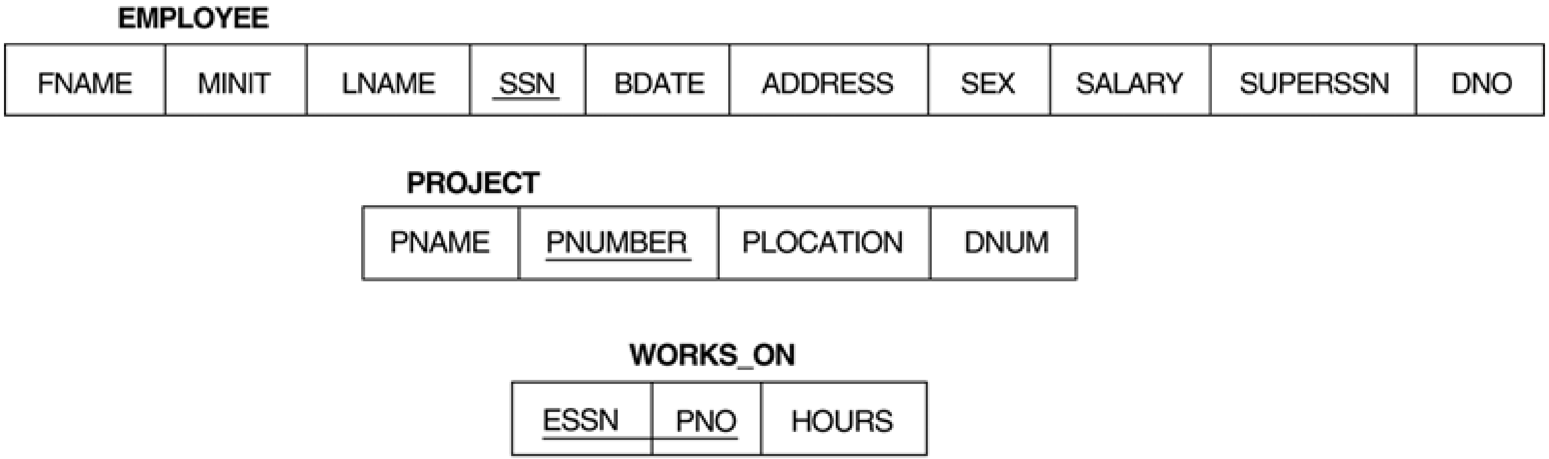
Views in SQL

- A view is a **“virtual” table** that is derived from other tables
- Allows for **limited update operations** (since the table may not physically be stored)
- **Allows full query operations**
- A convenience for expressing certain operations
  - simplify complex queries, and
  - define distinct conceptual interfaces for different users.

SQL Views: An Example



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

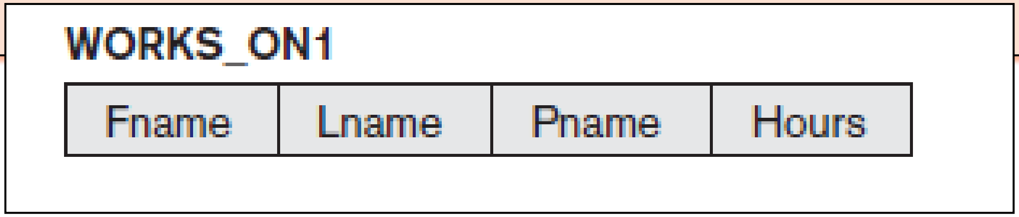
PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

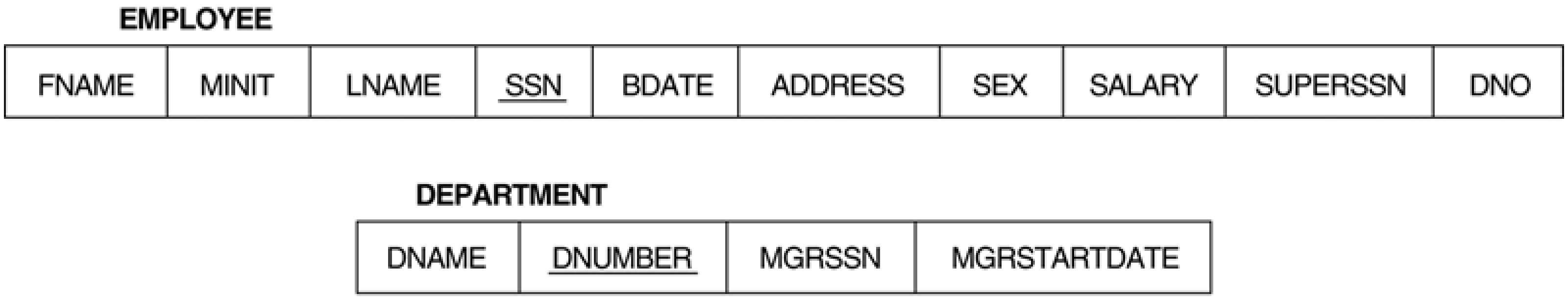
```
CREATE VIEW WORKS_ON1 AS
        SELECT FNAME, LNAME, PNAME, HOURS
        FROM EMPLOYEE, PROJECT, WORKS_ON
        WHERE SSN=ESSN AND PNO=PNUMBER
```



