**Q5a) Explain various types of constraints with an example.**

**SQL Constraints**

SQl Constraints are rules used to limit the type of data that can go into a table, to maintain the accuracy and integrity of the data inside table.

Constraints can be divided into following two types,

* **Column level constraints :** limits only column data
* **Table level constraints :** limits whole table data

Constraints are used to make sure that the integrity of data is maintained in the database. Following are the most used constraints that can be applied to a table.

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

**NOT NULL Constraint**

NOT NULL constraint restricts a column from having a NULL value. Once **NOT NULL** constraint is applied to a column, you cannot pass a null value to that column. It enforces a column to contain a proper value. One important point to note about NOT NULL constraint is that it cannot be defined at table level.

**Example using NOT NULL constraint**

CREATE table Student(s\_id int NOT NULL, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will not take NULL value.

**UNIQUE Constraint**

UNIQUE constraint ensures that a field or column will only have unique values. A UNIQUE constraint field will not have duplicate data. UNIQUE constraint can be applied at column level or table level.

**Example using UNIQUE constraint when creating a Table (Table Level)**

CREATE table Student(s\_id int NOT NULL UNIQUE, Name varchar(60), Age int);

The above query will declare that the **s\_id** field of **Student** table will only have unique values and wont take NULL value.

**Example using UNIQUE constraint after Table is created (Column Level)**

ALTER table Student add UNIQUE(s\_id);

The above query specifies that **s\_id** field of **Student** table will only have unique value.

**Primary Key Constraint**

Primary key constraint uniquely identifies each record in a database. A Primary Key must contain unique value and it must not contain null value. Usually Primary Key is used to index the data inside the table.

**Example using PRIMARY KEY constraint at Table Level**

CREATE table Student (s\_id int **PRIMARY KEY**, Name varchar(60) NOT NULL, Age int);

The above command will creates a PRIMARY KEY on the s\_id.

**Example using PRIMARY KEY constraint at Column Level**

ALTER table Student add PRIMARY KEY (s\_id);

The above command will creates a PRIMARY KEY on the s\_id.

**Foreign Key Constraint**

FOREIGN KEY is used to relate two tables. FOREIGN KEY constraint is also used to restrict actions that would destroy links between tables. To understand FOREIGN KEY, let's see it using two table.

**Customer\_Detail Table :**

|  |  |  |
| --- | --- | --- |
| **c\_id** | **Customer\_Name** | **address** |
| 101 | Adam | Noida |
| 102 | Alex | Delhi |
| 103 | Stuart | Rohtak |

**Order\_Detail Table :**

|  |  |  |
| --- | --- | --- |
| **Order\_id** | **Order\_Name** | **c\_id** |
| 10 | Order1 | 101 |
| 11 | Order2 | 103 |
| 12 | Order3 | 102 |

In **Customer\_Detail** table, c\_id is the primary key which is set as foreign key in **Order\_Detail** table. The value that is entered in c\_id which is set as foreign key in **Order\_Detail** table must be present in **Customer\_Detail**table where it is set as primary key. This prevents invalid data to be inserted into c\_id column of **Order\_Detail**table.

**Example using FOREIGN KEY constraint at Table Level**

CREATE table Order\_Detail(order\_id int PRIMARY KEY,

order\_name varchar(60) NOT NULL,

*c\_id int* **FOREIGN KEY** REFERENCES **Customer\_Detail**(*c\_id*));

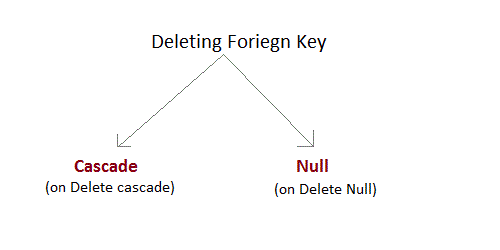
In this query, c\_id in table Order\_Detail is made as foriegn key, which is a reference of c\_id column of Customer\_Detail.

