In SQL Server, a *view* is a pre-written query that is stored on the database. A view consists of a SELECT statement, and when you run the view, you see the results of it like you would when opening a table. Some people like to think of a view as a virtual table. This is because a view can pull together data from multiple tables, as well as aggregate data, and present it as though it is a single table.

**Benefits of Views**

A view can be useful when there are multiple users with different levels of access, who all need to see portions of the data in the database (but not necessarily all of the data). Views can do the following:

* Restrict access to specific rows in a table
* Restrict access to specific columns in a table
* Join columns from multiple tables and present them as though they are part of a single table
* Present aggregate information (such as the results of the COUNT function)

**Creating a View::**

**CREATE VIEW "Alphabetical list of products" AS**

**SELECT Products.\*, Categories.CategoryName**

**FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID**

**WHERE (((Products.Discontinued)=0))**

|  |
| --- |
|  |

Modifying views:

ALTER VIEW "Alphabetical list of products" AS

SELECT Products.\*, Categories.CategoryName

FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID

WHERE (((Products.Discontinued)=0))

**Running a View**

SELECT \* FROM "Alphabetical list of products"

create view view\_Customer  
as  
select \* from customer join order on customer.customerId=order.customerId

examples::::

**create**table Billings (  
9>     BankerID           INTEGER,  
10>     BillingNumber      INTEGER,  
11>     BillingDate        datetime,  
12>     BillingTotal       INTEGER,  
13>     TermsID            INTEGER,  
14>     BillingDueDate     datetime ,  
15>     PaymentTotal       INTEGER,  
16>     CreditTotal        INTEGER  
17>  
18> );  
19> GO  
1>  
2> **INSERT**INTO Billings VALUES (1, 1, '2005-01-22', 165, 1,'2005-04-22',123,321);  
3> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (2, 2, '2001-02-21', 165, 1,'2002-02-22',123,321.);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (3, 3, '2003-05-02', 165, 1,'2005-04-12',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (4, 4, '1999-03-12', 165, 1,'2005-04-18',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (5, 5, '2000-04-23', 165, 1,'2005-04-17',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (6, 6, '2001-06-14', 165, 1,'2005-04-18',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (7, 7, '2002-07-15', 165, 1,'2005-04-19',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (8, 8, '2003-08-16', 165, 1,'2005-04-20',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (9, 9, '2004-09-17', 165, 1,'2005-04-21',123,321);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO Billings VALUES (0, 0, '2005-10-18', 165, 1,'2005-04-22',123,321);  
2> GO  
  
(1 rows affected)  
1>  
2>  
3> **create**table Bankers(  
4>    BankerID             Integer,  
5>    BankerName           VARCHAR(20),  
6>    BankerContactLName   VARCHAR(20),  
7>    BankerContactFName   VARCHAR(20),  
8>    BankerCity           VARCHAR(20),  
9>    BankerState          VARCHAR(20),  
10>    BankerZipCode        VARCHAR(20),  
11>    BankerPhone          VARCHAR(20)  
12> )  
13> GO  
1>  
2> **insert**into Bankers values (1, 'ABC Inc.','Joe','Smith','Vancouver','BC','11111','111-111-1111');  
3> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (2, 'DEF Inc.','Red','Rice', 'New York', 'DE','22222','222-222-2222');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (3, 'HJI Inc.','Kit','Cat',  'Paris',    'CA','33333','333-333-3333');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (4, 'QWE Inc.','Git','Black','Regina',   'ER','44444','444-444-4444');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (5, 'RTY Inc.','Wil','Lee',  'Toronto',  'YU','55555','555-555-5555');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (6, 'YUI Inc.','Ted','Larry','Calgary',  'TY','66666','666-666-6666');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (7, 'OIP Inc.','Yam','Act',  'San Franc','FG','77777','777-777-7777');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (8, 'SAD Inc.','Hit','Eat',  'Orland',   'PO','88888','888-888-8888');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (9, 'DFG Inc.','Sad','Lee',  'Wisler',   'PL','99999','999-999-9999');  
2> GO  
  
(1 rows affected)  
1> **insert**into Bankers values (0, 'GHJ Inc.','Bit','Lee',  'Ticker',   'MN','00000','000-000-0000');  
2> GO  
  
(1 rows affected)  
1>  
2>  
3>  
4> **CREATE**VIEW IBM\_Billings  
5> AS  
6> **SELECT**BillingNumber, BillingDate, BillingTotal  
7> **FROM**Billings  
8> **WHERE**BankerID = (**SELECT**BankerID **FROM**Bankers **WHERE**BankerName = 'IBM')  
9> GO  
1>  
4> **INSERT**INTO IBM\_Billings (BillingNumber, BillingDate, BillingTotal)  
5> VALUES ('8', '2002-07-31', 417)  
6> GO  
  