WORKS_ON1

Fname	Lname	Pname	Hours

SQL Views: An Example2

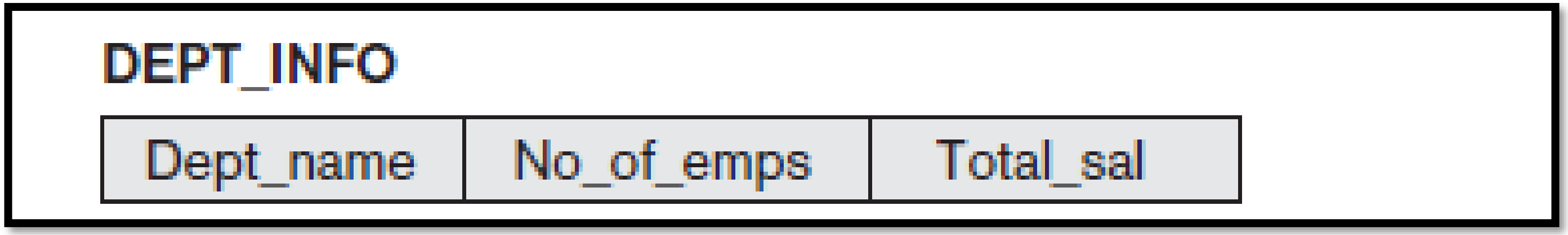


EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

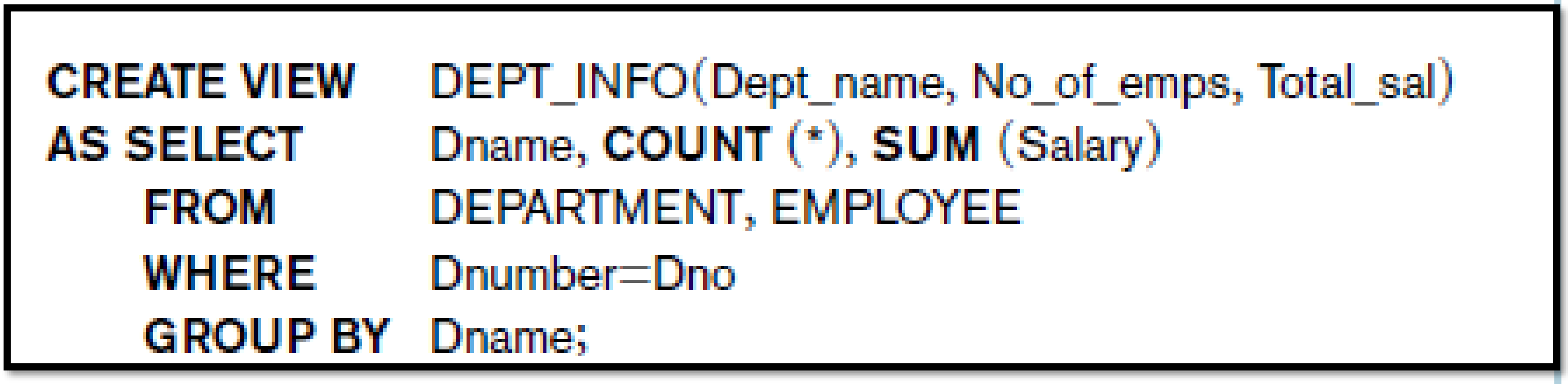
DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE



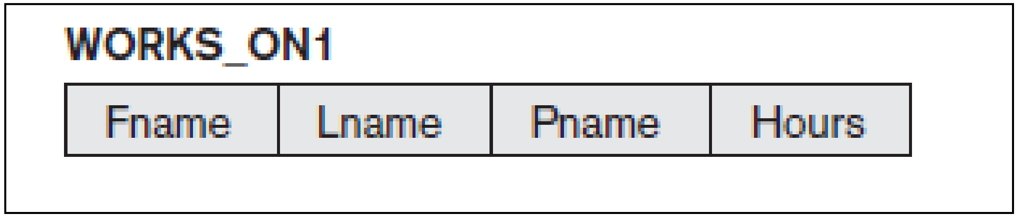
DEPT_INFO

Dept_name	No_of_emps	Total_sal



```
CREATE VIEW   DEPT_INFO(Dept_name, No_of_emps, Total_sal)
AS SELECT     Dname, COUNT (*), SUM (Salary)
     FROM     DEPARTMENT, EMPLOYEE
     WHERE    Dnumber=Dno
     GROUP BY Dname;
```

