Instructor Notes:

Instructor Notes:

Lesson Objectives

- In this lesson, you will learn:
  - Creating Indexes
  - Querying the sysindexes Table
  - Performance Considerations
  - Creating Views

Instructor Notes:

Index – An Overview

- Database systems generally use indexes to provide fast access to relational data
- An index is a separate physical data structure that enables queries to access one or more data rows fast
- This structure is known as B-Tree Structure
- Proper tuning of index is therefore a key for query performance
- Database Engine uses index to find the data just like one uses index in a book
- When a table is dropped , indexes also get dropped automatically
- Only the owner of the table can create indexes
- SQL Server supports two types of indexes
  - Clustered
  - Non clustered

Index – An Overview

Database systems generally use indexes to provide fast access to relational data. An index is a separate physical data structure that enables queries to access one or more data rows faster. This structure is known as B-Tree Structure. B-Tree helps the SQL Server find the row or rows associated with the key values.Proper tuning of indexes is therefore a key for query performance.

An index is in many ways similar to a book index.
When you are looking for a particular topic in a book, you use its index to find the page where that topic is described. Similarly, when you search for a row of a table, Database Engine uses an index to find its physical location.

However, there are two main differences between a book index and a database index:

- As a book reader, you have a choice to decide whether or not to use the book's index. This possibility generally does not exist if you use a database system
- A particular book's index is edited together with the book and does not change at all. This means that you can find a topic exactly on the page where it is determined in the index. In contrast, a database index can change each time the corresponding data is changed.

Instructor Notes:

How SQL Server access data?

- SQL Server accesses data in one of two ways:
- By scanning all the data pages in a table, which is called a table scan. When SQL Server performs a table scan, it:
  - Starts at the beginning of the table
  - Scans from page to page through all the rows in the table
  - Extracts the rows that meet the criteria of the query
- By using indexes. When SQL Server uses an index, it:
  - Traverses the index tree structure to find rows that the query requests
  - Extracts only the needed rows that meet the criteria of the query

How SQL Server access data?
If a table does not have an appropriate index, the database system uses the table scan method to retrieve rows. Table scan means that each row is retrieved and examined in sequence (from first to last) and returned in the result set if the search condition in the WHERE clause evaluates to true. Therefore, all rows are fetched according to their physical memory location. This method is less efficient than an index access, as explained next.

SQL Server first determines whether an index exists. Then the query optimizer—the component responsible for generating the optimal execution plan for a query - determines whether scanning a table or using the index is more efficient for accessing data.

Instructor Notes:

Clustered Index

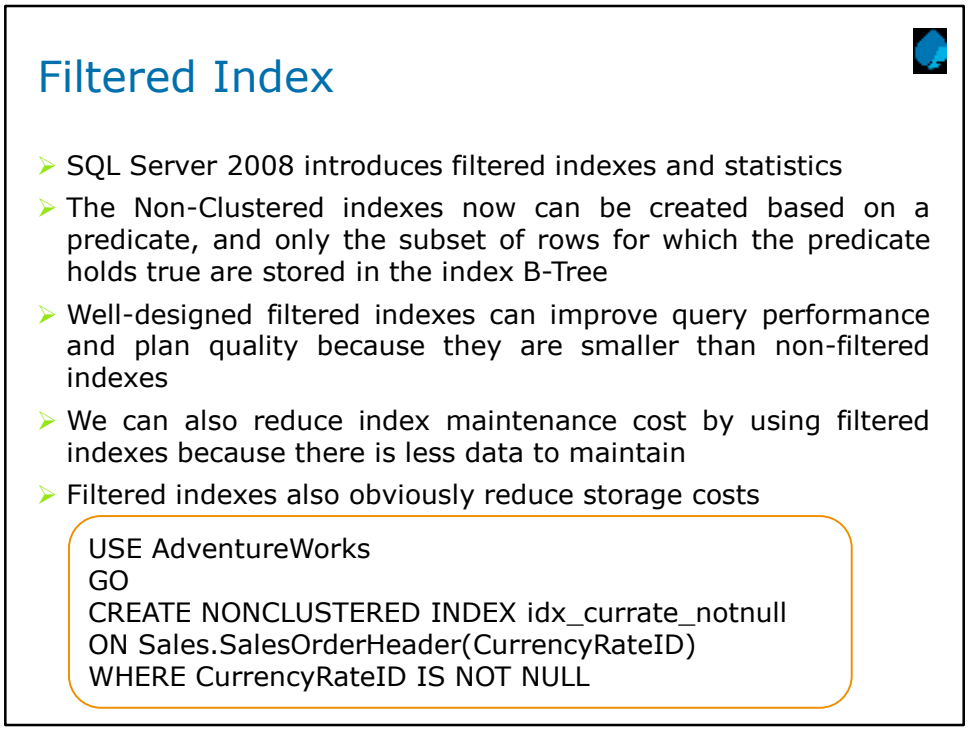
- A clustered index determines the physical order of the data in a table
- Database Engine allows the creation of a single clustered index per table
- If a clustered index is defined for a table, the table is called a clustered table
- A Unique Clustered index is built by default for each table, for which you define the primary key using the primary key constraint
- Also, each clustered index is unique by default that is, each data value can appear only once in a column for which the clustered index is defined

