Rebuilding an index in SQL Server 2005 and newer will also compact the large object (LOB) pages. You can choose not to by setting a value of `LOB_COMPACTION = OFF`. If you aren't worried about storage but you are concerned about how long your index reorganization is taking, this might be advisable to turn off.

When you use the `PAD_INDEX` setting while creating an index, it determines how much free space to leave on the index intermediate pages, which can help you deal with page splits. This is taken into account during the index rebuild, and the new pages will be set back to the original values you determined at the index creation unless you specify otherwise. I've almost never seen this make a major difference on most systems. You'll need to test on your system to determine whether it can help.

If you don't specify otherwise, the default behavior is to defragment all indexes across all partitions. If you want to control the process, you just need to specify which partition you want to rebuild when.

As shown previously, the `ALTER INDEX REBUILD` technique effectively reduces fragmentation. You can also use it to rebuild *all* the indexes of a table in one statement.

```
ALTER INDEX ALL ON dbo.Test1 REBUILD;
```

Although this is the most effective defragmentation technique, it does have some overhead and limitations.

- *Blocking*: Similar to the previous two index-rebuilding techniques, `ALTER INDEX REBUILD` introduces blocking in the system. It blocks all other queries trying to access the table (or any index on the table). It can also be blocked by those queries. You can reduce this using `ONLINE INDEX REBUILD`.
- *Transaction rollback*: Since `ALTER INDEX REBUILD` is fully atomic in action, if it is stopped before completion, then all the defragmentation actions performed up to that time are lost. You can run `ALTER INDEX REBUILD` using the `ONLINE` keyword, which will reduce the locking mechanisms, but it will increase the time involved in rebuilding the index.

Introduced in Azure SQL Database and available in SQL Server 2017, you now have the capacity to restart an index rebuild operation. You can restart a failed index rebuild, or you can pause the rebuild operation only to restart it later. To do this, you have to

be using the ONLINE rebuild option as well. The ONLINE option radically reduces the blocking associated with the rebuild operation. To rebuild an index that is both ONLINE and RESUMABLE, you must specify all this in the command.

```
ALTER INDEX i1 ON dbo.Test1 REBUILD WITH (ONLINE=ON, RESUMABLE=ON);
```

This will run the index rebuild operation until it is completed or until you issue the following command:

```
ALTER INDEX i1 ON dbo.Test1 PAUSE;
```

This will pause the ONLINE rebuild operation, and the table and indexes in question will remain accessible without any blocking. To restart the operation, use this:

```
ALTER INDEX i1 ON dbo.Test1 RESUME;
```

Executing the ALTER INDEX REORGANIZE Statement

For a rowstore index, ALTER INDEX REORGANIZE reduces the fragmentation of an index without rebuilding the index. It reduces external fragmentation by rearranging the existing leaf pages of the index in the logical order of the index key. It compacts the rows within the pages, reducing internal fragmentation, and discards the resultant empty pages. This technique doesn't use any new pages for defragmentation.

For a columnstore index, ALTER INDEX REORGANIZE will ensure that the deltastore within the columnstore index gets cleaned out and that all the logical deletes are taken care of. It does this while keeping the index online and accessible. This will ensure that the index is defragmented. Further, you have the option of forcing the compression of all the row groups. This will function similarly to running ALTER INDEX REBUILD, but it continues to keep the index online during the operation, unlike the REBUILD process. Because of this, ALTER INDEX REORGANIZE is preferred for columnstore indexes.

To avoid the blocking overhead associated with ALTER INDEX REBUILD, this technique uses a nonatomic online approach. As it proceeds through its steps, it requests a small number of locks for a short period. Once each step is done, it releases the locks and proceeds to the next step. While trying to access a page, if it finds that the page is being used, it skips that page and never returns to the page again. This allows other queries to run on the table along with the ALTER INDEX REORGANIZE operation. Also, if this operation is stopped intermediately, then all the defragmentation steps performed up to then are preserved.

For a rowstore index, since ALTER INDEX REORGANIZE doesn't use any new pages to reorder the index and it skips the locked pages, the amount of defragmentation provided by this approach is usually less than that of ALTER INDEX REBUILD. To observe the relative effectiveness of ALTER INDEX REORGANIZE compared to ALTER INDEX REBUILD, rebuild the test table used in the previous section on ALTER INDEX REBUILD.

Rebuild the fragmented rowstore table using the script from before. To reduce the fragmentation of the clustered rowstore index, use ALTER INDEX REORGANIZE as follows:

```
ALTER INDEX i1 ON dbo.Test1 REORGANIZE;
```

Figure 14-19 shows the resultant output from sys.dm_db_index_physical_stats.

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	0.239377618192699	80	6684	74.836236718557	20000	2022.999

Figure 14-19. *Results of ALTER INDEX REORGANIZE*

From the output, you can see that ALTER INDEX REORGANIZE doesn't reduce fragmentation as effectively as ALTER INDEX REBUILD, as shown in the previous section. For a highly fragmented index, the ALTER INDEX REORGANIZE operation can take much longer than rebuilding the index. Also, if an index spans multiple files, ALTER INDEX REORGANIZE doesn't migrate pages between the files. However, the main benefit of using ALTER INDEX REORGANIZE is that it allows other queries to access the table (or the indexes) simultaneously.

To see the results of defragmentation of a columnstore index, let's use the already fragmented columnstore index from the "Columnstore Overhead" section earlier in the chapter.

```
ALTER INDEX ClusteredColumnStoreTest ON dbo.bigTransactionHistory
REORGANIZE;
```

The results of the REORGANIZE statement are visible in Figure 14-20.

	TableName	IndexName	partition_number	row_group_id	state_description	total_rows	deleted_rows	PercentFull
1	bigTransactionHistory	cci_bigTransactionHistory	1	0	COMPRESSED	1048576	52759	94
2	bigTransactionHistory	cci_bigTransactionHistory	1	1	COMPRESSED	1048576	52497	94
3	bigTransactionHistory	cci_bigTransactionHistory	1	2	COMPRESSED	1048576	52191	95
4	bigTransactionHistory	cci_bigTransactionHistory	1	3	COMPRESSED	1048576	52482	94
5	bigTransactionHistory	cci_bigTransactionHistory	1	4	COMPRESSED	1048576	52294	95
6	bigTransactionHistory	cci_bigTransactionHistory	1	5	COMPRESSED	1048576	52819	94
7	bigTransactionHistory	cci_bigTransactionHistory	1	6	COMPRESSED	1048576	52647	94
8	bigTransactionHistory	cci_bigTransactionHistory	1	7	COMPRESSED	1048576	52477	94
9	bigTransactionHistory	cci_bigTransactionHistory	1	8	COMPRESSED	1048576	52472	94
10	bigTransactionHistory	cci_bigTransactionHistory	1	9	COMPRESSED	1048576	52370	95
11	bigTransactionHistory	cci_bigTransactionHistory	1	10	COMPRESSED	1048576	52472	94
12	bigTransactionHistory	cci_bigTransactionHistory	1	11	COMPRESSED	1048576	52773	94
13	bigTransactionHistory	cci_bigTransactionHistory	1	12	COMPRESSED	1048576	52639	94
14	bigTransactionHistory	cci_bigTransactionHistory	1	13	COMPRESSED	1048576	52323	95
15	bigTransactionHistory	cci_bigTransactionHistory	1	14	COMPRESSED	1048576	52360	95
16	bigTransactionHistory	cci_bigTransactionHistory	1	15	COMPRESSED	1048576	52276	95
17	bigTransactionHistory	cci_bigTransactionHistory	1	16	COMPRESSED	1048576	52741	94
18	bigTransactionHistory	cci_bigTransactionHistory	1	17	COMPRESSED	1048576	52329	95
19	bigTransactionHistory	cci_bigTransactionHistory	1	18	COMPRESSED	1048576	52474	94
20	bigTransactionHistory	cci_bigTransactionHistory	1	19	COMPRESSED	1048576	52647	94
21	bigTransactionHistory	cci_bigTransactionHistory	1	20	COMPRESSED	1048576	52046	95
22	bigTransactionHistory	cci_bigTransactionHistory	1	21	COMPRESSED	1048576	52673	94
23	bigTransactionHistory	cci_bigTransactionHistory	1	22	COMPRESSED	1048576	52668	94
24	bigTransactionHistory	cci_bigTransactionHistory	1	23	COMPRESSED	1048576	52885	94
25	bigTransactionHistory	cci_bigTransactionHistory	1	24	COMPRESSED	1048576	52453	94
26	bigTransactionHistory	cci_bigTransactionHistory	1	25	COMPRESSED	1048576	52414	95
27	bigTransactionHistory	cci_bigTransactionHistory	1	26	COMPRESSED	1048576	52114	95
28	bigTransactionHistory	cci_bigTransactionHistory	1	27	COMPRESSED	1048576	52500	94
29	bigTransactionHistory	cci_bigTransactionHistory	1	28	COMPRESSED	1048576	52434	94
30	bigTransactionHistory	cci_bigTransactionHistory	1	29	TOMBSTONE	104086	5230	94
31	bigTransactionHistory	cci_bigTransactionHistory	1	30	TOMBSTONE	750811	37419	95
32	bigTransactionHistory	cci_bigTransactionHistory	1	31	COMPRESSED	812248	0	100

Figure 14-20. *Results of REORGANIZE without compression on columnstore index*

You'll notice that most of the rowgroups are still somewhat fragmented. Two of the row groups (29,30) haven't been changed from `COMPRESSED` to `TOMBSTONE`. This means those rowgroups will be removed in the background at some later date. In short, only a few of the row groups were merged, and almost none of the deleted rows was dealt with. This is because the `REORGANIZE` command will clean up the deleted data only when more than 10 percent of the data in a rowgroup has been deleted. Let's remove more data from the table to bring some of the rowgroups to more than 10 percent deleted.

```
DELETE dbo.bigTransactionHistory
WHERE Quantity BETWEEN 8
                   AND     17;
```

Now when we look at the fragmentation results, we see a lot more activity, as shown in Figure 14-21.

