Recommendations

To take advantage of covering indexes, be careful with the column list in SELECT statements to move only the data you need to (thus the standard prohibition against SELECT *). It's also a good idea to use as few columns as possible to keep the index key size small for the covering indexes. Add columns using the INCLUDE statement in places where it makes sense. Since a covering index includes all the columns used in a query, it has a tendency to be very wide, increasing the maintenance cost of the covering indexes. You must balance the maintenance cost with the performance gain that the covering index brings. If the number of bytes from all the columns in the index is small compared to the number of bytes in a single data row of that table and you are certain the query taking advantage of the covered index will be executed frequently, then it may be beneficial to use a covering index.

Tip Covering indexes can also help resolve blocking and deadlocks, as you will see in Chapters 20 and 21.

Before building a lot of covering indexes, consider how SQL Server can effectively and automatically create covering indexes for queries on the fly using index intersection.

Index Intersections

If a table has multiple indexes, then SQL Server can use multiple indexes to execute a query. SQL Server can take advantage of multiple indexes, selecting small subsets of data based on each index and then performing an intersection of the two subsets (that is, returning only those rows that meet all the criteria). SQL Server can exploit multiple indexes on a table and then employ a join algorithm to obtain the *index intersection* between the two subsets.

In the following SELECT statement, for the WHERE clause columns, the table has a nonclustered index on the SalesPersonID column, but it has no index on the OrderDate column:

```
--SELECT * is intentionally used in this query
SELECT soh.*
FROM Sales.SalesOrderHeader AS soh
```

```
WHERE soh.SalesPersonID = 276
      AND soh.OrderDate
      BETWEEN '4/1/2005' AND '7/1/2005';
```

Figure 9-3 shows the execution plan for this query.

Figure 9-3. *Execution plan with no index on the OrderDate column*

As you can see, the optimizer didn't use the nonclustered index on the `SalesPersonID` column. Since the value of the `OrderDate` column is also required, the optimizer chose the clustered index to fetch the value of all the referred columns. The I/O and time for retrieving this data was as follows:

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 689
CPU time = 0 ms,  elapsed time = 3 ms.
```

To improve the performance of the query, the `OrderDate` column can be added to the nonclustered index on the `SalesPersonId` column or defined as an included column on the same index. But in this real-world scenario, you may have to consider the following while modifying an existing index:

- It may not be permissible to modify an existing index for various reasons.
- The existing nonclustered index key may be already quite wide.
- The cost of other queries using the existing index will be affected by the modification.

In such cases, you can create a new nonclustered index on the `OrderDate` column.

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader (OrderDate);
```

Run your `SELECT` command again.
Figure 9-4 shows the resultant execution plan of the `SELECT` statement.

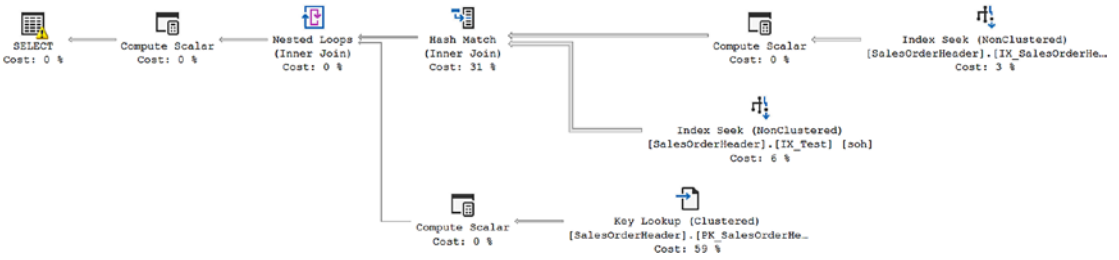

Figure 9-4. *Execution plan with an index on the OrderDate column*

As you can see, SQL Server exploited both the nonclustered indexes as index seeks (rather than scans) and then employed an intersection algorithm to obtain the index intersection of the two subsets. This is represented by the `Hash Join`. It then did a `Key Lookup` from the resulting data set to retrieve the rest of the data not included in the indexes. But, the complexity of the plan suggests that performance might be worse. Checking the statistics I/O and time, you can see that in fact you did get a good performance improvement:

```
Table 'SalesOrderHeader'. Scan count 2, logical reads 10
Table 'Workfile'. Scan count 0, logical reads 0,
Table 'Worktable'. Scan count 0, logical reads 0
CPU time = 0 ms,  elapsed time = 2 ms.
```

The reads dropped from 689 to 10 even though the plan used three different access points within the table and had to create storage for processing the `Hash Join`. The execution time also dropped (3,333µs to 2,279µs in Extended Events). You can also see there are additional operations occurring within the plan, such as the `Key Lookup`, that you might be able to eliminate with further adjustments to the indexes. However, it's worth noting, since you're returning all the columns through the `SELECT *` command, that you can't effectively eliminate the `Key Lookup` by using `INCLUDE` columns, so you may also need to adjust the query.

