CHAPTER 5

XML Indexes

As discussed in Chapters 3 and 4, SQL Server allows you to store data in tables, in a native XML format, using the XML data type. Like other large object types, it can store up to a maximum of 2GB per tuple. Although standard operators such as = and `LIKE` can be used against XML columns, you also have the option of using `XQuery` expressions (discussed in this chapter). They can be rather inefficient, however, unless you create XML indexes.

XML indexes will outperform full-text indexes for most queries against XML columns. SQL Server offers support for primary XML indexes and three types of secondary XML indexes: `PATH`, `VALUE`, and `PROPERTY`. Each of these indexes will be discussed in the following sections. First, however, I will briefly discuss clustered indexes, as a clustered index must exist on the table before you can create an XML index.

Preparing the Environment

Because the WideWorldImporters database has no tables that contain native XML columns, we will create an OrderSummary table, for demonstrations within this chapter. The table will contain three columns: an IDENTITY column (named ID), a CustomerID column, and an XML column (called OrderSummary), which will contain a summary of all orders that a customer has placed. The table can be created and populated using the script in Listing 5-1.

P. A. Carter, *SQL Server Advanced Data Types*,
https://doi.org/10.1007/978-1-4842-3901-8_5

Listing 5-1. Creating an OrderSummary Table

```
USE WideWorldImporters
GO

CREATE TABLE Sales.OrderSummary
(
        ID          INT          NOT NULL        IDENTITY,
        CustomerID  INT          NOT NULL,
        OrderSummary      XML
) ;

INSERT INTO Sales.OrderSummary (CustomerID, OrderSummary)
SELECT
        CustomerID,
        (
                SELECT
                         CustomerName 'OrderHeader/CustomerName'
                       , OrderDate 'OrderHeader/OrderDate'
                       , OrderID 'OrderHeader/OrderID'
                       , (
                               SELECT
                                         LineItems2.StockItemID
                                         '@ProductID'
                                       , StockItems.StockI
                                       temName '@ProductName'
                                       , LineItems2.UnitPrice
                                       '@Price'
                                       , Quantity '@Qty'
                              FROM Sales.OrderLines LineItems2
                               INNER JOIN Warehouse.StockItems
                               StockItems
```

```
                                        ON LineItems2.StockItemID
                                        = StockItems.StockItemID
                                WHERE LineItems2.OrderID =
                                Base.OrderID
                                FOR XML PATH('Product'), TYPE
                          ) 'OrderDetails'
                FROM
                (
                        SELECT DISTINCT
                                  Customers.CustomerName
                                , SalesOrder.OrderDate
                                , SalesOrder.OrderID
                        FROM Sales.Orders SalesOrder
                        INNER JOIN Sales.OrderLines LineItem
                                ON SalesOrder.OrderID =
                                LineItem.OrderID
                        INNER JOIN Sales.Customers Customers
                                ON Customers.CustomerID =
                                SalesOrder.CustomerID
                        WHERE customers.CustomerID = OuterCust.
                        CustomerID
                ) Base
                FOR XML PATH('Order'), ROOT ('SalesOrders'), TYPE
        ) AS OrderSummary
FROM Sales.Customers OuterCust ;
```

Clustered Indexes

A clustered index causes the data pages of a table to be logically stored in the order of the clustered index key. The clustered index key can be a single column or a set of columns. This is often the table's primary key, but

this is not enforced, and there are some circumstances in which you would want to use a different column. This will be discussed in more detail later in this chapter.

Tables Without a Clustered Index

When a table exists without a clustered index, it is known as a heap. A heap consists of an IAM (index allocation map) page (or pages) and a series of data pages that are not linked together or stored in order. The only way SQL Server can determine the pages of the table is by reading the IAM page(or pages). When a table is stored as a heap, every time the table is accessed, SQL Server must read every single page in the table, even if you only want to return one row. The diagram in Figure 5-1 illustrates how a heap is structured.