	TableName	IndexName	partition_number	row_group_id	state_description	total_rows	deleted_rows	PercentFull
1	bigTransactionHistory	cci_bigTransactionHistory	1	0	TOMBSTONE	1048576	104902	89
2	bigTransactionHistory	cci_bigTransactionHistory	1	1	TOMBSTONE	1048576	104961	89
3	bigTransactionHistory	cci_bigTransactionHistory	1	2	COMPRESSED	1048576	104147	90
4	bigTransactionHistory	cci_bigTransactionHistory	1	3	TOMBSTONE	1048576	105212	89
5	bigTransactionHistory	cci_bigTransactionHistory	1	4	TOMBSTONE	1048576	105101	89
6	bigTransactionHistory	cci_bigTransactionHistory	1	5	COMPRESSED	1048576	104770	90
7	bigTransactionHistory	cci_bigTransactionHistory	1	6	TOMBSTONE	1048576	104905	89
8	bigTransactionHistory	cci_bigTransactionHistory	1	7	TOMBSTONE	1048576	105097	89
9	bigTransactionHistory	cci_bigTransactionHistory	1	8	COMPRESSED	1048576	104170	90
10	bigTransactionHistory	cci_bigTransactionHistory	1	9	TOMBSTONE	1048576	105264	89
11	bigTransactionHistory	cci_bigTransactionHistory	1	10	COMPRESSED	1048576	104479	90
12	bigTransactionHistory	cci_bigTransactionHistory	1	11	COMPRESSED	1048576	104813	90
13	bigTransactionHistory	cci_bigTransactionHistory	1	12	TOMBSTONE	1048576	105276	89
14	bigTransactionHistory	cci_bigTransactionHistory	1	13	TOMBSTONE	1048576	105195	89
15	bigTransactionHistory	cci_bigTransactionHistory	1	14	TOMBSTONE	1048576	104944	89
16	bigTransactionHistory	cci_bigTransactionHistory	1	15	TOMBSTONE	1048576	105181	89
17	bigTransactionHistory	cci_bigTransactionHistory	1	16	TOMBSTONE	1048576	104969	89
18	bigTransactionHistory	cci_bigTransactionHistory	1	17	TOMBSTONE	1048576	105001	89
19	bigTransactionHistory	cci_bigTransactionHistory	1	18	TOMBSTONE	1048576	105264	89
20	bigTransactionHistory	cci_bigTransactionHistory	1	19	COMPRESSED	1048576	104274	90
21	bigTransactionHistory	cci_bigTransactionHistory	1	20	COMPRESSED	1048576	104847	90
22	bigTransactionHistory	cci_bigTransactionHistory	1	21	COMPRESSED	1048576	104635	90
23	bigTransactionHistory	cci_bigTransactionHistory	1	22	TOMBSTONE	1048576	104966	89
24	bigTransactionHistory	cci_bigTransactionHistory	1	23	COMPRESSED	1048576	104850	90
25	bigTransactionHistory	cci_bigTransactionHistory	1	24	TOMBSTONE	1048576	105189	89
26	bigTransactionHistory	cci_bigTransactionHistory	1	25	TOMBSTONE	1048576	105050	89
27	bigTransactionHistory	cci_bigTransactionHistory	1	26	COMPRESSED	1048576	104592	90
28	bigTransactionHistory	cci_bigTransactionHistory	1	27	TOMBSTONE	1048576	104965	89
29	bigTransactionHistory	cci_bigTransactionHistory	1	28	COMPRESSED	878208	87553	90
30	bigTransactionHistory	cci_bigTransactionHistory	1	29	COMPRESSED	1025265	102585	89
31	bigTransactionHistory	cci_bigTransactionHistory	1	30	COMPRESSED	943674	0	100
32	bigTransactionHistory	cci_bigTransactionHistory	1	31	COMPRESSED	943671	0	100
33	bigTransactionHistory	cci_bigTransactionHistory	1	32	COMPRESSED	943632	0	100
34	bigTransactionHistory	cci_bigTransactionHistory	1	33	COMPRESSED	943615	0	100
35	bigTransactionHistory	cci_bigTransactionHistory	1	34	COMPRESSED	943611	0	100
36	bigTransactionHistory	cci_bigTransactionHistory	1	35	COMPRESSED	943610	0	100
37	bigTransactionHistory	cci_bigTransactionHistory	1	36	COMPRESSED	943607	0	100
38	bigTransactionHistory	cci_bigTransactionHistory	1	37	COMPRESSED	943575	0	100
39	bigTransactionHistory	cci_bigTransactionHistory	1	38	COMPRESSED	943526	0	100
40	bigTransactionHistory	cci_bigTransactionHistory	1	39	COMPRESSED	943479	0	100
41	bigTransactionHistory	cci_bigTransactionHistory	1	40	COMPRESSED	943475	0	100
42	bigTransactionHistory	cci_bigTransactionHistory	1	41	COMPRESSED	943395	0	100
43	bigTransactionHistory	cci_bigTransactionHistory	1	42	COMPRESSED	943387	0	100
44	bigTransactionHistory	cci_bigTransactionHistory	1	43	COMPRESSED	943381	0	100
45	bigTransactionHistory	cci_bigTransactionHistory	1	44	COMPRESSED	943364	0	100
46	bigTransactionHistory	cci_bigTransactionHistory	1	45	COMPRESSED	943312	0	100
47	bigTransactionHistory	cci_bigTransactionHistory	1	46	COMPRESSED	943312	0	100
48	bigTransactionHistory	cci_bigTransactionHistory	1	47	COMPRESSED	943300	0	100

Figure 14-21. *REORGANIZE without compression against a more fragmented index*

You can see that many more rowgroups have been marked as TOMBSTONE and that all the new pages have zero deleted rows. You'll note that the row_group_id values have been generated for the row groups that have been compressed. The old row_group_id won't be reused. If you look at the table after the cleanup is complete, you'll see only the COMPRESSED row groups, bringing the total down to 30 because of the removed data, but you'll see gaps.

If we were to rerun the REORGANIZE command but include the row group, the command would look like this:

```
ALTER INDEX cci_bigTransactionHistory
ON dbo.bigTransactionHistory
REORGANIZE
WITH (COMPRESS_ALL_ROW_GROUPS = ON);
```

The command COMPRESS_ALL_ROW_GROUPS will ensure that any OPEN or CLOSED rowgroups in the deltastore will get moved into the columnstore going through the compression and everything else associated with a columnstore index.

Before running this, though, let's delete a little more data to push the rowgroups that are at 10 percent fragmentation over the top.

```
DELETE dbo.bigTransactionHistory
WHERE Quantity BETWEEN 6
               AND     8;
```

The results shown in Figure 14-22 include removing the TOMBSTONE, as well as reorganizing the index completely.

	TableName	IndexName	partition_number	row_group_id	state_description	total_rows	deleted_rows	PercentFull
1	bigTransactionHistory	cci_bigTransactionHistory	1	2	TOMBSTONE	1048576	125061	88
2	bigTransactionHistory	cci_bigTransactionHistory	1	5	TOMBSTONE	1048576	125925	87
3	bigTransactionHistory	cci_bigTransactionHistory	1	8	TOMBSTONE	1048576	125388	88
4	bigTransactionHistory	cci_bigTransactionHistory	1	10	TOMBSTONE	1048576	125386	88
5	bigTransactionHistory	cci_bigTransactionHistory	1	11	TOMBSTONE	1048576	125754	88
6	bigTransactionHistory	cci_bigTransactionHistory	1	19	TOMBSTONE	1048576	125134	88
7	bigTransactionHistory	cci_bigTransactionHistory	1	20	TOMBSTONE	1048576	125572	88
8	bigTransactionHistory	cci_bigTransactionHistory	1	21	TOMBSTONE	1048576	125648	88
9	bigTransactionHistory	cci_bigTransactionHistory	1	23	TOMBSTONE	1048576	125789	88
10	bigTransactionHistory	cci_bigTransactionHistory	1	26	TOMBSTONE	1048576	125558	88
11	bigTransactionHistory	cci_bigTransactionHistory	1	28	TOMBSTONE	878208	105032	88
12	bigTransactionHistory	cci_bigTransactionHistory	1	29	TOMBSTONE	1025265	123191	87
13	bigTransactionHistory	cci_bigTransactionHistory	1	30	COMPRESSED	943674	21049	97
14	bigTransactionHistory	cci_bigTransactionHistory	1	31	COMPRESSED	943671	20823	97
15	bigTransactionHistory	cci_bigTransactionHistory	1	32	COMPRESSED	943632	20967	97
16	bigTransactionHistory	cci_bigTransactionHistory	1	33	COMPRESSED	943615	21016	97
17	bigTransactionHistory	cci_bigTransactionHistory	1	34	COMPRESSED	943611	21007	97
18	bigTransactionHistory	cci_bigTransactionHistory	1	35	COMPRESSED	943610	21167	97
19	bigTransactionHistory	cci_bigTransactionHistory	1	36	COMPRESSED	943607	20897	97
20	bigTransactionHistory	cci_bigTransactionHistory	1	37	COMPRESSED	943575	21229	97
21	bigTransactionHistory	cci_bigTransactionHistory	1	38	COMPRESSED	943526	20900	97
22	bigTransactionHistory	cci_bigTransactionHistory	1	39	COMPRESSED	943479	20752	97
23	bigTransactionHistory	cci_bigTransactionHistory	1	40	COMPRESSED	943475	21333	97
24	bigTransactionHistory	cci_bigTransactionHistory	1	41	COMPRESSED	943395	21005	97
25	bigTransactionHistory	cci_bigTransactionHistory	1	42	COMPRESSED	943387	20963	97
26	bigTransactionHistory	cci_bigTransactionHistory	1	43	COMPRESSED	943381	20811	97
27	bigTransactionHistory	cci_bigTransactionHistory	1	44	COMPRESSED	943364	20995	97
28	bigTransactionHistory	cci_bigTransactionHistory	1	45	COMPRESSED	943312	21074	97
29	bigTransactionHistory	cci_bigTransactionHistory	1	46	COMPRESSED	943312	21218	97
30	bigTransactionHistory	cci_bigTransactionHistory	1	47	COMPRESSED	943300	20891	97
31	bigTransactionHistory	cci_bigTransactionHistory	1	48	COMPRESSED	923515	0	100
32	bigTransactionHistory	cci_bigTransactionHistory	1	49	COMPRESSED	923442	0	100
33	bigTransactionHistory	cci_bigTransactionHistory	1	50	COMPRESSED	923190	0	100
34	bigTransactionHistory	cci_bigTransactionHistory	1	51	COMPRESSED	923188	0	100
35	bigTransactionHistory	cci_bigTransactionHistory	1	52	COMPRESSED	923018	0	100
36	bigTransactionHistory	cci_bigTransactionHistory	1	53	COMPRESSED	923004	0	100
37	bigTransactionHistory	cci_bigTransactionHistory	1	54	COMPRESSED	922928	0	100
38	bigTransactionHistory	cci_bigTransactionHistory	1	55	COMPRESSED	922822	0	100
39	bigTransactionHistory	cci_bigTransactionHistory	1	56	COMPRESSED	922787	0	100
40	bigTransactionHistory	cci_bigTransactionHistory	1	57	COMPRESSED	922651	0	100
41	bigTransactionHistory	cci_bigTransactionHistory	1	58	COMPRESSED	902074	0	100
42	bigTransactionHistory	cci_bigTransactionHistory	1	59	COMPRESSED	773176	0	100

Figure 14-22. *Compression and defragmentation for the columnstore index*

Effectively, the order of the row groups doesn't really matter. As they get defragmented, they get moved from their original location within the index to a new location later.

If you don't want to deal with the 10 percent limitation, you can use the `REBUILD` option on the columnstore index, but you will have to deal with the fact that you're taking the index offline during that process.

Table 14-1 summarizes the characteristics of these four defragmentation techniques on a rowstore index.

Table 14-1. *Characteristics of Rowstore Defragmentation Techniques*

Characteristics/ Issues	**Drop and Create Index**	**Create Index with `DROP_ EXISTING`**	**`ALTER INDEX REBUILD`**	**`ALTER INDEX REORGANIZE`**
Rebuild nonclustered indexes on clustered index fragmentation	Twice	No	No	No
Missing indexes	Yes	No	No	No
Defragment index with constraints	Highly complex	Moderately complex	Easy	Easy
Defragment multiple indexes together	No	No	Yes	Yes
Concurrency with others	Low	Low	Medium, depending on concurrent user activity	High
Intermediate cancellation	Dangerous with no transaction	Progress lost	Progress lost	Progress preserved
Degree of defragmentation	High	High	High	Moderate to low
Apply new fill factor	Yes	Yes	Yes	No
Statistics are updated	Yes	Yes	Yes	No

You can also reduce internal fragmentation by compressing more rows within a page, reducing free spaces within the pages. The maximum amount of compression that can be done within the leaf pages of an index is controlled by the fill factor, as you will see next.

When dealing with large databases and the indexes associated, it may become necessary to split up the tables and the indexes across disks using partitioning. Indexes on partitions can also become fragmented as the data within the partition changes.

When dealing with a portioned index, you will need to determine whether you want to either reorganize or rebuild (with `REORGANIZE` or `REBUILD`, respectively) one, some, or all partitions as part of the `ALTER INDEX` command. Partitioned indexes cannot be rebuilt online. Keep in mind that doing anything that affects all partitions is likely to be a costly operation.

If compression is specified on an index, even on a partitioned index, you must be sure to set the compression while performing the `ALTER INDEX` operation to what it was before; if you don't, it will be lost, and you'll have to rebuild the index. This is especially important for nonclustered indexes, which will not inherit the compression setting from the table.

Defragmentation and Partitions

If you have massive databases, a standard mechanism for effectively managing the data is to break it up into partitions. While partitions can, in some rare cases, help with performance, they are foremost for managing data. But, one of the issues with indexes and partitions is that if you rebuild the index, it's unavailable during the rebuild. This means that with partitions, which are on massive indexes, you can expect to have a major portion of your data offline during the rebuild. SQL Server 2012 introduced the ability to do an online rebuild. If you had a partitioned index, it would look like this:

```
ALTER INDEX i1 ON dbo.Test1
REBUILD PARTITION = ALL
WITH (ONLINE = ON);
```

This can rebuild the entire partition and do it as an online operation, meaning it keeps the index largely available while it does the rebuild. But, for some partitions, this is a massive undertaking that will probably result in excessive load on the server and the need for a lot more tempdb storage. SQL Server 2014 introduced new functionality that lets you designate individual partitions.

```
ALTER INDEX i1 ON dbo.Test1
REBUILD PARTITION = 1
WITH (ONLINE = ON);
```

This reduces the overhead of the rebuild operation while still keeping the index mostly available during the rebuild. I do emphasize that it is "mostly" online because there is still some degree of locking and contention that will occur during the rebuild. It's not a completely free operation. It's just radically improved over the alternative.