(1 rows affected)  
1>  
2>  
3> drop view IBM\_Billings;  
4> GO  
1>  
2> drop table Bankers;  
3> GO  
1>  
2>  
3> drop table Billings;  
4> GO

UPDATEABLE VIEW

**CREATE**TABLE employee(  
8>    id          INTEGER NOT NULL PRIMARY KEY,  
9>    first\_name  VARCHAR(10),  
10>    last\_name   VARCHAR(10),  
11>    salary      DECIMAL(10,2),  
12>    start\_Date  DATETIME,  
13>    region      VARCHAR(10),  
14>    city        VARCHAR(20),  
15>    managerid   INTEGER  
16> );  
17> GO  
1> **INSERT**INTO employee VALUES (1, 'Jason' ,  'Martin', 5890,'2005-03-22','North','Vancouver',3);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (2, 'Alison',  'Mathews',4789,'2003-07-21','South','Utown',4);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (3, 'James' ,  'Smith',  6678,'2001-12-01','North','Paris',5);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (4, 'Celia' ,  'Rice',   5567,'2006-03-03','South','London',6);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (5, 'Robert',  'Black',  4467,'2004-07-02','East','Newton',7);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (6, 'Linda' ,  'Green' , 6456,'2002-05-19','East','Calgary',8);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (7, 'David' ,  'Larry',  5345,'2008-03-18','West','New York',9);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (8, 'James' ,  'Cat',    4234,'2007-07-17','West','Regina',9);  
2> GO  
  
(1 rows affected)  
1> **INSERT**INTO employee VALUES (9, 'Joan'  ,  'Act',    6123,'2001-04-16','North','Toronto',10);  
2> GO  
  
(1 rows affected)  
1>  
2> **select**\* **from**employee;  
3> GO  
id          first\_name last\_name  salary       start\_Date              region     city                 managerid  
----------- ---------- ---------- ------------ ----------------------- ---------- -------------------- -----------  
          1 Jason      Martin          5890.00 2005-03-22 00:00:00.000 North      Vancouver                      3  
          2 Alison     Mathews         4789.00 2003-07-21 00:00:00.000 South      Utown                          4  
          3 James      Smith           6678.00 2001-12-01 00:00:00.000 North      Paris                          5  
          4 Celia      Rice            5567.00 2006-03-03 00:00:00.000 South      London                         6  
          5 Robert     Black           4467.00 2004-07-02 00:00:00.000 East       Newton                         7  
          6 Linda      Green           6456.00 2002-05-19 00:00:00.000 East       Calgary                        8  
          7 David      Larry           5345.00 2008-03-18 00:00:00.000 West       New York                       9  
          8 James      Cat             4234.00 2007-07-17 00:00:00.000 West       Regina                         9  
          9 Joan       Act             6123.00 2001-04-16 00:00:00.000 North      Toronto                       10  
  
(9 rows affected)  
1>  
2>  
3>  
4> Create VIEW vwEmployee  
5> **As**  
6> **SELECT**ID, First\_Name  
7> From Employee  
8> WITH CHECK OPTION  
9> GO  
1>  
2>  
3>  
4> drop view vwEmployee  
5> GO  
1>  
2> drop table employee;  
3> GO

**Protecting the Data WITH CHECK OPTION in Views**

**Views are designed to control access to data. As such, there are several options that protect the data or the view.Then the question arise how to Locking Down the View**

**The with check option causes the where clause of the view to check the data being inserted or updated through the view in addition to the data being retrieved. In a sense, it makes the where clause a two-way restriction.  
This option is useful when the view should limit inserts and updates with the same restrictions applied to the where clause.  
To understand the need for the with check option, it’s important to first understand how views function without the check option. The following view will generate a list of tours for the Cape Hatteras base camp:**

**ALTER VIEW dbo.tbl\_AjantaTour  
AS  
SELECT TourName, BaseCampID  
FROM dbo.Tour WHERE BaseCampID = 2  
SELECT \* FROM tbl\_AjantaTour**

|  |  |
| --- | --- |
| TourName | BaseCampID |
| Outer Banks Lighthouses | 2 |

**If the Ashville base camp adds a Blue Ridge Parkway Hike tour and inserts it through the view without the check option, the insert is permitted:**

