Physical Database Design in SQL Server

Physical Structures

- Primary structures
  - Heap
  - Ordered with sparse B+-tree
- Secondary indexes: dense B+-tree

CREATE TABLE

```
CREATE TABLE     [database_name . [schema_name] .
    | schema_name.] table_name
    ( { <column_definition> |
    <computed_column_definition> }
        [ <table_constraint> ] [ ,...n ] )
    [ ON { partition_scheme_name
    (partition_column_name ) | filegroup | "default" } ]
     [ { TEXTIMAGE_ON { filegroup | "default" } ]
```

CREATE TABLE

- Creates table table_name
- [ON { *partition_scheme_name* (*partition_column_name*) | *filegroup* | "default" }]
  - Specifies the partition scheme or filegroup on which the table is stored.
  - If **"**default**"** is specified, or if ON is not specified at all, the table is stored on the default filegroup.
  - The storage mechanism of a table as specified in CREATE TABLE cannot be subsequently altered.

TEXTIMAGE_ON

- [{ TEXTIMAGE_ON { *filegroup* | "default" }]
  - indicate that the **text**, **ntext**, **image**, **xml**, **varchar(max)**, **nvarchar(max)**, **varbinary(max)**, and CLR user-defined type columns are stored on the specified filegroup.
  - TEXTIMAGE_ON is not allowed if there are no large value columns in the table.
  - TEXTIMAGE_ON cannot be specified if <partition_scheme> is specified.
  - If **"**default**"** is specified, or if TEXTIMAGE_ON is not specified at all, the large value columns are stored in the default filegroup. The storage of any large value column data specified in CREATE TABLE cannot be subsequently altered.

<column_definition>

<column_definition> ::=*column_name* <data_type>
[COLLATE *collation_name*]
[NULL | NOT NULL]
[
[CONSTRAINT *constraint_name*] DEFAULT
*constant_expressio*n
| IDENTITY [(*seed* **,***increment*)]
]
[ROWGUIDCOL]
[<column_constraint> [...*n*]]

<column_definition>

- NULL | NOT NULL
  - Determine whether null values are allowed in the column.

IDENTITY

- When a new row is added to the table, the Database Engine provides a unique, incremental value for the column.
- Identity columns are typically used with PRIMARY KEY constraints to serve as the unique row identifier for the table.
- The IDENTITY property can be assigned to **tinyint**, **smallint**, **int**, **bigint**, **decimal(p,0)**, or **numeric(p,0)** columns.
- Only one identity column can be created per table.
- DEFAULT constraints cannot be used with an identity column.
- Both the seed and increment or neither must be specified. If neither is specified, the default is (1,1).

ROWGUIDCOL

- Although the IDENTITY property automates row numbering within one table, separate tables, each with its own identifier column, can generate the same values.
- If an application must generate an identifier column that is unique across the database, or every database on every networked computer in the world, use the ROWGUIDCOL property, the **uniqueidentifier** data type, and the NEWID function.

ROWGUIDCOL

- When you use the ROWGUIDCOL property to define a Globally Unique Identifier (GUID) column, consider the following:
  - A table can have only one ROWGUIDCOL column, and that column must be defined by using the **uniqueidentifier** data type.
  - The Database Engine does not automatically generate values for the column. To insert a globally unique value, create a DEFAULT definition on the column that uses the NEWID function to generate a globally unique value.
  - Because the ROWGUIDCOL property does not enforce uniqueness, the UNIQUE constraint should be used to guarantee that unique values are inserted into the ROWGUIDCOL column.

<column_constraint>

<column_constraint> ::=

[CONSTRAINT *constraint_name*]

{ { PRIMARY KEY | UNIQUE }

[CLUSTERED | NONCLUSTERED]

[ON { *partition_scheme_name* **(** *partition_column_name* **)** | *filegroup* | **"**default**"** }]

| [FOREIGN KEY]

REFERENCES [*schema_name* .] *referenced_table_name*

[**(** *ref_column* **)**]

[ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]

[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]

| CHECK **(** *logical_expression* **)** }

<column_constraint>

- PRIMARY KEY
  - Is a constraint that enforces entity integrity for a specified column or columns through a unique index. Only one PRIMARY KEY constraint can be created per table.
- UNIQUE
  - Is a constraint that provides entity integrity for a specified column or columns through a unique index. A table can have multiple UNIQUE constraints.
- CLUSTERED | NONCLUSTERED
  - Indicate that a clustered or a nonclustered index is created for the PRIMARY KEY or UNIQUE constraint. PRIMARY KEY constraints default to CLUSTERED, and UNIQUE constraints default to NONCLUSTERED.