To improve the performance of a query, SQL Server can use multiple indexes on a table, although it is somewhat rare since it requires good statistics and precise indexes for the specific query. Generally, I try to use smaller, narrower keys on my indexes instead of wide keys. SQL Server can use indexes together frequently, and even when it doesn't, performance is better with narrow indexes. While creating a covering index, identify the existing nonclustered indexes that include most of the columns required by the covering index. You may already have two existing nonclustered indexes that jointly

serve all the columns required by the covering index. If it is possible, rearrange the column order of the existing nonclustered indexes appropriately, allowing the optimizer to consider an index intersection between the two nonclustered indexes.

At times, it is possible that you may have to create a separate nonclustered index for the following reasons:

- Reordering the columns in one of the existing indexes is not allowed.
- Some of the columns required by the covering index may not be included in the existing nonclustered indexes.
- The total number of columns in the existing nonclustered indexes may be more than the number of columns required by the covering index.

In such cases, you can create a nonclustered index on the remaining columns. If the combined column order of the new index and an existing nonclustered index meets the requirement of the covering index, the optimizer may be able to use index intersection. While identifying the columns and their order for the new index, try to maximize their benefit by keeping an eye on other queries, too.

Don't count on frequently getting index intersection to work. It's dependent on the choices made internally by the optimizer. However, there's nothing wrong with striving in this direction when creating your indexes.

Drop the index that was created for the tests.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Index Joins

The *index join* is a variation of index intersection, where the covering index technique is applied to the index intersection. If no single index covers a query but multiple indexes together can cover the query, SQL Server can use an index join to satisfy the query fully without going to the base table.

Let's look at this indexing technique at work. Make a slight modification to the query from the "Index Intersections" section like this:

```
SELECT soh.SalesPersonID,
       soh.OrderDate
FROM Sales.SalesOrderHeader AS soh
```

```
WHERE soh.SalesPersonID = 276
      AND soh.OrderDate
      BETWEEN '4/1/2013' AND '7/1/2013';
```

The execution plan for this query is shown in Figure 9-5, and the reads are as follows:

```
Table  'SalesOrderHeader'.  Scan count 1,  logical reads 689
CPU time = 0 ms,  elapsed time = 2 ms. (2345 us)
```

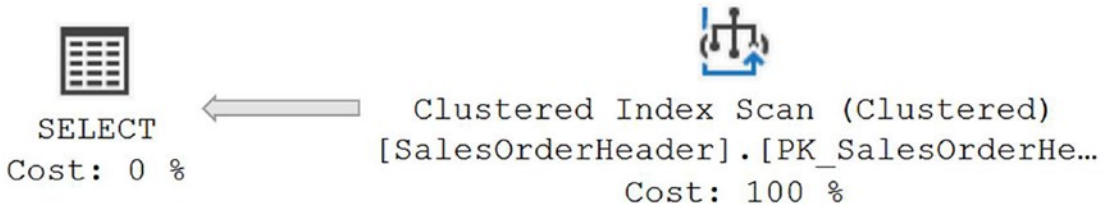

Figure 9-5. *Execution plan with no index join*

As shown in Figure 9-5, the optimizer didn't use the existing nonclustered index on the `SalesPersonID` column. Since the query requires the value of the `OrderDate` column also, the optimizer selected the clustered index to retrieve values for all the columns referred to in the query. If an index is created on the `OrderDate` column like this:

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader (OrderDate ASC);
```

and the query is rerun, then Figure 9-6 shows the result, and you can see the reads here:

```
Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'SalesOrderHeader'. Scan count 2, logical reads 10
CPU time = 0 ms,  elapsed time = 1 ms (1657 us).
```

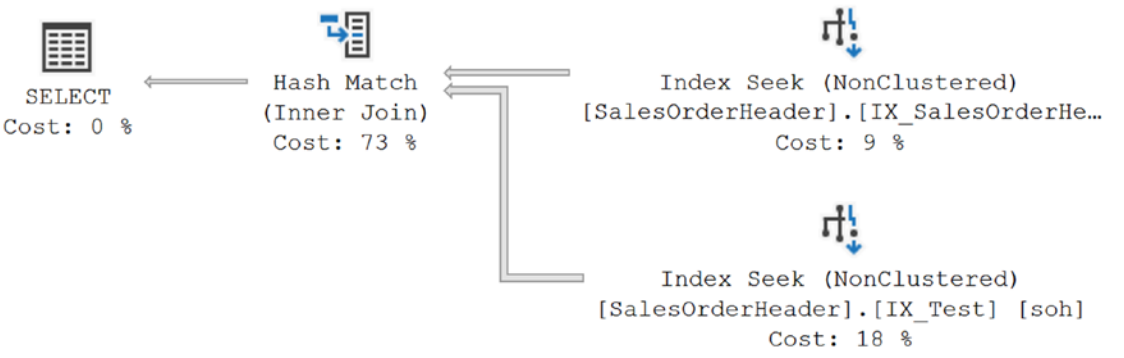

Figure 9-6. *Execution plan with an index join*

The combination of the two indexes acts like a covering index, reducing the reads against the table from 689 to 10 because it's using two Index Seek operations joined together instead of a clustered index scan.

But what if the WHERE clause didn't result in both indexes being used? Instead, you know that both indexes exist and that a seek against each would work like the previous query, so you choose to use an index hint.

```
SELECT soh.SalesPersonID,
       soh.OrderDate
FROM Sales.SalesOrderHeader AS soh
             WITH (INDEX(IX_Test, IX_SalesOrderHeader_SalesPersonID))
WHERE soh.OrderDate
BETWEEN '4/1/2013' AND '7/1/2013';
```

The results of this new query are shown in Figure 9-7, and the I/O is as follows:

```
Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'SalesOrderHeader'. Scan count 2, logical reads 64
CPU time = 0 ms,  elapsed time = 68 ms.
```

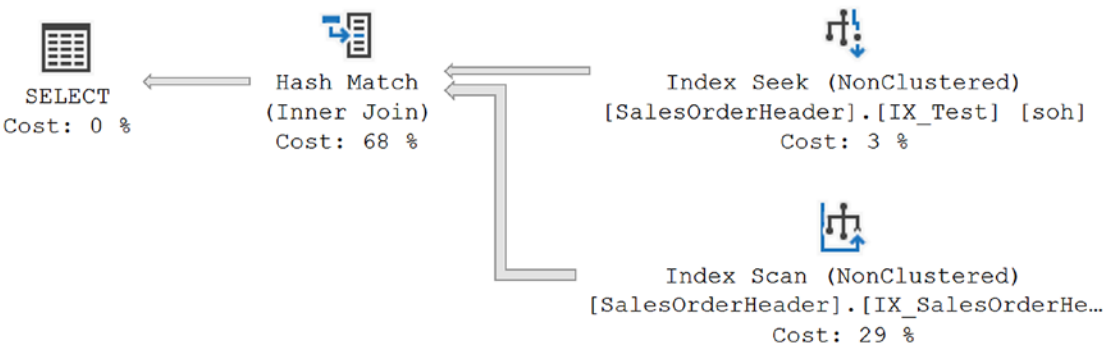

Figure 9-7. *Execution plan with index join through a hint*

The reads have clearly increased as has the execution time. Most of the time, the optimizer makes good choices when it comes to indexes and execution plans. Although query hints are available to allow you to take control from the optimizer, this control can cause as many problems as it solves. In attempting to force an index join as a performance benefit, instead the forced selection of indexes slowed down the execution of the query.

Remove the test index before continuing.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Note While generating a query execution plan, the SQL Server optimizer goes through the optimization phases not only to determine the type of index and join strategy to be used but also to evaluate the advanced indexing techniques such as index intersection and index join. Therefore, in some cases, instead of creating wide covering indexes, consider creating multiple narrow indexes. SQL Server can use them together to serve as a covering index yet use them separately where required. But you will need to test to be sure which works better in your situation—wider indexes or index intersections and joins.

Filtered Indexes

A filtered index is a nonclustered index that uses a filter, basically a `WHERE` clause, to ideally create a highly selective set of keys against a column or columns that may not have good selectivity otherwise. For example, a column with a large number of `NULL` values may be stored as a sparse column to reduce the overhead of those `NULL` values. Adding a filtered index using the column will allow you to have an index available on the data that is not `NULL`. The best way to understand this is to see it in action.

The `Sales.SalesOrderHeader` table has more than 30,000 rows. Of those rows, 27,000+ have a `null` value in the `PurchaseOrderNumber` column and the `SalesPersonId` column. If you wanted to get a simple list of purchase order numbers, the query might look like this:

```
SELECT soh.PurchaseOrderNumber,
       soh.OrderDate,
       soh.ShipDate,
       soh.SalesPersonID
FROM Sales.SalesOrderHeader AS soh
WHERE PurchaseOrderNumber LIKE 'PO5%'
      AND soh.SalesPersonID IS NOT NULL;
```

Running the query results in, as you might expect, a clustered index scan, and the following I/O and execution time, as shown in Figure 9-8:

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 689
CPU time = 0 ms,  elapsed time = 52 ms.
```

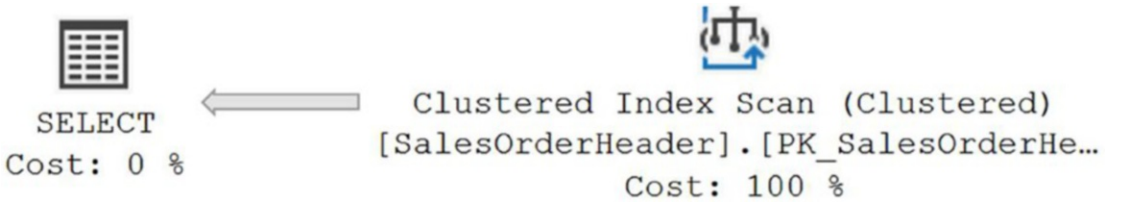

Figure 9-8. *Execution plan without an index*

To fix this, it is possible to create an index and include some of the columns from the query to make this a covering index.

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader
(
    PurchaseOrderNumber,
    SalesPersonID
)
INCLUDE
(
    OrderDate,
    ShipDate
);
```

When you rerun the query, the performance improvement is fairly radical (see Figure 9-9 and the I/O and time in the following result).

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 5
CPU time = 0 ms,  elapsed time = 40 ms.
```

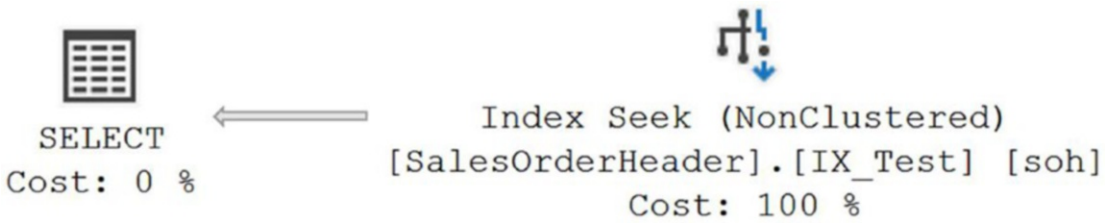

Figure 9-9. *Execution plan with a covering index*

As you can see, the covering index dropped the reads from 689 to 5 and the time from 52ms to 40ms. Normally, this would be considered a decent improvement and may be adequate for the system. Assume for a moment that this query has to be called frequently. Now, every bit of speed you can wring from it will pay dividends. Knowing that so much of the data in the indexed columns is `null`, you can adjust the index so that it filters out the `null` values, which aren't used by the index anyway, reducing the size of the tree and therefore the amount of searching required.

```
CREATE NONCLUSTERED INDEX IX_Test
ON Sales.SalesOrderHeader
(
    PurchaseOrderNumber,
    SalesPersonID
)
INCLUDE
(
    OrderDate,
    ShipDate
)
WHERE PurchaseOrderNumber IS NOT NULL
      AND SalesPersonID IS NOT NULL
WITH (DROP_EXISTING = ON);
```

The final run of the query resulted in the following performance metrics:

```
Table 'SalesOrderHeader'. Scan count 1, logical reads 4
CPU time = 0 ms,    elapsed time = 38 ms.
```

The execution plan is going to look identical, with an `Index Seek`. To see the differences between the plan for the covering index and the plan for the filtered, covering index, we can use SSMS to compare the plans. Save the first plan as a file (right-click the plan and select Save Execution Plan As), and then, from the second plan, right-click inside the plan and select Compare Plan. You'll then see something similar to Figure 9-10.