**INSERT tbl\_AjantaTour(TourName, BaseCampID)  
VALUES (‘Blue Ridge Parkway Hike’, 1)  
(1 row(s) affected)**

**The insert worked and the new row is in the database, but the row is not visible through the view because the where clause of the view filters out the inserted row. This phenomenon is called disappearing rows:**

**SELECT \* FROM dbo.tbl\_AjantaTour**

|  |  |
| --- | --- |
| TourName | BaseCampID |
| Outer Banks Lighthouses | 2 |

**If the purpose of the view was to give users at the Cape access to their tours alone, then the view failed. Although they can see only the Cape’s tours, they successfully modified another base camp’s tours.  
The with check option would have prevented this fault. The following code will back out the insert and redo the same scenario, but this time the view will include the with check option:**

**DELETE dbo.tbl\_AjantaTour WHERE TourName = ‘Blue Ridge Parkway Hike’  
ALTER VIEW dbo.tbl\_AjantaTour  
AS  
SELECT TourName, BaseCampID FROM dbo.Tour   
WHERE BaseCampID = 2  
WITH CHECK OPTION  
INSERT dbo.tbl\_AjantaTour(TourName, BaseCampID)  
VALUES (‘Blue Ridge Parkway Hike’, 1)**

**Server: Msg 550, Level 16, State 1, Line 1  
The attempted insert or update failed because the target view either  
specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK  
OPTION and one or more rows resulting from the operation did not qualify  
under the CHECK OPTION constraint.  
The statement has been terminated.**

**CREATE**VIEW IBM\_Billings  
5> AS  
6> **SELECT**BillingNumber, BillingDate, BillingTotal  
7> **FROM**Billings  
8> **WHERE**BankerID = (**SELECT**BankerID **FROM**Bankers **WHERE**BankerName = 'IBM')  
9> GO  
1>  
4> **INSERT**INTO IBM\_Billings (BillingNumber, BillingDate, BillingTotal)  
5> VALUES ('8', '2002-07-31', 417)  
6> GO  
  
(1 rows affected)  
1>  
2>  
3> drop view IBM\_Billings;  
4> GO  
1>  
2> drop table Bankers;  
3> GO  
1>  
2>  
3> drop table Billings;  
4> GO

Altering Constraints:

select \* from emp9

alter table emp add doj datetime

update emp

set doj='12/12/12'

where eno='001'

alter table emp alter column eno int

create table emp90

(

eno varchar(5) not null,

ename varchar(10),

eadd varchar(10),

esal int)

insert into emp90 values(1,'aaaa','sdds',9000)

ALTER TABLE emp90

ADD CONSTRAINT pk\_emp90 PRIMARY KEY (Eno)

DISABLING CONSTRAINTS

ALTER TABLE employee

ADD CONSTRAINT pk\_employee PRIMARY KEY (EmployeeId)

ALTER TABLE employee

DROP CONSTRAINT pk\_employee

ALTER TABLE Persons  
ADD CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName)

ALTER TABLE Persons  
DROP CONSTRAINT pk\_PersonID

create table dept90

(

dno varchar(5),

dname varchar(10),

eno varchar(5),

dloc varchar(10))

drop table dept90

ALTER TABLE dept90

ADD CONSTRAINT fk\_PerOrders

FOREIGN KEY (eno)

REFERENCES emp90(eno)

ALTER TABLE Orders  
ADD CONSTRAINT fk\_PerOrders  
FOREIGN KEY (P\_Id)  
REFERENCES Persons(P\_Id)

ALTER TABLE Orders

DROP CONSTRAINT fk\_PerOrders

CREATE TABLE Persons  
(  
P\_Id int NOT NULL,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255),  
CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')  
)

ALTER TABLE Persons  
ADD CHECK (P\_Id>0)

ALTER TABLE Persons  
ADD CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')

ALTER TABLE Persons  
DROP CONSTRAINT chk\_Person

ALTER TABLE EMP95

ADD CONSTRAINT DD DEFAULT 'MUMBAI' FOR EADD

ALTER TABLE EMP95

DROP CONSTRAINT DD

To change the data type of a column in a table, use the following syntax:

ALTER TABLE table\_name  
ALTER COLUMN column\_name datatype

To disable constraints of a table

alter table em9 nocheck constraint all

to reenable constraints of a table

alter table em9 nocheck constraint all

alter table em9 check constraint all

alter table em9 check constraint all

**SQL CREATE INDEX Statement**

The CREATE INDEX statement is used to create indexes in tables.

Indexes allow the database application to find data fast; without reading the whole table.

