Contents

[Normalization 2](#_Toc520836352)

[Insertion Anomaly 2](#_Toc520836353)

[Updation Anomaly 2](#_Toc520836354)

[Deletion Anomaly 2](#_Toc520836355)

[Normalization Rules 2](#_Toc520836356)

[1NF 2](#_Toc520836357)

[2NF 3](#_Toc520836358)

[3NF 4](#_Toc520836359)

[Isolation Levels 6](#_Toc520836360)

[Read Committed 6](#_Toc520836361)

[Read Uncommitted 8](#_Toc520836362)

[Serializable 8](#_Toc520836363)

[Snapshot 10](#_Toc520836364)

[ACID 12](#_Toc520836365)

# Normalization

* Problem:
  + Data redundancy

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rollno** | **name** | **Branch** | **hod** | **office\_tel** |
| 401 | Akon | CSE | Mr. X | 53337 |
| 402 | Bkon | CSE | Mr. X | 53337 |
| 403 | Ckon | CSE | Mr. X | 53337 |
| 404 | Dkon | CSE | Mr. X | 53337 |

* + Branch, HOD and Office\_Tel is repeated

## Insertion Anomaly

* New student. Has to enter Branch details
* OR has to keep it NULL
* To insert 100s of students of the same branch, Branch info will be repeated for all of them

## Updation Anomaly

* If Mr. X leaves the college or not HOD anymore, all student records to be updated
* Leave out some by mistake, data inconsistency

## Deletion Anomaly

* Two different info kept together: Student and Branch
* At end of the academic year, if Student records deleted, Branch info is also lost

## Normalization Rules

### 1NF

For a table to be in the First Normal Form, it should follow the following 4 rules:

1. It should only have single (atomic) valued attributes/columns.
   * Each column of your table should be single valued which means they should not contain multiple values
2. Values stored in a column should be of the same domain
   * This is more of a "Common Sense" rule. In each column the values stored must be of the same kind or type
   * If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' of some of them in that column along with 'date of birth' of others in that column. It should hold only 'date of birth' for all the records/rows
3. All the columns in a table should have unique names.
4. And the order in which data is stored, does not matter.

Example:

Problem:

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Akon | OS, CN |
| 103 | Ckon | Java |
| 102 | Bkon | C, C++ |

Solution:

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Akon | OS |
| 101 | Akon | CN |
| 103 | Ckon | Java |
| 102 | Bkon | C |
| 102 | Bkon | C++ |

**Note**: Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

### 2NF

For a table to be in the Second Normal Form,

1. It should be in the First Normal form.
2. And, it should not have Partial Dependency.

**Example: Student table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id (PK)** | **Name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | New York |
| 11 | Akon | 08-WY | IT | Maine |

* Using a student\_id, we can fetch details (like name) of any student uniquely identifying each row
* This is **Functional Dependency**

**So, what is Partial Dependency?**

**Example: Subject table**

|  |  |
| --- | --- |
| **subject\_id (PK)** | **subject\_name** |
| 1 | Java |
| 2 | C++ |
| 3 | Php |

**Score table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **teacher** |
| 1 | 10 | 1 | 70 | Java Teacher |
| 2 | 10 | 2 | 75 | C++ Teacher |
| 3 | 11 | 1 | 80 | Java Teacher |

* Student\_Id + Subject\_Id are ***composite key => PK***
  + To uniquely identify each row for a student and a subject
* Column *teacher* is dependent only on subject
  + Teacher’s name depends only on subject
* This is ***Partial Dependency***
* When for a composite key, any attribute of the table depends on only part of the PK and not on the complete PK

Solution:

**Subject table**

|  |  |  |
| --- | --- | --- |
| **subject\_id** | **subject\_name** | **teacher** |
| 1 | Java | Java Teacher |
| 2 | C++ | C++ Teacher |
| 3 | Php | Php Teacher |

**Score table**

|  |  |  |  |
| --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** |
| 1 | 10 | 1 | 70 |
| 2 | 10 | 2 | 75 |
| 3 | 11 | 1 | 80 |

### 3NF

For a table to be in the Third Normal Form when,

1. It is in the Second Normal form.
2. And, it doesn't have Transitive Dependency.

Example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Student Table** | |  |  |  |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | Kerala |
| 11 | Akon | 08-WY | IT | Gujarat |
| 12 | Bkon | 09-WY | IT | Rajasthan |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Subject Table** | |  |  |  | |
| **subject\_id** | **subject\_name** | **teacher** |  |  | |
| 1 | Java | Java Teacher |  |  | |
| 2 | C++ | C++ Teacher |  |  | |
| 3 | Php | Php Teacher |  |  | |
| **Score Table** | |  |  |  | |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_name** | **total\_marks** | |
| 1 | 10 | 1 | 70 |  |  | |
| 2 | 10 | 2 | 75 |  |  | |
| 3 | 11 | 1 | 80 |  |  | |

* Student\_Id + Subject\_Id are ***composite key => PK***
* Exam\_name depends on both keys
  + E.g.;
    - Mechanical Engg student has Workshop exam
    - Science student has Practical exam
* Total\_marks???
  + Depends on exam\_name
  + Exam\_name is not a PK or even part of the CK-PK
  + This is ***Transitive Dependency***
* When a non-prime attribute depends on another non-prime attribute rather than on the PK, we have Transitive Dependency

Solution:

Take out the columns exam\_name and total\_marks from Score table and put them in an Exam table and use the exam\_id wherever required

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Score Table: In 3rd Normal Form** | | | | |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_id** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| **The new Exam table** | | | | |
| **exam\_id** | **exam\_name** | **total\_marks** |  |  |
| 1 | Workshop | 200 |  |  |
| 2 | Mains | 70 |  |  |
| 3 | Practicals | 30 |  |  |

The advantage of removing transitive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

# Isolation Levels

Following are the different types of isolations available in SQL Server.

* READ COMMITTED
* READ UNCOMMITTED
* REPEATABLE READ
* SERIALIZABLE
* SNAPSHOT

## Read Committed

In select query it will take only committed values of table. If any transaction is opened and incomplete on table in others sessions then select query will wait till no transactions are pending on same table.

Read Committed is the default transaction isolation level.

**Read committed example 1:**

*Session 1*

begin tran

update emp set Salary=999 where ID=1

waitfor delay '00:00:15'

commit

*Session 2*

set transaction isolation level read committed

select Salary from Emp where ID=1

Run both sessions side by side.

*Output*

*999*

In second session, it returns the result only after execution of complete transaction in first session because of the lock on Emp table. We have used wait command to delay 15 seconds after updating the Emp table in transaction.

**Read committed example 2**

*Session1*

begin tran

select \* from Emp

waitfor delay '00:00:15'

commit

*Session2*

set transaction isolation level read committed

select \* from Emp

Run both sessions side by side.

*Output*

*1000*

In session2, there won't be any delay in execution because in session1 Emp table is used under transaction but it is not used update or delete command hence Emp table is not locked.

**Read committed example 3**

*Session 1*

begin tran

select \* from emp

waitfor delay '00:00:15'

update emp set Salary=999 where ID=1

commit

*Session 2*

set transaction isolation level read committed

select Salary from Emp where ID=1

Run both sessions side by side.

*Output*

*1000*

In session2, there won't be any delay in execution because when session2 is executed Emp table in session1 is not locked (used only select command, locking on Emp table occurs after wait delay command).

## Read Uncommitted

If any table is updated(insert or update or delete) under a transaction and same transaction is not completed that is not committed or roll backed then uncommitted values will displaly(Dirty Read) in select query of "Read Uncommitted" isolation transaction sessions. There won't be any delay in select query execution because this transaction level does not wait for committed values on table.

**Read uncommitted example 1**

*Session 1*

begin tran

update emp set Salary=999 where ID=1

waitfor delay '00:00:15'

rollback

*Session 2*

set transaction isolation level read uncommitted

select Salary from Emp where ID=1

Run both sessions at a time one by one.

*Output*

*999*

Select query in Session2 executes after update Emp table in transaction and before transaction rolled back. Hence 999 is returned instead of 1000.

## Serializable

Serializable Isolation is similar to Repeatable Read Isolation but the difference is it prevents Phantom Read. This works based on range lock. If table has index then it locks records based on index range used in WHERE clause(like where ID between 1 and 3). If table doesn't have index then it locks complete table.

**Serializable Example 1**

Assume table does not have index column.

*Session 1*

set transaction isolation level serializable

begin tran

select \* from emp

waitfor delay '00:00:15'

select \* from Emp

rollback

*Session 2*

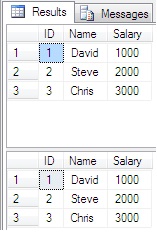
insert into Emp(ID,Name,Salary)

values( 11,'Stewart',11000)

Run both sessions side by side.

*Output*

*Result in Session 1.*



Complete Emp table will be locked during the transaction in Session 1. Unlike "Repeatable Read", insert query in Session 2 will wait till session 1 execution is completed. Hence Phantom read is prevented and both queries in session 1 will display same number of rows.

**Serializable Example 2**

Assume table has primary key on column "ID". In our example script, primary key is not added. Add primary key on column Emp.ID before executing below examples.

*Session 1*

set transaction isolation level serializable

begin tran

select \* from emp where ID between 1 and 3

waitfor delay '00:00:15'

select \* from Emp where ID between 1 and 3

rollback

*Session 2*

insert into Emp(ID,Name,Salary)

values( 11,'Stewart',11000)

Run both sessions side by side.

*Output*

Since Session 1 is filtering IDs between 1 and 3, only those records whose IDs range between 1 and 3 will be locked and these records can not be modified and no new records with ID range between 1 to 3 will be inserted. In this example, new record with ID=11 will be inserted in Session 2 without any delay.

## Snapshot

Snapshot isolation is similar to Serializable isolation. The difference is Snapshot does not hold lock on table during the transaction so table can be modified in other sessions. Snapshot isolation maintains versioning in Tempdb for old data in case of any data modification occurs in other sessions then existing transaction displays the old data from Tempdb.

**Snapshot Example 1**

*Session 1*

set transaction isolation level snapshot

begin tran

select \* from Emp

waitfor delay '00:00:15'

select \* from Emp

rollback

*Session 2*

insert into Emp(ID,Name,Salary) values( 11,'Stewart',11000)

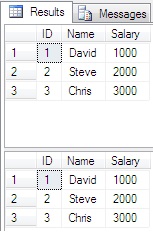
update Emp set Salary=4444 where ID=4

select \* from Emp

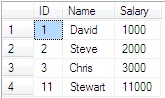
Run both sessions side by side.

*Output*

*Result in Session 1.*



*Result in Session 2.*



Session 2 queries will be executed in parallel as transaction in session 1 won't lock the table Emp.

# ACID

* **Atomicity**
  + All or Nothing Txns
* **Consistency**
  + Guarantees that a transaction never leaves your database in a half-finished state
* **Isolation**
  + Keeps transactions separated from each other until they’re finished
* **Durability**
  + Guarantees that the database will keep track of pending changes in such a way that the server can recover from an abnormal termination