Query using a Virtual Table



WORKS_ON1

Fname	Lname	Pname	Hours

- We can specify SQL queries on a newly created view:

```
SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNAME='ProductX';
```

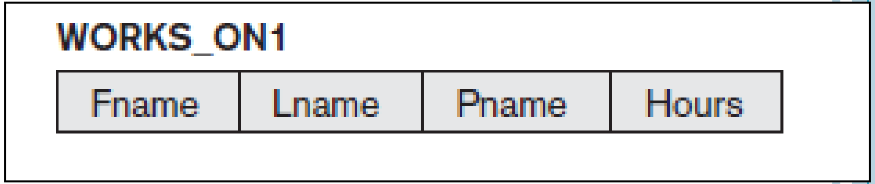
- DBMS is responsible to keep view always up-to-date
- When no longer needed, a view can be dropped:

DROP WORKS_ON1;

Efficient View Implementation

Query modification: present the view query in terms of a query on the underlying base tables

```
SELECT FNAME, LNAME
FROM WORKS_ON1
WHERE PNAME='ProductX'
```



WORKS_ON1

Fname	Lname	Pname	Hours

```
SELECT FNAME, LNAME
FROM (EMPLOYEE JOIN PROJECT on SSN=ESSN ) JOIN
                    WORKS_ON on PNO=PNUMBER
WHERE PNAME='PRODUCTX'
```

Disadvantage:

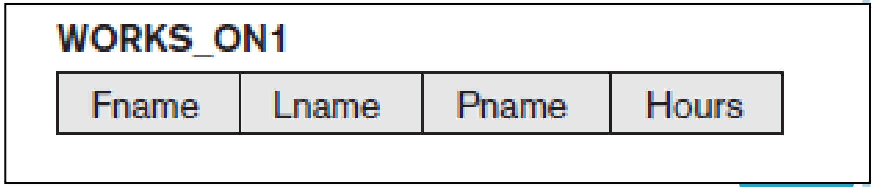
Inefficient for views defined via complex queries

Esp. if additional queries are to be applied within a short time period

Efficient View Implementation

View materialization: involves physically creating and keeping a temporary table

- assumption: other queries on the view will follow
- concerns: maintaining correspondence between the base table and the view when the base table is updated
- strategy: incremental update



WORKS_ON1

Fname	Lname	Pname	Hours

View Update

Single view without aggregate operations:

- update may map to an update on the underlying base table

Views involving joins:

- an update *may* map to an update on the underlying base relations
- not always possible

EXAMPLE – Complex View Update

- **Example:**

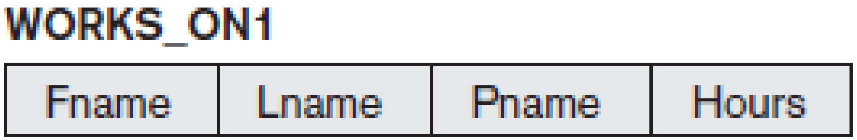
UPDATE WORKS_ON1

SET PNAME=COMPUTERIZATION

WHERE FNAME=‘JOHN AND

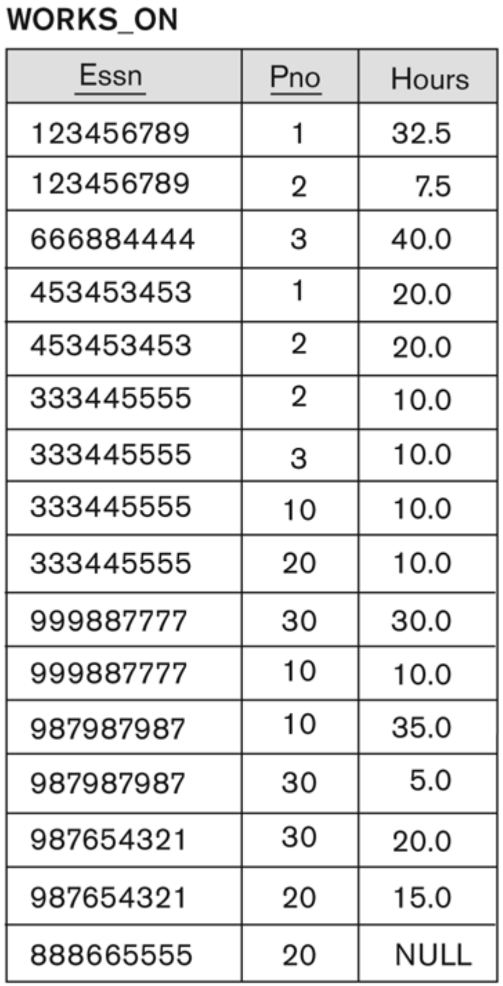
LNAME='SMITH' AND

PNAME=‘PRODUCTX’