**Indexes**

An index can be created in a table to find data more quickly and efficiently.

The users cannot see the indexes, they are just used to speed up searches/queries.

**Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So you should only create indexes on columns (and tables) that will be frequently searched against.

**SQL CREATE INDEX Syntax**

Creates an index on a table. Duplicate values are allowed:

CREATE INDEX index\_name  
ON table\_name (column\_name)

**SQL CREATE UNIQUE INDEX Syntax**

Creates a unique index on a table. Duplicate values are not allowed:

CREATE UNIQUE INDEX index\_name  
ON table\_name (column\_name)

**Note:** The syntax for creating indexes varies amongst different databases. Therefore: Check the syntax for creating indexes in your database.

**CREATE INDEX Example**

The SQL statement below creates an index named "PIndex" on the "LastName" column in the "Persons" table:

CREATE INDEX PIndex  
ON Persons (LastName)

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

CREATE INDEX PIndex  
ON Persons (LastName, FirstName)

**DROP INDEX Syntax for MS SQL Server:**

DROP INDEX table\_name.index\_name

USE AdventureWorks  
GO  
-- Disable the constraint  
ALTER TABLE HumanResources.Employee  
NOCHECK CONSTRAINT CK\_Employee\_BirthDate

GO  
-- Enable the constraint  
ALTER TABLE HumanResources.Employee  
WITH CHECK CHECK CONSTRAINT CK\_Employee\_BirthDate  
GO

FOR DISABLING ALL CONSTRAINTS IN A TABLE:

ALTER TABLE '+ @TblName + ' NOCHECK CONSTRAINT ALL'

###CREATE TABLE Books (

BookID INT NOT NULL PRIMARY KEY,

AuthorID INT NOT NULL,

BookName VARCHAR(100) NOT NULL,

Price MONEY NOT NULL

)

GO

CREATE TABLE Authors (

AuthorID INT NOT NULL PRIMARY KEY,

Name VARCHAR(100) NOT NULL

)

GO

ALTER TABLE Books

ADD CONSTRAINT fk\_author

FOREIGN KEY (AuthorID)

REFERENCES Authors (AuthorID) ON DELETE CASCADE

GO

ALTER TABLE Persons  
ADD PRIMARY KEY (P\_Id)

One of the important parts of SQL Server development and optimization is the creation of indexes. In order to create proper indexing strategies it is necessary to understand how indexes work. This tutorial will guide you step by step to understand some index basics.

There are only two different types of indexes. Clustered and NonClustered. There can only be one clustered index on a table and the reason is simple:

* A Clustered index is the data of table sorted according to the columns you choose.
* A NonClustered index is just like the index of a book. It contains data sorted so that it’s easy to find, then once found, it points back to the actual page that contains the data. (In other words, it points back to the clustered index)

Suppose we are reading a book about biographical information of all the U.S. Presidents, and the book itself orders the biographies starting from the first president to the latest president. This ordering of the pages would represent the clustered index.

Now suppose you asked two different people to find Franklin D. Roosevelt’s biography. Person-A was a historian and Person-B was unschooled. Person-A would quickly be able to find the presidents biography while Person-B would have to scan through each page in order to find the biography. Even if the Person-B used the book’s index (akin to the non-clustered index), he would still have to search for the page after he found the page number.

So it is always faster to find information off of the clustered index because the data in already at the “leaf-level” off the index.

http://sqlserverplanet.com/wp-content/themes/sqlserverplanet/images/ico9.gif

The clustered index should be a key that does not get modified. It should also ideally be sequential so that the underlying data pages do not become fragmented [[more information](http://sqlserverplanet.com/indexes/choosing-the-best-clustered-index)]

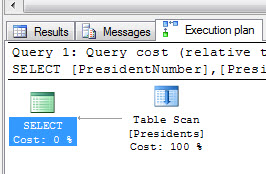
With this information, how do we determine what the clustered index should be? Well, it depends on the population of the people searching for the biographies. If it is mostly unschooled people, then it would be more efficient to sort the book alphabetically rather than the historical order of the presidents.

Now let’s say that 75% of the population are historians and the other 25% are unschooled. Let’s assume the data the historians will need consists of a lot of different information regarding the president’s biographies, while all the unschooled need is the president’s age at the time they took office. In this scenario, it is more plausible to keep the ordering of the book (or the clustered index) based on the order of the president, then simply add the age of the president in the back index of the book (the non clustered index). That way the unschooled people do not have to look into the front of the book (clustered index) for the president’s age. They could simply find it in the back of the book by doing one single lookup. The presidents age being stored in the rear index would be considered at the “leaf” level. This would satisfy both requirements and would be efficient for both historians and the unschooled group.