**Example using FOREIGN KEY constraint at Column Level**

ALTER table Order\_Detail add **FOREIGN KEY** (c\_id) REFERENCES Customer\_Detail(c\_id);

**Behaviour of Foriegn Key Column on Delete**

There are two ways to maintin the integrity of data in Child table, when a particular record is deleted in main table. When two tables are connected with Foriegn key, and certain data in the main table is deleted, for which record exit in child table too, then we must have some mechanism to save the integrity of data in child table.



* **On Delete Cascade :** This will remove the record from child table, if that value of foriegn key is deleted from the main table.
* **On Delete Null :** This will set all the values in that record of child table as NULL, for which the value of foriegn key is eleted from the main table.
* If we don't use any of the above, then we cannot delete data from the main table for which data in child table exists. We will get an error if we try to do so.

ERROR : Record in child table exist

**CHECK Constraint**

CHECK constraint is used to restrict the value of a column between a range. It performs check on the values, before storing them into the database. Its like condition checking before saving data into a column.

**Example using CHECK constraint at Table Level**

create table Student(s\_id int NOT NULL **CHECK(s\_id > 0)**,

Name varchar(60) NOT NULL,

Age int);

The above query will restrict the s\_id value to be greater than zero.

**Example using CHECK constraint at Column Level**

ALTER table Student add CHECK(s\_id > 0);

Q6a]1)generalization and aggregation

Ans>

Generalization:

The refinement from an initial entity set into successive levels of entity subgroupings

represents a **top-down** design process inwhich distinctions are made explicit.

The design process may also proceed in a **bottom-up** manner, in which multiple

entity sets are synthesized into a higher-level entity set on the basis of common

features. The database designer may have first identified:

• *instructor* entity set with attributes *instructor id*, *instructor name*, *instructor*

*salary*, and *rank*.

• *secretary* entity set with attributes *secretary id*, *secretary name*, *secretary salary*,

and *hours per week*.

There are similarities between the *instructor* entity set and the *secretary* entity

set in the sense that they have several attributes that are conceptually the same

across the two entity sets: namely, the identifier, name, and salary attributes.

This commonality can be expressed by **generalization**, which is a containment

relationship that exists between a *higher-level* entity set and one ormore *lower-level*

entity sets. In our example, *employee* is the higher-level entity set and *instructor* and

*secretary* are lower-level entity sets. In this case, attributes that are conceptually

the same had different names in the two lower-level entity sets. To create a

generalization, the attributes must be given a common name and represented

with the higher-level entity *person*. We can use the attribute names *ID*, *name*,

*address*, as we saw in the example in Section 7.8.1.

**298 Chapter 7 Database Design and the E-R Model**

Higher- and lower-level entity sets also may be designated by the terms

**superclass** and **subclass**, respectively. The *person* entity set is the superclass of the

*employee* and *student* subclasses.

For all practical purposes, generalization is a simple inversion of specialization.

We apply both processes, in combination, in the course of designing the E-R

schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish

between specialization and generalization. New levels of entity representation

are distinguished (specialization) or synthesized (generalization) as the design

schema comes to express fully the database application and the user requirements

of the database. Differences in the two approaches may be characterized

by their starting point and overall goal.

Specialization stems from a single entity set; it emphasizes differences among

entities within the set by creating distinct lower-level entity sets. These lowerlevel

entity sets may have attributes, or may participate in relationships, that do

not apply to all the entities in the higher-level entity set. Indeed, the reason a

designer applies specialization is to represent such distinctive features. If *student*

and *employee* have exactly the same attributes as *person* entities, and participate

in exactly the same relationships as *person* entities, there would be no need to

specialize the *person* entity set.

Generalization proceeds from the recognition that a number of entity sets

share some common features (namely, they are described by the same attributes

and participate in the same relationship sets). On the basis of their commonalities,

generalization synthesizes these entity sets into a single, higher-level entity set.

Generalization is used to emphasize the similarities among lower-level entity sets

and to hide the differences; it also permits an economy of representation in that

shared attributes are not repeated.

Aggregation:

One limitation of the E-R model is that it cannot express relationships among

relationships. To illustrate the need for such a construct, consider the ternary

relationship *proj guide*, which we saw earlier, between an *instructor*, *student* and

*project* (see Figure 7.13).