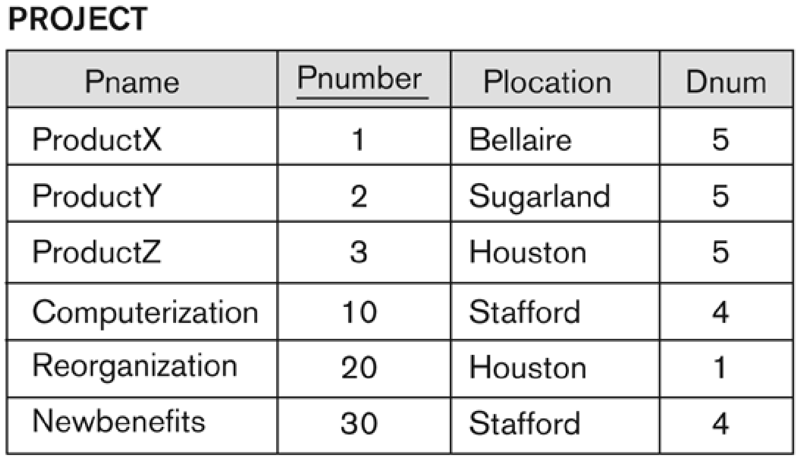
WORKS_ON1

Fname	Lname	Pname	Hours



WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL



PROJECT

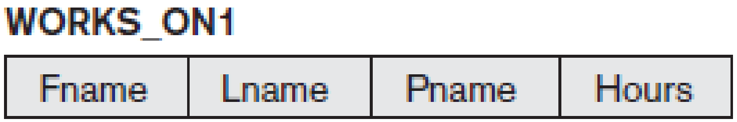
Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

EXAMPLE – Complex View Update

UPDATE WORKS_ON1

SET PNAME=COMPUTERIZATION

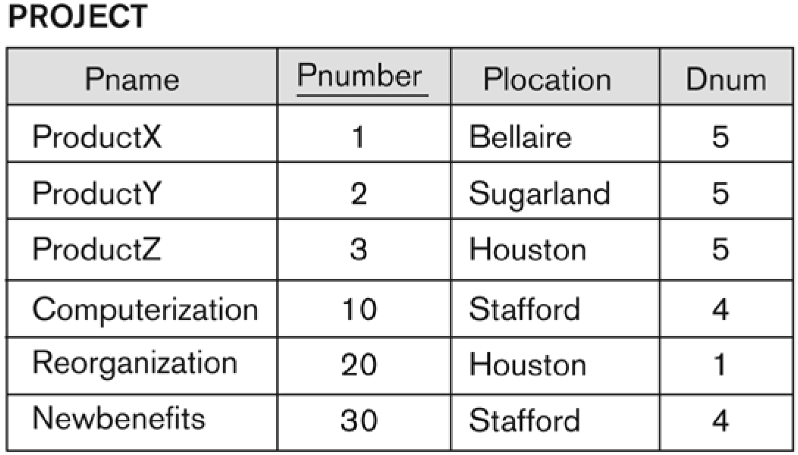
WHERE FNAME=‘JOHN AND LNAME='SMITH' AND PNAME=‘PRODUCTX’



WORKS_ON1

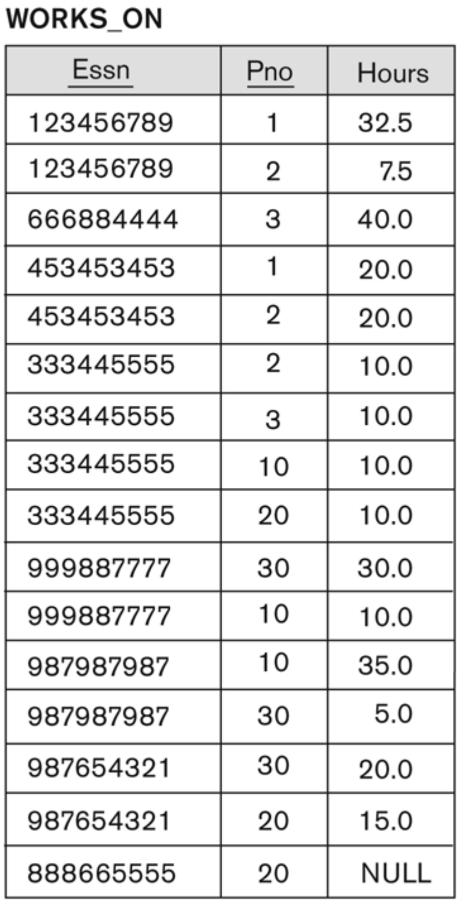
Fname	Lname	Pname	Hours

A) UPDATE PROJECT
SET PNAME=‘COMPUTERIZATION’
WHERE PNAME=‘PRODUCTX’



PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4



WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

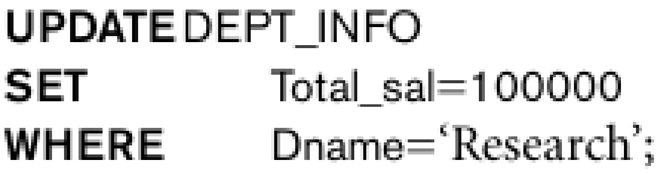
View Update

```
UPDATE WORKS_ON1
SET PNAME=COMPUTERIZATION
WHERE  FNAME=‘JOHN  AND LNAME='SMITH' AND PNAME=‘PRODUCTX’
```

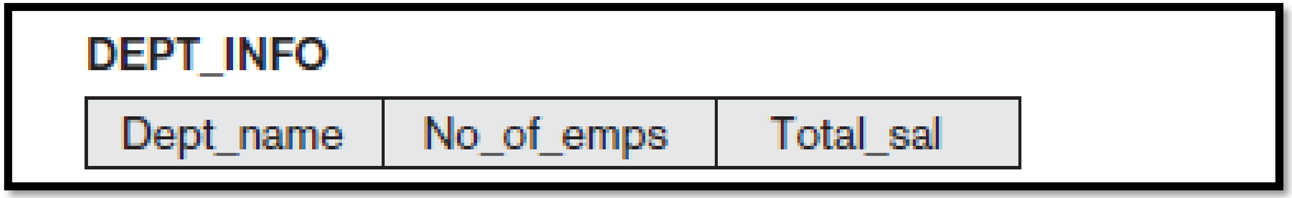
```
B)UPDATE WORKS_ON
SET PNO =       (SELECT PNUMBER
                 FROM PROJECT
                 WHERE PNAME=‘COMPUTERIZATION’)
WHERE ESSN IN  (SELECT SSN
                    FROM EMPLOYEE
                     WHERE LNAME=‘SMITH’ AND FNAME=‘JOHN’)
      AND
      PNO = (SELECT PNUMBER FROM PROJECT
                 WHERE PNAME=‘PRODUCTX’)
```

Un-updatable Views

- Views defined using groups and aggregate functions are not updateable



```
UPDATE DEPT_INFO
SET        Total_sal=100000
WHERE      Dname=‘Research’;
```



DEPT_INFO

Dept_name	No_of_emps	Total_sal

- Views defined on multiple tables using joins are generally not updateable

SQL Server indexed view

- Regular SQL Server views provide query simplicity and security. But do not improve the query performance.

- SQL Server indexed views are **materialized views** that stores data physically like a table
- **Indexed views** provide some the performance benefit if they are used appropriately.

SQL Server indexed view

```
CREATE VIEW WORKS_ON2 WITH SCHEMABINDING
AS
        SELECT SSN,FNAME, LNAME, PNAME, HOURS
        FROM EMPLOYEE, PROJECT, WORKS_ON
        WHERE SSN=ESSN AND PNO=PNUMBER
```

```
CREATE UNIQUE CLUSTERED INDEX idx1 ON
WORKS_ON2 (SSN)
```

This statement materializes the view, so it have a physical existence in the database.

https://www.sqlservertutorial.net/sql-server-views/sql-server-indexed-view/

SQL Server indexed view

```
CREATE VIEW WORKS_ON2 WITH SCHEMABINDING
AS
        SELECT SSN,FNAME, LNAME, PNAME, HOURS
        FROM EMPLOYEE, PROJECT, WORKS_ON
        WHERE SSN=ESSN AND PNO=PNUMBER
```

```
CREATE NONCLUSTERED INDEX idx2 ON
Company.WORKS_ON1 (SSN);
```

This statement materializes the view, so it have a physical existence in the database.

SQL Index

- **Indexes** are used to retrieve data from the database more quickly than otherwise.
- The users cannot see the indexes, they are just used to speed up searches/queries.

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX
index_name ON <object> ( column_name [ ASC | DESC ] [ ,...n ] )
```

HW : Read and implement

Clustered and Non clustered Index

https://learn.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql?view=sql-server-ver16

SQL INDEX

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

```
CREATE INDEX idx_E ON Employee(SSN);
```

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX
index_name ON <object> ( column_name [ ASC | DESC ] [ ,...n ] )
```

```
CREATE UNIQUE INDEX uidx_E ON Employee(SSN)
```

```
CREATE INDEX idxFL ON Employee(Fname, Lname);
```