<column_constraint>

  - In a CREATE TABLE statement, CLUSTERED can be specified for only one constraint. If CLUSTERED is specified for a UNIQUE constraint and a PRIMARY KEY constraint is also specified, the PRIMARY KEY defaults to NONCLUSTERED.
  - The clause ON indicates where the index will be stored
- CHECK
  - Is a constraint that enforces domain integrity by limiting the possible values that can be entered into a column or columns.
- *logical_expression*
  - Is a logical expression that returns TRUE or FALSE.

Examples of <column_definition>

EmployeeID int
PRIMARY KEY CLUSTERED

SalesPersonID int NULL
REFERENCES SalesPerson(SalesPersonID)

Name nvarchar(100) NOT NULL
UNIQUE NONCLUSTERED

Computed Columns

- A computed column is computed from an expression that uses other columns in the same table.
- It is not physically stored in the table, unless the column is marked PERSISTED.
- For example, a computed column can have the definition: **cost** AS **price** * **qty**. The expression can be a noncomputed column name, constant, function, variable, and any combination of these connected by one or more operators. The expression cannot be a subquery.

Computed Columns

- Computed columns can be used in select lists, WHERE clauses, ORDER BY clauses, or any other locations in which regular expressions can be used, with the following exceptions:
  - A computed column can be used as a key column in an index or as part of any PRIMARY KEY or UNIQUE constraint, if the computed column value is defined by a deterministic expression and the data type of the result is allowed in index columns.
    - For example, if the table has integer columns **a** and **b**, the computed column **a+b** may be indexed, but the computed column **a+DATEPART(dd, GETDATE())** cannot be indexed because the value may change in subsequent invocations.
  - A computed column cannot be the target of an INSERT or UPDATE statement.

<computed_column_definition>

<computed_column_definition> ::=*column_name* AS
computed_column_expression
[PERSISTED [NOT NULL]]
[[CONSTRAINT *constraint_name*]
{ PRIMARY KEY | UNIQUE }
[CLUSTERED | NONCLUSTERED]
[WITH FILLFACTOR = *fillfactor*
| WITH **(** < index_option > [**,** ...*n*] **)**]
| [FOREIGN KEY]
REFERENCES [*schema_name* .] *referenced_table_name*
[**(** *ref_column* **)**]
[ON DELETE { NO ACTION | CASCADE }]
[ON UPDATE { NO ACTION }]
| CHECK **(** *logical_expression* **)** }
[ON { *partition_scheme_name* **(** *partition_column_name* **)** | *filegroup* |
"default**"** }]
]

PERSISTED

- Specifies that the SQL Server Database Engine will physically store the computed values in the table, and update the values when any other columns on which the computed column depends are updated.
- Any computed column that is used as a partitioning column of a partitioned table must be explicitly marked PERSISTED.
- *computed_column_expression* must be deterministic when PERSISTED is specified

<table_constraint>

< table_constraint > ::=

[CONSTRAINT *constraint_name*]
{ { PRIMARY KEY | UNIQUE }
[CLUSTERED | NONCLUSTERED]
(*column* [ASC | DESC] [**,**...*n*] **)**
[ON { *partition_scheme_name* **(***partition_column_name***)** | *filegroup* |
"default" }]
| FOREIGN KEY **(** *column* [**,**...*n*] **)**
REFERENCES *referenced_table_name* [**(** *ref_column* [**,**...*n*] **)**]
[ON DELETE { NO ACTION | CASCADE | SET NULL |
SET DEFAULT}]
[ON UPDATE { NO ACTION | CASCADE | SET NULL |
SET DEFAULT }]
| CHECK **(** *logical_expression* **)** }

<table_constraint>

- Used for defining multicolumn constraints
- *column*: is a column or list of columns, in parentheses, used to indicate the columns used in the constraint definition.
- [ASC | DESC]: specifies the order in which the column or columns participating in table constraints are sorted. The default is ASC.

Example of <table_constraint>

FOREIGN KEY (SalesPersonID) REFERENCES SalesPerson(SalesPersonID)

CONSTRAINT FK_SpecialOfferProduct_SalesOrderDetail FOREIGN KEY (ProductID, SpecialOfferID)

REFERENCES SpecialOfferProduct (ProductID, SpecialOfferID)