Now suppose that each instructor guiding a student on a project is required to

file a monthly evaluation report.We model the evaluation report as an entity *evaluation*,

with a primary key *evaluation id*. One alternative for recording the (*student*,

*project*, *instructor*) combination to which an *evaluation* corresponds is to create a

quaternary (4-way) relationship set *eval for* between *instructor*, *student*, *project*, and

*evaluation*. (A quaternary relationship is required—a binary relationship between

*student* and *evaluation*, for example, would not permit us to represent the (*project*,

*instructor*) combination to which an *evaluation* corresponds.) Using the basic E-R

modeling constructs, we obtain the E-R diagram of Figure 7.22. (We have omitted

the attributes of the entity sets, for simplicity.)

It appears that the relationship sets *proj guide* and *eval for* can be combined

into one single relationship set. Nevertheless, we should not combine them into

a single relationship, since some *instructor*, *student*, *project* combinations may not

have an associated *evaluation*.

There is redundant information in the resultant figure, however, since every

*instructor*, *student*, *project* combination in *eval for* must also be in *proj guide*. If the

*evaluation* were a value rather than a entity, we could instead make *evaluation*

a multivalued composite attribute of the relationship set *proj guide*. However,

this alternative may not be be an option if an *evaluation* may also be related

to other entities; for example, each evaluation report may be associated with a

*secretary* who is responsible for further processing of the evaluation report to

make scholarship payments.

The best way to model a situation such as the one just described is to use aggregation.

**Aggregation** is an abstraction through which relationships are treated

as higher-level entities. Thus, for our example, we regard the relationship set

*proj guide* (relating the entity sets *instructor*, *student*, and *project*) as a higher-level

entity set called *proj guide*. Such an entity set is treated in the same manner as is

any other entity set.We can then create a binary relationship *eval for* between *proj*

**302 Chapter 7 Database Design and the E-R Model**

*project*

*evaluation*

*instructor student*

*eval\_ for*

*proj\_ guide*

**Figure 7.22** E-R diagram with redundant relationships.

*guide* and *evaluation* to represent which (*student*, *project*, *instructor*) combination

an *evaluation* is for. Figure 7.23 shows a notation for aggregation commonly used

to represent this situation

Q6a)4] shadow page recovery

Ans>This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, *n*—for recovery purposes. A **directory**with *n*entries is constructed, where the i th entry points to the i th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory.**The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never*modified. When a write\_item operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten.*Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory, and the new version by the current directory.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle **garbage collection**when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

Q6.5]

Cost based quesy optimization

## Ans>What is cost based optimization? Also, what is the difference between a cost based and rule based optimizer?

As you may already know, a query optimizer is a part of the relational database software which is meant to analyze a SQL query and then figure out what the best to run that query. That is why it is called a query ***optimizer***, because it’s goal is to optimize the query for the sake of efficiency.

**How does a cost based optimizer work?**

|  |
| --- |
|  |

A cost based optimizer will look at all of the possible ways or scenarios in which a query can be executed – and each scenario will be assigned a ‘cost’, which indicates how efficiently that query can be run. Then, the cost based optimizer will pick the scenario that has the least cost and execute the query using that scenario, because that is the most efficient way to run the query.

**Cost based optimizers use statistics from the database**

Cost based optimizers have to use certain statistics that they collect from the database. Some examples of the types of statistics used by cost based optimizers include the number of number of unique values an indexed column has or even just the number of rows in a table. See our article on [selectivity](http://www.programmerinterview.com/index.php/database-sql/selectivity-in-sql-databases/) for more details on how cost based optimizers can use database statistics to make decisions.

**Cost based versus rule-based optimizers**

A rule based optimizer is an optimizer that just applies a set of rules to a SQL statement instead of looking at cost estimates in order to determine what the best way is to execute that SQL statement. Oracle actually allows you to use either the rule based or cost based optimizer, although Oracle says that rule based optimization will be deprecated in a future release, so it highly recommends the use of cost based optimization.