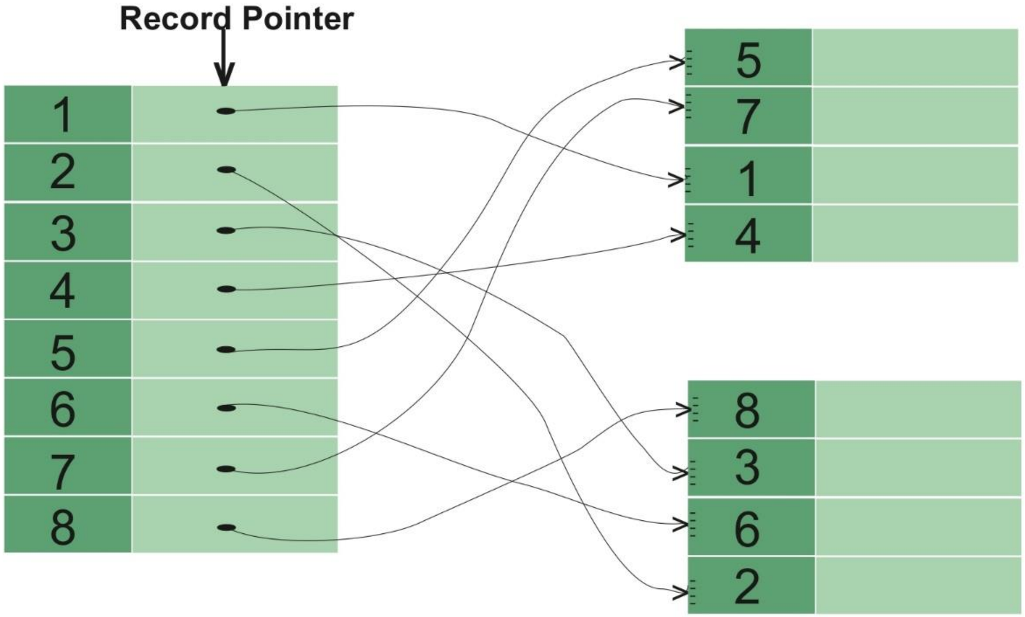
HW : Read and implement

Clustered and Non clustered Index

https://learn.microsoft.com/en-us/sql/t-sql/statements/create-index-transact-sql?view=sql-server-ver16

SQL Index

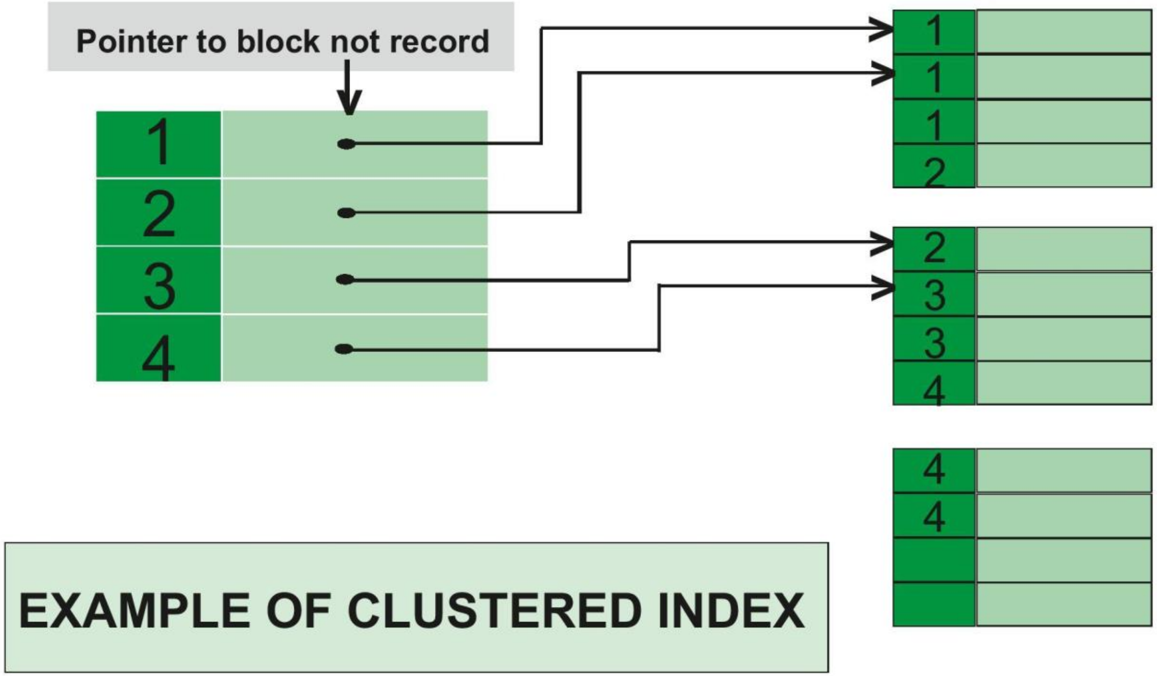
Non-Clustered Index is similar to the index of a book



EXAMPLE OF NON-CLUSTERED INDEX

SQL Index

- Clustered index sort the data in the table.
- You can create only one clustered index in a table like primary key.
- Clustered index is as same as dictionary where the data is arranged by alphabetical order.