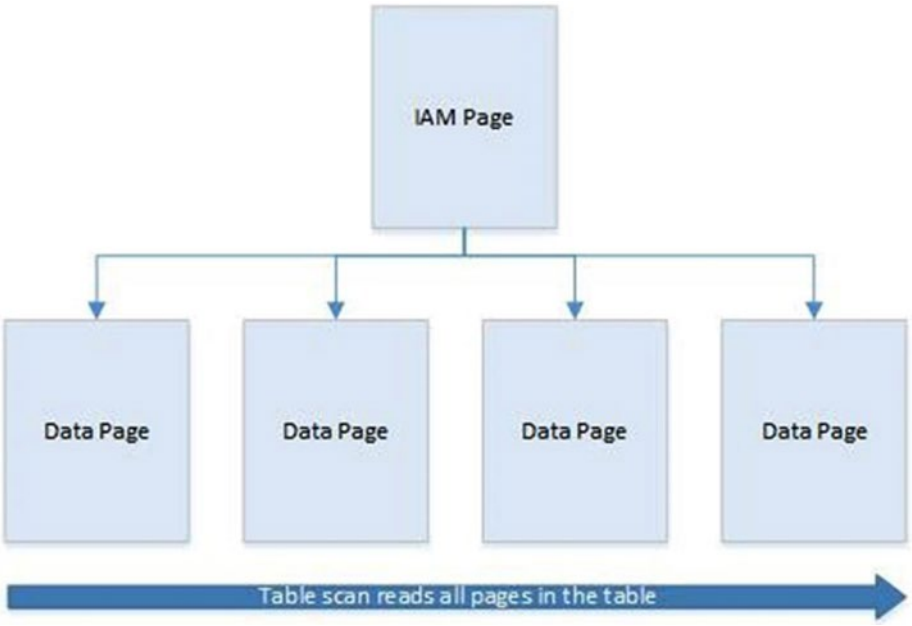

Figure 5-1. *Heap structure*

When data is stored on a heap, SQL Server must maintain a unique identifier for each row. It does this by creating a RID (row identifier). Even if a table has nonclustered indexes, it is still stored as a heap, unless there is a clustered index. When nonclustered indexes are created on a heap, the RID is used as a pointer, so that nonclustered indexes can link back to the correct row in the base table. Nonclustered indexes store the RID with a format of FileID: Page ID: Slot Number.

Tables with a Clustered Index

When you create a clustered index on a table, a B-Tree (balanced tree) structure is created. This allows for more efficient search operations to be performed, by creating a tiered set of pointers to the data, as illustrated in Figure 5-2. The page at the top level of this hierarchy is called the root node. The bottom level of the structure is called the leaf level, and with a clustered index, the leaf level consists of the actual data pages of the table. There can be one or more intermediate levels of B-Tree structures, depending on the size of the table.

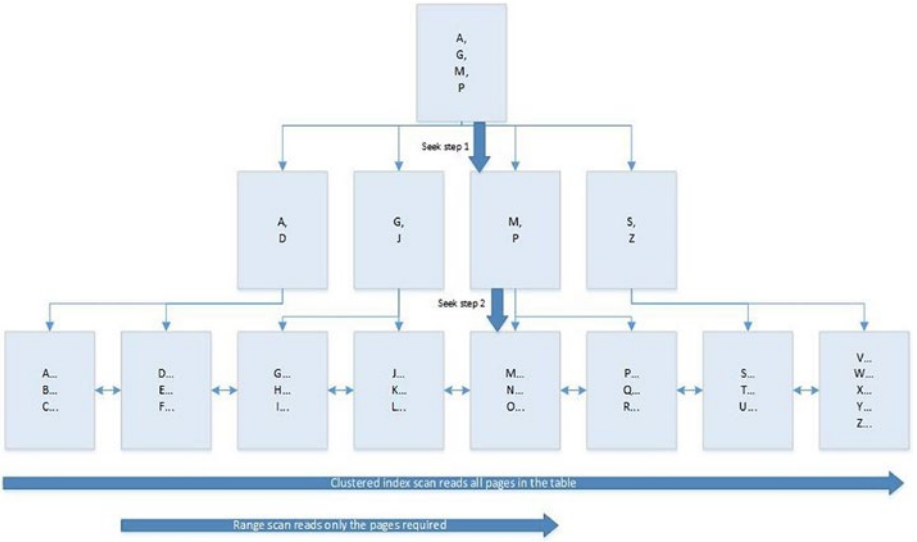

Figure 5-2. *Clustered index structure*

The diagram in Figure 5-2 shows that while the leaf level is the data itself, the levels above contain pointers to the pages below them in the tree. This allows for SQL Server to perform a seek operation. This is a very efficient method of returning a small number of rows. It works by navigating its way down the B-Tree, using the pointers to find the row(s) it requires. We can see that, if required, SQL Server can still scan all pages of the table, in order to retrieve the required rows. This is known as a Clustered Index Scan. Alternatively, SQL Server may decide to combine these two methods, to perform a range scan. Here, SQL Server will seek the first value of the required range and then scan the leaf level, until it encounters the first value that is not required. SQL Server can do this because the table is ordered by the index key, meaning that it can guarantee that there will be no other matching values later in the table.

Clustering the Primary Key

The primary key of a table is often the natural choice for the clustered index. In fact, by default, unless you specify otherwise, or unless a clustered index already exists on the table, creating a primary key will automatically generate a clustered index on that key. There are circumstances in which the primary key is not the correct choice for the clustered index. An example of this that I have witnessed is a third-party application that required the primary key of the table to be a GUID. A GUID (globally unique identifier) is used to guarantee uniqueness across the entire network.

This introduces two major problems if the clustered index were to be built on the primary key. The first is size. A GUID is 16 bytes long. When a table has nonclustered indexes, the clustered index key is stored in every nonclustered index. For unique nonclustered indexes, it is stored for every row at the leaf level, and for non-unique nonclustered indexes, it is also stored at every row in the root and intermediate levels of the index as well.

When you multiply 16 bytes by millions of rows, this will drastically increase the size of the indexes, making them less efficient.