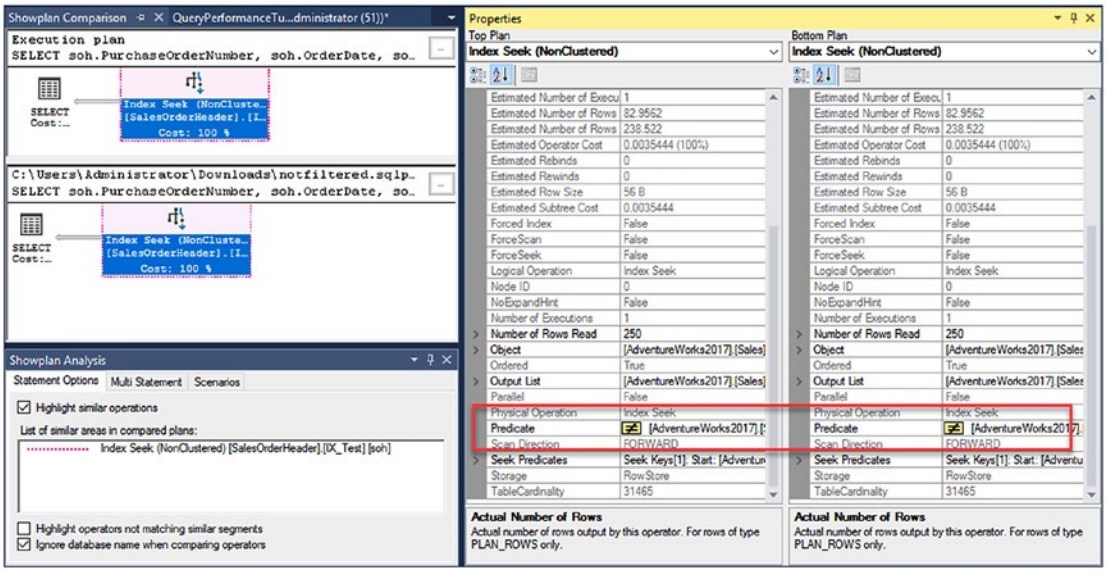

Figure 9-10. *Comparison of the two plans*

There are almost no direct indicators of differences in the execution plans. In the properties to the right, I've highlighted the one big difference. While the queries were identical, because of the index that filters out all null values, the predicate gets changed to remove `IS NOT NULL` because it's no longer needed. This is part of a process within the optimizer called *simplification*.

Although in terms of sheer numbers reducing the reads from 5 to 4 isn't much, it is a 20 percent reduction in the I/O cost of the query, and if this query were running hundreds or even thousands of times in a minute, like some queries do, that 20 percent reduction would be a great payoff indeed. Another visible evidence of the payoff is in the execution time, which dropped again from 40ms to 38ms.

Filtered indexes improve performance in many ways.

- Improving the efficiency of queries by reducing the size of the index
- Reducing storage costs by making smaller indexes
- Cutting down on the costs of index maintenance because of the reduced size

But, everything does come with a cost. You may see issues with parameterized queries not matching the filtered index, therefore preventing its use. Statistics are not updated based on the filtering criteria but rather on the entire table just like a regular index. Like with any of the suggestions in this book, test in your environment to ensure that filtered indexes are helpful.

One of the first places suggested for their use is just like the previous example, eliminating `NULL` values from the index. You can also isolate frequently accessed sets of data with a special index so that the queries against that data perform much faster. You can use the `WHERE` clause to filter data in a fashion similar to creating an indexed view (covered in more detail in the "Indexed Views" section) without the data maintenance headaches associated with indexed views by creating a filtered index that is a covering index, just like the earlier example.

Filtered indexes require a specific set of ANSI settings when they are accessed or created.

- ON: `ANSI_NULLS`, `ANSI_PADDING`, `ANSI_WARNINGS`, `ARITHABORT`, `CONCAT_NULL_YIELDS_NULL`, `QUOTED_IDENTIFIER`
- OFF: `NUMERIC_ROUNDABORT`

When completed, drop the testing index.

```
DROP INDEX Sales.SalesOrderHeader.IX_Test;
```

Indexed Views

A database view in SQL Server does not store any data. A view is simply a `SELECT` statement that is stored. You create a view using the `CREATE VIEW` statement. You can write queries against a view exactly as if it were a table. When a view gets queried, the optimizer receives the full definition of the `SELECT` statement and uses that as the basis for optimizing the query against the view. Through the optimization process, some or all of the definition of the `SELECT` statement may be used to satisfy the query against the view. What degree of simplification occurs here is determined by a combination of the `SELECT` statement itself and the query against that `SELECT` statement.

A database view can be materialized on the disk by creating a unique clustered index on the view. Such a view is referred to as an *indexed view* or a *materialized view*. After a unique clustered index is created on the view, the view's result set is materialized immediately and persisted in physical storage in the database, saving the overhead of

performing costly operations during query execution. After the view is materialized, multiple nonclustered indexes can be created on the indexed view. Effectively, this turns a view (again, just a query) into a real table with defined storage.

Benefit

You can use an indexed view to increase the performance of a query in the following ways:

- Aggregations can be precomputed and stored in the indexed view to minimize expensive computations during query execution.
- Tables can be prejoined, and the resulting data set can be materialized.
- Combinations of joins or aggregations can be materialized.

Overhead

Indexed views can produce major overhead on an OLTP database. Some of the overheads of indexed views are as follows:

- Any change in the base tables has to be reflected in the indexed view by executing the view's `SELECT` statement.
- Any changes to a base table on which an indexed view is defined may initiate one or more changes in the nonclustered indexes of the indexed view. The clustered index will also have to be changed if the clustering key is updated.
- The indexed view adds to the ongoing maintenance overhead of the database.
- Additional storage is required in the database.

The restrictions on creating an indexed view include the following:

- The first index on the view must be a unique clustered index.
- Nonclustered indexes on an indexed view can be created only after the unique clustered index is created.
- The view definition must be *deterministic*—that is, it is able to return only one possible result for a given query. (A list of deterministic and nondeterministic functions is provided in SQL Server Books Online.)

- The indexed view must reference only base tables in the same database, not other views.
- The indexed view may contain float columns. However, such columns cannot be included in the clustered index key.
- The indexed view must be schema bound to the tables referred to in the view to prevent modifications of the table schema (frequently a major problem).
- There are several restrictions on the syntax of the view definition. (A list of the syntax limitations on the view definition is provided in SQL Server Books Online.)
- The list of `SET` options that must be fixed are as follows:
  - `ON`: `ARITHABORT`, `CONCAT_NULL_YIELDS_NULL`, `QUOTED_IDENTIFIER`, `ANSI_NULLS`, `ANSI_ PADDING`, and `ANSI_WARNING`
  - `OFF`: `NUMERIC_ROUNDABORT`

Note If the query connection settings don't match these ANSI standard settings, you may see errors on the insert/update/delete of tables that are used within the indexed view.

Usage Scenarios

Reporting systems benefit the most from indexed views. OLTP systems with frequent writes may not be able to take advantage of the indexed views because of the increased maintenance cost associated with updating both the view and the underlying base tables within a single transaction. The net performance improvement provided by an indexed view is the difference between the total query execution savings offered by the view and the cost of storing and maintaining the view.

If you are using the Enterprise edition of SQL Server, an indexed view need not be referenced explicitly in the query for the query optimizer to use it during query execution. This allows existing applications to benefit from the newly created indexed views without changing those applications. Otherwise, you would need to directly reference it within your T-SQL code on editions of SQL Server other than Enterprise. The

query optimizer considers indexed views only for queries with nontrivial cost. You may also find that the new columnstore index will work better for you than indexed views, especially when you're running aggregation or analysis queries against the data. I'll cover the columnstore index later in this chapter.

Let's see how indexed views work with the following example. Consider the following three queries:

```
SELECT  p.[Name] AS ProductName,
        SUM(pod.OrderQty) AS OrderOty,
        SUM(pod.ReceivedQty) AS ReceivedOty,
        SUM(pod.RejectedQty) AS RejectedOty
FROM    Purchasing.PurchaseOrderDetail AS pod
        JOIN Production.Product AS p
        ON p.ProductID = pod.ProductID
GROUP BY p.[Name];

SELECT  p.[Name] AS ProductName,
        SUM(pod.OrderQty) AS OrderOty,
        SUM(pod.ReceivedQty) AS ReceivedOty,
        SUM(pod.RejectedQty) AS RejectedOty
FROM    Purchasing.PurchaseOrderDetail AS pod
        JOIN Production.Product AS p
        ON p.ProductID = pod.ProductID
GROUP BY p.[Name]
HAVING  (SUM(pod.RejectedQty) / SUM(pod.ReceivedQty)) > .08;

SELECT  p.[Name] AS ProductName,
        SUM(pod.OrderQty) AS OrderQty,
        SUM(pod.ReceivedQty) AS ReceivedQty,
        SUM(pod.RejectedQty) AS RejectedQty
FROM    Purchasing.PurchaseOrderDetail AS pod
        JOIN Production.Product AS p
        ON p.ProductID = pod.ProductID
WHERE   p.[Name] LIKE 'Chain%'
GROUP BY p.[Name];
```

All three queries use the aggregation function SUM on columns of the PurchaseOrderDetail table. Therefore, you can create an indexed view to precompute these aggregations and minimize the cost of these complex computations during query execution.

Here are the number of logical reads performed by these queries to access the appropriate tables:

```
Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'Product'. Scan count 1, logical reads 6
Table 'PurchaseOrderDetail'. Scan count 1, logical reads 66
CPU time = 0 ms,  elapsed time = 31 ms.

Table 'Workfile'. Scan count 0, logical reads 0
Table 'Worktable'. Scan count 0, logical reads 0
Table 'Product'. Scan count 1, logical reads 6
Table 'PurchaseOrderDetail'. Scan count 1, logical reads 66
CPU time = 0 ms,  elapsed time = 16 ms.

Table 'PurchaseOrderDetail'. Scan count 5, logical reads 894
Table 'Product'. Scan count 1, logical reads 2
CPU time = 0 ms,  elapsed time = 1 ms.
```

I'll use the following script to create an indexed view to precompute the costly computations and join the tables:

```
CREATE OR ALTER VIEW Purchasing.IndexedView
WITH SCHEMABINDING
AS
SELECT pod.ProductID,
       SUM(pod.OrderQty) AS OrderQty,
       SUM(pod.ReceivedQty) AS ReceivedQty,
       SUM(pod.RejectedQty) AS RejectedQty,
       COUNT_BIG(*) AS Count
FROM Purchasing.PurchaseOrderDetail AS pod
GROUP BY pod.ProductID;
GO
```

```
CREATE UNIQUE CLUSTERED INDEX iv
ON Purchasing.IndexedView (ProductID);
GO
```

Certain constructs such as AVG are disallowed. (For the complete list of disallowed constructs, refer to SQL Server Books Online.) If aggregates are included in the view, like in this one, you must include COUNT_BIG by default.

The indexed view materializes the output of the aggregate functions on the disk. This eliminates the need for computing the aggregate functions during the execution of a query interested in the aggregate outputs. For example, the third query requests the sum of ReceivedQty and RejectedQty for certain products from the PurchaseOrderDetail table. Because these values are materialized in the indexed view for every product in the PurchaseOrderDetail table, you can fetch these preaggregated values using the following SELECT statement on the indexed view:

```
SELECT  iv.ProductID,
        iv.ReceivedQty,
        iv.RejectedQty
FROM    Purchasing.IndexedView AS iv;
```

As shown in the execution plan in Figure 9-11, the SELECT statement retrieves the values directly from the indexed view without accessing the base table (PurchaseOrderDetail).

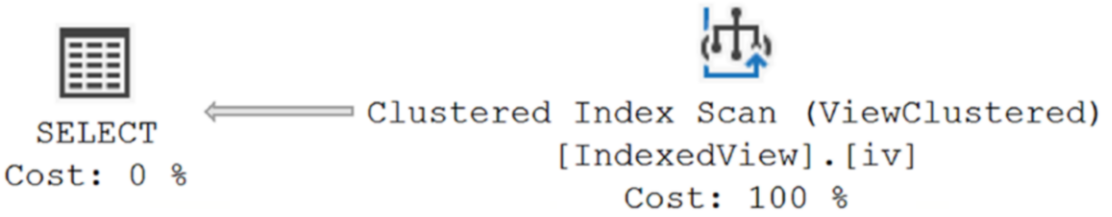

Figure 9-11. *Execution plan with an indexed view*

The indexed view benefits not only the queries based on the view directly but also other queries that may be interested in the materialized data. For example, with the indexed view in place, the three queries on `PurchaseOrderDetail` benefit without being rewritten (see the execution plan in Figure 9-12 for the execution plan from the first query), and the number of logical reads decreases, as shown here:

```
Table 'Product'. Scan count 1, logical reads 13
Table 'IndexedView'. Scan count 1, logical reads 4
CPU time = 0 ms,  elapsed time = 53 ms.

Table 'Product'. Scan count 1, logical reads 13
Table 'IndexedView'. Scan count 1, logical reads 4
CPU time = 0 ms,  elapsed time = 1 ms.

Table 'IndexedView'. Scan count 0, logical reads 10
Table 'Product'. Scan count 1, logical reads 2
CPU time = 0 ms,  elapsed time = 0 ms. (214 us)
```

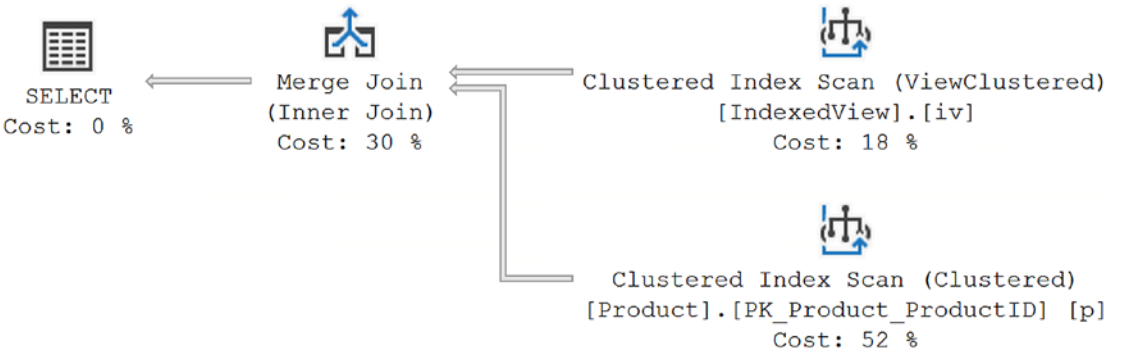

Figure 9-12. *Execution plan with the indexed view automatically used*

Even though the queries are not modified to refer to the new indexed view, the optimizer still uses the indexed view to improve performance. Thus, even existing queries in the database application can benefit from new indexed views without any modifications to the queries. If you do need different aggregations than what the indexed view offers, you'll be out of luck. Here again the columnstore index shines.

Make sure to clean up.

```
DROP VIEW Purchasing.IndexedView;
```

Index Compression

Data and index compression were introduced in SQL Server 2008 (available in the Enterprise and Developer editions, currently in all editions). *Compressing* an index means getting more key information onto a single page. This can lead to significant performance improvements because fewer pages and fewer index levels are needed to store the index. There will be overhead in the CPU as the key values in the index are compressed and decompressed, so this may not be a solution for all indexes. Memory benefits also because the compressed pages are stored in memory in a compressed state.

By default, an index will be not be compressed. You have to explicitly call for the index to be compressed when you create the index. There are two types of compression: row- and page-level compression. *Row-level compression* identifies columns that can be compressed (for details, look in Books Online) and compresses the storage of that column and does this for every row. *Page-level compression* is actually using row-level compression and then adding additional compression on top to reduce storage size for the nonrow elements stored on a page. Nonleaf pages in an index receive no compression under the page type. To see index compression in action, consider the following index:

```
CREATE NONCLUSTERED INDEX IX_Test
ON Person.Address
(
    City ASC,
    PostalCode ASC
);
```

This index was created earlier in the chapter. If you were to re-create it as defined here, this creates a row type of compression on an index with the same two columns as the first test index `IX_Test`.

```
CREATE NONCLUSTERED INDEX IX_Comp_Test
ON Person.Address
(
    City,
    PostalCode
)
WITH (DATA_COMPRESSION = ROW);
```

Create one more index.

```
CREATE NONCLUSTERED INDEX IX_Comp_Page_Test
ON Person.Address
(
    City,
    PostalCode
)
WITH (DATA_COMPRESSION = PAGE);
```

To examine the indexes being stored, modify the original query against `sys.dm_db_index_physical_stats` to add another column, `compressed_page_count`.

```
SELECT i.name,
       i.type_desc,
       s.page_count,
       s.record_count,
       s.index_level,
       s.compressed_page_count
FROM sys.indexes AS i
    JOIN sys.dm_db_index_physical_stats(DB_ID(N'AdventureWorks2017'),
                                        OBJECT_ID(N'Person.Address'),
                                        NULL,
                                        NULL,
                                        'DETAILED') AS s
        ON i.index_id = s.index_id
WHERE i.object_id = OBJECT_ID(N'Person.Address');
```

Running the query, you get the results in Figure 9-13.

	name	type_desc	page_count	record_count	index_level	compressed_page_count
12	IX_Comp_Test	NONCLUSTERED	63	19614	0	0
13	IX_Comp_Test	NONCLUSTERED	1	63	1	0
14	IX_Comp_Page_Test	NONCLUSTERED	25	19614	0	25
15	IX_Comp_Page_Test	NONCLUSTERED	1	25	1	0
16	IX_Test	NONCLUSTERED	106	19614	0	0
17	IX_Test	NONCLUSTERED	1	106	1	0

Figure 9-13. *sys.dm_db_index_physical_stats output about compressed indexes*