EXAMPLE OF CLUSTERED INDEX

SQL Triggers

Triggers monitors a database and executes when an event occurs in the database server.

- like insertion,
- deletion or
- updation of data.

It is a database object which is bound to a table and is executed automatically.

You can’t explicitly invoke **trigger**s.

- The only way to do this is by performing the required action on the table that they are assigned to.

SQL Triggers

Objective: to monitor a database and take action when a condition occurs

Triggers include the following:

- event (e.g., an update operation)
- condition
- action (to be taken when the condition is satisfied)

Triggers are classified into two main types:

- After Triggers (For Triggers)
- Instead Of Triggers

SQL Triggers: An Example

Using a trigger with a reminder message

```
CREATE TRIGGER Reminder
ON Employee
AFTER INSERT, UPDATE
AS PRINT 'Notify employee added or updated’
```

SQL Triggers: An Example

A trigger to compare an employee’s salary to his/her supervisor after insert or update operations:

```
CREATE TRIGGER Emp_Salary ON Employee
   FOR INSERT, UPDATE
   AS
   IF EXISTS (SELECT * FROM inserted as i JOIN Employee as e ON
                    i.super_SSN= e.SSN WHERE i.salary > e.salary)
   BEGIN
         PRINT ‘Employee salary is greater than the Supervisor Salary'
   END
```

```
INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, Super_SSN, Salary)
VALUES ('Richard', 'Marini', '653298653','123456789',500000)
```

SQL Triggers

- **CREATE** TRIGGER SampleTrigger **ON** Employee
- **INSTEAD OF INSERT**
- **AS**
-    SELECT * FROM Employee

- To fire the trigger we can insert a row in table and it will show list of all user instead of inserting into the table

INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, Super_SSN, Salary)
VALUES ('Richard', 'Marini', '653298653','123456789',500000)

INSTEAD OF triggers are usually used to correctly update views that are based on multiple tables.

SQL Triggers: An Example

Using a trigger with a reminder message

```
CREATE TRIGGER reminder2
ON  employee
AFTER INSERT, UPDATE, DELETE
AS
        EXEC msdb.dbo.sp_send_dbmail
        @profile_name = ‘The Administrator’,
        @recipients = 'danw@Adventure-Works.com’,
        @body = 'Don''t forget to print a report’,
        @subject = 'Reminder’;
```

https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql?view=sql-server-2017

Trigger

It is required that a team do not submit more than two proposals. Write a SQL query or trigger or view to solve this issue?

- **INSTEAD OF** triggers are run in place of the Insert command.
  - If you run insert command in instead of trigger it will again call the trigger so on.
- You can either use After (FOR) trigger
  - check if the inserted row has violated the given condition. If yes then delete it.

OR

- you can handle it at frontend application using Sql query to check if the given team has already submitted two projects then do not insert.

SQL Stored Procedure

- A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.
- You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

```
CREATE PROCEDURE SelectAllEmp AS
SELECT * FROM Employee
```

Execute the stored procedure above as follows:

```
EXEC SelectAllEmp;
```

https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/create-a-stored-procedure?view=sql-server-ver16

SQL Stored Procedure

```
CREATE PROCEDURE SelectAllEmp @DNO INTEGER AS
SELECT * FROM Employee WHERE DNO = @Dnum
```

```
EXEC SelectAllEmp @DNO=4
```

```
CREATE PROCEDURE SelectAllEmp @DNO INTEGER, @Fname
varchar(30) AS
SELECT * FROM Employee WHERE DNO = @Dnum
```

```
EXEC SelectAllEmp @DNO=4 AND Fname =‘John’
```

Why Stored Procedure

- **Enhances Performance**
  - **Stored Procedure can reuse compiled and cached query plans**.
  - In the **first execution of a stored procedure**, its execution plan is stored in the query plan cache and this query plan is used in the next execution of the procedure.
- **Provides an important layer of security between the user interface and the DB**.
  - It supports security through data access controls because end users may enter or change data, but do not write procedures.

Transactions?

- *Transaction* is a process involving database queries and/or modification.

- Database systems are normally being accessed by many users or processes at the same time.

- Example- ATM

- Formed in SQL from single statements or explicit programmer control

ACID Transactions

Atomic

- Whole transaction or none is done.

Consistent

- Database constraints preserved.

Isolated

- It appears to the user as if only one process executes at a time.

Durable

- Effects of a process survive a crash.

Optional: weaker forms of transactions are often supported as well.

