - **Prefix compression** –stores repeated prefix values for a column in a row in a special compression information (CI) structure that is immediately after the page header. The repeated prefix values in the column are replaced with a reference to the corresponding prefix in the CI structure (even partial matches can be indicated)
  - **Dictionary compression** – searches the page for repeating values and stores them in another CI area. Dictionary compression is not column specific and operates on the page.
- Unicode compression
  - This is implemented automatically with row and page compression can't have this without the others (one of the two).
  - The implementations benefits depend on the locale and involve compressing Unicode values that don't require localization with one byte instead of two bytes.

That's a whole lot of information regarding table compression, but what does it all mean? For all forms of compression, the data pages are compressed both on disk and in memory. Whenever SQL Server needs access that page, it must rely on the CPU to decompress the page before it can be read and then additional overhead to perform the compression of any modified pages.

The amount of strain placed on the CPU to compress and decompress the data is directly related to how much work is done by the CPU to perform the compression. Thus, page compression is the most CPU intensive, and also most effective, form of data compression. It will generally require anywhere from 20-30% of additional CPU capacity (I emphasize generally). Row compression is generally a 10% hit to your CPU (again, emphasis on generally).

With Great Power Comes Great Responsibility

Easy there, champ! Let's not go getting all crazy with applying compression now that you feel you are a good candidate for it. Compression is extremely flexible and so we should take advantage of that by being selective when choosing what to compress. By flexible I mean that you can compress:

- whole heaps,
- whole clustered indexes,
- whole indexed views,
- whole nonclustered indexes, and
- individual or ranges of partitions for partitioned tables and nonclustered indexes (partition aligned indexes, let's not be silly).

When I investigated this issue initially, I came across a SQL Server 2008 article on SQL Server Compression authored by Sanjay Mishra (a principal program manager of the SQLCat team over at Microsoft). Sanjay's article – which is definitely worth a full read if you're interested in compression – has fantastic guidance in identifying where and what type of compression is most appropriate in specific cases. The article even contains a sample table that contains, what I would refer to as, a balanced scorecard to use for evaluating compression options. I merely took the queries he was kind enough to provide in the article and did some work around them to get them to build the balanced scorecard manually for me instead of having to do it in excel (I'm lazy, what can I say).

Of note:

- I did not perform this work with partitioning in mind,
- Stats are only as representative as the duration of time SQL Server was last started or the database was opened,
- This is at the database level, so it will iterate through all objects within the currently active database

Not to Get All Management-y, But It’s Balanced Scorecard Time

Below is the query that will build the balanced scorecard that comes from aforementioned SQLCat article:

```
if object_id( N'tempdb..#index_stats' ) is not null
      drop table #index_stats;

select
      SN.name as Schema_Name,
      T.name as Table_Name,
      OS.partition_number AS Partition,
      OS.index_id AS Index_ID,
      I.type_desc AS Index_Type,
      ( OS.leaf_update_count * 100.0 / (OS.range_scan_count +
OS.leaf_insert_count + OS.leaf_delete_count + OS.leaf_update_count +
OS.leaf_page_merge_count + OS.singleton_lookup_count ) ) AS Percent_Update,
      ( OS.range_scan_count * 100.0 / (OS.range_scan_count + OS.leaf_insert_count
+ OS.leaf_delete_count + OS.leaf_update_count + OS.leaf_page_merge_count +
OS.singleton_lookup_count ) ) AS Percent_Scan
into
      #index_stats
from
      sys.dm_db_index_operational_stats( db_id(), null, null, null ) as OS
            inner join sys.tables as T on OS.object_id = T.object_id
            inner join sys.indexes as I on I.object_id = OS.object_id and
OS.index_id = I.index_id
            outer apply( select name from sys.schemas where schema_id =
T.schema_id ) as SN
where
      (OS.range_scan_count + OS.leaf_insert_count + OS.leaf_delete_count +
OS.leaf_update_count + OS.leaf_page_merge_count + OS.singleton_lookup_count) <> 0
and
      objectproperty( I.object_id, 'IsUserTable' ) = 1;

if object_id( 'tempdb..#row_compression_results' ) is not null
      drop table #row_compression_results;

create table #row_compression_results;
create table #row_compression_results
(
	object_name sysname,
	schema_name sysname,
	index_id int,
	partition_number int,
	size_with_current_compression_setting_KB bigint,
	size_with_requested_compression_setting_KB bigint,
	sample_size_with_current_compression_setting_KB bigint,
	sample_size_with_requested_compression_setting_KB bigint
);

if object_id( 'tempdb..#page_compression_results' ) is not null
	drop table #page_compression_results;

create table #page_compression_results
(
	object_name sysname,
	schema_name sysname,
	index_id int,
	partition_number int,
	size_with_current_compression_setting_KB bigint,
	size_with_requested_compression_setting_KB bigint,
	sample_size_with_current_compression_setting_KB bigint,
	sample_size_with_requested_compression_setting_KB bigint
);

declare @currSchema as sysname;
declare @currTable as sysname;

declare dbCursor cursor fast_forward for
	select distinct
		SCHEMA_NAME,
		table_name
	from
		#index_stats
	order by
		Schema_Name,
		Table_Name;

open dbCursor;

fetch next from dbCursor into @currSchema, @currTable;

while @@FETCH_STATUS = 0
	begin
		insert into #row_compression_results
			exec sp_estimate_data_compression_savings
					@schema_name = @currSchema,
					@object_name = @currTable,
					@index_id = null,
					@partition_number = null,
					@data_compression = 'ROW';
```

```
insert into #page_compression_results
                    exec sp_estimate_data_compression_savings
                                 @schema_name = @currSchema,
                                 @object_name = @currTable,
                                 @index_id = null,
                                 @partition_number = null,
                                 @data_compression = 'PAGE';

              fetch next from dbCursor into @currSchema, @currTable;
       end;

close dbCursor;
deallocate dbCursor;

select
       IXS.Schema_Name,
       IXS.Table_Name,
       IXS.Partition,
       IXS.Index_ID,
       IXS.Index_Type,
       IXS.Percent_Update,
       IXS.Percent_Scan,
       PCR.size_with_current_compression_setting_KB / 1024 as
Current_Compression_Size_MB,
       RCR.size_with_requested_compression_setting_KB / 1024 as
Row_Compression_Size_MB,
       PCR.size_with_requested_compression_setting_KB / 1024 as
Page_Compression_Size_MB,
       100. - cast( ( cast( PCR.size_with_requested_compression_setting_KB as
numeric( 18, 6 ) ) / cast( PCR.size_with_current_compression_setting_KB as
numeric( 18, 6 ) ) ) * 100 as numeric( 5, 2 ) ) as
Page_Compression_Savings_Percentage,
       100. - cast( ( cast( RCR.size_with_requested_compression_setting_KB as
numeric( 18, 6 ) ) / cast( RCR.size_with_current_compression_setting_KB as
numeric( 18, 6 ) ) ) * 100 as numeric( 5, 2 ) ) as
Row_Compression_Savings_Percentage
from
       #index_stats as IXS
              left join #row_compression_results as RCR on IXS.schema_name =
RCR.schema_name and IXS.table_name = RCR.object_name and IXS.index_ID =
RCR.index_id
              left join #page_compression_results as PCR on IXS.schema_name =
PCR.schema_name and IXS.table_name = PCR.object_name and IXS.index_ID =
PCR.index_id
where
       RCR.size_with_requested_compression_setting_KB <> 0 and
       PCR.size_with_requested_compression_setting_KB <> 0
order by
       PCR.size_with_current_compression_setting_KB desc;
```

These Aren’t the Droids You’re Looking For

Now that you have your balanced scorecard, what are you looking for? This is how I look for good candidates:

Page compression – most CPU impactful so we’re a little more careful

- Low update percentage to total table activity (< 30%) – compressing a very active table is going to cause overhead on update/insert/delete activity, which is in turn going to cause other issues
- High percentage of scans to total table activity (>65%) – scans are going to give you the most bang for your buck fitting more data on each page, decreasing the amount of pages needing to be retrieved from storage and stored in memory
- Database size is large (there’s a reason I sort the results descending by size) – 25 KB tables that compress to 2KB, really don’t matter because they’ll fit in memory compressed and uncompressed
- Large difference between Row_Compression_Savings_Percentage and Page_Compression_Savings_Percentage – if you only get 25% additional savings from page compression, is it really worth the extra CPU overhead? Again, this depends on the size of the table because if the table is 100GB, 25% is going to be 25GB and that’s a whole lot fewer pages

Row compression – least CPU impactful so we can be a little more liberal

- Reasonable update percentage to total table activity (<65%) – we can compress tables that are a bit more active with Row vs. Compression as there will be less overhead on the insert/update/delete operations
- Low to moderate percentage of scans compared to total table activity (<40%) – as you will likely only be touching a subset of pages with high seek activity, you still benefit, just not as much
- Database size is reasonably large – same reason as page compression
- Small difference between Row_Compression_Savings_Percentage and Page_Compression_Savings_Percentage – if the gain from page compression over row compression is small, why bother with the added overhead of page compression?

The tables that don’t fit for either page or row compression? Those are indeed not the droids you are looking for.

Ready...Set...Compress!

There are ideal ways to go about compressing your data:

- One compress operation at a time or many at the same time – I like one at a time as it lets you keep track of the system impact you are having much better. Additionally it keeps your memory, CPU and disk requirements at a more predictable level
- Compress small to large – as you complete the compression operations, you are releasing more space to data files so that they can hopefully absorb the space requirements for the compression on the larger objects without having to grow
- Set online ON or OFF for the rebuild operation
  - **OFF** is faster and requires less resources to complete. Offline re builds apply a table lock for the duration of the index operation.
  - **ON** is slower and requires more resources to complete. Online rebuilds apply an Intent Share (IS) lock on the source table for the duration of index operation – except at the beginning (a shared lock - S Lock) and the end (schema mod lock – SCH-M).
  - **Off** requires a maintenance window due to the fact that you are taking the index offline while you are rebuilding it
- Sort in tempdb – especially with online rebuilds, I like this option as it allows you to offload a lot of the IO activity – the results of the sort runs specifically. This option specifically allows you to store the intermediate results of the rebuild operation in the tempdb vs. the same database as the index, specifically the destination filegroup

That's All, Folks!

I'm really not going to cover the actual TSQL behind enabling compression. It's all covered in the BoL quite extensively and pretty darn well. Indexes and tables have data compression enabled through:

Alter Index... Rebuild statements - http://technet.microsoft.com/en-us/library/ms188388.aspx
Alter Table... Rebuild statements - http://technet.microsoft.com/en-us/library/ms190273.aspx

I hope this was a beneficial starting point for identifying whether or not data compression may help with your specific environment!

About the Author:

Jerrod Early is a SQL Consultant with Oakwood's Managed Services group. When not furthering his knowledge in all things data and technology, he enjoys spending time with his family and being mercilessly trolled in video games.