The second issue is that when a GUID is generated, it is a random value. Because the data in your table is stored in the order of the clustered index key, for good performance, you need the values of this key to be generated in sequential order. Generating random values for your clustered index key will result in the index becoming more and more fragmented every time you insert a new row. Fragmentation will be discussed later in this chapter.

There is a workaround for the second issue, however. SQL Server has a function called `NEWSEQUENTIALID()` that will always generate a GUID value higher than previous values generated on the server. Therefore, if you use this function in the `Default` constraint of your primary key, you can enforce sequential inserts.

Caution After the server has been restarted, `NEWSEQUENTIALID()` can start with a lower value. This may lead to fragmentation.

If the primary key must be a GUID, or another wide column, such as National Insurance Number, or a set of columns forming a natural key, such as Customer ID, Order Date, and Product ID, it is highly recommended that you create an additional column in your table. This column could be an `INT` or `BIGINT`, depending on the number of rows you expect the table to have, and could use either the `IDENTITY` property or a `SEQUENCE` to create a narrow, sequential key that can be used for your clustered index. I recommend ensuring a narrow key, as it will be included in all nonclustered indexes on the table. It will also use less memory when joining tables.

Caution If you intend to use XML indexes, the clustered index must be created on the primary key.

Performance Considerations for Clustered Indexes

Because an IAM page lists the extents of a heap table in the order in which they are stored in the data file, as opposed to the order of the index key, a table scan of a heap may prove to be slightly faster than a clustered index scan, unless the clustered index has 0% fragmentation, which is rare.

Inserts into a clustered index may be faster than inserts into a heap, when the clustered index key is ever-increasing. This is especially true when there are multiple inserts happening in parallel, because a heap will experience more contention on system pages (GAM/SGAM/PFS) when the database engine is looking for spaces to place the new data. If the clustered index key is not ever-increasing, however, then inserts will lead to page splits and fragmentation. The knock-on effect is that inserts would be slower than they would be into a heap. A large insert into a heap may also be faster, if you take out a table lock and take advantage of minimally logged inserts. This is because of reduced IO to the transaction log.

Updates that cause a row to relocate, due to a change in size, will be faster when performed against a clustered index, as opposed to a heap. This is, for the same reason as mentioned above for insert operations, where there will be more contention against the system pages. When updated, rows may change in size, for reasons such as updating a `VARCHAR` column with a longer string. If the update to the row can be made in place (without relocating the row), there is likely to be little difference in performance. Deletes may also be slightly faster into a clustered index than into a heap, but the difference will be less noticeable than for update operations.

Creating a Clustered Index

With SSMS (SQL Server Management Studio), we can create a clustered index on the `ID` column of the OrderSummary table, by expanding Databases ➤ WideWorldImporters ➤ Tables ➤ Sales.OrderSummary in Object Explorer, right-clicking the Indexes node, and selecting New index ➤ Clustered index. This will cause the New Index dialog box to be invoked, as shown in Figure 5-3.

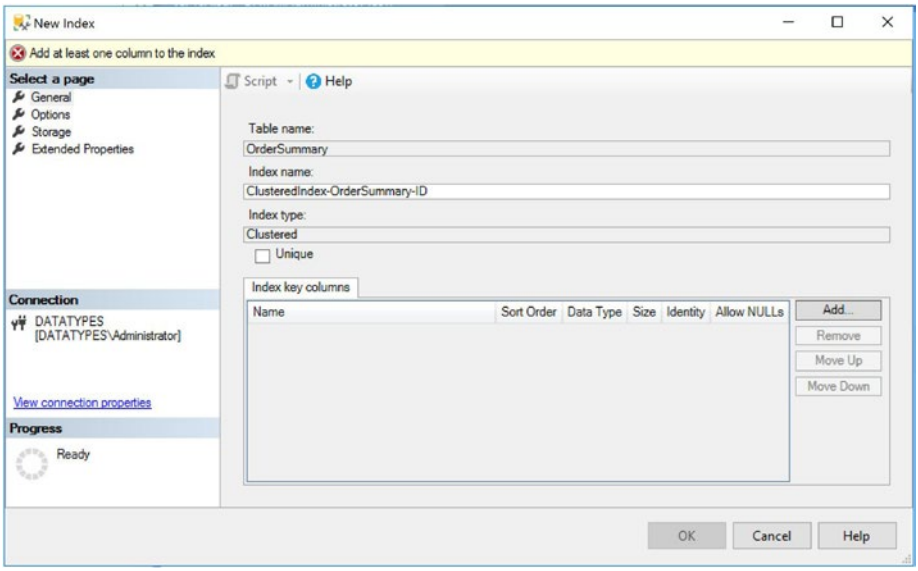

Figure 5-3. *New Index dialog box*

Caution If you plan to follow later demonstrations in this chapter, do not execute the steps illustrated in Figures 5-3 and 5-4. Also, avoid executing the script in Listing 5-2. If you do create the index, you will have to drop it before running further examples.

On the General page of the dialog box, give the index a descriptive name, then use the Add button, to select the column(s) that the index will be built on, as shown in Figure 5-4.

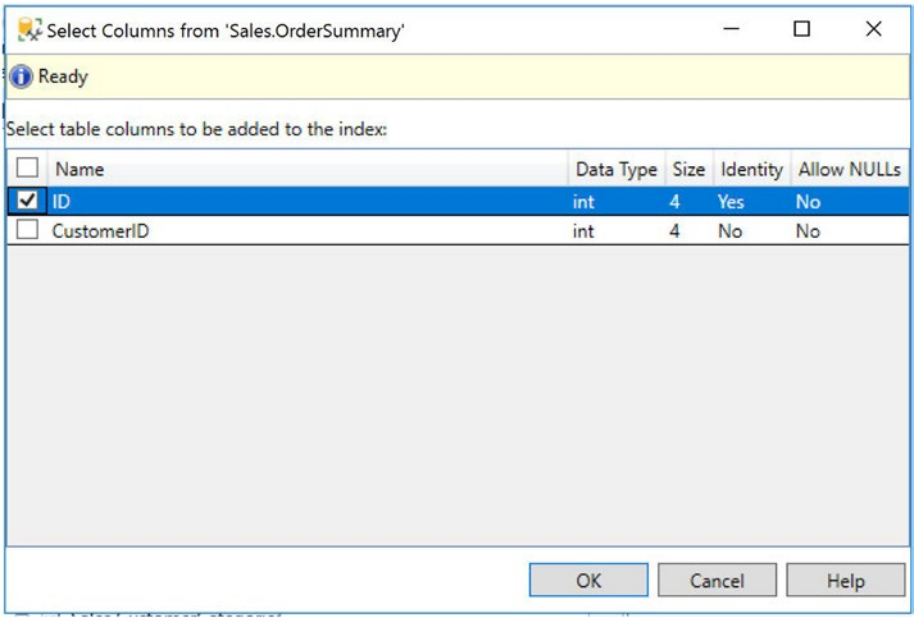

Figure 5-4. *Add columns dialog box*

Alternatively, this clustered index could be created using the script in Listing 5-2.

Listing 5-2. Creating a Clustered Index

```
USE WideWorldImporters
GO

CREATE CLUSTERED INDEX [ClusteredIndex-OrderSummary-ID] ON
Sales.OrderSummary (ID) ;
GO
```

Note Advanced options for creating clustered indexes are beyond the scope of this book, but further information can be found in *Pro SQL Server Administration* (Apress, 2015), available at www.apress.com/gb/book/9781484207116.

Because our XML indexes require the clustered index to be built on a primary key, instead of executing the preceding script, we should instead run the script in Listing 5-3. This script will create a primary key on the ID column and then a clustered index on the primary key.

Listing 5-3. Creating a Primary Key and Clustered Index

```
USE WideWorldImporters
GO

ALTER TABLE Sales.OrderSummary ADD CONSTRAINT
        PK_OrderSummary PRIMARY KEY CLUSTERED (ID) ;
```

Primary XML Indexes

A primary XML index is actually a multicolumn clustered index on an internal system table called the Node table. This table stores a shredded representation on the XML objects within an XML column, along with the clustered index key of the base table. This means that a table must have a clustered index before a primary XML index can be created. Additionally, the clustered index must be created on the primary key and must consist of 32 columns or less.

The system table stores enough information that the scalar or XML subtrees required by a query can be reconstructed from the index itself. This information includes the node ID and name, the tag name and URI, a tokenized version of the node's data type, the first position of the node

value in the document, pointers to the long node value and binary value, the nullability of the node, and the value of the base table's clustered index key for the corresponding row.

Primary XML indexes can provide a performance improvement when a query must shred scalar values from an XML document (or documents) or return a subset of nodes from an XML document (or documents).

Creating Primary XML Indexes

To create a primary XML index using SSMS, drill through Databases ➤ WideWorldImporters ➤ Tables ➤ Sales.OrderSummary in Object Explore. Then select New Index ➤ Primary XML Index from the context menu of indexes. This will cause the General page of the New Index dialog box to be displayed. This is shown in Figure 5-5.

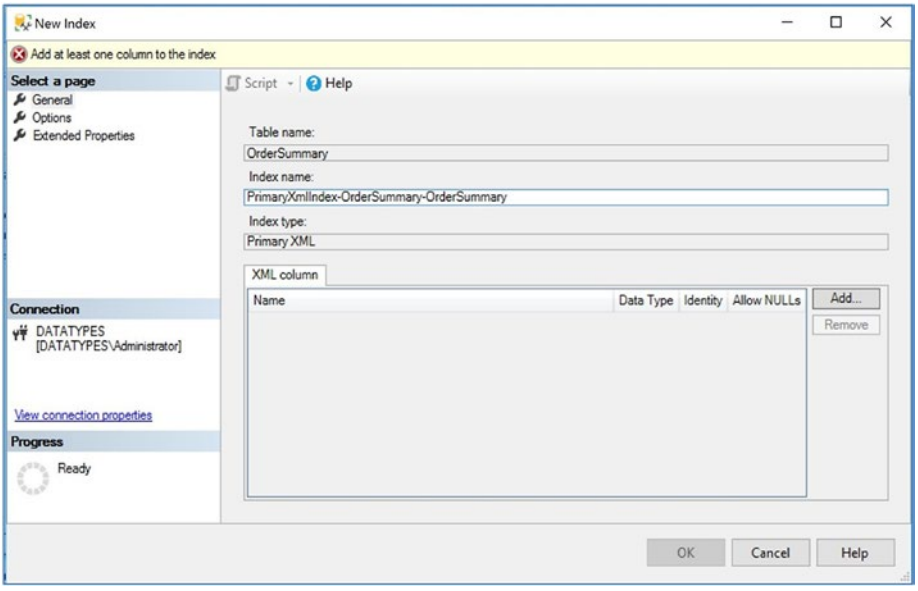

Figure 5-5. *New Index dialog box (Primary XML)*

Here, we will give the index a descriptive name and then use the Add button, to add the required XML column, as shown in Figure 5-6.

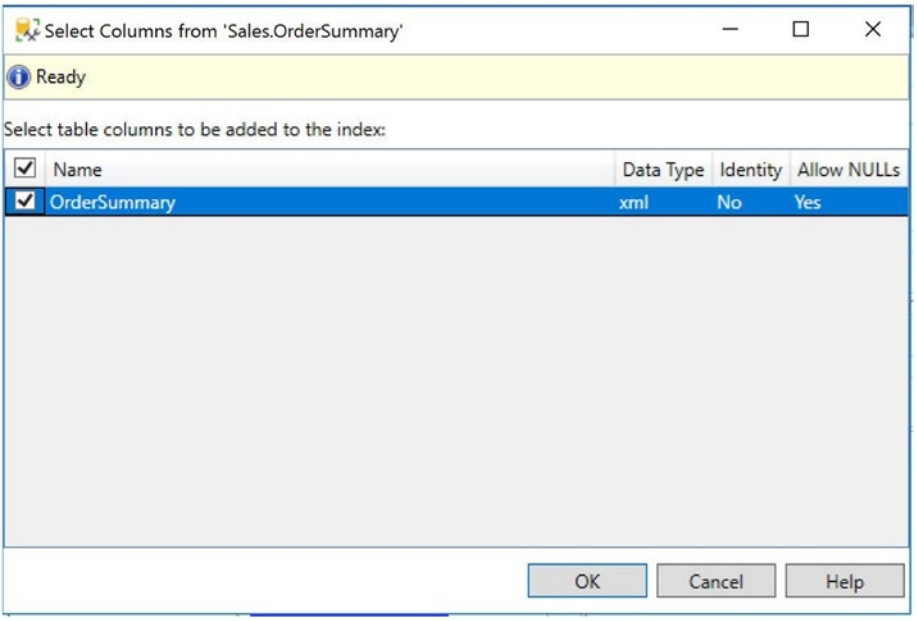

Figure 5-6. *Add column dialog box (Primary XML)*

From the Options tab of the New Index dialog box (Figure 5-7), we can set the options detailed in Table 5-1.

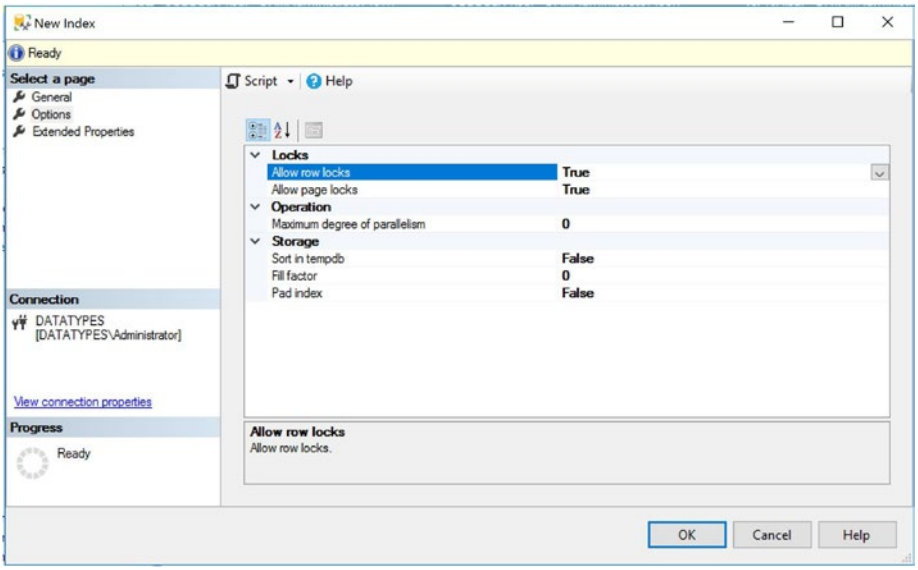

Figure 5-7. *New Index dialog box—Options page (Primary XML)*

Table 5-1. *Primary XML Index Options*

Option	Description
Allow Row Locks	Specifies if row locks can be acquired when accessing the index
Allow Page Locks	Specifies if page locks can be acquired when accessing the index
MaxDoP	Has no effect for building primary XML indexes, as this operation is always single threaded
Sort in TempDB	If specified, sort in TempDB will cause the intermediate result set to be stored in TempDB, as opposed to the user database. This could mean that the index is built faster.

(*continued*)

Table 5-1. (*continued*)

Option	Description
Fill Factor	Specifies a percentage of free space that will be left on each index page at the lowest level of the index. The default is 0 (100% full), meaning that only enough space for a single row will be left. Specifying a percentage lower than 100, for example, specifying 70, will leave 30% free space and can reduce page splits, if there are likely to be frequent row inserts.
Pad Index	Applies a fill factor (see preceding) to the intermediate levels of a B-Tree

Alternatively, to create the index via T-SQL, you could use the script in Listing 5-4.

Listing 5-4. Creating a Primary XML Index

```
USE WideWorldImporters
GO

CREATE PRIMARY XML INDEX [PrimaryXmlIndex-OrderSummary-
OrderSummary]
        ON Sales.OrderSummary ([OrderSummary]) ;
GO
```

Secondary XML Indexes

Secondary XML indexes can only be created on XML columns that already have a primary XML index. Behind the scenes, secondary XML indexes are actually nonclustered indexes on the internal Node table. Secondary XML indexes can improve query performance for queries that use specific types of XQuery processing.

A PATH secondary XML index is built on the Node ID and VALUE columns of the Node table. This type of index offers performance improvements to queries that use path expressions, such as the exists() XQuery method. A VALUE secondary XML index is the reverse of this and is built on the VALUE and Node ID columns. This type of index will offer performance improvements to queries that search for values, without knowing the name of the XML element or attribute that contains the value being searched for.

Finally, a PROPERTY secondary XML index is built on the clustered index key of the base table, the Node ID, and the VALUE columns of the Node table. This type of index performs very well if the query is trying to retrieve nodes from multiple tuples of the column.

Creating Secondary XML Indexes

To create a secondary XML index in SSMS, drill through Databases ➤ WideWorldImporters ➤ Tables ➤ OrderSummary in Object Explorer. Next, select New Index ➤ Secondary XML Index from the context menu of the Indexes node. This will cause the New Index dialog box to be displayed, as illustrated in Figure 5-8.

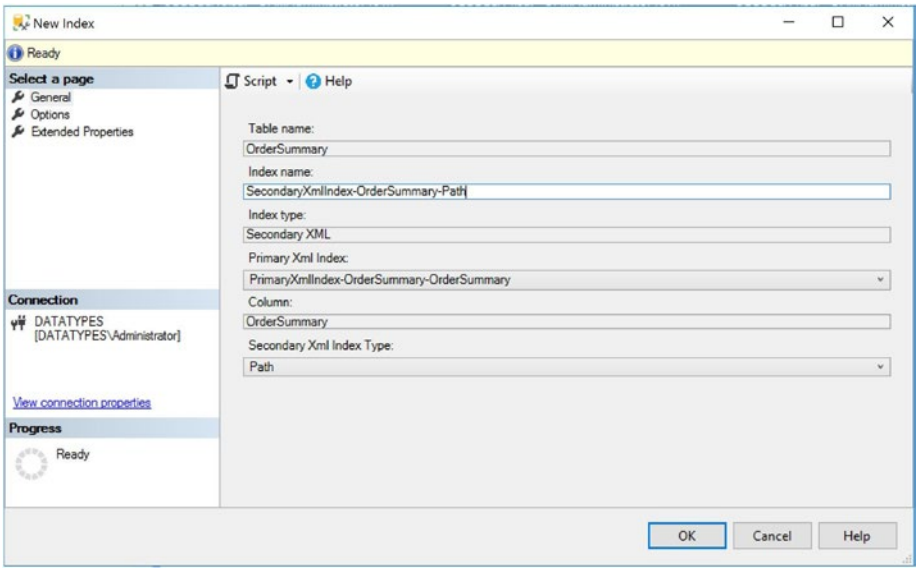

Figure 5-8. *New Index dialog box (Secondary XML)*

On the General tab of the New Index dialog box, we have first given the index a descriptive name. Next, we select the appropriate primary XML index from the Primary XML Index drop-down list. Finally, we select the type of secondary XML index that we wish to create, from the Secondary XML Index Type drop-down box. In this case, we have chosen to create a PATH index.

Figure 5-9 illustrates the Options tab of the New Index dialog box. For details of each option, please refer to Table 5-1.

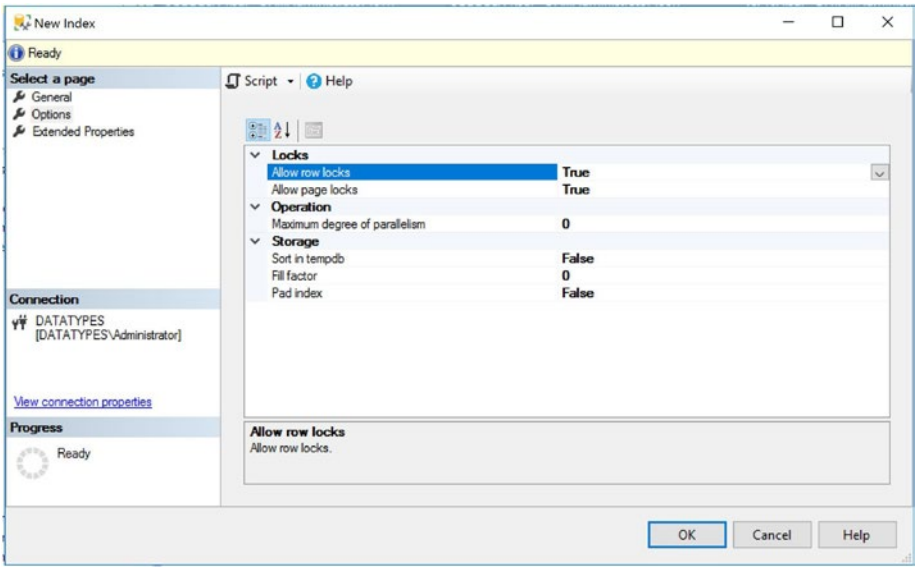

Figure 5-9. *New Index dialog box—Options tab (Secondary XML Index)*

Alternatively, to create this index with T-SQL, you could use the script in Listing 5-5.

Listing 5-5. Creating a Secondary XML Index

```
USE WideWorldImporters
GO

CREATE XML INDEX [SecondaryXmlIndex-OrderSummary-Path]
        ON Sales.OrderSummary (OrderSummary)
USING XML INDEX [PrimaryXmlIndex-OrderSummary-OrderSummary] FOR
PATH ;
GO
```

Performance Considerations for XML Indexes

In order to discuss the performance of XML indexes, let's write a query that is well-suited to the PATH secondary XML index that we have created on the OrderSummary table. The query in Listing 5-6 runs a query against the OrderSummary table and returns all rows indicating customers who have ordered the Chocolate echidnas 250g product, which has a StockItemID of 223. The first part of the script removes unchanged pages from the buffer cache and drops the plan cache, making it a fair test. The middle part of the script turns on time statistics, so we can accurately tell how long the query took to run.

Tip Performance will vary, based on the specification of your server and how many resources are being consumed by concurrent processes. You should always check performance within your own environment.

Listing 5-6. Return Rows Where Customers Have Ordered StockItemID 223

```
--Clear buffer cache and plan cache
DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
GO

--Turn on IO statistics to appear with results
SET STATISTICS TIME ON
GO

--Run query
SELECT *
```

```
FROM Sales.OrderSummary
WHERE OrderSummary.exist('/SalesOrders/Order/OrderDetails/
Product/.[@ProductID = 223]') = 1 ;
```

The statistics shown in Figure 5-10 show that the query took 1.95 seconds to complete.

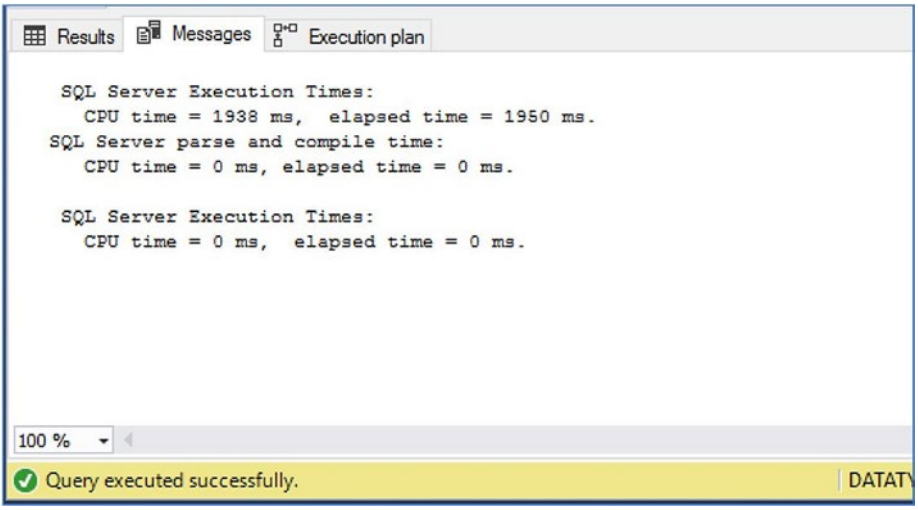

Figure 5-10. *Query results with PATH index*

Now let's use the script in Listing 5-7 to drop the PATH index and run the query again. This time, only the primary XML index is available for use.

Listing 5-7. Run Query Without PATH Index

```
DROP INDEX [SecondaryXmlIndex-OrderSummary-Path] ON Sales.
OrderSummary ;
GO

DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
GO
```

```
SET STATISTICS TIME ON
GO

SELECT *
FROM Sales.OrderSummary
WHERE OrderSummary.exist('/SalesOrders/Order/OrderDetails/
Product/.[@ProductID = 223]') = 1 ;
```

This time, as we can see from the statistics in Figure 5-11, the query took more than 2.7 seconds to complete.

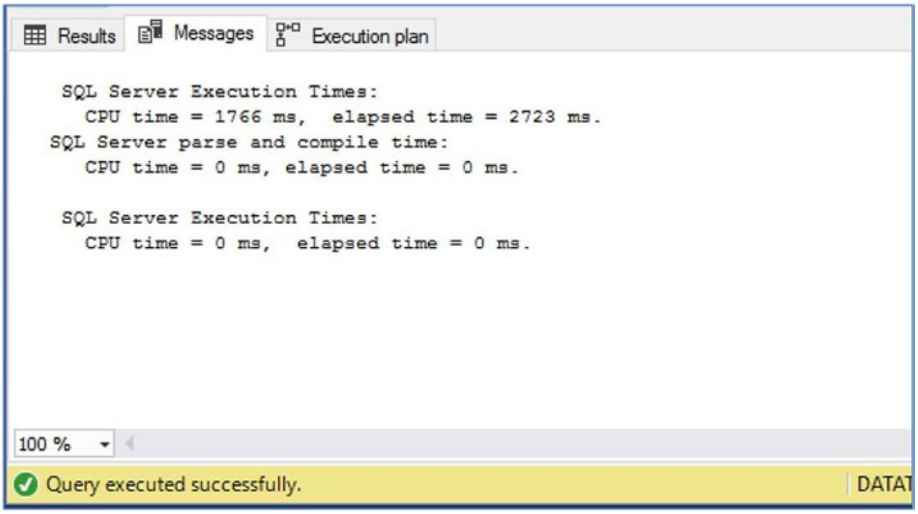

Figure 5-11. *Query results without PATH index*

Finally, let's use the script in Listing 5-8 to drop the primary XML Index, and run the query again, with no XML index support.

Listing 5-8. Drop Primary XML Index and Rerun Query

```
DROP INDEX [PrimaryXmlIndex-OrderSummary-OrderSummary] ON
Sales.OrderSummary ;
GO

DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
GO

SET STATISTICS TIME ON
GO

SELECT *
FROM Sales.OrderSummary
WHERE OrderSummary.exist('/SalesOrders/Order/OrderDetails/
Product/.[@ProductID = 223]') = 1 ;
```

You will notice from the statistics in Figure 5-12 that the query execution time has now risen to more than 4 seconds. While our table only has fewer than 700 rows and results you see will vary, depending on the performance of your machine, this example shows why creating XML indexes is so important.

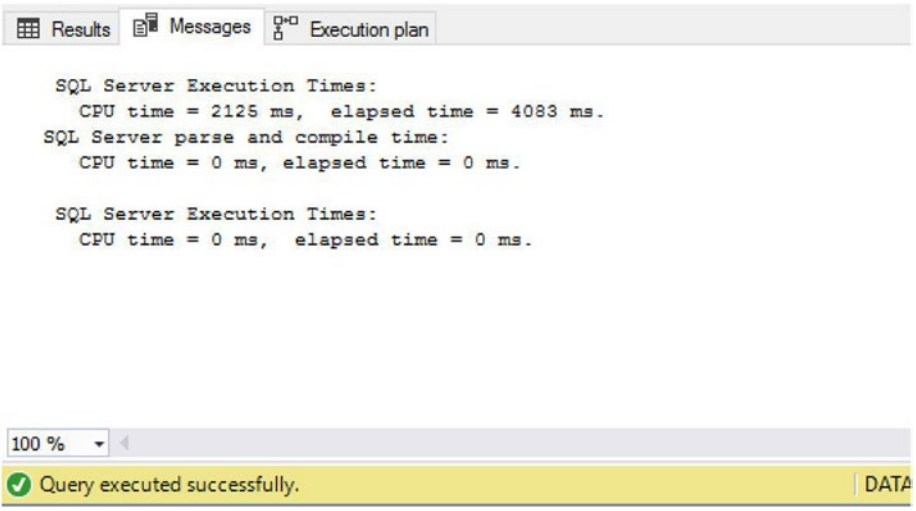

Figure 5-12. *Results of query with no XML indexes*

Summary

Specialized XML indexes can be created on XML columns, to improve the performance of queries that rely on interrogating XML data. There are four types of XML Index: Primary, Secondary `PATH`, Secondary `VALUE`, and Secondary `PROPERTY`.

A primary XML index cannot be created on an XML column, unless the table has a clustered primary key (a clustered index built on a primary key column). A secondary XML cannot be created unless a primary XML index already exists on the XML column. XML indexes can be created before a table is populated with data, however.

Queries that interrogate XML columns can be quite inefficient and perform poorly, unless the correct XML indexes are created to support them. XML indexes will always be more efficient on XML columns than full-text indexes will be. As demonstrated in this chapter, XML query performance is significantly impaired if XML indexes are not created appropriately.