Talking about the locking involved with index rebuild operations in partitions, you also have one other new piece of functionality introduced in SQL Server 2014. You can now modify the lock priority used during the rebuild operation by again adjusting the `REBUILD` command.

```
ALTER INDEX i1
ON dbo.Test1
REBUILD PARTITION = 1
WITH (ONLINE = ON (WAIT_AT_LOW_PRIORITY (MAX_DURATION = 20,
                                         ABORT_AFTER_WAIT = SELF)));
```

What this does is set the duration that the rebuild operation is willing to wait, in minutes. Then, it allows you to determine which processes get aborted in order to clear the system for the index rebuild. You can have it stop itself or the blocking process. The most interesting thing is that the waiting process is set to low priority, so it's not using a lot of system resources, and any transactions that come in won't be blocked by this process.

Significance of the Fill Factor

On rowstore indexes, the internal fragmentation of an index is reduced by getting more rows per leaf page in an index. Getting more rows within a leaf page reduces the total number of pages required for the index and in turn decreases disk I/O and the logical reads required to retrieve a range of index rows. On the other hand, if the index key values are highly transactional, then having fully used index pages will cause page splits. Therefore, for a transactional table, a good balance between maximizing the number of rows in a page and avoiding page splits is required.

SQL Server allows you to control the amount of free space within the leaf pages of the index by using the *fill factor*. If you know that there will be enough `INSERT` queries on the table or `UPDATE` queries on the index key columns, then you can pre-add free space to the index leaf page using the fill factor to minimize page splits. If the table is read-only, you

can create the index with a high fill factor to reduce the number of index pages. It's a good idea to have some free space, though, when dealing with inserts against an IDENTITY column (or any index key that contains ordered data that will tend to create a hot page).

The default fill factor is 0, which means the leaf pages are packed to 100 percent, although some free space is left in the branch nodes of the B-tree structure. The fill factor for an index is applied only when the index is created. As keys are inserted and updated, the density of rows in the index eventually stabilizes within a narrow range. As you saw in the previous chapter's sections on page splits caused by UPDATE and INSERT, when a page split occurs, generally half the original page is moved to a new page, which happens irrespective of the fill factor used during the index creation.

To understand the significance of the fill factor, let's use a small test table with 24 rows.

```
DROP TABLE IF EXISTS dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT,
                        C2 CHAR(999));

WITH Nums
AS (SELECT 1 AS n
    UNION ALL
    SELECT n + 1
    FROM Nums
    WHERE n < 24)
INSERT INTO dbo.Test1 (C1,
                       C2)
SELECT n * 100,
       'a'
FROM Nums;
```

Increase the maximum number of rows in the leaf (or data) page by creating a clustered index with the default fill factor.

```
CREATE CLUSTERED INDEX FillIndex ON Test1(C1);
```

Since the average row size is 1,010 bytes, a clustered index leaf page (or table data page) can contain a maximum of eight rows. Therefore, at least three leaf pages are required for the 24 rows. You can confirm this in the sys.dm_db_index_physical_stats output shown in Figure 14-23.

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	33.3333333333333	2	3	100	24	1010

Figure 14-23. *Fill factor set to default value of 0*

Note that `avg_page_space_used_in_percent` is 100 percent since the default fill factor allows the maximum number of rows to be compressed in a page. Since a page cannot contain a part row to fill the page fully, `avg_page_space_used_in_percent` will be often a little less than 100 percent, even with the default fill factor.

To reduce the initial frequency of page splits caused by `INSERT` and `UPDATE` operations, create some free space within the leaf (or data) pages by re-creating the clustered index with a fill factor as follows:

```
ALTER INDEX FillIndex ON dbo.Test1 REBUILD
WITH  (FILLFACTOR= 75);
```

Because each page has a total space for eight rows, a fill factor of 75 percent will allow six rows per page. Thus, for 24 rows, the number of leaf pages should increase to four, as in the `sys.dm_db_index_physical_stats` output shown in Figure 14-24.

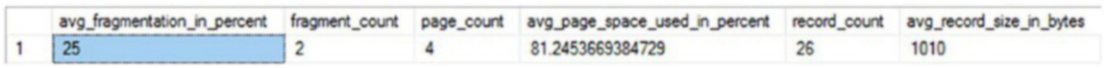
	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	25	2	4	74.9938225846306	24	1010

Figure 14-24. *Fill factor set to 75*

Note that `avg_page_space_used_in_percent` is about 75 percent, as set by the fill factor. This allows two more rows to be inserted in each page without causing a page split. You can confirm this by adding two rows to the first set of six rows (`C1` = 100 – 600, contained in the first page).

```
INSERT  INTO dbo.Test1
VALUES  (110, 'a'),  --25th row
        (120, 'a') ;  --26th row
```

Figure 14-25 shows the current fragmentation.



	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	25	2	4	81.2453669384729	26	1010

Figure 14-25. *Fragmentation after new records*

From the output, you can see that the addition of the two rows has not added any pages to the index. Accordingly, `avg_page_space_used_in_percent` increased from 74.99 percent to 81.25 percent. With the addition of two rows to the set of the first six rows, the first page should be completely full (eight rows). Any further addition of rows within the range of the first eight rows should cause a page split and thereby increase the number of index pages to five.

```
INSERT  INTO dbo.Test1
VALUES  (130, 'a') ;  --27th row
```

Now `sys.dm_db_index_physical_stats` displays the difference in Figure 14-26.

	avg_fragmentation_in_percent	fragment_count	page_count	avg_page_space_used_in_percent	record_count	avg_record_size_in_bytes
1	40	3	5	67.4919693600198	27	1010

Figure 14-26. *Number of pages goes up*

Note that even though the fill factor for the index is 75 percent, `Avg. Page Density (full)` has decreased to 67.49 percent, which can be computed as follows:

```
Avg. Page Density (full)
= Average rows per page / Maximum rows per page
= (27 / 5) / 8
= 67.5%
```

From the preceding example, you can see that the fill factor is applied when the index is created. But later, as the data is modified, it has no significance. Irrespective of the fill factor, whenever a page splits, the rows of the original page are distributed between two pages, and `avg_page_space_used_in_percent` settles accordingly. Therefore, if you use a nondefault fill factor, you should ensure that the fill factor is reapplied regularly to maintain its effect.

You can reapply a fill factor by re-creating the index or by using `ALTER INDEX REORGANIZE` or `ALTER INDEX REBUILD`, as was shown. `ALTER INDEX REORGANIZE` takes the fill factor specified during the index creation into account. `ALTER INDEX REBUILD` also takes the original fill factor into account, but it allows a new fill factor to be specified, if required.

Without periodic maintenance of the fill factor, for both default and nondefault fill factor settings, `avg_page_space_used_in_percent` for an index (or a table) eventually settles within a narrow range.

An argument can be made that rather than attempt to defragment indexes over and over again, with all the overhead that implies, you could be better off settling on a fill factor that allows for a fairly standard set of distribution across the pages in your indexes. Some people do use this method, sacrificing some read performance and disk space to avoid page splits and the associated issues in which they result. Testing on your own systems to both find the right fill factor and determine whether that method works will be necessary.

Automatic Maintenance

In a database with a great deal of transactions, tables and indexes become fragmented over time (assuming you're not using the fill factor method just mentioned). Thus, to improve performance, you should check the fragmentation of the tables and indexes regularly, and you should defragment the ones with a high amount of fragmentation. You also may need to take into account the workload and defragment indexes as dictated by the load as well as the fragmentation level of the index. You can do this analysis for a database by following these steps:

1. Identify all user tables in the current database to analyze fragmentation.
2. Determine fragmentation of every user table and index.
3. Determine user tables and indexes that require defragmentation by taking into account the following considerations:
    - A high level of fragmentation where `avg_fragmentation_in_percent` is greater than 20 percent
    - Not a very small table/index—that is, `pagecount` is greater than 8
4. Defragment tables and indexes with high fragmentation.

For a fully functional script that includes a large degree of capability, I strongly recommend using the Minion Reindex application located at `http://bit.ly/2EGsmYU` or Ola Hollengren's scripts at `http://bit.ly/JijaNI`.

In addition to those scripts, you can use the maintenance plans built into SQL Server. However, I don't recommend them because you surrender a lot of control for a little bit of ease of use. You'll be much happier with the results you get from one of the sets of scripts recommended earlier.

Summary

As you learned in this chapter, in a highly transactional database, page splits caused by `INSERT` and `UPDATE` statements may fragment the tables and indexes, increasing the cost of data retrieval. You can avoid these page splits by maintaining free spaces within the pages using the fill factor. Since the fill factor is applied only during index creation, you should reapply it at regular intervals to maintain its effectiveness. Data manipulation of columnstore indexes also leads to fragmentation and performance degradation. You can determine the amount of fragmentation in an index (or a table) using `sys.dm_db_index_physical_stats` for a rowstore index or using `sys.column_store_row_groups` for a columnstore index. Upon determining a high amount of fragmentation, you can use either `ALTER INDEX REBUILD` or `ALTER INDEX REORGANIZE`, depending on the required amount of defragmentation, the database concurrency, and whether you are dealing with a rowstore or columnstore index.

Defragmentation rearranges the data so that its physical order on the disk matches its logical order in the table/index, thus improving the performance of queries. However, unless the optimizer decides upon an effective execution plan for the query, query performance even after defragmentation can remain poor. Therefore, it is important to have the optimizer use efficient techniques to generate cost-effective execution plans.

In the next chapter, I explain execution plan generation and the techniques the optimizer uses to decide upon an effective execution plan.

CHAPTER 15

Execution Plan Generation

The performance of any query depends on the effectiveness of the execution plan decided upon by the optimizer, as you learned in previous chapters. Because the overall time required to execute a query is the sum of the time required to generate the execution plan plus the time required to execute the query based on this execution plan, it is important that the cost of generating the execution plan itself is low or that a plan gets reused from cache, avoiding that cost altogether. The cost incurred when generating the execution plan depends on the process of generating the execution plan, the process of caching the plan, and the reusability of the plan from the plan cache. In this chapter, you will learn how an execution plan is generated.

In this chapter, I cover the following topics:

- Execution plan generation and caching
- The SQL Server components used to generate an execution plan
- Strategies to optimize the cost of execution plan generation
- Factors affecting parallel plan generation

Execution Plan Generation

SQL Server uses a cost-based optimization technique to determine the processing strategy of a query. The optimizer considers both the metadata of the database objects, such as unique constraints or index size, and the current distribution statistics of the columns referred to in the query when deciding which index and join strategies should be used.

The cost-based optimization allows a database developer to concentrate on implementing a business rule, rather than on the exact syntax of the query. At the same time, the process of determining the query-processing strategy remains quite complex and can consume a fair amount of resources. SQL Server uses a number of techniques to optimize resource consumption.

G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_15

- Syntax-based optimization of the query
- Trivial plan match to avoid in-depth query optimization for simple queries
- Index and join strategies based on current distribution statistics
- Query optimization in stepped phases to control the cost of optimization
- Execution plan caching to avoid the unnecessary regeneration of query plans

The following techniques are performed in order, as shown in Figure 15-1.