Example of *Fund Transfer*

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*

- **Atomicity requirement :**
  - if the transaction **fails** after step 3 and before step 6,
    - the **system** should **ensure** that :
      - its **updates** are *not reflected* in the database,
      - else an *inconsistency* will result.

Example of *Fund Transfer*

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*

- **Consistency requirement :**
  - the **sum** of **A** and **B** is:
    - **unchanged** by the execution of the transaction.

Example of *Fund Transfer* (Cont.)

- Transaction to transfer $50 from account **A** to account **B**:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B +* 50
  6. **write**(*B)*
- **Isolation requirement** —
  - if between steps 3 and 6, another transaction is allowed to access the partially updated database,
    - it will see an inconsistent database (the sum *A + B* will be less than it should be).
  - Isolation can be **ensured** trivially by:
    - running transactions **serially***,* that is **one** after the **other**.
  - *However*, executing multiple *transactions* **concurrently** has significant **benefits**.

Example of *Fund Transfer* (Cont.)

- Transaction to transfer $50 from account **A** to account **B**:

  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*

- **Durability requirement :**
  - once the user has been notified that the transaction has **completed** :
    - (i.e., the transfer of the $50 has taken place),
    - the **updates** to the database by the transaction **must persist**
      - despite *failures*.

T-SQL AND Transactions

SQL has following transaction modes.

- Autocommit transactions
  - Each individual SQL statement = transaction.

- Explicit transactions

  BEGIN TRANSACTION

  [SQL statements]

  COMMIT   or   ROLLBACK

Transaction Support in TSQL

- BEGIN TRAN
- UPDATE Department
- SET Mgr_ssn = 123456789
- WHERE DNumber = 1
- UPDATE Department
- SET Mgr_start_date = '1981-06-19'
- WHERE Dnumber = 1
- COMMIT TRAN

Transaction Support in SQL

Potential problem with lower isolation levels:

- **Dirty Read**
  - Reading a value that was written by a failed transaction.

- **Nonrepeatable Read**
  - Allowing another transaction to write a new value between multiple reads of one transaction.
    - A transaction T1 reads a given value from a table.
    - If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.

Transaction Support in SQL

- Potential problem with lower isolation levels (contd.):
  - Phantoms
    - New rows being read using the same read with a condition.
      - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE clause.
      - Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE clause condition of T1, into the table used by T1.
      - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

Transaction Support in TSQL

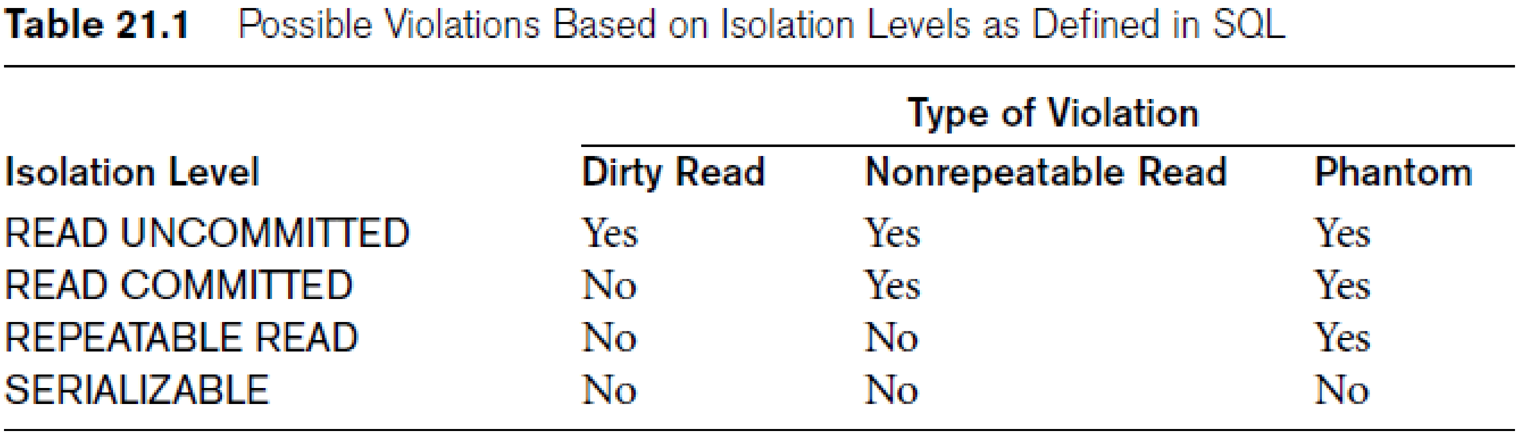


Table 21.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Transaction Support in TSQL

1. “Dirty reads”

   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

2. “Committed reads”

   SET TRANSACTION ISOLATION LEVEL READ COMMITTED

3. “Repeatable reads”

   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

4. Serializable transactions (default):

   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE