SQL ServerCentral.com
redgate | Hub
Insights Learning Forums Ideas
Discover the new Redgate Hub
a community service from redgate
Try SQL Clone

Keep up to date - daily newsletter:
your@email.com
Sign up

Write For Us
Find out more

Proactively monitor your growing SQL Server environment.
Try free
SQL Monitor

Want more great articles like this? Sign up for fresh SQL Server knowledge delivered daily.
your@email.com
Sign up

Thank this author by sharing: 0
Rate this | Join the discussion | Add to briefcase

Unique Indexes in SQL Server: Stairway to SQL Server Indexes Level 8

By David Durant, 2011/09/14

The Series

This article is part of the Stairway Series: Stairway to SQL Server Indexes

Indexes are fundamental to database design, and tell the developer using the database a great deal about the intentions of the designer. Unfortunately indexes are too often added as an afterthought when performance issues appear. Here at last is a simple series of articles that should bring any database professional rapidly 'up to speed' with them

In this level we examine *Unique Indexes*. Unique indexes are special in that they provide more than just a performance benefit; they also provide a data integrity benefit. In SQL Server, unique indexes are the only reasonable way to enforce primary key and candidate key constraints.

Unique Indexes and Constraints

A unique index is no different than any other index, except that multiple entries with the same index key value are prohibited. Since each entry in an index maps to a row in the table, preventing an entry from being added to the index prevents the corresponding row from being added to the table. This is why unique indexes are the enforcers of primary key and candidate key constraints.

Declaring a PRIMARY KEY or UNIQUE constraint causes SQL Server to automatically create an index. You can have a unique index without having a matching constraint; but you cannot have either of these constraints without having a unique index. Defining the constraint will cause the index to be created, with the index name being the same as the constraint name. You will be unable to drop the index without first dropping the constraint, for the constraint cannot exist without the index. Dropping the constraint will cause its associated index to be dropped.

It is possible to have more than one unique index per table. The AdventureWork's *Product* table, for example, has four unique indexes: one each on the *ProductID*, *ProductNumber*, *rowguid*, and *ProductName*columns. The designers of the AdventureWorks database chose *ProductID* for the primary key;

Related Articles
FORUM Unique Constraint
2009/03/14
Unique Constraint
ARTICLE Unique Indexes Are Code; Non-Unique Indexes Are Data
2016/04/12
Unique indexes are the database developer's responsibility. Non-unique indexes can be more easily ma...
BLOG Using unique constraints
FORUM
FORUM

Stay up to date:
Daily newsletters with brand new articles, scripts, editorials and a Question of the Day help you keep on top of SQL Server.
your@email.com
Sign up
No thanks

Tags
indexing
stairway series

Write For Us
Find out more

Copyright © 2002-2018 Redgate. All Rights Reserved. Privacy Policy. Terms of Use. Report Abuse.

the other three are alternate keys, sometimes called candidate keys.

You can create a unique index with either a CREATE INDEX statement, as shown here:

```
CREATE UNIQUE NONCLUSTERED INDEX [AK_Produc
ON Production.Product ( [Name] );
```

Or by defining a constraint, as in:

```
ALTER TABLE Production.Product
  ADD CONSTRAINT PK_Product_ProductID PRIMA
    (
      ProductID    );
```

In the first case, you are ensuring that no two products will ever have the same name; in the second, you are ensuring that no two products will have the same *ProductID* value.

Because defining a primary key or alternate key constraint causes an index to be created, you must specify the necessary index information in the constraint definition; hence the "CLUSTERED" keyword in the ALTER TABLE statement shown above.

If the table contains data that would violate the constraint or violate the index restriction, the CREATE INDEX statement will fail.

If the index can be created, then any subsequent DML (Data Manipulation Language) statement that would violate the constraint or index will fail. For example, suppose we try to insert a row with a duplicate product name, as shown in Listing 1:

```
INSERT Production.Product
(
Name, ProductNumber,
Color,
SafetyStockLevel, ReorderPoint, StandardCos
ListPrice, Size,
SizeUnitMeasureCode, WeightUnitMeasureCode,
DaysToManufacture, ProductLine, Class, Styl
ProductSubcategoryID, ProductModelID,
SellStartDate, SellEndDate, DiscontinuedDat
)
VALUES
(
'Full-Finger Gloves, M', 'A unique product
'Black',
4, 3, 20.00, 40.00, 'M',
NULL, NULL, NULL,
0, 'M', NULL, 'U', 20, 3,
GETDATE(), GETDATE(), NULL
);
```

Listing 1: INSERT a duplicate value for product name

The statement fails to execute and we receive the following error message:

```
Msg 2601, LEVEL 14, State 1, Line 1
Cannot INSERT duplicate KEY row IN object '
```

```
WITH UNIQUE INDEX  'AK_Product_Name'.
The statement has been terminated.
```

The message informs us that our *AK_Product_Name* index has successfully defended our table against an attempted insert of a row that contains an already existing product name.

Primary Key or Unique Constraint or No Constraint at All

There are some small differences between a PRIMARY KEY constraint and a UNIQUE constraint.

- A PRIMARY KEY constraint prohibits NULLs; no index key column of any index entry can contain a null. A UNIQUE constraint does allow NULLs. However, since a UNIQUE constraint treats two NULLs as duplicates of each other, only one search key value containing NULLs in every column can exist in the index.
- Creating a PRIMARY KEY constraint results in the creation of a clustered index unless either of the following is true:
- - The table already is a clustered index.
  - You specify NONCLUSTERED when you define the constraint.
- Creating a UNIQUE constraint results in a nonclustered index unless you specify CLUSTERED when you define the constraint and the table is not already clustered.
- There can be only one PRIMARY KEY constraint per table. There can be multiple UNIQUE constraints per table.

When deciding whether to create a unique constraint or just a unique index, follow the guideline set down in the SQL Server documentation in the MSDN library:

"There are no significant differences between creating a UNIQUE constraint and creating a unique index that is independent of a constraint. Data validation occurs in the same manner, and the query optimizer does not differentiate between a unique index created by a constraint or manually created. However, you should create a UNIQUE constraint on the column when data integrity is the objective. This makes the objective of the index clear."

The property of unique indexes mentioned above, that they allow only one NULL, is often in conflict with a common business requirement. Often, we would like to enforce uniqueness of existing values in a column but allow multiple rows to have no value at all for that column.

For instance, suppose you were a supplier of products, some of which you obtain from third party vendors, and you keep your product information in a table called *ProductDemo*. You have your own *ProductID* value that you assign to all products. You also track the UPC (Universal Product Code) value, but not all vendors' products have a UPC. Your table, in part, has the values shown in Table 1:

ProductID	UPCode	Other Columns
(Primary Key)	(Unique, but not a key)	
14AJ-W	036000291452	
23C2-M		
23C2-L		

18MM-J	044000865867	

Table 1: Partial contents of the ProductDemotable

In the second column, you need to enforce uniqueness of UPCs while still allowing NULLs. The best way to provide this functionality is to combine a unique index with a filtered index. (Filtered indexes were the subject of Level 7, and were introduced in SQL Server 2008.)

To illustrate this, we create a simple table containing the columns shown above:

```
CREATE TABLE ProductDemo
     (
       ProductID NCHAR(6)  NOT NULL PRIMARY
       UPCode    NCHAR(12) NULL
     );
```

Now, when we insert the multiple NULL UPCode rows shown below, all four rows are added to the table.

```
INSERT ProductDemo (ProductID , UPCode) VAL
, ('23CZ-M', NULL)
, ('23CZ-L', NULL)
, ('18MM-J', '044000865867');
```

However, when we try to add a row with a duplicate UPCode value:

```
INSERT ProductDemo (ProductID , UPCode) VAL
```

We receive the following error message and the row is not inserted.

Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'dbo.ProductDemo' with unique index 'AK_UPCode'.
The statement has been terminated.

As it did in the earlier example in this level, our index has provided the data integrity that we requested. But this time it also allows multiple rows to have NULLs.

Choosing the Correct IGNORE_DUP_KEY Option

When you create a unique index, you have the option of specifying the IGNORE_DUP_KEY option. Thus, our original CREATE INDEX statement could have been worded:

```
CREATE UNIQUE NONCLUSTERED INDEX AK_Product
ON Production.Product ( [Name] )
WITH ( IGNORE_DUP_KEY = OFF );
```

The name of this option is slightly misleading, for a duplicate key is never ignored when a unique index is present; more correctly, a duplicate key is never allowed in a unique index. This option controls a behavior that is applicable only during a multi-row insert operation.

For instance, if you had two tables, *TableA* and *TableB*, that had identical structure, you could submit the following INSERT statement to SQL Server:

```
INSERT INTO TableA
SELECT *
FROM TableB;
```

SQL Server would attempt to copy all the *TableB* rows into *TableA*. What if two of the *TableB* rows could not be copied to *TableA* because of duplicate values? Would you want all the other rows to be copied, and only the two rows to fail, or would you want the entire INSERT statement to fail?

The choice is yours to make. When you create the unique index, you make the choice as to what should happen if an INSERT statement attempts to add duplicate values to a unique key. The two settings for IGNORE_DUP_KEY are interpreted as follows:

- IGNORE_DUP_KEY = OFF

The entire INSERT statement will fail.
An error message will be issued.
Note: This choice is the default.

- IGNORE_DUP_KEY = ON

Only the rows with index key values that are duplicates of existing rows will fail.

A warning message will be issued.

Note: This choice cannot be used if the unique index is also a filtered index.

The IGNORE_DUP_KEY option only effects INSERT statements. It is ignored by UPDATE, CREATE INDEX, and ALTER INDEX statements. The IGNORE_DUP_KEY option can also be specified when adding a PRIMARY KEY or UNIQUE constraint to a table.

Unique indexes can provide an unexpected performance benefit. This is because they provide information to SQL Server that we often take for granted, but which SQL Server can never assume. Two of the unique indexes on the *AdventureWork's Product* table, *ProductID* and *ProductName*, provide an example of this.

Suppose, you receive a request from the warehouse staff for a query that will show them the following information for each product in the *Product* table:

- The product name.
- The count of the number of times this product has been sold.
- The total value of those sales.

In response, you write the following query:

```
SELECT [Name]
     , COUNT(*) AS 'RowCount'
     , SUM(LineTotal) AS 'TotalValue'
  FROM Production.Product P
  JOIN Sales.SalesOrderDetail D ON D.Produc
  GROUP BY ProductID
```

The warehouse staff is very pleased with your query, as it gives them the results they wanted; one row per product, each row containing the product name, sales count and total sales value. This output is shown, in part, below:

```
Name                    RowCount   TotalValue
------------------------ -------- ---------------------
Sport-100 Helmet, Red      3083     157772.394392
```

Sport-100 Helmet, Black	3007	160939.517626
Mountain Bike Socks, M	188	6080.398200
Mountain Bike Socks, L	44	513.000000
Sport-100 Helmet, Blue	3090	165406.617049
AWC Logo Cap	3952	51229.445623
Long-Sleeve Logo Jersey, S	429	21445.710000
Long-Sleeve Logo Jersey, M	1218	115249.214976
Long-Sleeve Logo Jersey, L	1635	198754.975360
Long-Sleeve Logo Jersey, XL	1078	95611.197090
HL Road Frame - Red, 62	218	394255.572400

.

.

You, however, are concerned about the potential cost of this query. *SalesOrderDetail* is the larger of the two tables referenced in the query; and its rows must be grouped by product name, a value that is carried in the *Product* table, not the *SalesOrderDetail* table.

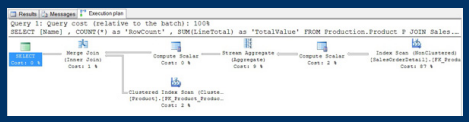
Using SQL Server Management Studio, you note that the *SalesOrderDetail* table is clustered on its primary key, *SalesOrderID / SalesOrderDetailID*; which will not be of benefit when trying to group rows by product name.

If you ran the code in *Level 5 – Included Columns*, you created the following nonclustered index on the *SalesOrderDetail.ProductID* foreign key column:

```
CREATE NONCLUSTERED INDEX FK_ProductID_Modi
ON Sales.SalesOrderDetail
(
ProductID,
ModifiedDate
)
INCLUDE
(
OrderQty,
UnitPrice,
LineTotal
);
```

You feel that this index should help your query because it contains all the information required by your query, except the product name, and it is in *ProductID* sequence. But you are still concerned about having to group information in one table by a value that is in a different table.

You return to SQL Server Management Studio, turn on the **Show Actual Execution Plan** option, run the query, and note the execution plan shown in Figure 1. (Viewing and evaluating query plans is the subject of Level 9).



Zoom in | Open in new window

Figure 1: Execution plan when grouping by Name

At first you are surprised to see that the *Product*table's product name index, *Product.AK_Product_Name*, is never used, even though its column is the aggregation key of the GROUP BY clause. Then you realize that having a unique index on *Product.Name* and a unique index on *Product.ProductID* informs SQL Server that there is one product per product name and one

product per product id. Therefore, GROUP BY [Name] or GROUP BY ProductID is the same grouping, that is, they both produce one group per product.

Thus, the query optimizer realized that your query was identical to the query shown below, and, therefore, that the two *ProductID* indexes would simultaneously support both joining and grouping of the requested data.

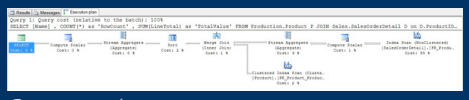
```
SELECT[Name]
, COUNT(*)AS 'RowCount'
, SUM(LineTotal)AS 'TotalValue'
FROM Production.Product P
JOIN Sales.SalesOrderDetail D OND.ProductID
GROUP BY ProductID
```

SQL Server was able to simultaneously scan the covering index on the *SalesOrderDetail* table and the clustered indexed *Product* table, both of which are in *ProductID*sequence, generate the totals for each group, and merge in the product name, without having to do any sorting or hashing. In short, SQL Server generated the most efficient plan possible for your query.

If you drop the *Product.AK_Product_Name* index, like so:

```
IF EXISTS ( SELECT *
FROM sys.indexes
WHERE OBJECT_ID = OBJECT_ID(N'Production.Pr
AND name = N'AK_Product_Name')
DROP INDEX AK_Product_Name
ON Production.Product;
```

The new query plan (shown below) is less efficient, requiring additional sort and merge operations.



Zoom in | Open in new window

Figure 2: Execution plan when grouping by Name, after dropping the index on Name

You can see that although the primary purpose of a unique index is to provide for the integrity of your data, it can also help the query optimizer determine the most efficient way to gather that data, even if that index is not used to access the data.

Unique indexes provide support for primary key and alternate key constraints. A unique index may exist with a corresponding constraint, but a constraint cannot exist without its index.

A unique index can also be a filtered index. This allows for enforcement of uniqueness of values in columns that permit multiple NULLs within the column.

The IGNORE_DUP_KEY option influences the behavior of multi-row insert statements.

A unique index can provide for better query performance, even if the index is not used by the query.