Instructor Notes:

Non-Clustered Index

- A Non-Clustered index has the same index structure as a clustered index
- A Non-Clustered index does not change the physical order of the rows in the table
- A table can have more than non clustered index
- Unique Non-Clustered index will be created automatically when you create unique key on a column to enforce uniqueness of key value

Instructor Notes:



Filtered Indexes

SQL Server 2008 introduces filtered indexes. You can now create a nonclustered index based on a predicate, and only the subset of rows for which the predicate holds true are stored in the index B-Tree. Well-designed filtered indexes can improve query performance and plan quality because they are smaller than nonfiltered indexes.

You can also reduce index maintenance cost by using filtered indexes because there is less data to maintain. This includes modifications against the index, index rebuilds, and the cost of updating statistics. Filtered indexes also obviously reduce storage costs.

Points to remember when creating Filtered Index

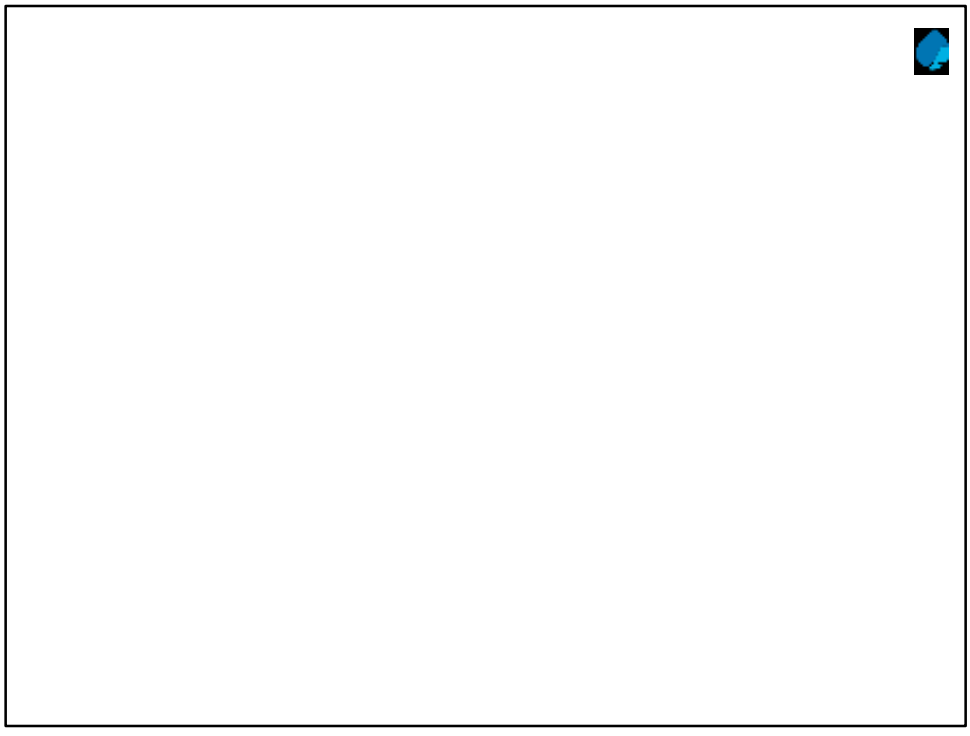
- They can be created only as Nonclustered Index
- They can be used on Views only if they are persisted views.
- They cannot be created on full-text Indexes.

Let's look at a few examples that demonstrate filtered indexes.

The following code creates an index on the CurrencyRateID column in the Sales.SalesOrderHeader table, with a filter that excludes NULLs:

```
USE AdventureWorks;
GO
CREATE NONCLUSTERED INDEX idx_currate_notnull
ON Sales.SalesOrderHeader(CurrencyRateID)
WHERE CurrencyRateID IS NOT NULL;
```

Instructor Notes:



Considering query filters, besides the IS NULL predicate that explicitly looks for NULLs, all other predicates exclude NULLs, so the optimizer knows that there is the potential to use the index.

```
SELECT *
FROM Sales.SalesOrderHeader
WHERE CurrencyRateID = 4;
```

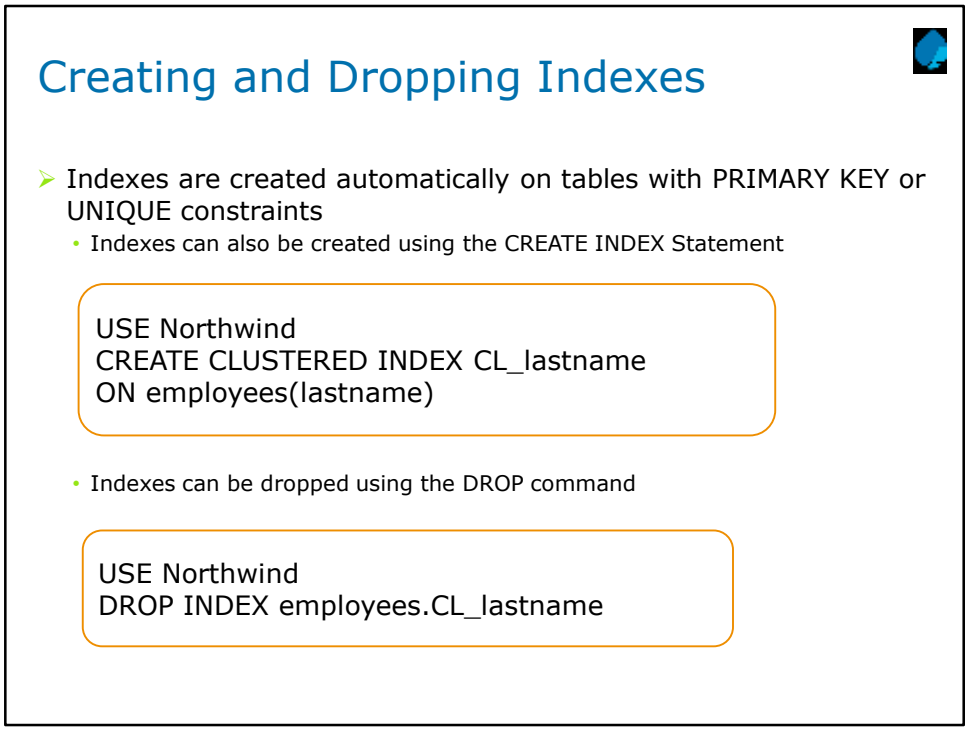
The CurrencyRateID column has a large percentage of NULLs; therefore, this index consumes substantially less storage than a nonfiltered one on the same column. You can also create similar indexes on sparse columns.
The following code creates a nonclustered index on the Freight column, filtering rows where the Freight is greater than or equal to 5000.00:

```
CREATE NONCLUSTERED INDEX idx_freight_5000_or_more
ON Sales.SalesOrderHeader(Freight)
WHERE Freight >= 5000.00;

SELECT *
FROM Sales.SalesOrderHeader
WHERE Freight BETWEEN 5500.00 AND 6000.00;
```

Filtered indexes can also be defined as UNIQUE and have an INCLUDE clause as with regular nonclustered indexes.

Instructor Notes:



Creating and dropping Indexes
Indexes are created in the following ways:

By defining a PRIMARY KEY or UNIQUE constraint on a column by using CREATE TABLE or ALTER TABLE

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 unique 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.

By creating an index independent of a constraint by using the CREATE INDEX statement, or **New Index** dialog box in SQL Server Management Studio Object Explorer

You must specify the name of the index, table, and columns to which the index applies. Index options and index location, filegroup or partition scheme, can also be specified. By default, a nonclustered, nonunique index is created if the clustered or unique options are not specified.

Instructor Notes:

Creating and Dropping Indexes

- To create non clustered index ncl_deptno

```
USE Northwind
    CREATE NON CLUSTERED INDEX NCL_deptno
    ON employees(deptno)
```

- Using the DROP INDEX Statement

```
USE Northwind
DROP INDEX employees.NCL_deptno
```

Instructor Notes:

Creating Unique Indexes

- Unique index can be non clustered or clustered
- Unique non clustered index is automatically created when a column has UNIQUE constraint
- Unique Clustered index is automatically created when column has a PRIMARY KEY constraint
- Ensures column(s) have unique value
- There is no difference in the way Unique constraint and Unique index work, except for syntax

```
USE Northwind
CREATE UNIQUE NONCLUSTERED INDEX U_CustID
ON customers(CustomerID)
```

Creating a unique index guarantees that any attempt to duplicate key values fails. 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.
Unique indexes are implemented in the following ways:
PRIMARY KEY or UNIQUE constraint

When you create a PRIMARY KEY constraint, a unique clustered index on the column or columns is automatically created if a clustered index on the table does not already exist and you do not specify a unique nonclustered index. The primary key column cannot allow NULL values.

You can specify a unique clustered index if a clustered index on the table does not already exist.
Index independent of a constraint

Multiple unique nonclustered indexes can be defined on a table.

Instructor Notes:

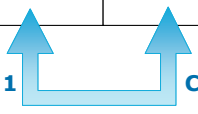
Creating Composite Indexes

- Index of two /more columns are said to be composite

```
USE Northwind
CREATE UNIQUE NONCLUSTERED INDEX
U_OrdID_ProdID
ON [Order Details] (OrderID, ProductID)
```

Order Details

OrderID	ProductID	UnitPrice	Quantity	Discount
10248	11	14.000	12	0.0
10248	42	9.800	10	0.0
10248	72	34.800	5	0.0



Column 1 **Column 2** **Composite Key**

Instructor Notes:

Columnstore Indexes

- SQL Server 2012 introduces ColumnStore Indexes.

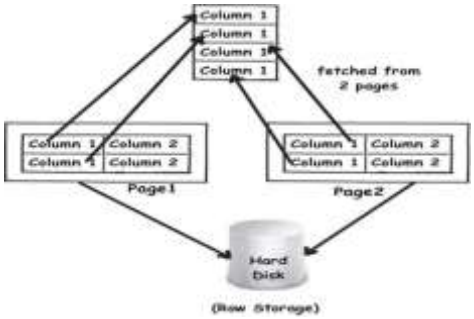
Benefits of using SQL Server ColumnStore Indexes

- Faster query performance for common data warehouse queries as only required columns/pages in the query are fetched from disk
- Data is stored in a highly compressed form to reduce the storage space
- Frequently accessed columns (pages that contains data for these columns) remain in memory because a high ratio of compression is used in the pages and less pages are involved

Instructor Notes:

Columnstore Indexes

- Relational database store data "row wise". These rows are further stored in 8 KB page size.
- For instance you can see in the below figure we have table with two columns "Column1" and "Column2". You can see how the data is stored in two pages i.e. "page1" and "page2". "Page1" has two rows and "page2" also has two rows.



Instructor Notes:

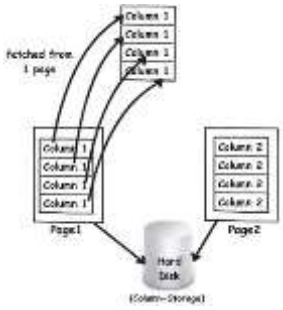
Columnstore Indexes

- If you want to fetch only "column1", you have to pull records from two pages i.e. "Page1" and "Page2".
- As we have to fetch data from two pages its bit performance intensive.

Instructor Notes:

Columnstore Indexes

- If somehow we can store data column wise we can avoid fetching data from multiple pages.
- That's what column store indexes do.
- When you create a column store index it stores same column data in the same page. You can see from the diagram (shown below), we now need to fetch "column1" data only from one page rather than querying multiple pages.



Instructor Notes:

Limitations of SQL Server ColumnStore Indexes

There are several limitations of using SQL Server ColumnStore indexes over Row Store indexes including:

- A table with a ColumnStore Index cannot be updated
- ColumnStore index creation takes more time (1.5 times almost) than creating a B-tree index (on same set of columns) because the data is compressed
- A table can have only one ColumnStore Index and hence you should consider including all columns or at least all those frequently used columns of the table in the index
- A ColumnStore Index can only be non cluster and non unique index; you cannot specify ASC/DESC or INCLUDE clauses
- Not all data types (binary, varbinary, image, text, ntext, varchar(max), nvarchar(max), etc.) are supported

Instructor Notes:

Limitations of SQL Server ColumnStore Indexes (Contd…)

- The definition of a ColumnStore Index cannot be changed with the ALTER INDEX command, you need to drop and create the index or disable it then rebuild it
- You can create a ColumnStore index on a table which has compression enabled, but you cannot specify the compression setting for the column store index
- A ColumnStore Index cannot be created on view
- A ColumnStore Index cannot be created on table which uses features like Replication, Change Tracking, Change Data Capture and Filestream

Instructor Notes:

Columnstore Indexes

```
CREATE NONCLUSTERED COLUMNSTORE INDEX
idx_colSale
ON myTable (OrderDate, ProductID, SaleAmount)
```

Instructor Notes:

Obtaining information on Indexes

- Using the sp_helpindex System Stored Procedure

```
USE Northwind
EXEC sp_helpindex Customers
```

- Using the sp_help tablename System Stored Procedure

Instructor Notes:

Indexes – Performance Considerations

- Create indexes on foreign keys
- Create the clustered index before nonclustered indexes
- Consider before creating composite indexes
- Create multiple indexes for a table that is read frequently
- Use the index tuning wizard get statics of index usage

Instructor Notes:

Demo

- Creating Indexes

Instructor Notes:

Views – An Overview

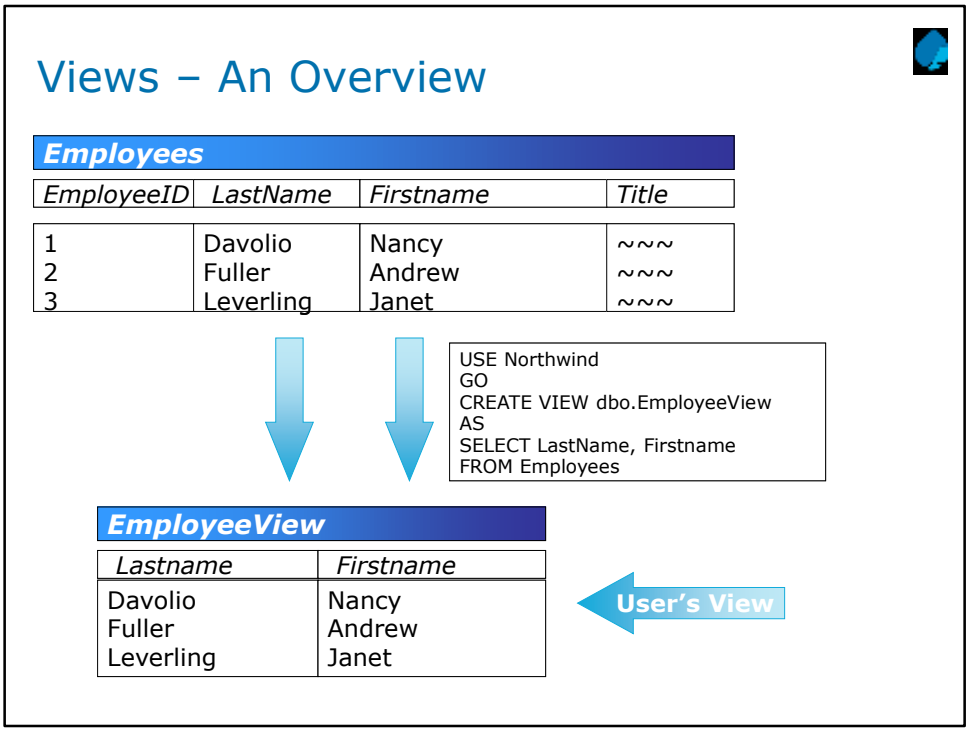
- Views are Virtual tables, which provides access to a subset of columns from one or more tables
- Created from one or more base tables or other views
- Internally Views are stored queries
- Views are created when
  - To hide the complexity of the underlying database schema, or customize the data and schema for a set of users.
  - To control access to rows and columns of data.
- Objective of creating views is Abstraction , not performance

A view is a virtual table whose contents are defined by a query. Like a real table, a view consists of a set of named columns and rows of data., a Views does not exist with a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.

A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases. Distributed queries can also be used to define views that use data from multiple heterogeneous sources. This is useful, for example, if you want to combine similarly structured data from different servers, each of which stores data for a different region of your organization.

There are no restrictions on querying through views and few restrictions on modifying data through them.

Instructor Notes:



Instructor Notes:

Views – Advantages

- Focus the Data for Users
  - Focus on important or appropriate data only
  - Limit access to sensitive data
- Mask Database Complexity
  - Hide complex database design
  - Simplify complex queries, including distributed queries to heterogeneous data
- Simplify Management of User Access on Data

Instructor Notes:

Views – Types

- Standard Views
- Indexed Views
- Partitioned Views

In SQL Server, you can create standard views, indexed views, and partitioned views.

Standard Views

Combining data from one or more tables through a standard view lets you satisfy most of the benefits of using views. These include focusing on specific data and simplifying data manipulation. These benefits are described in more detail in Scenarios for Using Views.

Indexed Views

An indexed view 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.

Partitioned Views

A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers. This makes the data appear as if from one table. A view that joins member tables on the same instance of SQL Server is a local partitioned view.

When a view joins data from tables across servers, it is a distributed partitioned view. Distributed partitioned views are used to implement a federation of database servers. A federation is a group of servers administered independently, but which cooperate to share the processing load of a system.

Instructor Notes:

Defining Views

- Creating views
- Altering and dropping views
- Locating view definition information
- Hiding view definitions

Instructor Notes:

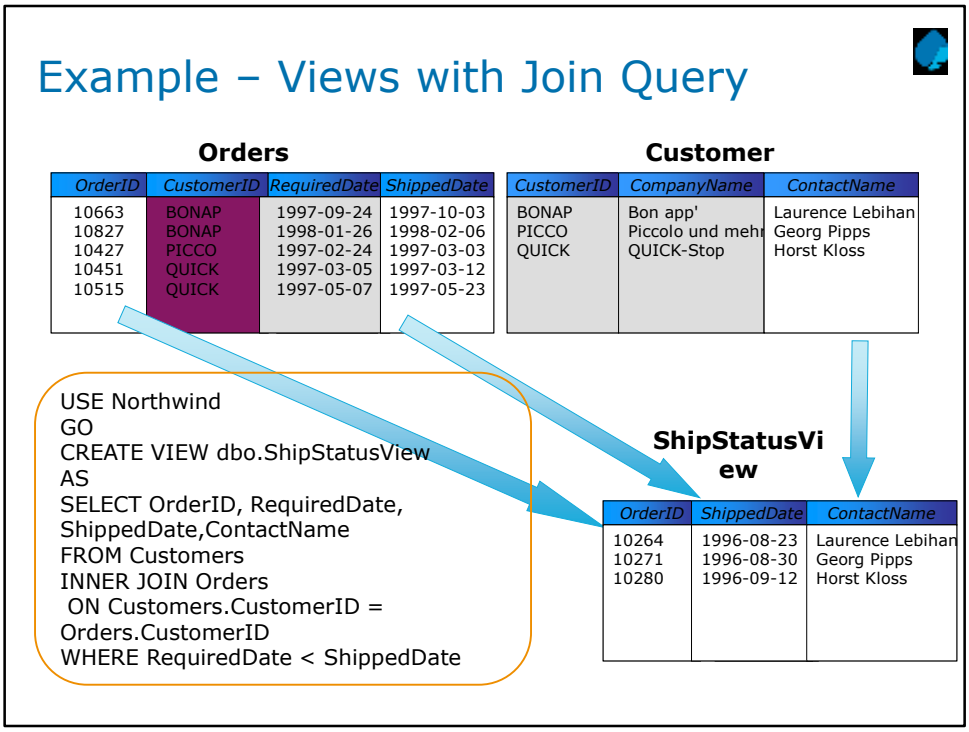
Creating Views

- Creating a View

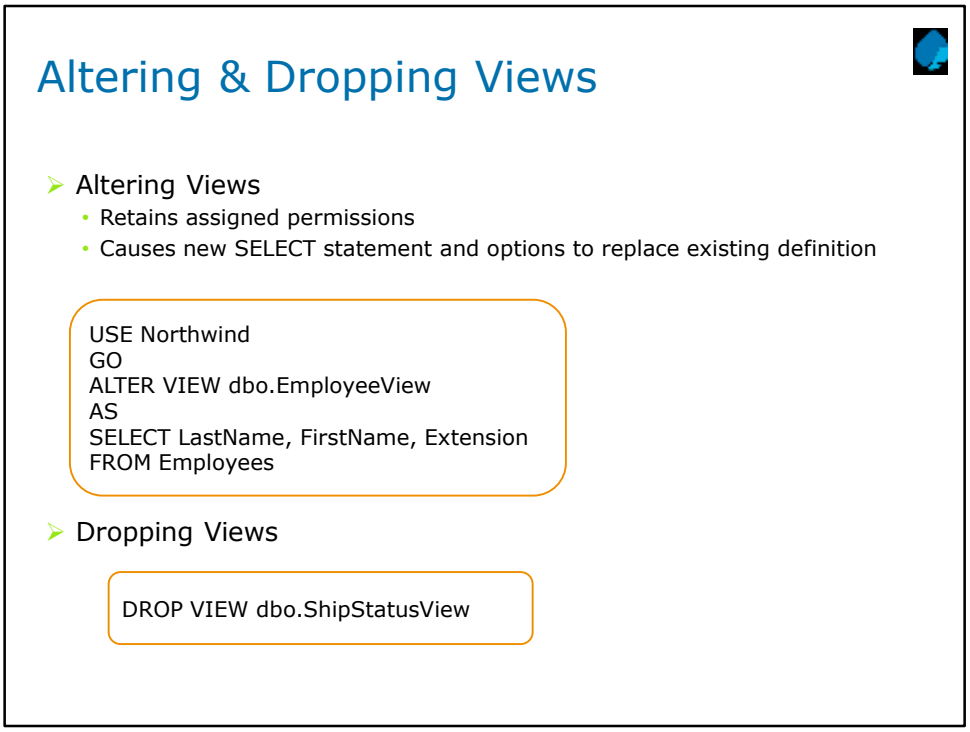
```
CREATE VIEW dbo.OrderSubtotalsView (OrderID, Subtotal)
AS
SELECT OD.OrderID,
 SUM(CONVERT(money,(OD.UnitPrice*Quantity*(1-
Discount)/100))*100)
FROM [Order Details] OD
GROUP BY OD.OrderID
GO
```

- Restrictions on View Definitions
  - Cannot include ORDER BY clause
  - Cannot include INTO keyword

Instructor Notes:



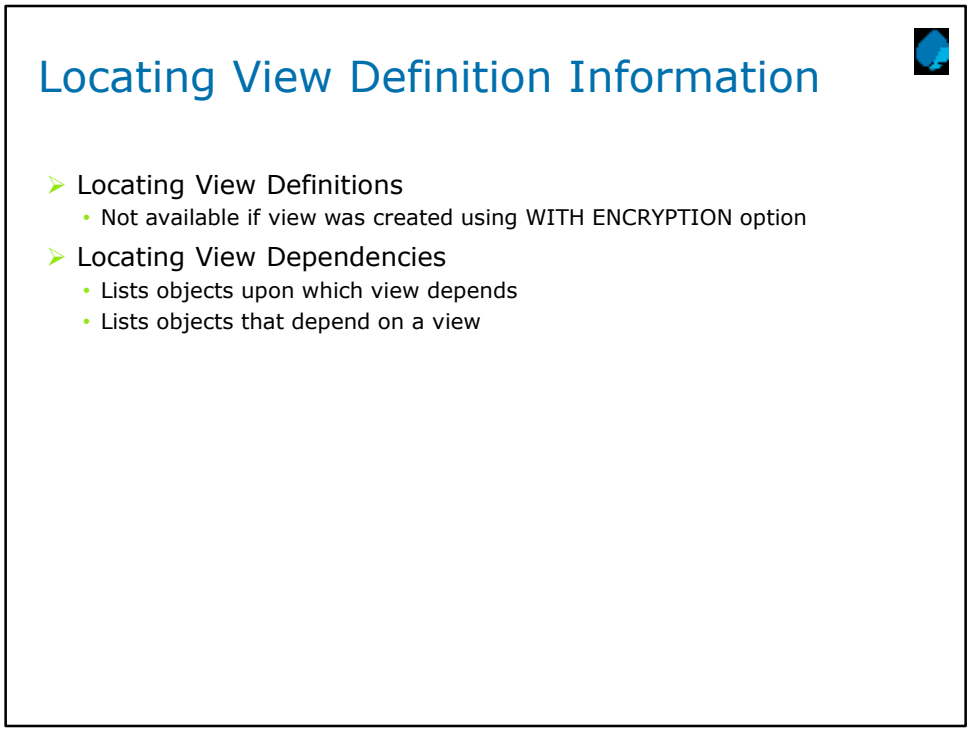
Instructor Notes:



Modifies a previously created view. This includes an indexed view. ALTER VIEW does not affect dependent stored procedures or triggers and does not change permissions.
ALTER VIEW [schema_name .] view_name [(column [,...n])] [WITH <view_attribute> [,...n]] AS select_statement [WITH CHECK OPTION] [;]
If a view currently used is modified by using ALTER VIEW, the Database Engine takes an exclusive schema lock on the view. When the lock is granted, and there are no active users of the view, the Database Engine deletes all copies of the view from the procedure cache. Existing plans referencing the view remain in the cache but are recompiled when invoked.
ALTER VIEW can be applied to indexed views; however, ALTER VIEW unconditionally drops all indexes on the view.
Removes one or more views from the current database. DROP VIEW can be executed against indexed views.
DROP VIEW [schema_name .] view_name [...,n] [;]

Note : when base table are dropped , views dependent on them DON'T get dropped

Instructor Notes:



ENCRYPTION
Encrypts the entries in sys.syscomments that contain the text of the CREATE VIEW statement. Using WITH ENCRYPTION prevents the view from being published as part of SQL Server replication

Dependencies:
Contains a row for each dependency on a referenced (independent) entity as referenced in the SQL expression or statements that define some other referencing (dependent) object. The **sys.sql_dependencies** view is meant to track by-name dependencies between entities. For each row in **sys.sql_dependencies**, the referenced entity appears by-name in a persisted SQL expression of the referencing object.

Instructor Notes:

Hiding View Definition

- Use the WITH ENCRYPTION Option
- Do not delete entries in the syscomments table

```
USE Northwind
GO
CREATE VIEW dbo.[Order Subtotals]
   WITH ENCRYPTION
AS
SELECT OrderID,
 Sum(CONVERT(money, (UnitPrice * Quantity * (1 - Discount) /
100)) * 100) AS Subtotal
FROM [Order Details]
GROUP BY OrderID
GO
```

Instructor Notes:

Modifying Data through View

- Cannot affect more than one underlying table
- Cannot be made to columns having aggregation
- Depends on the constraints placed on the base tables
- Are verified if the WITH CHECK OPTION has been specified

You can modify the data of an underlying base table through a view, in the same manner as you modify data in a table by using UPDATE, INSERT and DELETE statements or by using the **bcp** utility and BULK INSERT statement. However, the following restrictions apply to updating views, but do not apply to tables:

- Any modifications, including UPDATE, INSERT, and DELETE statements, must reference columns from only one base table.
- The columns that are being modified in the view must reference the underlying data in the table columns directly. They cannot be derived in any other way, such as through:
    - An aggregate function (AVG, COUNT, SUM, MIN, MAX, GROUPING, STDEV, STDEVP, VAR and VARP).
    - A computation; the column cannot be computed from an expression using other columns. Columns formed using set operators (UNION, UNION ALL, CROSSJOIN, EXCEPT, and INTERSECT) amount to a computation and are also not updatable
- The columns that are being modified cannot be affected by GROUP BY, HAVING, or DISTINCT clauses.
- TOP cannot be used anywhere in the *select_statement* of the view when WITH CHECK OPTION is also specified.

Instructor Notes:

Views – Recommended Practices

- Use a Standard Naming Convention
- dbo Should Own All Views
- Verify Object Dependencies Before You Drop Objects
- Never Delete Entries in the syscomments Table
- Carefully Evaluate Creating Views Based on Views

Instructor Notes:

Demo

- Working with Views

Instructor Notes:

Summary

- In this lesson, you have learnt:
  - Creating Indexes
  - Types of indexes
    - Clustered Index, Non clustered index ,Filtered Indexes ,Column store Indexes
  - Creating and modifying Views

Instructor Notes:

Answers for the Review Questions:

Answer 1: Unique clustered index

Answer 2: With Encryption

Answer 3: True

Review Question

- Question 1: -------- Gets created automatically for Primary key constrain
  - clustered index
  - Unique clustered index
  - Unique Non clustered index
- Question 2: -------- option with views will not stored base query of views in syscomments table
- Question 3: A table can have multiple unique non clustered index
  - True/False