Complete Example

```
CREATE TABLE [dbo].[PurchaseOrderDetail]
(    [PurchaseOrderID] [int] NOT NULL
        REFERENCES
     Purchasing.PurchaseOrderHeader(PurchaseOrderID),
    [LineNumber] [smallint] NOT NULL,
    [ProductID] [int] NULL
        REFERENCES Production.Product(ProductID),
    [DueDate] [datetime] NULL,
    [rowguid] [uniqueidentifier] ROWGUIDCOL  NOT NULL
        CONSTRAINT [DF_PurchaseOrderDetail_rowguid] DEFAULT
     (newid()),
    [ModifiedDate] [datetime] NOT NULL
        CONSTRAINT [DF_PurchaseOrderDetail_ModifiedDate] DEFAULT
     (getdate()),
    [UnitPrice] [money] NULL, [OrderQty] [smallint] NULL,
    [LineTotal]  AS (([UnitPrice]*[OrderQty])),
CONSTRAINT [PK_PurchaseOrderDetail_PurchaseOrderID_LineNumber]
    PRIMARY KEY CLUSTERED ([PurchaseOrderID], [LineNumber])
) ON [PRIMARY]
```

Partitions

- A large table or index can be divided into partitions
- A partition is an horizontal portion of a table or index
- A column is chosen on the basis of which performing partitioning
- The partitions can be spread across more than one filegroup in a database.
- The table or index is treated as a single logical entity when queries or updates are performed on the data.
- All partitions of a single index or table must reside in the same database.

Partitions

- Example: A transaction table
- The current month of data is primarily used for INSERT, UPDATE, and DELETE operations
- Previous months are used primarily for SELECT queries
- Partitioning by month can be useful for
  - Maintenance operations: index rebuilds
  - Moving one month of read-only data from this table to a data warehouse table for analysis. With partitioning, subsets of data can be separated quickly from a table and then added as partitions to another existing partitioned table, assuming these tables are all in the same database instance.
  - Improving query performance

Partitioning

- In SQL Server, all tables and indexes in a database are considered partitioned, even if they are made up of only one partition.
- Essentially, partitions form the basic unit of organization in the physical architecture of tables and indexes.

Partitioned Indexes

- Partitioned indexes can be implemented independently from their base tables
- However, it generally makes sense to partition the index in the same way as the underlying table. We say the index is *aligned* with the table
- When you design a partitioned table and then create an index on the table, SQL Server automatically partitions the index by using the same partition scheme and partitioning column as the table.

Partitioning

- Before partitioning a table or index you need to create the following database objects:
  - Partition function: determines the values of the thresholds on the partitioning column
  - Partition scheme: determines in which filegroup each partition is stored

Example

CREATE PARTITION FUNCTION myRangePF1 (int)

AS RANGE LEFT FOR VALUES (1, 100, 1000);

GO

CREATE PARTITION SCHEME myRangePS1

AS PARTITION myRangePF1

TO (test1fg, test2fg, test3fg, test4fg);

The partitions of a table that uses partition function myRangePF1 on partitioning column **col1** would be assigned as shown in the following table

Partition	1	2	3	4
Values	**col1** <= 1	**col1** > 1 AND **col1** <= 100	**col1**>100 AND **col1** <= 1000	**col1** > 1000
Filegroup	**test1fg**	**test2fg**	**test3fg**	**test4fg**

Example

```
CREATE TABLE PartitionTable (col1 int, col2 char(10))
ON myRangePS1 (col1) ;
GO
```

Views

- A view can be thought of as either a virtual table or a stored query.
- Unless a view is indexed, its data is not stored in the database as a distinct object. What is stored in the database is a SELECT statement. The result set of the SELECT statement forms the virtual table returned by the view.
- A user can use this virtual table by referencing the view name in Transact-SQL statements in the same way in which a table is referenced.

Types of Views

- **Standard views**
- **Indexed Views:**
  - it is a view that has been materialized. This means it has been computed and stored. You index a view by creating a unique clustered index on it.
  - Indexed views dramatically improve the performance of some types of queries. Indexed views work best for queries that aggregate many rows. They are not well-suited for underlying data sets that are frequently updated.

CREATE VIEW

CREATE VIEW [*schema_name* **.**] *view_name* [
(*column* [**,**...*n*] **)**]
[WITH <view_attribute> [**,**...n]]
AS *select_statement* [;]
[WITH CHECK OPTION]

<view_attribute> ::= {
[SCHEMABINDING]
…}

column

- *column*
  - If *column* is not specified, the view columns acquire the same names as the columns in the SELECT statement.
  - If it is specified, it is the name to be used for a column in a view. A column name is required only when a column is derived from an arithmetic expression, a function, or a constant; when two or more columns may otherwise have the same name, typically because of a join; or when a column in a view is given a name different from that of the column from which it is derived.

WITH CHECK OPTION

- Forces all data modification statements executed against the view to satisfy the criteria set within *select_statement*. When a row is modified through a view, the WITH CHECK OPTION makes sure the data remains visible through the view after the modification is committed.

SCHEMABINDING

- When SCHEMABINDING is specified, the base table or tables cannot be modified in a way that would affect the view definition.
- The view definition itself must first be modified or dropped to remove dependencies on the table that is to be modified, otherwise an error is returned

Updatable Views

- You can modify the data of an underlying base table through a view, as long as the following conditions are true:
  - Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table.
  - The columns being modified in the view must directly reference the underlying data in the table columns. The columns cannot be derived in any other way, such as through the following:
    - An aggregate function: AVG, COUNT, SUM, MIN, MAX, GROUPING
    - A computation. The column cannot be computed from an expression that uses other columns.

Updatable Views

- The columns being modified are not affected by GROUP BY, HAVING, or DISTINCT clauses.
- TOP is not used anywhere in the *select_statement* of the view together with the WITH CHECK OPTION clause.

View Example

```
USE AdventureWorks ;
GO
CREATE VIEW hiredate_view
AS
SELECT c.FirstName, c.LastName, e.EmployeeID,
    e.HireDate
FROM HumanResources.Employee e JOIN
    Person.Contact c on e.ContactID = c.ContactID ;
GO
```

An UPDATE that modifies an employee LastName and HireDate returns an error

View Example

```
USE AdventureWorks ;
GO
CREATE VIEW SeattleOnly
AS
SELECT p.LastName, p.FirstName, p.City,
FROM Person p WHERE p.City = 'Seattle'
WITH CHECK OPTION ;
GO
```

An UPDATE that changes the city of a Person returns an error.

Index Types

Index type	Description
Clustered	A clustered index sorts and stores the data rows of the table or view in order based on the clustered index key.
Nonclustered	A nonclustered index can be defined on a table or view with a clustered index or on a heap. Each index row in the nonclustered index contains the nonclustered key value and a row locator.
Unique	A unique index ensures that the index key contains no duplicate values and therefore every row in the table or view is in some way unique. Both clustered and nonclustered indexes can be unique.

Index Types

Index with included columns	A nonclustered index that is extended to include nonkey columns in addition to the key columns.
Indexed views	An index on a view materializes the view and the result set is permanently stored in a unique clustered index in the same way a table with a clustered index is stored. Nonclustered indexes on the view can be added after the clustered index is created.
Full-text	A special type of token-based functional index that is built and maintained by the Microsoft Full-Text Engine for SQL Server (MSFTESQL) service. It provides efficient support for sophisticated word searches in character string data.
XML	A persisted, representation of the XML binary large objects (BLOBs) in the **xml** data type column.

Index Creation

- Indexes are automatically created when PRIMARY KEY and UNIQUE constraints are defined on table columns.
- A unique index is created to enforce the uniqueness requirements of a PRIMARY KEY or UNIQUE constraint.
- By default, a unique clustered index is created to enforce a PRIMARY KEY constraint, unless a clustered index already exists on the table, or you specify a nonclustered index.
- By default, a unique nonclustered index is created to enforce a UNIQUE constraint unless a unique clustered index is explicitly specified and a clustered index on the table does not exist.
- An index created as part of a PRIMARY KEY or UNIQUE constraint is automatically given the same name as the constraint name.

Index Creation

- An index can be built with
  - CREATE INDEX of T-SQL or
  - New Index of Management Studio

Relational Index Creation

```
CREATE [ UNIQUE ] [ CLUSTERED |
   NONCLUSTERED ] INDEX index_name
   ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
   [ WITH ( <relational_index_option> [ ,...n ] ) ]
   [ ON { partition_scheme_name ( column_name )
        | filegroup_name
        | "default" }
   ]
<object> ::= table_or_view
```

Unique Indexes

- The benefits of unique indexes include the following:
  - Data integrity of the defined columns is ensured.
  - Additional information helpful to the query optimizer is provided
- There are no significant differences between creating a UNIQUE constraint and creating a unique index independently 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.
- Create a UNIQUE or PRIMARY KEY constraint on the column when data integrity is the objective.

Unique Indexes

- For indexing purposes, NULL values compare as equal. Therefore, you cannot create a unique index, or UNIQUE constraint, if the key values are NULL in more than one row. Select columns that are defined as NOT NULL when you choose columns for a unique index or unique constraint.

Sort Order

- You should consider whether the data for the index key column should be stored in ascending or descending order.
- Ascending is the default Keywords: ASC (ascending) and DESC (descending)
- Specifying the order in which key values are stored in an index is useful when queries referencing the table have ORDER BY clauses that specify order directions for the columns in that index.
- In these cases, the index can remove the need for a SORT operator in the query plan; therefore, this makes the query more efficient.

Sort Order Example

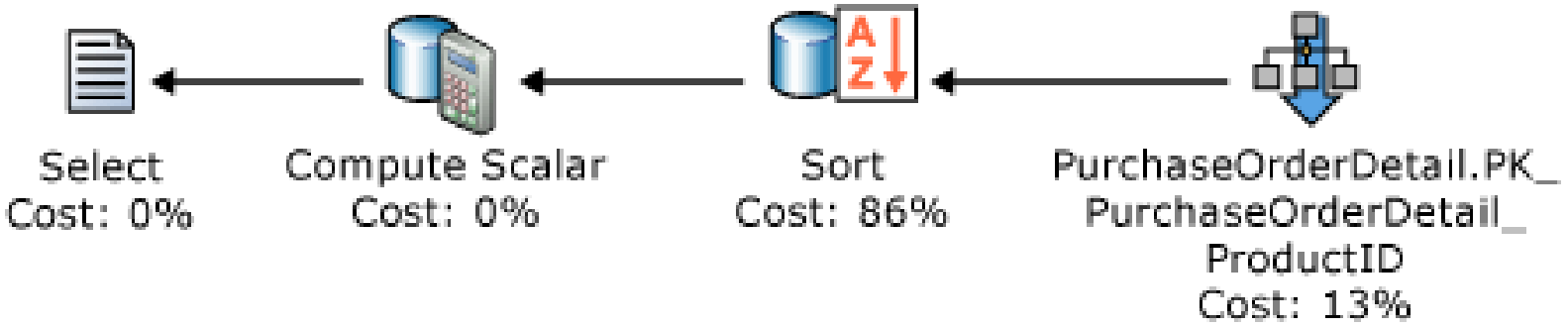
- The buyers in the Adventure Works Cycles purchasing department have to evaluate the quality of products they purchase from vendors. The buyers are most interested in finding products sent by these vendors with a high rejection rate.
- Retrieving the data to meet this criteria requires the RejectedQty column in the Purchasing.PurchaseOrderDetail table to be sorted in descending order (large to small) and the ProductID column to be sorted in ascending order (small to large).

Sort Order Example

```
USE AdventureWorks;
GO
SELECT RejectedQty, ((RejectedQty/OrderQty)*100)
    AS RejectionRate, ProductID, DueDate
FROM Purchasing.PurchaseOrderDetail
ORDER BY RejectedQty DESC, ProductID ASC;
```

Sort Order Example

- The following execution plan for this query shows that the query optimizer used a SORT operator to return the result set in the order specified by the ORDER BY clause.



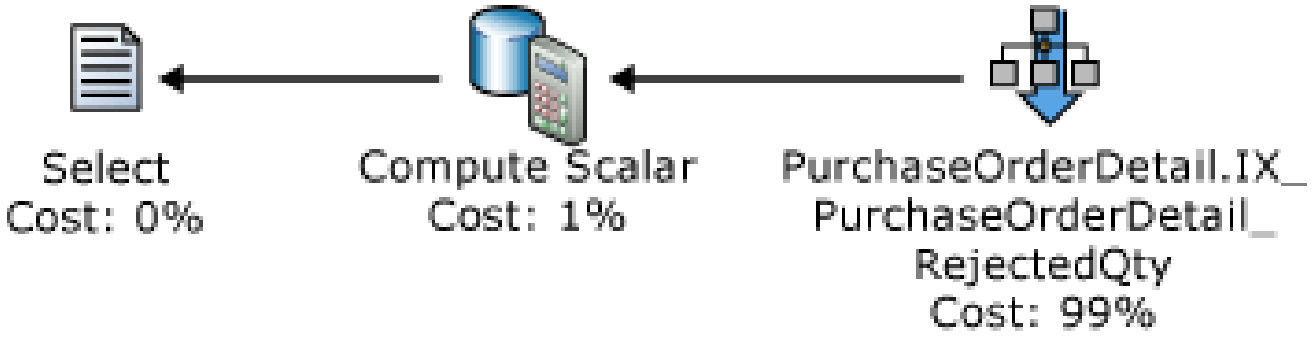
Sort Order Example

- If an index is created with key columns that match those in the ORDER BY clause in the query, the SORT operator can be eliminated in the query plan and the query plan is more efficient:

```
CREATE NONCLUSTERED INDEX
    IX_PurchaseOrderDetail_RejectedQty
ON Purchasing.PurchaseOrderDetail
    (RejectedQty DESC, ProductID ASC);
```

Sort Order Example

- After the query is executed again, the following execution plan shows that the SORT operator has been eliminated and the newly created nonclustered index is used.



Sort Order

- SQL Server can move equally efficiently in either direction. An index defined as (RejectedQty DESC, ProductID ASC) can still be used for a query in which the sort direction of the columns in the ORDER BY clause are reversed.
- For example, a query with the ORDER BY clause ORDER BY RejectedQty ASC, ProductID DESC can use the index.
- A query with the ORDER BY clause ORDER BY RejectedQty ASC, ProductID ASC can not use the index.

Index Position

- ON *filegroup_name*
  - Creates the specified index on the specified filegroup. If no location is specified and the table or view is not partitioned, the index uses the same filegroup as the underlying table or view. The filegroup must already exist.
- ON **"**default**"**
  - Creates the specified index on the default filegroup.
  - The term default, in this context, is not a keyword. It is an identifier for the default filegroup and must be delimited, as in ON **"**default**"** or ON **[**default**]**.

CREATE INDEX Example

- The following example creates a nonclustered index on the VendorID column of the Purchasing.ProductVendor table.

```
USE AdventureWorks;
GO
CREATE INDEX IX_ProductVendor_VendorID
    ON Purchasing.ProductVendor (VendorID);
GO
```

CREATE INDEX Example

- The following example creates a nonclustered composite index on the SalesQuota and SalesYTD columns of the Sales.SalesPerson table.

```
USE AdventureWorks
GO
CREATE NONCLUSTERED INDEX
   IX_SalesPerson_SalesQuota_SalesYTD
   ON Sales.SalesPerson (SalesQuota, SalesYTD);
GO
```

CREATE INDEX Example

- The following example creates a unique nonclustered index on the Name column of the Production.UnitMeasure table.

```
USE AdventureWorks;
GO
CREATE UNIQUE INDEX AK_UnitMeasure_Name
   ON Production.UnitMeasure(Name);
GO
```

- AK stands for Alternate Key

CREATE INDEX Example

Attempting to insert a row with the same value as that in an existing row.

```
INSERT INTO Production.UnitMeasure
(UnitMeasureCode, Name, ModifiedDate) VALUES
('OC', 'Ounces', GetDate());
GO
```

Result:

```
Server: Msg 2601, Level 14, State 1, Line 1
Cannot insert duplicate key row in object 'UnitMeasure'
with unique index 'AK_UnitMeasure_Name'. The
statement has been terminated.
```

Indexes on Views

- The first index created on a view must be a unique clustered index. After the unique clustered index has been created, you can create additional nonclustered indexes.
- The naming conventions for indexes on views are the same as for indexes on tables. The only difference is that the table name is replaced with a view name.
- The view must be defined with SCHEMABINDING to create an index on it
- An indexed view is stored in the database in the same way a table with a clustered index is stored.

Index Creation

- Whether the index will be created on an empty table or one that contains data is an important factor to consider.
- Creating an index on an empty table has no performance implications at the time the index is created; however, performance will be affected when data is added to the table.
- Creating indexes on large tables should be planned carefully so database performance is not hindered. The preferred way to create indexes on large tables is to start with the clustered index and then build any nonclustered indexes.

Index Creation

- If a clustered index is created on a heap with several existing nonclustered indexes, all the nonclustered indexes must be rebuilt so that they contain the clustering key value instead of the row identifier (RID). Similarly, if a clustered index is dropped on a table that has several nonclustered indexes, the nonclustered indexes are all rebuilt as part of the DROP operation. This may take significant time on large tables.

Index Creation

- In SQL Server you can create, rebuild, or drop indexes online with the ONLINE option set to ON.
- It allows concurrent user access to the underlying table or clustered index data and any associated nonclustered indexes during index operations.
- For example, while a clustered index is being rebuilt by one user, that user and others can continue to update and query the underlying data.
- When you perform DDL operations offline, such as building or rebuilding a clustered index; these operations hold long-term exclusive locks on the underlying data and associated indexes. This prevents modifications and queries to the underlying data until the index operation is complete.

Fragmentation

- Over time insert, update, or delete operations can cause the information in the index to become scattered in the database (fragmented).
- Fragmentation exists when indexes have pages in which the logical ordering, based on the key value, does not match the physical ordering inside the data file.
- Heavily fragmented indexes can degrade query performance and cause your application to respond slowly.
- You can remedy index fragmentation by either reorganizing an index or by rebuilding an index.

Detecting Fragmentation

- By using the system function **sys.dm_db_index_physical_stats**, you can detect fragmentation in a specific index, all indexes on a table or indexed view, all indexes in a database, or all indexes in all databases.
- For partitioned indexes, **sys.dm_db_index_physical_stats** also provides fragmentation information for each partition.
- The result set returned by this function includes the following column:
  - **avg_fragmentation_in_percent:** the percent of logical fragmentation (out-of-order pages in the index).

Correcting Fragmentation

- After the degree of fragmentation is known, use the following table to determine the best method to correct the fragmentation:

avg_fragmentation_in_percent value	Corrective statement
> 5% and < = 30%	ALTER INDEX REORGANIZE
> 30%	ALTER INDEX REBUILD

Example

- Rebuilding online all indexes on table Production.Product

```
USE AdventureWorks;
GO
ALTER INDEX ALL ON Production.Product
REBUILD WITH (ONLINE = ON);
```

Dropping Indexes

- When a clustered index is dropped, the data rows that were stored in the leaf level of the clustered index are stored in an unordered table (heap).
- Dropping a clustered index can take time because all nonclustered indexes on the table must be rebuilt to replace the clustered index keys with row pointers to the heap.
- When dropping all indexes on a table, drop the nonclustered indexes first and the clustered index last.

Dropping Indexes Example

The following example drops the index IX_ProductVendor_VendorID in the ProductVendor table.

```
USE AdventureWorks;
GO
DROP INDEX IX_ProductVendor_VendorID
    ON Purchasing.ProductVendor;
GO
```

Dropping Indexes Example

- The following example drops a clustered index with the ONLINE option set to ON. The resulting unordered table (heap) is stored in the same filegroup as the index was stored.

```
USE AdventureWorks;
GO
DROP INDEX
    AK_BillOfMaterials_ProductAssemblyID_ComponentID_StartDate
    ON Production.BillOfMaterials WITH (ONLINE = ON);
GO
```

Statistics

- Statistical information can be created regarding the distribution of values in a column.
- The query optimizer uses this statistical information to determine the optimal query plan by estimating the cost of using an index to evaluate the query.
- A statistics on a column consist of an *histogram* dividing the values in the column in up to 200 intervals.
- The histogram specifies how many rows exactly match each interval value, how many rows fall within an interval, and a calculation of the density of values, or the incidence of duplicate values, within an interval.

Statistics

- Statistics can be created in three ways:
  - Automatically by creating an index
  - Automatically when a column is used in a predicate when the AUTO_CREATE_STATISTICS database option is set to ON (default),
  - Explicitly with CREATE STATISTICS

Visualizing Statistics

DBCC SHOW_STATISTICS **(** {'*table_name*' | '*view_name*'}**,** *target* **)**

[WITH < option > [**,** *n*]]

<option > :: = STAT_HEADER | DENSITY_VECTOR | HISTOGRAM

target is the name of the object (index name, statistics name or column name) for which to display statistics information.

STAT_HEADER | DENSITY_VECTOR | HISTOGRAM

- Specifying one or more of these options limits the result sets returned by the statement.

Visualing Statistics

- Alternatively, in Management Studio right click on a statistic and Properties

Example

DBCC SHOW_STATISTICS ('Person.Address',
IX_Address_StateProvinceID);

Returns 3 tables:

- STAT_HEADER
- DENSITY_VECTOR
- HISTOGRAM

Columns of STAT_HEADER

Column name	Description
Name	Name of the statistic.
Updated	Date and time the statistics were last updated.
Rows	Number of rows in the table.
Rows Sampled	Number of rows sampled for statistics information.
Steps	Number of distribution steps.
Density	Selectivity of the first index column prefix excluding the **EQ_ROWS**, which are described in the section about the HISTOGRAM option result set.
Average key length	Average length of all the index columns.
String Index	Yes indicates that the statistics contain a string summary index to support estimation of result set sizes for LIKE conditions. Applies only to leading columns of **char**, **varchar**, **nchar**, and **nvarchar**, **varchar(max)**, **nvarchar(max)**, **text**, and **ntext** data types.

Columns of HISTOGRAM

Column name	Description
RANGE_HI_KEY	Upper bound value of a histogram step.
RANGE_ROWS	Estimated number of rows from the table that fall within a histogram step, excluding the upper bound.
EQ_ROWS	Estimated number of rows from the table that are equal in value to the upper bound of the histogram step.
DISTINCT_RANGE_ROWS	Estimated number of distinct values within a histogram step, excluding the upper bound.
AVG_RANGE_ROWS	Average number of duplicate values within a histogram step, excluding the upper bound (RANGE_ROWS / DISTINCT_RANGE_ROWS).

Example of HISTOGRAM

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	0	25	0	1
3	0	7	0	1
6	0	18	0	1
7	0	1579	0	1
8	0	231	0	1
9	0	4564	0	1
10	0	11	0	1
11	0	9	0	1
14	0	1954	0	1
15	0	31	0	1
17	0	17	0	1
.....				

Statistics Update

- As the data in a column changes, statistics can become out-of-date and cause the query optimizer to make less-than-optimal decisions on how to process a query.
- Out-of-date or missing statistics are indicated as warnings (table name in red text) when the execution plan of a query is graphically displayed using SQL Server Management Studio

Statistics Update

- When the AUTO_UPDATE_STATISTICS database option is set to ON (the default), the query optimizer automatically updates this statistical information periodically as the data in the tables changes.
  - Unless the statistic has been created with an index for which the STATISTICS_NORECOMPUTE option was specified in the CREATE INDEX statement
- Almost always, statistical information is updated when approximately 20 percent of the data rows has changed.

UPDATE STATISTICS

- You can manually update the statistics with

UPDATE STATISTICS *table* | *view*
[
  { { *index* | *statistics_name* }
  | **(** { *index* |*statistics_name* } [**,**...*n*] **)** }
]
[WITH
  [FULLSCAN
  | SAMPLE *number* { PERCENT | ROWS }
]

UPDATE STATISTICS

- The FULLSCAN clause specifies that all data in the table is scanned to gather statistics,
- The SAMPLE clause can be used to specify either the percentage of rows to sample or the number of rows to sample.

Physical Design

- In order to chose the physical structures to use (clustered indexes, nonclustered indexes, indexed views, partitions) we can use the Database Engine Tuning Advisor
- Two interfaces are available
  - A standalone graphical user interface tool
  - A command-line utility program, **dta.exe**, for Database Engine Tuning Advisor functionality in software programs and scripts.

Database Engine Tuning Advisor

- It analyzes the performance effects of *workloads* run against one or more databases.
- A workload is a set of Transact-SQL statements that executes against databases you want to tune.
- After analyzing the effects of a workload on your databases, Database Engine Tuning Advisor provides recommendations to add, remove, or modify physical design structures in databases in order to reduce the execution time of the workload

Workloads

- A workload consists of a Transact-SQL script (.sql), a SQL Server Profiler trace saved to a file (.trc) or table or an XML file (.xml) containing the statements plus configuration information

Recommendations

- A recommendation consists of Transact-SQL statements
- After Database Engine Tuning Advisor has suggested a recommendation, you can optionally:
  - Implement it immediately.
  - Save it to a Transact-SQL script and implement it later
  - Modify it so to apply only a subset of recommendations

Reports

- Database Engine Tuning Advisor also returns a number of reports
  - Improvement in percent
  - Use of existing physical structures
  - …

Exploratory Analysis

- The user can also investigate the impact on the execution time of hypothetical structures
- Example:
  - An administrator just finished using Database Engine Tuning Advisor to tune a database and received the recommendation (R).
  - After reviewing R, the administrator would like to fine tune R by modifying it.
  - After modifying R, the administrator uses the modified recommendation as input to Database Engine Tuning Advisor and tunes again to measure the performance impact of her modifications.

Tuning Load

- Tuning a large workload can create significant overhead on the server that is being tuned.
- The overhead results from the many calls made by Database Engine Tuning Advisor to the query optimizer during the tuning process.
- We can eliminate this overhead problem by using a **test server** in addition to the **production server**.
- **Production server**: the server used by clients in day to day real operations
- **Test server**: a server used by developers for testing new database configurations

Test Server

- Database Engine Tuning Advisor creates a shell database on the test server. To create this shell database and tune it, Database Engine Tuning Advisor extracts from the production server the following information:
  - metadata on the production database. This metadata includes empty tables, indexes, views, stored procedures, triggers, and so on.
  - statistics
  - hardware parameters specifying the number of processors and available memory on the production server

Tuning

- After Database Engine Tuning Advisor finishes tuning the test server shell database, it generates a tuning recommendation.
- You can apply the recommendation received from tuning the test server to the production server.

Showing the Execution Plan of a Query

SET SHOWPLAN_XML ON

This statement causes SQL Server not to execute Transact-SQL statements. Instead, Microsoft SQL Server returns execution plan information about how the statements are going to be executed in a well-formed XML document.

SET SHOWPLAN_TEXT ON

After this SET statement is executed, SQL Server returns the execution plan information for each query in text. The Transact-SQL statements or batches are not executed.

Showing the Execution Plan of a Query

- SET SHOWPLAN_ALL ON
  This statement is similar to SET SHOWPLAN_TEXT, except that the output is in a format more verbose than that of SHOWPLAN_TEXT