After running, let’s turn on the execution plan (In SQL Server Managment Studio place your mouse in the query window and select Query -> Include Actual Execution Plan)

Now execute the following query:

SELECT   
    PresidentNumber  
    ,President  
    ,YearsInOffice  
    ,YearFirstInaugurated  
FROM Presidents  
WHERE PresidentNumber = 32

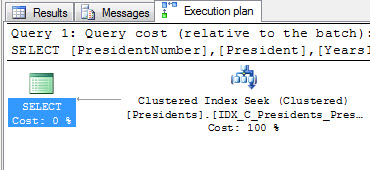
Now let’s view the execution plan:  
[](http://sqlserverplanet.com/wp-content/uploads/2009/06/tablescanexecutionplan1.jpg)  
Without a clustered index, our book is in no particular order. To find president 32, we need to scan every page.

Now let’s add a clustered index so we can organize our book according to PresidentNumber:

CREATE UNIQUE CLUSTERED INDEX IDX\_C\_Presidents\_PresidentNumber ON Presidents(PresidentNumber)

And let’s run our query again:

SELECT   
    PresidentNumber  
    ,President  
    ,YearsInOffice  
    ,YearFirstInaugurated  
FROM Presidents  
WHERE PresidentNumber = 32

[](http://sqlserverplanet.com/wp-content/uploads/2009/06/tableseekexecutionplan1.jpg)  
Our execution plan now shows a “clustered index seek”. Meaning we did not have to look through every page of our book. We jumped right to page 32 and found the information on our president there.

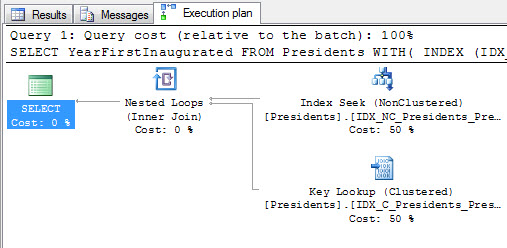
In summary, when we look up information based on the clustered index (the way the table is physically sorted), we naturally find all the information we are looking for already there (President, YearsInOffice, YearFirstInaugurated).

Now, let’s create a non clustered index and look up the YearFirstInaugurated by president’s name:

CREATE NONCLUSTERED INDEX IDX\_NC\_Presidents\_President ON Presidents(President)

Now let’s run our query to find the YearFirstInaugurated:

-- Force our query to use the index  
-- (table is so small SQL Server bypasses it)  
SELECT   
    YearFirstInaugurated  
FROM Presidents WITH(INDEX(IDX\_NC\_Presidents\_President))  
WHERE President = 'Franklin Roosevelt'

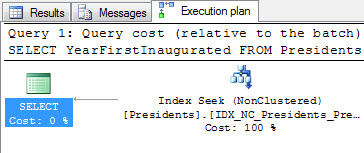
If we look at our execution plan now, we will see that we initially looked the president’s name up in our index, then after finding the page where the presidents biography was located, we went to that page to grab the YearFirstInaugurated. This is denoted by the “Key Lookup”. (Also known as “Bookmark Lookup”)  
[](http://sqlserverplanet.com/wp-content/uploads/2009/06/bookmarklookup1.jpg)  
This is a more expensive operation because our data is not at the “leaf-level” (or inline with the index we just searched), rather it is in the clustered index instead.

So how do we fix this? In SQL Server 2005, a new feature was introduced called “included columns”. This allows us to include data at the leaf-level of an index. So rather than looking up YearFirstInaugurated in the clustered index, we can find it in the nonclustered index. Let’s drop our index and include YearFirstInagurated in our nonclustered index:

DROP INDEX Presidents.IDX\_NC\_Presidents\_President  
GO  
CREATE NONCLUSTERED INDEX IDX\_NC\_Presidents\_President ON Presidents(President) INCLUDE(YearFirstInaugurated)

And run our query one more time:

-- Force our query to use the index  
-- (table is so small SQL Server bypasses it)  
SELECT   
    YearFirstInaugurated  
FROM Presidents WITH(INDEX(IDX\_NC\_Presidents\_President))  
WHERE President = 'Franklin Roosevelt'

Now we only have an index seek. Because as soon as we looked the president up in the index, we immediately also found the YearFirstInaugurated:  
[](http://sqlserverplanet.com/wp-content/uploads/2009/06/indexseek1.jpg)