Data Administration in Information Systems

Lab 3: Indexing

Note: This lab assumes that you are using the provided virtual machine, or have otherwise installed SQL Server, SQL Server Management Studio, and the AdventureWorks database.

1. Open **SQL Server Management Studio** and connect to SQL Server.

2. In **Object Explorer** expand **Databases > AdventureWorks > Tables**.

3. Right-click the table **Person.Person** and select the **Design** option.

4. At the top of the window, in the menu **Table Designer**, select **Indexes/Key**.

5. In the **Indexes/Keys** window:
   - On the left pane, select the index associated with the primary key.
   - On the right pane, check the **Description** for this index and other properties.

   Is this index clustered or non-clustered?

6. Select the index associated with LastName, FirstName, MiddleName, and check its properties.
   Is this index clustered or non-clustered?

7. Close the **Indexes/Keys** window.

8. In **Object Explorer**, right-click the **AdventureWorks** database and select **New Query**.

9. Execute the following command:

```
EXEC sp_helpindex 'Person.Person';
```

10. In the **Results** tab, check that the information agrees with what you have seen before.

11. Write the following query in the same query window:

```
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'PK_Person_BusinessEntityID',
                     'IsClustered');
```

12. Before executing the query, note the following:
    - The function INDEXPROPERTY provides information about the properties of an index (or statistics) on a given table. The first argument is the object (table) ID, the second argument is the index (or statistics) name, and the third argument is the desired property.
    - In this case, we are retrieving the property **IsClustered** of an index on table **Person**. Because the index is on the primary key, we should expect the index to be clustered.

13. Execute the query above and confirm that the index is clustered (1 = true, 0 = false).

14. Change the property from **IsClustered** to **IndexDepth**:

```
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'PK_Person_BusinessEntityID',
                     'IndexDepth');
```

15. Execute the query, and note the following:
    - The result is the depth of the B+ tree that SQL Server built for this index.
    - SQL Server uses B+ trees for on-disk indexes, and hash indexes for in-memory tables.

16. Change the property from **IndexDepth** to **IsUnique**:

```
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'PK_Person_BusinessEntityID',
                     'IsUnique');
```

17. Execute the query, and note the following:
    - Because the index is on the primary key, it should be unique, i.e. there are no duplicate values in the column (or combination of columns) being indexed.
    - Therefore, the **IsUnique** property for this index should be 1 (true).

18. Execute this command again to check the name of the non-clustered index on LastName, FirstName, MiddleName:

```
EXEC sp_helpindex 'Person.Person';
```

19. Check the **IsClustered** property of this index to confirm that the index is non-clustered:

```
SELECT INDEXPROPERTY(OBJECT_ID('Person.Person'),
                     'IX_Person_LastName_FirstName_MiddleName',
                     'IsClustered');
```

20. Check the **IndexDepth** and **IsUnique** properties as well.

21. Besides the index structure, SQL Server also collects statistics on the indexed values. To show the statistics for the clustered index, execute the following command:

```
DBCC SHOW_STATISTICS ('Person.Person', 'PK_Person_BusinessEntityID');
```

22. In the **Results** tab, SQL Server will show three results sets:
    i) When the statistics were last updated and how many rows the table had by then.
    ii) Density, which is calculated as 1 / distinct values.
    iii) A histogram of values, where:
        - RANGE_HI_KEY is the upper bound of each histogram bin;
        - RANGE_ROWS is the number of values that fall inside the bin (excluding the upper bound);
        - EQ_ROWS is the number of values equal to the upper bound;
        - DISTINCT_RANGE_ROWS is the number of distinct values that fall inside the bin (excluding the upper bound);
        - AVG_RANGE_ROWS is the average number of duplicate values inside the bin (excluding the upper bound).

23. Note that, for a clustered index (i.e. an index with unique values):
    - DISTINCT_RANGE_ROWS = RANGE_ROWS
    - AVG_RANGE_ROWS = 1

24. Execute the following command to show the statistics for the non-clustered index:

```
DBCC SHOW_STATISTICS ('Person.Person', 'IX_Person_LastName_FirstName_MiddleName');
```

25. In the **Results** tab, note the following:
    - Several density values are being presented, one for each prefix of columns.
    - The histogram shows the distribution of values only for the first column in the index.
    - DISTINCT_RANGE_ROWS ≤ RANGE_ROWS because there are repeated values.
    - AVG_RANGE_ROWS ≥ 1 for the same reason.

26. In **Object Explorer**, expand **Databases > AdventureWorks > Tables > Person.Person > Statistics**.

27. Right-click on the statistics for the non-clustered index on LastName, FirstName, MiddleName, and select **Properties**.

28. In the **Statistics Properties** window, change to the **Details** page.

29. Check that the information agrees with what you have seen earlier with the command DBCC SHOW_STATISTICS.

30. Open a new query window.

31. Execute the following command to see the indexes on table **Person.Address**:

```
EXEC sp_helpindex 'Person.Address';
```

32. Locate the clustered index associated with the primary key.

33. In **Object Explorer**, expand **Databases > AdventureWorks > Tables > Person.Address > Indexes**, and check that the indexes agree with the results of the previous command.

34. Also, expand **Person.Address > Statistics**, and check that each index has its own statistics.

35. Finally, expand **Person.Address > Keys**, and note that the table has a primary key and a foreign key to table **Person.StateProvince**.

36. In the **Results** tab of the query window, note that this foreign key column (**StateProvinceID**) is being indexed by its own non-clustered index.

37. In the query window, write the following commands:

```
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
```

38. Before executing these commands, note the following:
    - The first command turns on statistics about the amount of disk activity generated by queries.
    - The second turns on statistics about the time it takes to parse, compile, and execute queries.

39. Highlight the two commands above and **Execute** them.

40. In the toolbar, press the button **Include Actual Execution Plan** (the button will remain pressed). *Note: If you cannot find it in the toolbar, the same option is available in the Query menu.*

41. Write the following query to see the contents of table **Person.Address**:

```
SELECT * FROM Person.Address;
```

42. Execute the query and inspect the results, paying special attention to the indexed columns:
    - AddressID (primary key and clustered index)
    - StateProvinceID (foreign key and non-clustered index)
    - AddressLine1, AddressLine2, City, StateProvinceID, PostalCode (non-clustered index)

43. Note that there is a **Spatial results** tab because the results include geographical data, but we will not be using it in this course.

44. In the **Messages** tab, check the number of **logical reads**.
    Note: The number of physical reads may be zero if the data is already in memory.

45. In the **Execution plan** tab, check that the system is doing a **Clustered Index Scan** using the clustered index on the primary key.

46. Hover the mouse (or click) over the **Clustered Index Scan**, and a large tooltip will appear.

47. Check the **Number of Rows Read** and the **Object** that the system is operating on.

48. Write the following query:

```
SELECT * FROM Person.Address WHERE AddressID = 1000;
```

49. Take a moment to think about how the system should execute this query. (Is it possible to use an index?)

50. Execute the query and check that the desired record has been retrieved.

51. In the **Messages** tab, check the number of **logical reads**. (This was the number of page reads required to traverse the index.)

52. In the **Execution plan** tab, check that the system is doing a **Clustered Index Seek** (note that this is different from a **Clustered Index Scan**).

53. Hover the mouse over the **Clustered Index Seek**, and check the **Number of Rows Read** and the **Object** that the system is operating on.

54. We will try removing the index to see the impact on the query execution plan.
    For this purpose, write the following statement:

```
ALTER TABLE Person.Address DROP CONSTRAINT PK_Address_AddressID;
```

55. Before executing the statement, note that we are attempting to drop the index on the primary key by dropping the primary key itself! (This is because SQL Server does not allow DROP INDEX on indexes created by primary keys.)

56. When trying to execute the statement above, you should get the following error:

```
The constraint 'PK_Address_AddressID' is being referenced by table 'SalesOrderHeader',
foreign key constraint 'FK_SalesOrderHeader_Address_ShipToAddressID'.
```

57. In fact, the primary key is being referenced by foreign keys on multiple tables.
    To find those tables, write the following code:

```
SELECT OBJECT_NAME(fk.parent_object_id) AS [table],
       OBJECT_NAME(fk.object_id) AS [constraint]
FROM sys.foreign_keys AS fk
WHERE fk.referenced_object_id = OBJECT_ID('Person.Address');
```

58. Before executing the code, note the following:
    - The system view **sys.foreign_keys** returns a row for each foreign key constraint in the database.
    - Here we are interested in the foreign key constraints that reference the **Person.Address** table.
    - The query returns the tables that contain such references.

59. Execute the query above to find the tables and constraints that reference the primary key of **Person.Address**.

60. Write the following code to drop those constraints:

```
ALTER TABLE Person.BusinessEntityAddress
DROP CONSTRAINT FK_BusinessEntityAddress_Address_AddressID;

ALTER TABLE Sales.SalesOrderHeader
DROP CONSTRAINT FK_SalesOrderHeader_Address_BillToAddressID;

ALTER TABLE Sales.SalesOrderHeader
DROP CONSTRAINT FK_SalesOrderHeader_Address_ShipToAddressID;
```

61. Before executing the code, check that the constraint names agree with the results of the previous query.

62. Execute the code to drop the constraints.

63. Now try dropping the primary key again, this time it should succeed:

```
ALTER TABLE Person.Address DROP CONSTRAINT PK_Address_AddressID;
```

64. In **Object Explorer**, right-click **Person.Address** and select **Refresh**.

65. Expand **Keys**, **Indexes** and **Statistics** to confirm that the primary key and its index are no longer there.

66. Execute the following query again:

```
SELECT * FROM Person.Address WHERE AddressID = 1000;
```

67. In the **Messages** tab, check the number of **logical reads**.

68. In the **Execution plan** tab, check that, in the absence of the index, the system is now doing a **Table Scan** (when earlier, with the index, it was doing a **Clustered Index Seek**).

69. Hover the mouse over the **Table Scan**, and check the **Number of Rows Read** and the **Object** that the system is operating on.

70. Execute the following command to re-create the primary key and its clustered index:

```
ALTER TABLE Person.Address
ADD CONSTRAINT PK_Address_AddressID PRIMARY KEY(AddressID);
```

71. In **Object Explorer**, right-click **Person.Address** and select **Refresh**.

72. Expand **Keys**, **Indexes** and **Statistics** to confirm that the primary key and its clustered index are back.

73. Now execute the query again:

```
SELECT * FROM Person.Address WHERE AddressID = 1000;
```

74. In the **Messages** tab, check the number of **logical reads**.

75. In the **Execution plan** tab, check that the system is doing a **Clustered Index Seek** using the index on the primary key.

76. Execute the following query to obtain the index depth:

```
SELECT INDEXPROPERTY(OBJECT_ID('Person.Address'),
                     'PK_Address_AddressID',
                     'IndexDepth');
```

77. Check that the index depth agrees with the number of logical reads for the previous query.

78. Open a new query window.

79. In the toolbar, press the button **Include Actual Execution Plan** (the button will remain pressed). *Note: If you cannot find it in the toolbar, the same option is available in the Query menu.*

80. Execute the following query:

```
SELECT ModifiedDate FROM Person.Address WHERE ModifiedDate = '2014-01-01';
```

81. In the execution plan, check that the system is going through all the records in the table by scanning the clustered index associated with the primary key.

82. Execute the following statement to create an index on **ModifiedDate**:

```
CREATE INDEX IX_Address_ModifiedDate ON Person.Address(ModifiedDate);
```

83. In **Object Explorer**, refresh **Person.Address** and expand **Indexes** to check that the new index has been created, together with its corresponding statistics.

84. Right-click the index and select **Properties**, then check that the index is non-clustered.

85. Execute the same query again:

```
SELECT ModifiedDate FROM Person.Address WHERE ModifiedDate = '2014-01-01';
```

86. In the execution plan, check that the system is using the new index.

87. Note that the new index is a *covering index* for the query, i.e. the index contains all the information needed for the query, so the query can be answered based on the index alone.

88. Modify the query to select all the columns:

```
SELECT * FROM Person.Address WHERE ModifiedDate = '2014-01-01';
```

89. Execute and check the execution plan for this query.

90. Note that system is now using two indexes:
    - It uses the non-clustered index on ModifiedDate to locate the records with the desired date.
    - It uses the clustered index on the primary key to retrieve all the columns for those records.