1. Parsing
2. Binding
3. Query optimization
4. Execution plan generation, caching, and hash plan generation
5. Query execution

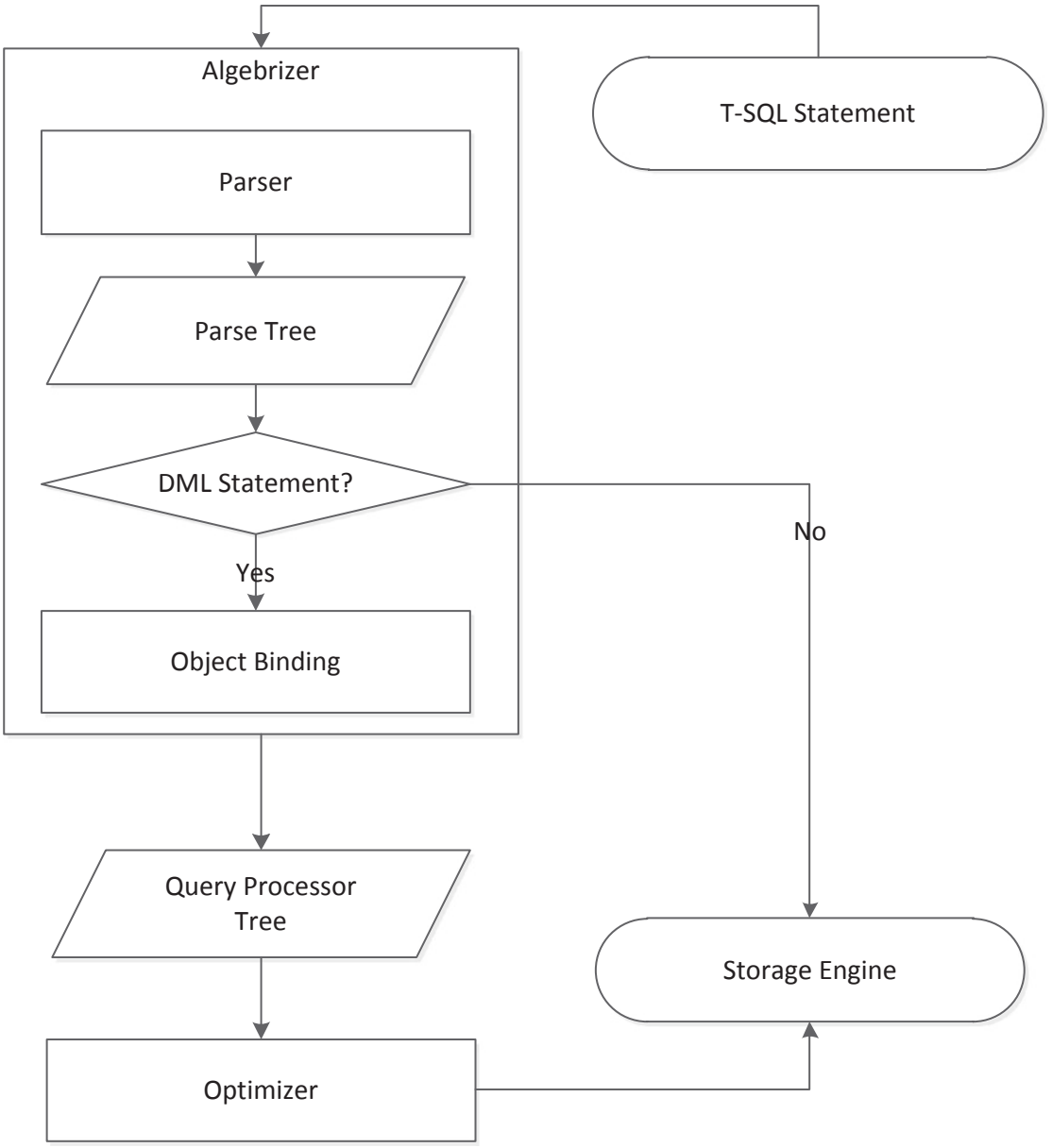


Figure 15-1. *SQL Server techniques to optimize query execution*

Let's take a look at these steps in more detail.

Parser

When a query is submitted, SQL Server passes it to the algebrizer within the *relational engine*. (This relational engine is one of the two main parts of SQL Server data retrieval and manipulation, with the other being the *storage engine*, which is responsible for data access, modifications, and caching.) The relational engine takes care of parsing, name and type resolution, and optimization. It also executes a query as per the query execution plan and requests data from the storage engine.

The first part of the algebrizer process is the parser. The parser checks an incoming query, validating it for the correct syntax. The query is terminated if a syntax error is detected. If multiple queries are submitted together as a batch as follows (note the error in syntax), then the parser checks the complete batch together for syntax and cancels the complete batch when it detects a syntax error. (Note that more than one syntax error may appear in a batch, but the parser goes no further than the first one.)

```
CREATE TABLE dbo.Test1 (c1 INT);
INSERT  INTO dbo.Test1
VALUES  (1);
CEILEKT * FROM dbo.t1; --Error:  I meant,  SELECT * FROM t1
```

On validating a query for correct syntax, the parser generates an internal data structure called a *parse tree* for the algebrizer. The parser and algebrizer taken together are called *query compilation.*

Binding

The parse tree generated by the parser is passed to the next part of the algebrizer for processing. The algebrizer now resolves all the names of the different objects, meaning the tables, the columns, and so on, that are being referenced in the T-SQL in a process called *binding*. It also identifies all the various data types being processed. It even checks for the location of aggregates (such as `GROUP BY` and `MAX`). The output of all these verifications and resolutions is a binary set of data called a *query processor tree.*

To see this part of the algebrizer in action, if the following batch query is submitted, then the first three statements before the error statement are executed, and the errant statement and the one after it are cancelled.

```
IF (SELECT OBJECT_ID('dbo.Test1')) IS NOT NULL
    DROP TABLE dbo.Test1;
GO
CREATE TABLE dbo.Test1 (C1 INT);
INSERT INTO dbo.Test1
VALUES (1);
SELECT 'Before Error',
       C1
FROM dbo.Test1 AS t;
SELECT 'error',
       c1
FROM dbo.no_Test1;
--Error:  Table doesn't exist
SELECT 'after error' AS c1
FROM dbo.Test1 AS t;
```

If a query contains an implicit data conversion, then the normalization process adds an appropriate step to the query tree. The process also performs some syntax-based transformation. For example, if the following query is submitted, then the syntax-based optimization transforms the syntax of the query, as shown in the T-SQL in Figure 15-2 taken from the SELECT operator properties in the execution plan, where BETWEEN becomes >= and <=.

```
SELECT soh.AccountNumber,
       soh.OrderDate,
       soh.PurchaseOrderNumber,
       soh.SalesOrderNumber
FROM Sales.SalesOrderHeader AS soh
WHERE soh.SalesOrderID BETWEEN 62500
                       AND     62550;
```

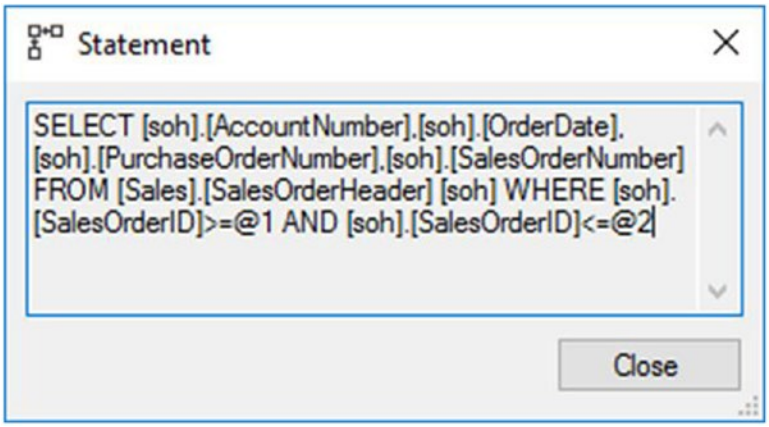


Figure 15-2. *Syntax-based optimization*

You can also see some evidence of parameterization, discussed in more detail later in this chapter. The execution plan generated from the query looks like Figure 15-3.

Figure 15-3. *Execution plan with a warning*

You should also note the warning indicator on the `SELECT` operator. Looking at the properties for this operator, you can see that `SalesOrderID` is actually getting converted as part of the process and the optimizer is warning you.

```
Type conversion in expression (CONVERT(nvarchar(23),[soh].
[SalesOrderID],0)) may affect "CardinalityEstimate" in query plan choice
```

I left this example in, with the warning, to illustrate a couple of points. First, warnings can be unclear. In this case, the warning is coming from the calculated column, `SalesOrderNumber`. It's doing a conversion of the `SalesOrderID` to a string and adding a value to it. In the way it does it, the optimizer recognizes that this could be problematic, so it gives you a warning. But, you're not referencing the column in any kind of filtering fashion such as the `WHERE` clause, `JOIN`s, or `HAVING`. Because of that, you can safely ignore the warning. I also left it in because it illustrates just fine that AdventureWorks is a good example database because it has the same types of poor choices that are sometimes in databases in the real world too.

For most Data Definition Language (DDL) statements (such as `CREATE TABLE`, `CREATE PROC`, and so on), after passing through the algebrizer, the query is compiled directly for execution, since the optimizer need not choose among multiple processing strategies. For one DDL statement in particular, `CREATE INDEX`, the optimizer can determine an efficient processing strategy based on other existing indexes on the table, as explained in Chapter 8.

For this reason, you will never see any reference to `CREATE TABLE` in an execution plan, although you will see reference to `CREATE INDEX`. If the normalized query is a Data Manipulation Language (DML) statement (such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`), then the query processor tree is passed to the optimizer to decide the processing strategy for the query.

Optimization

Based on the complexity of a query, including the number of tables referred to and the indexes available, there may be several ways to execute the query contained in the query processor tree. Exhaustively comparing the cost of all the ways of executing a query can take a considerable amount of time, which may sometimes override the benefit of finding the most optimized query. Figure 15-4 shows that to avoid a high optimization overhead compared to the actual execution cost of the query, the optimizer adopts different techniques, namely, the following:

- Simplification
- Trivial plan match
- Multiple optimization phases
- Parallel plan optimization

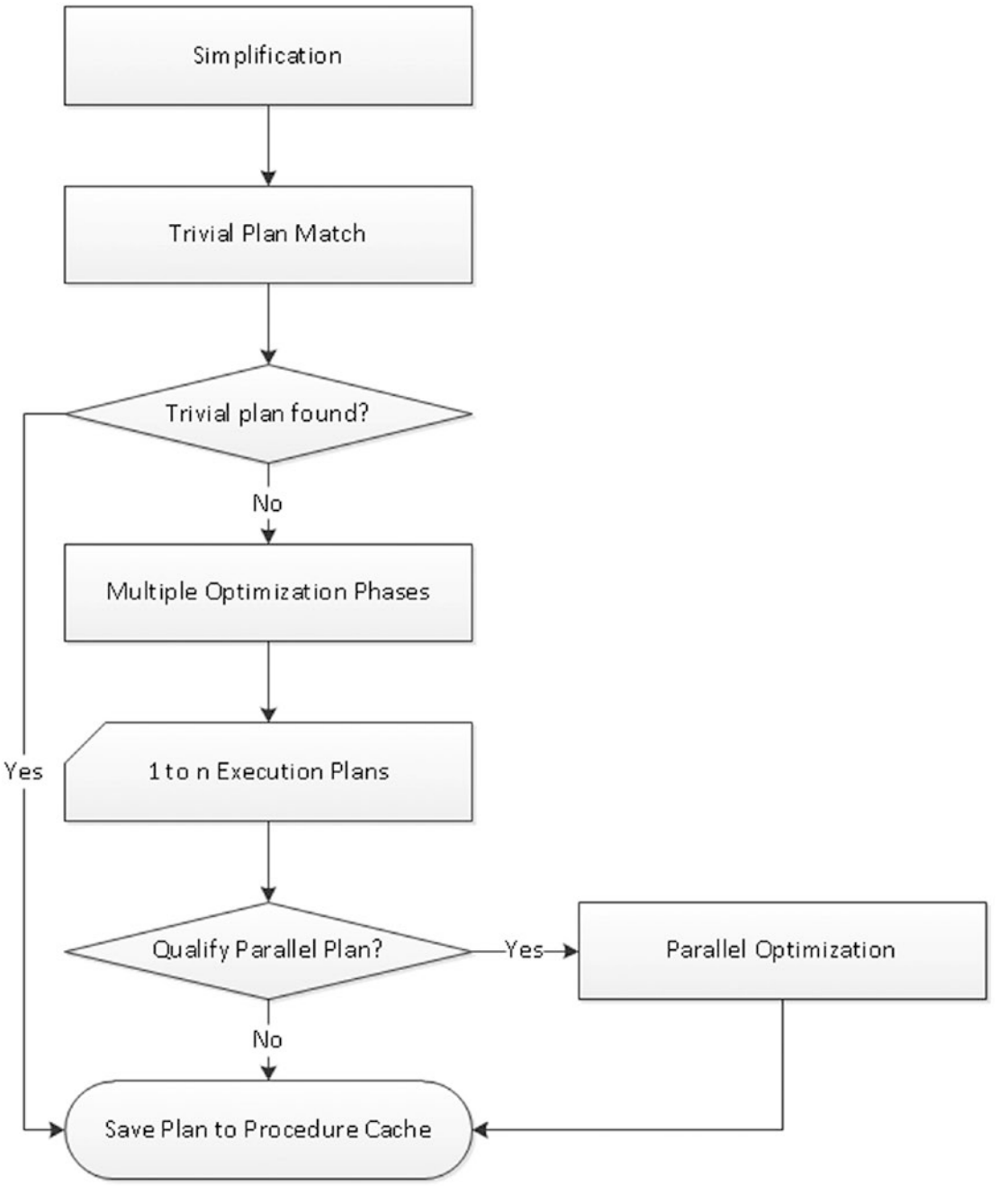


Figure 15-4. *Query optimization steps*

Simplification

Before the optimizer begins to process your query, the logical engine has already identified all the objects referenced in your database. When the optimizer begins to construct your execution plan, it first ensures that all objects being referenced are actually used and necessary to return your data accurately. If you were to write a query with a three-table join but only two of the tables were actually referenced by either the `SELECT` criteria or the `WHERE` clauses, the optimizer may choose to leave the other table out of the processing. This is known as the *simplification* step. It's actually part of a larger set of processing that gathers statistics and starts the process of estimating the cardinality of the data involved in your query. The optimizer also gathers the necessary information about your constraints, especially the foreign key constraints, that will help it later make decisions about the join order, which it can rearrange as needed to arrive at a good enough plan. Also during the Simplification process subqueries get transformed into joins. Other processes of simplification include the removal of redundant joins.

Trivial Plan Match

Sometimes there might be only one way to execute a query. For example, a heap table with no indexes can be accessed in only one way: via a table scan. To avoid the runtime overhead of optimizing such queries, SQL Server maintains a list of patterns that define a trivial plan. If the optimizer finds a match, then a similar plan is generated for the query without any optimization. The generated plans are then stored in the procedure cache. Eliminating the optimization phase means that the cost for generating a trivial plan is very low. This is not to imply that trivial plans are desired or preferable to more complex plans. Trivial plans are available only for extremely simple queries. Once the complexity of the query rises, it must go through optimization.

Multiple Optimization Phases

For a nontrivial query, the number of alternative processing strategies to be analyzed can be high, and it may take a long time to evaluate each option. Therefore, the optimizer goes through three different levels of optimizations. These are referred to as search 0, search 1, and search 2. But it's easier to think of them as *transaction*, *quick plan*, and *full optimization*. Depending on the size and complexity of the query, these different optimizations may be tried one at a time, or the optimizer might skip straight to full optimization. Each of the optimizations takes into account using different join

techniques and different ways of accessing the data through scans, seeks, and other operations.

The index variations consider different indexing aspects, such as single-column index, composite index, index column order, column density, and so forth. Similarly, the join variations consider the different join techniques available in SQL Server: nested loop join, merge join, and hash join. (Chapter 4 covers these join techniques in detail.) Constraints such as unique values and foreign key constraints are also part of the optimization decision-making process.

The optimizer considers the statistics of the columns referred to in the `WHERE`, `JOIN`, and `HAVING` clauses to evaluate the effectiveness of the index and the join strategies. Based on the current statistics, it evaluates the cost of the configurations in multiple optimization phases. The cost includes many factors, including (but not limited to) usage of CPU, memory, and disk I/O (including random versus sequential I/O estimation) required to execute the query. After each optimization phase, the optimizer evaluates the cost of the processing strategy. This cost is an estimation only, not an actual measure or prediction of behavior; it's a mathematical construct based on the statistics and the processes under consideration.

Note The cost estimates are just that, estimates. Further, any one set of estimates represented by an execution plan may or may not in actuality be costlier than another set of estimates, a different execution plan. Comparing the costs between plans can be a dangerous approach.

If the cost is found to be cheap enough, then the optimizer stops further iteration through the optimization phases and quits the optimization process. Otherwise, it keeps iterating through the optimization phases to determine a cost-effective processing strategy.

Sometimes a query can be so complex that the optimizer needs to extensively progress through the optimization phases. While optimizing the query, if it finds that the cost of the processing strategy is more than the cost threshold for parallelism, then it evaluates the cost of processing the query using multiple CPUs. Otherwise, the optimizer proceeds with the serial plan. You may also see that after the optimizer picks a parallel plan, that plan's cost may actually be less than the cost threshold for parallelism and less than the cost of the serial plan.

You can find out some detail of what occurred during the multiple optimization phases via two sources. Take, for example, this query:

```
SELECT soh.SalesOrderNumber,
       sod.OrderQty,
       sod.LineTotal,
       sod.UnitPrice,
       sod.UnitPriceDiscount,
       p.Name AS ProductName,
       p.ProductNumber,
       ps.Name AS ProductSubCategoryName,
       pc.Name AS ProductCategoryName
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
    JOIN Production.Product AS p
        ON sod.ProductID = p.ProductID
    JOIN Production.ProductModel AS pm
        ON p.ProductModelID = pm.ProductModelID
    JOIN Production.ProductSubcategory AS ps
        ON p.ProductSubcategoryID = ps.ProductSubcategoryID
    JOIN Production.ProductCategory AS pc
        ON ps.ProductCategoryID = pc.ProductCategoryID
WHERE soh.CustomerID = 29658;
```

When this query is run, the execution plan in Figure 15-5, a nontrivial plan for sure, is returned.

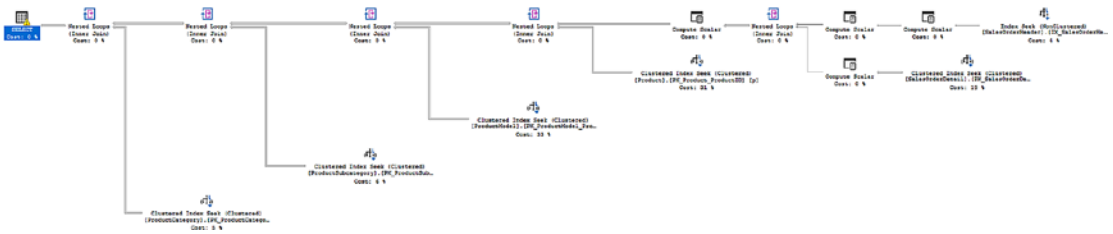
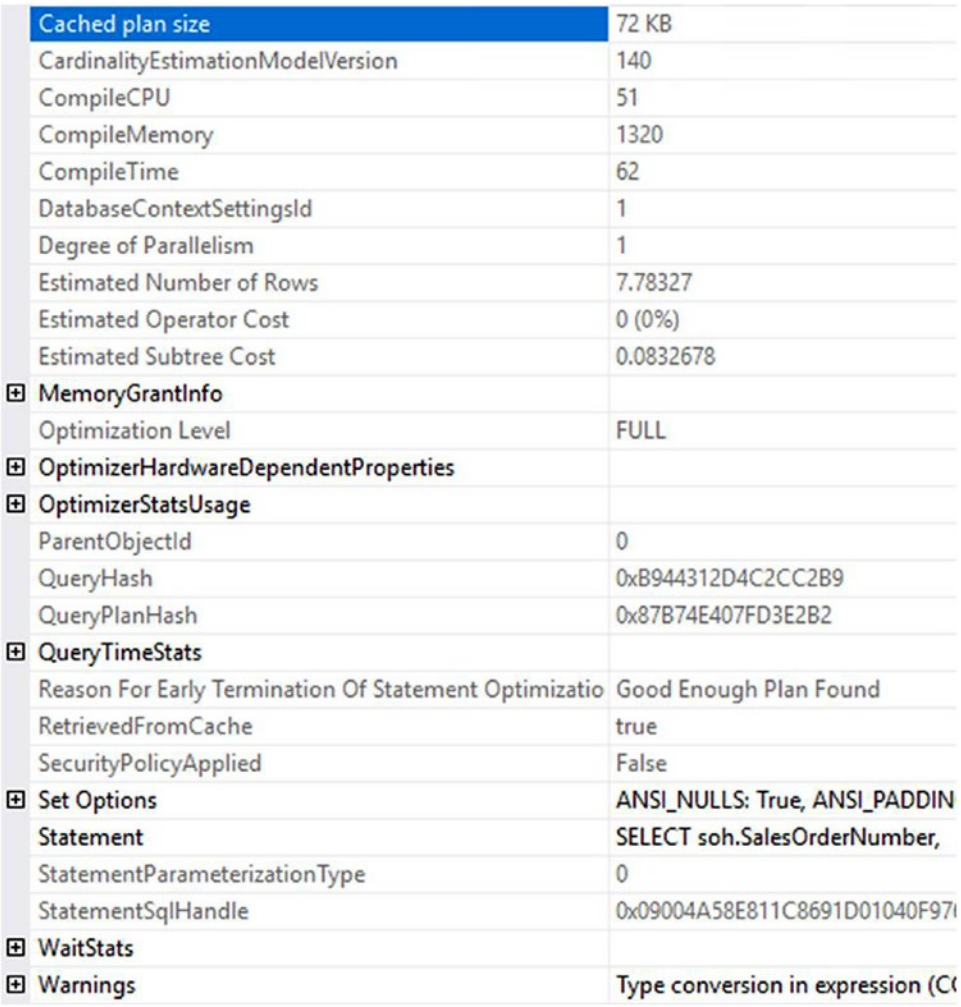


Figure 15-5. *Nontrivial execution plan*

I realize that this execution plan is hard to read. The intent here is not to read through this plan. The important point is that it involves quite a few tables, each with indexes and statistics that all had to be taken into account to arrive at this execution plan. The first place you can go to look for information about the optimizer's work on this execution plan is the property sheet of the first operator, in this case the T-SQL SELECT operator, at the far left of the execution plan. Figure 15-6 shows the property sheet.



Cached plan size	72 KB
CardinalityEstimationModelVersion	140
CompileCPU	51
CompileMemory	1320
CompileTime	62
DatabaseContextSettingsId	1
Degree of Parallelism	1
Estimated Number of Rows	7.78327
Estimated Operator Cost	0 (0%)
Estimated Subtree Cost	0.0832678
⊞ MemoryGrantInfo	
Optimization Level	FULL
⊞ OptimizerHardwareDependentProperties	
⊞ OptimizerStatsUsage	
ParentObjectId	0
QueryHash	0xB944312D4C2CC2B9
QueryPlanHash	0x87B74E407FD3E2B2
⊞ QueryTimeStats	
Reason For Early Termination Of Statement Optimizatio	Good Enough Plan Found
RetrievedFromCache	true
SecurityPolicyApplied	False
⊞ Set Options	ANSI_NULLS: True, ANSI_PADDIN
Statement	SELECT soh.SalesOrderNumber,
StatementParameterizationType	0
StatementSqlHandle	0x09004A58E811C8691D01040F97
⊞ WaitStats	
⊞ Warnings	Type conversion in expression (C

Figure 15-6. *SELECT operator property sheet*

Starting at the top, you can see information directly related to the creation and optimization of this execution plan.

- The size of the cached plan, which is 72KB
- The number of CPU cycles used to compile the plan, which is 51ms
- The amount of memory used, which is 1329KB
- The compile time, which is 62ms

The Optimization Level property (`StatementOptmLevel` in the XML plan) shows what type of processing occurred within the optimizer. In this case, `FULL` means that the optimizer did a full optimization. This is further displayed in the property Reason for Early Termination of Statement, which is Good Enough Plan Found. So, the optimizer took 62ms to track down a plan that it deemed good enough in this situation. You can also see the `QueryPlanHash` value, also known as the *fingerprint*, for the execution plan (you can find more details on this in the section "Query Plan Hash and Query Hash"). The properties of the `SELECT` (and the `INSERT`, `UPDATE`, and `DELETE`) operators are an important first stopping point when evaluating any execution plan because of this information.

Added in SQL Server 2017, you can also see the `QueryTimeStats` and `WaitStats` for any actual execution plan that you capture. This can be a useful way to capture query metrics.

The second source for optimizer information is the dynamic management view `sys.dm_exec_query_optimizer_info`. This DMV is an aggregation of the optimization events over time. It won't show the individual optimizations for a given query, but it will track the optimizations performed. This isn't as immediately handy for tuning an individual query, but if you are working on reducing the costs of a workload over time, being able to track this information can help you determine whether your query tuning is making a positive difference, at least in terms of optimization time. Some of the data returned is for internal SQL Server use only. Figure 15-7 shows a truncated example of the useful data returned in the results from the following query:

```
SELECT deqoi.counter,
       deqoi.occurrence,
       deqoi.value
FROM sys.dm_exec_query_optimizer_info AS deqoi;
```

	counter	occurrence	value
1	optimizations	3911	1
2	elapsed time	3905	0.00768245838668374
3	final cost	3905	34.4891834584044
4	trivial plan	611	1
5	tasks	3294	1687.9335154827
6	no plan	0	NULL
7	search 0	1538	1
8	search 0 time	1564	0.0444891304347826
9	search 0 tasks	1564	2743.25959079284
10	search 1	1746	1
11	search 1 time	1760	0.0159017045454545
12	search 1 tasks	1760	588.889204545455

Figure 15-7. *Output from sys.dm_exec_query_optimizer_info*

Running this query before and after another query can show you the changes that have occurred in the number and type of optimizations completed. However, if you can isolate your queries on a test box, you can be more assured that you get before and after differences that are directly related only to the query you're attempting to measure.

Parallel Plan Optimization

The optimizer considers various factors while evaluating the cost of processing a query using a parallel plan. Some of these factors are as follows:

- Number of CPUs available to SQL Server
- SQL Server edition
- Available memory
- Cost threshold for parallelism
- Type of query being executed
- Number of rows to be processed in a given stream
- Number of active concurrent connections

If only one CPU is available to SQL Server, then the optimizer won't consider a parallel plan. The number of CPUs available to SQL Server can be restricted using the *affinity* setting of the SQL Server configuration. The affinity value is set either to specific CPUs or to specific NUMA nodes. You can also set it to ranges. For example, to allow SQL Server to use only CPUO to CPU3 in an eight-way box, execute these statements:

```
USE master;
EXEC sp_configure 'show advanced option','1';
RECONFIGURE;
ALTER SERVER CONFIGURATION SET PROCESS AFFINITY CPU = 0 TO 3;
GO
```

This configuration takes effect immediately. `affinity` is a special setting, and I recommend you use it only in instances where taking control away from SQL Server makes sense, such as when you have multiple instances of SQL Server running on the same machine and you want to isolate them from each other.

Even if multiple CPUs are available to SQL Server, if an individual query is not allowed to use more than one CPU for execution, then the optimizer discards the parallel plan option. The maximum number of CPUs that can be used for a parallel query is governed by the `max degree of parallelism` setting of the SQL Server configuration. The default value is 0, which allows all the CPUs (availed by the `affinity mask` setting) to be used for a parallel query. You can also control parallelism through the Resource Governor.
If you want to allow parallel queries to use no more than two CPUs out of CPUO to CPU3, limited by the preceding `affinity mask` setting, execute the following statements:

```
USE master;
EXEC sp_configure 'show advanced option','1';
RECONFIGURE;
EXEC sp_configure 'max degree of parallelism',2;
RECONFIGURE;
```

This change takes effect immediately, without any restart. The `max degree of parallelism` setting can also be controlled at a query level using the `MAXDOP` query hint.

```
SELECT  *
FROM    dbo.t1
WHERE   C1 = 1
OPTION  (MAXDOP 2);
```

Changing the `max degree of parallelism` setting is best determined by the needs of your application and the workloads on it. I will usually leave the max degree of parallelism set to the default value unless indications arise that suggest a change is necessary. I will usually immediately adjust the cost threshold for parallelism up from its default value of 5. However, it's really important to understand that the cost threshold is set for the server. Picking a single value that is optimal for all databases on the server may be somewhat tricky.

Since parallel queries require more memory, the optimizer determines the amount of memory available before choosing a parallel plan. The amount of memory required increases with the degree of parallelism. If the memory requirement of the parallel plan for a given degree of parallelism cannot be satisfied, then SQL Server decreases the degree of parallelism automatically or completely abandons the parallel plan for the query in the given workload context. You can see this part of the evaluation in the `SELECT` properties of Figure 15-6.

Queries with a very high CPU overhead are the best candidates for a parallel plan. Examples include joining large tables, performing substantial aggregations, and sorting large result sets, all common operations on reporting systems (less so on OLTP systems). For simple queries usually found in transaction-processing applications, the additional coordination required to initialize, synchronize, and terminate a parallel plan outweighs the potential performance benefit.

Whether a query is simple is determined by comparing the estimated execution cost of the query with a cost threshold. This cost threshold is controlled by the `cost threshold for parallelism` setting of the SQL Server configuration. By default, this setting's value is 5, which means that if the estimated execution cost (CPU and IO) of the serial plan is more than 5, then the optimizer considers a parallel plan for the query. For example, to modify the cost threshold to 35, execute the following statements:

```
USE master;
EXEC sp_configure 'show advanced option','1';
RECONFIGURE;
EXEC sp_configure 'cost threshold for parallelism',35;
RECONFIGURE;
```

This change takes effect immediately, without any restart. If only one CPU is available to SQL Server, then this setting is ignored. I've found that OLTP systems suffer when the cost threshold for parallelism is set this low. Usually increasing the value

to somewhere between 30 and 50 will be beneficial. A lower value can be better for analytical queries. Be sure to test this suggestion against your system to ensure it works well for you.

Another option is to simply look at the plans in your cache and then make an estimate, based on the queries there and the type of workload they represent to arrive at a specific number. You can separate your OLTP queries from your reporting queries and then focus on the reporting queries most likely to benefit from parallel execution. Take an average of those costs and set your cost threshold to that number.

The DML action queries (`INSERT`, `UPDATE`, and `DELETE`) are executed serially. However, the `SELECT` portion of an `INSERT` statement and the `WHERE` clause of an `UPDATE` or a `DELETE` statement can be executed in parallel. The actual data changes are applied serially to the database. Also, if the optimizer determines that the estimated cost is too low, it does not introduce parallel operators.

Note that, even at execution time, SQL Server determines whether the current system workload and configuration information allow for parallel query execution. If parallel query execution is allowed, SQL Server determines the optimal number of threads and spreads the execution of the query across those threads. When a query starts a parallel execution, it uses the same number of threads until completion. SQL Server reexamines the optimal number of threads before executing the parallel query the next time.

Once the processing strategy is finalized by using either a serial plan or a parallel plan, the optimizer generates the execution plan for the query. The execution plan contains the detailed processing strategy decided by the optimizer to execute the query. This includes steps such as data retrieval, result set joins, result set ordering, and so on. A detailed explanation of how to analyze the processing steps included in an execution plan is presented in Chapter 4. The execution plan generated for the query is saved in the plan cache for future reuse.

With all that then, we can summarize the process. The optimizer starts by simplifying and normalizing the input tree. From there it generates possible logical trees that are the equivalent of that simplified tree. Then the optimizer transforms the logical trees into possible physical trees, costs them, and selects the cheapest tree. That's the optimization process in a nutshell.

Execution Plan Caching

The execution plan of a query generated by the optimizer is saved in a special part of SQL Server's memory pool called the *plan cache*. Saving the plan in a cache allows SQL Server to avoid running through the whole query optimization process again when the same query is resubmitted. SQL Server supports different techniques such as *plan cache aging* and *plan cache types* to increase the reusability of the cached plans. It also stores two binary values called the *query hash* and the *query plan hash.*

Note I discuss the techniques supported by SQL Server for improving the effectiveness of execution plan reuse in this Chapter 15.

Components of the Execution Plan

The execution plan generated by the optimizer contains two components.

- *Query plan*: This represents the commands that specify all the physical operations required to execute a query.
- *Execution context*: This maintains the variable parts of a query within the context of a given user.

I will cover these components in more detail in the next sections.

Query Plan

The query plan is a reentrant, read-only data structure, with commands that specify all the physical operations required to execute the query. The reentrant property allows the query plan to be accessed concurrently by multiple connections. The physical operations include specifications on which tables and indexes to access, how and in what order they should be accessed, the type of join operations to be performed between multiple tables, and so forth. No user context is stored in the query plan.

Execution Context

The execution context is another data structure that maintains the variable part of the query. Although the server keeps track of the execution plans in the procedure cache, these plans are context neutral. Therefore, each user executing the query will have a separate execution context that holds data specific to their execution, such as parameter values and connection details.

Aging of the Execution Plan

The plan cache is part of SQL Server's buffer cache, which also holds data pages. As new execution plans are added to the plan cache, the size of the plan cache keeps growing, affecting the retention of useful data pages in memory. To avoid this, SQL Server dynamically controls the retention of the execution plans in the plan cache, retaining the frequently used execution plans and discarding plans that are not used for a certain period of time.

SQL Server keeps track of the frequency of an execution plan's reuse by associating an age field to it. When an execution plan is generated, the age field is populated with the cost of generating the plan. A complex query requiring extensive optimization will have an age field value higher than that for a simpler query.

At regular intervals, the current cost of all the execution plans in the plan cache is examined by SQL Server's lazy writer process (which manages most of the background processes in SQL Server). If an execution plan is not reused for a long time, then the current cost will eventually be reduced to 0. The cheaper the execution plan was to generate, the sooner its cost will be reduced to 0. Once an execution plan's cost reaches 0, the plan becomes a candidate for removal from memory. SQL Server removes all plans with a cost of 0 from the plan cache when memory pressure increases to such an extent that there is no longer enough free memory to serve new requests. However, if a system has enough memory and free memory pages are available to serve new requests, execution plans with a cost of 0 can remain in the plan cache for a long time so that they can be reused later, if required.

As well as changing the costs downward, execution plans can also find their costs increased to the max cost of generating the plan every time the plan is reused (or to the current cost of the plan for ad hoc plans). For example, suppose you have two execution plans with generation costs equal to 100 and 10. Their starting cost values will therefore

be 100 and 10, respectively. If both execution plans are reused immediately, their age fields will be set back to that maximum cost. With these cost values, the lazy writer will bring down the cost of the second plan to 0 much earlier than that of the first one, unless the second plan is reused more often. Therefore, even if a costly plan is reused less frequently than a cheaper plan, because of the effect of the initial cost, the costly plan can remain at a nonzero cost value for a longer period of time.

Summary

SQL Server's cost-based query optimizer decides upon an effective execution plan based not on the exact syntax of the query but on evaluating the cost of executing the query using different processing strategies. The cost evaluation of using different processing strategies is done in multiple optimization phases to avoid spending too much time optimizing a query. Then, the execution plans are cached to save the cost of execution plan generation when the same queries are reexecuted.

In the next chapter, I will discuss how the plans get reused from the cache in different ways depending on how they're called.

CHAPTER 16

Execution Plan Cache Behavior

Once all the processing necessary to generate an execution plan has been completed, it would be crazy for SQL Server to throw away that work and do it all again each time a query gets called. Instead, it saves the plans created in a memory space on the server called the *plan cache*. This chapter will walk through how you can monitor the plan cache to see how SQL Server reuses execution plans.

In this chapter, I cover the following topics:

- How to analyze execution plan caching
- Query plan hash and query hash as mechanisms for identifying queries to tune
- Ways to improve the reusability of execution plan caching
- Interactions between the Query Store and the plan cache

Analyzing the Execution Plan Cache

You can obtain a lot of information about the execution plans in the plan cache by accessing various dynamic management objects. The initial DMO for working with execution plans is `sys.dm_exec_cached_plans`.

```
SELECT decp.refcounts,
       decp.usecounts,
       decp.size_in_bytes,
       decp.cacheobjtype,
```


G. Fritchey, *SQL Server 2017 Query Performance Tuning*, https://doi.org/10.1007/978-1-4842-3888-2_16

```
        decp.objtype,
        decp.plan_handle
FROM sys.dm_exec_cached_plans AS decp;
```

Table 16-1 shows some of the useful information provided by `sys.dm_exec_cached_plans`.

Table 16-1. *sys.dm_exec_cached_plans*

Column Name	Description
`refcounts`	This represents the number of other objects in the cache referencing this plan.
`usecounts`	This is the number of times this object has been used since it was added to the cache.
`size_in_bytes`	This is the size of the plan stored in the cache.
`cacheobjtype`	This specifies what type of plan this is; there are several, but of particular interest are these: Compiled plan: A completed execution plan Compiled plan stub: A marker used for ad hoc queries (you can find more details in the "Ad Hoc Workload" section of this chapter) Parse tree: A plan stored for accessing a view
`Objtype`	This is the type of object that generated the plan. Again, there are several, but these are of particular interest: Proc Prepared Adhoc View

Using the DMV `sys.dm_exec_cached_plans` all by itself gets you only a small part of the information. DMOs are best used in combination with other DMOs and other system views. For example, using the dynamic management function `sys.dm_exec_query_plan(plan_handle)` in combination with `sys.dm_exec_cached_plans` will also bring back the XML execution plan itself so that you can display it and work with it. If you then bring in `sys.dm_exec_sql_text(plan_handle)`, you can also retrieve the

original query text. This may not seem useful while you're running known queries for the examples here, but when you go to your production system and begin to pull in execution plans from the cache, it might be handy to have the original query. To get aggregate performance metrics about the cached plan, you can use `sys.dm_exec_query_stats` for batches, `sys.dm_exec_procedure_stats` for procedures and in-line functions, and `sys.dm_exec_trigger_stats` for returning that same data for triggers. Among other pieces of data, the query hash and query plan hash are stored in this DMF. Finally, to find your way to execution plans for queries that are currently executing, you can use `sys.dm_exec_requests`.

In the following sections, I'll explore how the plan cache works with actual queries of these DMOs.

Execution Plan Reuse

When a query is submitted, SQL Server checks the plan cache for a matching execution plan. If one is not found, then SQL Server performs the query compilation and optimization to generate a new execution plan. However, if the plan exists in the plan cache, it is reused with the private execution context. This saves the CPU cycles that otherwise would have been spent on the plan generation. In the event that a plan is not in the cache but that plan is marked as forced in the Query Store, optimization proceeds as normal, but the forced plan is used instead, assuming it's still a valid plan.

Queries are often submitted to SQL Server with filter criteria to limit the size of the result set. The same queries are often resubmitted with different values for the filter criteria. For example, consider the following query:

```
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = 29690
      AND sod.ProductID = 711;
```

When this query is submitted, the optimizer creates an execution plan and saves it in the plan cache to reuse in the future. If this query is resubmitted with a different filter criterion value—for example, `soh.CustomerlD = 29500`—it will be beneficial to reuse the existing execution plan for the previously supplied filter criterion value (unless this is a bad parameter sniffing scenario). Whether the execution plan created for one filter criterion value can be reused for another filter criterion value depends on how the query is submitted to SQL Server.

The queries (or workload) submitted to SQL Server can be broadly classified into two categories that determine whether the execution plan will be reusable as the value of the variable parts of the query changes.

- Ad hoc
- Prepared

Tip To test the output of `sys.dm_exec_cached_plans` for this chapter, it will be necessary to remove the plans from cache on occasion by executing `DBCC FREEPROCCACHE`. Do not run this on your production server except when you use the methods outlined here, passing a plan handle. Otherwise, you will flush the cache and will require all execution plans to be rebuilt as they are executed, placing a serious strain on your production system for no good reason. You can use `DBCC FREEPROCCACHE(plan_handle)` to target specific plans. Retrieve the `plan_handle` using the DMOs I've already talked about and as demonstrated later. You can also flush the cache for a single database using `ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE CACHE`. However, here again, I do not recommend running this on a production server except when you have intention of removing all plans for that database.

Ad Hoc Workload

Queries can be submitted to SQL Server without explicitly isolating the variables from the query. These types of queries executed without explicitly converting the variable parts of the query into parameters are referred to as *ad hoc workloads* (or queries). Most of the examples in the book so far are ad hoc queries, such as the previous listing.

If the query is submitted as is, without explicitly converting either of the hard-coded values to a parameter (that can be supplied to the query when executed), then the query is an ad hoc query. Setting the values to local variables using the `DECLARE` statement is not the same as parameters.

In this query, the filter criterion value is embedded in the query itself and is not explicitly parameterized to isolate it from the query. This means you cannot reuse the execution plan for this query unless you use the same values and all the spacing and carriage returns are identical. However, the places where values are used in the queries can be explicitly parameterized in three different ways that are jointly categorized as a prepared workload.

Prepared Workload

Prepared workloads (or queries) explicitly parameterize the variable parts of the query so that the query plan isn't tied to the value of the variable parts. In SQL Server, queries can be submitted as prepared workloads using the following three methods:

- *Stored procedures*: Allows saving a collection of SQL statements that can accept and return user-supplied parameters.
- *`sp_executesql`*: Allows executing a SQL statement or a SQL batch that may contain user-supplied parameters, without saving the SQL statement or batch.
- *Prepare/execute model*: Allows a SQL client to request the generation of a query plan that can be reused during subsequent executions of the query with different parameter values, without saving the SQL statements in SQL Server. This is the most common practice for ORM tools such as Entity Framework.

For example, the `SELECT` statement shown previously can be explicitly parameterized using a stored procedure as follows:

```
CREATE OR ALTER PROC dbo.BasicSalesInfo
    @ProductID INT,
    @CustomerID INT
```

```
AS
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = @CustomerID
      AND sod.ProductID = @ProductID;
```

The plan of the SELECT statement included within the stored procedure will embed the parameters (@ProductID and @Customerld), not variable values. I will cover these methods in more detail shortly.

Plan Reusability of an Ad Hoc Workload

When a query is submitted as an ad hoc workload, SQL Server generates an execution plan and stores that plan in the cache, where it can be reused if the same ad hoc query is resubmitted. Since there are no parameters, the hard-coded values are stored as part of the plan. For a plan to be reused from the cache, the T-SQL must match exactly. This includes all spaces and carriage returns plus any values supplied with the plan. If any of these change, the plan cannot be reused.

To understand this, consider the ad hoc query you've used before, shown here:

```
SELECT  soh.SalesOrderNumber,
        soh.OrderDate,
        sod.OrderQty,
        sod.LineTotal
FROM    Sales.SalesOrderHeader AS soh
JOIN    Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE   soh.CustomerID = 29690
        AND sod.ProductID = 711;
```

The execution plan generated for this ad hoc query is based on the exact text of the query, which includes comments, case, trailing spaces, and hard returns. You'll have to use the exact text to pull the information out of `sys.dm_exec_cached_plans`.

```
SELECT    c.usecounts
    ,c.cacheobjtype
    ,c.objtype
FROM       sys.dm_exec_cached_plans c
    CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE      t.text =  'SELECT  soh.SalesOrderNumber,
        soh.OrderDate,
        sod.OrderQty,
        sod.LineTotal
FROM    Sales.SalesOrderHeader AS soh
JOIN    Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE   soh.CustomerID = 29690
        AND sod.ProductID = 711;';
```

Figure 16-1 shows the output of `sys.dm_exec_cached_plans`.

	usecounts	cacheobjtype	objtype
1	1	Compiled Plan	Adhoc

Figure 16-1. *sys.dm_exec_cached_plans output*

You can see from Figure 16-1 that a compiled plan is generated and saved in the plan cache for the preceding ad hoc query. To find the specific query, I used the query itself in the `WHERE` clause. You can see that this plan has been used once up until now (`usecounts = 1`). If this ad hoc query is reexecuted, SQL Server reuses the existing executable plan from the plan cache, as shown in Figure 16-2.

	usecounts	cacheobjtype	objtype
1	2	Compiled Plan	Adhoc

Figure 16-2. *Reusing the executable plan from the plan cache*

In Figure 16-2, you can see that the usecounts value for the preceding query's executable plan has increased to 2, confirming that the existing plan for this query has been reused. If this query is executed repeatedly, the existing plan will be reused every time.

Since the plan generated for the preceding query includes the filter criterion value, the reusability of the plan is limited to the use of the same filter criterion value. Reexecute the query, but change son.CustomerlD to 29500.

```
SELECT soh.SalesOrderNumber,
       soh.OrderDate,
       sod.OrderQty,
       sod.LineTotal
FROM Sales.SalesOrderHeader AS soh
    JOIN Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID
WHERE soh.CustomerID = 29500
      AND sod.ProductID = 711;
```

The existing plan can't be reused, and if sys.dm_exec_cached_plans is rerun as is, you'll see that the execution count hasn't increased (Figure 16-3).

	usecounts	cacheobjtype	objtype
1	2	Compiled Plan	Adhoc

Figure 16-3. *sys.dm_exec_cached_plans shows that the existing plan is not reused*

Instead, I'll adjust the query against sys.dm_exec_cached_plans.

```
SELECT  c.usecounts,
        c.cacheobjtype,
        c.objtype,
        t.text,
        c.plan_handle
FROM    sys.dm_exec_cached_plans c
CROSS APPLY sys.dm_exec_sql_text(c.plan_handle) t
WHERE   t.text LIKE 'SELECT  soh.SalesOrderNumber,
        soh.OrderDate,
```

```
        sod.OrderQty,
        sod.LineTotal
FROM    Sales.SalesOrderHeader AS soh
JOIN    Sales.SalesOrderDetail AS sod
        ON soh.SalesOrderID = sod.SalesOrderID%';
```

You can see the output from this query in Figure 16-4.

	usecounts	cacheobjtype	objtype	text	plan_handle
1	1	Compiled Plan	Adhoc	SELECT soh.SalesOrderNumber, soh.Order...	0x06000500FE3F7918600F257302000000010000000000000...
2	2	Compiled Plan	Adhoc	SELECT soh.SalesOrderNumber, soh.Order...	0x060005003176D518401F257302000000010000000000000...

Figure 16-4. *sys.dm_exec_cached_plans showing that the existing plan can't be reused*

From the `sys.dm_exec_cached_plans` output in Figure 16-4, you can see that the previous plan for the query hasn't been reused; the corresponding `usecounts` value remained at the old value of 2. Instead of reusing the existing plan, a new plan is generated for the query and is saved in the plan cache with a new `plan_handle`. If this ad hoc query is reexecuted repeatedly with different filter criterion values, a new execution plan will be generated every time. The inefficient reuse of the execution plan for this ad hoc query increases the load on the CPU by consuming additional CPU cycles to regenerate the plan.

To summarize, ad hoc plan caching uses statement-level caching and is limited to an exact textual match. If an ad hoc query is not complex, SQL Server can implicitly parameterize the query to increase plan reusability by using a feature called *simple parameterization.* The definition of a query for simple parameterization is limited to quite basic cases such as ad hoc queries with only one table. As shown in the previous example, most queries requiring a join operation cannot be autoparameterized.

Optimize for an Ad Hoc Workload

If your server is going to primarily support ad hoc queries, it is possible to achieve a small degree of performance improvement. One server option is called `optimize for ad hoc workloads`. Enabling this for the server changes the way the engine deals with ad hoc queries. Instead of saving a full compiled plan for the query the first time it's called, a compiled plan stub is stored. The stub does not have a full execution plan associated,

saving the storage space required for it and the time saving it to the cache. This option can be enabled without rebooting the server.

```
EXEC sp_configure 'show advanced option', '1';
GO
RECONFIGURE
GO
EXEC sp_configure 'optimize for ad hoc workloads', 1;
GO
RECONFIGURE;
```

After changing the option, flush the cache and then rerun the ad hoc query. Modify the query against `sys.dm_exec_cached_plans` so that you include the `size_in_bytes` column; then run it to see the results in Figure 16-5.

	usecounts	cacheobjtype	objtype	text	size_in_bytes
1	1	Compiled Plan Stub	Adhoc	SELECT soh.SalesOrderNumber, soh.OrderDa...	424

Figure 16-5. *sys.dm_exec_cached_plans showing a compiled plan stub*

Figure 16-5 shows in the `cacheobjtype` column that the new object in the cache is a compiled plan stub. Stubs can be created for lots more queries with less impact on the server than full compiled plans. But the next time an ad hoc query is executed, a fully compiled plan is created. To see this in action, run the query one more time and check the results in `sys.dm_exec_cachedplans`, as shown in Figure 16-6.

	usecounts	cacheobjtype	objtype	text	size_in_bytes
1	1	Compiled Plan	Adhoc	SELECT soh.SalesOrderNumber, soh.OrderDa...	73728

Figure 16-6. *The compiled plan stub has become a compiled plan*

Check the `cacheobjtype` value. It has changed from Compiled Plan Stub to Compiled Plan. Finally, to see the real difference between a stub and a full plan, check the `sizeinbytes` column in Figure 16-5 and Figure 16-6. The size changed from 424 in the stub to 73728 in the full plan. This shows precisely the savings available when working with lots of ad hoc queries. Before proceeding, be sure to disable `optimize for ad hoc workloads`.

```
EXEC sp_configure 'optimize for ad hoc workloads', 0;
GO
RECONFIGURE;
GO
EXEC sp_configure 'show advanced option', '0';
GO
RECONFIGURE;
```

Personally, I see little downside to implementing this on just about any system. Like with all recommendations, you should test it to ensure your system isn't exceptional. However, the cost of writing the plan into memory when it's called a second time is extremely trivial to the savings in memory overall that you see by not storing plans that are only ever going to be used once. In all my testing and experience, this is a pure benefit with little downside. You can now use a database-scoped configuration setting to enable this in your Azure SQL Database too:

```
ALTER DATABASE SCOPED CONFIGURATION SET OPTIMIZE_FOR_AD_HOC_WORKLOADS = ON;
```

Simple Parameterization

When an ad hoc query is submitted, SQL Server analyzes the query to determine which parts of the incoming text might be parameters. It looks at the variable parts of the ad hoc query to determine whether it will be safe to parameterize them automatically and use the parameters (instead of the variable parts) in the query so that the query plan can be independent of the variable values. This feature of automatically converting the variable part of a query into a parameter, even though not parameterized explicitly (using a prepared workload technique), is called *simple parameterization.*

During simple parameterization, SQL Server ensures that if the ad hoc query is converted to a parameterized template, the changes in the parameter values won't widely change the plan requirement. On determining the simple parameterization to be safe, SQL Server creates a parameterized template for the ad hoc query and saves the parameterized plan in the plan cache.

To understand the simple parameterization feature of SQL Server, consider the following query:

```
SELECT *
FROM Person.Address AS a
WHERE a.AddressID = 42;
```

When this ad hoc query is submitted, SQL Server can treat this query as it is for plan creation. However, before the query is executed, SQL Server tries to determine whether it can be safely parameterized. On determining that the variable part of the query can be parameterized without affecting the basic structure of the query, SQL Server parameterizes the query and generates a plan for the parameterized query. You can observe this from the `sys.dm_exec_cached_plans` output shown in Figure 16-7.

	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtype, t.text FROM s...
2	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.AddressID = 42;
3	1	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [Person].[Address] [a] WHERE [a].[AddressID]=@1
4	2	Parse Tree	View	CREATE FUNCTION sys.dm_exec_sql_text(@handle varbinary(64)) RETURNS...

Figure 16-7. *sys.dm_exec_cached_plans output showing an autoparameterized plan*

The `usecounts` of the executable plan for the parameterized query appropriately represents the number of reuses as 1. Also, note that the `objtype` for the autoparameterized executable plan is no longer `Adhoc`; it reflects the fact that the plan is for a parameterized query, Prepared.

The original ad hoc query, even though not executed, gets compiled to create the query tree required for the simple parameterization of the query. The compiled plan for the ad hoc query will be saved in the plan cache. But before creating the executable plan for the ad hoc query, SQL Server figured out that it was safe to autoparameterize and thus autoparameterized the query for further processing.

The parameter values are based on the value of the ad hoc query. Let's edit the previous query to use a different `AddressID` value.

```
SELECT *
FROM Person.Address AS a
WHERE a.AddressID = 42000;
```

If we requery `sys.dm_exec_cached_plans`, we'll see an additional plan has been added, as shown in Figure 16-8.

	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
2	1	Compiled Plan	Prepared	(@1 int)SELECT * FROM [Person].[Address] [a] WHERE ...
3	2	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtyp...
4	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
5	1	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [Person].[Address] [a] WHE...
6	2	Parse Tree	View	CREATE FUNCTION sys.dm_exec_sql_text(@handle var...

Figure 16-8. *An additional plan with simple parameterization*

As you can see in Figure 16-8, a new plan with a parameter with a data type of `int` has been created. You can see plans for `smallint` and `bigint`. This does add some overhead to the cache but not as much as would be added by the large number of additional plans necessary for the wide variety of values. Here's the full query text from the simple parameterization:

```
(@1 int)SELECT * FROM [Person].[Address] [a] WHERE [a].[AddressID]=@1
```

Since this ad hoc query has been autoparameterized, SQL Server will reuse the existing execution plan if you reexecute the query with a different value for the variable part.

```
SELECT *
FROM Person.Address AS a
WHERE a.AddressID = 52;
```

Figure 16-9 shows the output of `sys.dm_exec_cached_plans`.

	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtyp...
2	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
3	1	Compiled Plan	Adhoc	SELECT * FROM Person.Address AS a WHERE a.Addr...
4	2	Compiled Plan	Prepared	(@1 tinyint)SELECT * FROM [Person].[Address] [a] WHE...

Figure 16-9. *sys.dm_exec_cached_plans output showing reuse of the autoparameterized plan*

From Figure 16-9, you can see that although a new plan has been generated for this ad hoc query, the ad hoc one using an `AddressId` value of 52, the existing prepared plan is reused as indicated by the increase in the corresponding `usecounts` value to 2. The ad hoc query can be reexecuted repeatedly with different filter criterion values, reusing the existing execution plan—all this despite that the original text of the two queries does not match. The parameterized query for both would be the same, so it was reused.

There is one more aspect to note in the parameterized query for which the execution plan is cached. In Figure 16-7, observe that the body of the parameterized query doesn't exactly match with that of the ad hoc query submitted. For instance, in the ad hoc query, there are no square brackets on any of the objects.

On realizing that the ad hoc query can be safely autoparameterized, SQL Server picks a template that can be used instead of the exact text of the query.

To understand the significance of this, consider the following query:

```
SELECT  a.*
FROM    Person.Address AS a
WHERE   a.AddressID BETWEEN 40 AND 60;
```

Figure 16-10 shows the output of `sys.dm_exec_cached_plans`.

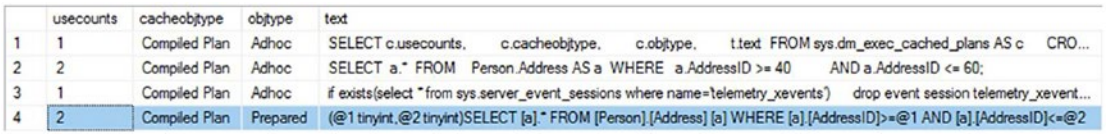
	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtype, t.text FROM sys.dm_exec_cached_plans AS c CRO...
2	1	Compiled Plan	Adhoc	SELECT a.* FROM Person.Address AS a WHERE a.AddressID BETWEEN 40 AND 60;
3	1	Compiled Plan	Prepared	(@1 tinyint,@2 tinyint)SELECT [a].* FROM [Person].[Address] [a] WHERE [a].[AddressID]>=@1 AND [a].[AddressID]<=@2

Figure 16-10. *sys.dm_exec_cached_plans output showing plan simple parameterization using a template*

From Figure 16-10, you can see that SQL Server put the query through the simplification process and substituted a pair of >= and <= operators, which are equivalent to the `BETWEEN` operator. Then the parameterization step modified the query again. That means instead of resubmitting the preceding ad hoc query using the `BETWEEN` clause, if a similar query using a pair of >= and <= is submitted, SQL Server will be able to reuse the existing execution plan. To confirm this behavior, let's modify the ad hoc query as follows:

```
SELECT  a.*
FROM    Person.Address AS a
WHERE   a.AddressID >= 40
        AND a.AddressID <= 60;
```

Figure 16-11 shows the output of `sys.dm_exec_cached_plans`.



	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT c.usecounts, c.cacheobjtype, c.objtype, t.text FROM sys.dm_exec_cached_plans AS c CRO...
2	2	Compiled Plan	Adhoc	SELECT a.* FROM Person.Address AS a WHERE a.AddressID >= 40 AND a.AddressID <= 60;
3	1	Compiled Plan	Adhoc	if exists(select * from sys.server_event_sessions where name='telemetry_xevents') drop event session telemetry_xevent...
4	2	Compiled Plan	Prepared	(@1 tinyint,@2 tinyint)SELECT [a].* FROM [Person].[Address] [a] WHERE [a].[AddressID]>=@1 AND [a].[AddressID]<=@2

Figure 16-11. *sys.dm_exec_cached_plans output showing reuse of the autoparameterized plan*

From Figure 16-11, you can see that the existing plan is reused, even though the query is syntactically different from the query executed earlier. The autoparameterized plan generated by SQL Server allows the existing plan to be reused not only when the query is resubmitted with different variable values but also for queries with the same template form.

Simple Parameterization Limits

SQL Server is highly conservative during simple parameterization because the cost of a bad plan can far outweigh the cost of generating a new plan. The conservative approach prevents SQL Server from creating an unsafe autoparameterized plan. Thus, simple parameterization is limited to fairly simple cases, such as ad hoc queries with only one table. An ad hoc query with a join operation between two (or more) tables (as shown in the early part of the "Plan Reusability of an Ad Hoc Workload" section) is not considered safe for simple parameterization.

In a scalable system, do not rely on simple parameterization for plan reusability. The simple parameterization feature of SQL Server makes an educated guess as to which variables and constants can be parameterized. Instead of relying on SQL Server for simple parameterization, you should actually specify it programmatically while building your application.

Forced Parameterization

If the system you're working on consists primarily of ad hoc queries, you may want to attempt to increase the number of queries that accept parameterization. You can modify a database to attempt to force, within certain restrictions, all queries to be parameterized just like in simple parameterization.

To do this, you have to change the database option PARAMETERIZATION to FORCED using ALTER DATABASE like this:

```
ALTER DATABASE AdventureWorks2017 SET PARAMETERIZATION FORCED;
```

But, if you have a query that is in any way complicated, you won't get simple parameterization.

```
SELECT ea.EmailAddress,
       e.BirthDate,
       a.City
FROM Person.Person AS p
    JOIN HumanResources.Employee AS e
        ON p.BusinessEntityID = e.BusinessEntityID
    JOIN Person.BusinessEntityAddress AS bea
        ON e.BusinessEntityID = bea.BusinessEntityID
    JOIN Person.Address AS a
        ON bea.AddressID = a.AddressID
    JOIN Person.StateProvince AS sp
        ON a.StateProvinceID = sp.StateProvinceID
    JOIN Person.EmailAddress AS ea
        ON p.BusinessEntityID = ea.BusinessEntityID
WHERE ea.EmailAddress LIKE 'david%'
      AND sp.StateProvinceCode = 'WA';
```

When you run this query, simple parameterization is not applied, as you can see in Figure 16-12.

	usecounts	cacheobjtype	objtype	text
1	1	Compiled Plan	Adhoc	SELECT ea.EmailAddress, e.BirthDate, a.City ...

Figure 16-12. *A more complicated query doesn't get parameterized*

No prepared plans are visible in the output from sys.dm_exec_cached_plans. But if we use the previous script to set PARAMETERIZATION to FORCED, we can rerun the query after clearing the cache.

```
ALTER DATABASE SCOPED CONFIGURATION CLEAR PROCEDURE_CACHE;
```