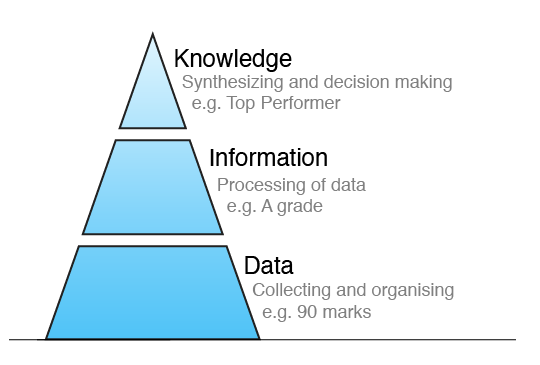
**What is Data?**

Before we understand databases, we need to understand the concept of data. Data is collection of facts about the object of interest. For e.g. data about an employee would include information like name, address, age, educational qualifications etc. Software Applications need to store data as it is required to answer a question e.g. how many employees are above 40 years of age? Data is also required to convey a story e.g. why are we successful as an organization?

Data is raw, just a set of facts which by itself does not convey anything. We need to understand patterns between factual data and give it a meaning. This is called information which helps us with answers to questions like who, when, what, where etc. Synthesis of data and information leads us to answer the how question and take business decisions. This is referred to as Knowledge.



**Requirements From Data :**

Software Applications that use data are expected to meet several requirements from end users. Let us take the example of Facebook application.

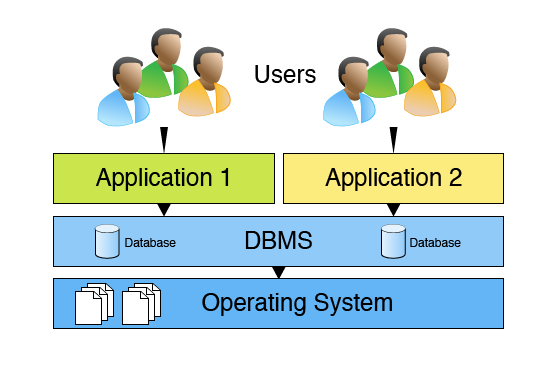
| **Requirement** | **Description** |
| --- | --- |
| Integrity | Data should be accurate e.g. my facebook profile should contain valid country name. |
| Availability | I should be able to access facebook and see my data at all times. |
| Security | Only my friends should be able to see my posts and no one else. |
| Independent of Application | I should be able to access the same data from my Android app as well as from web browser on my laptop. |
| Concurrent | All my friends should be able to see my posts at the same time. |

**What is DBMS ?**

A **Database** is a shared collection of logically related data and description of these data, designed to meet the information needs of an organization

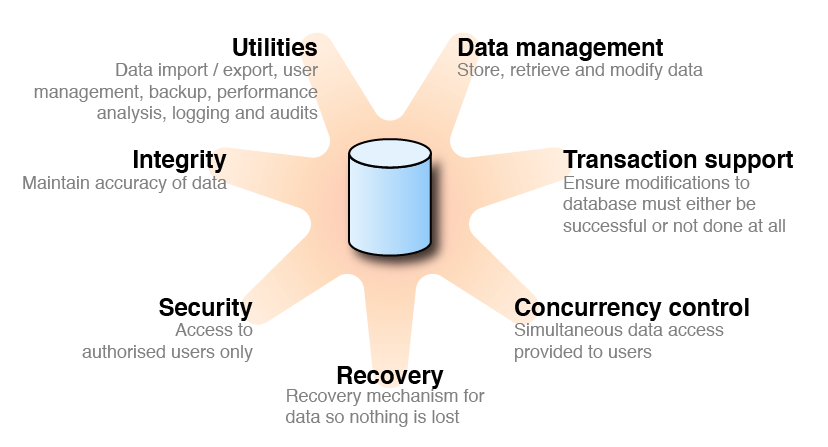
A **Database Management System** is a software system that enables users to define, create, maintain, and control access to the database. Database Systems typically have high cost and they require high end hardware configurations.

An **Application Program** interacts with a database by issuing an appropriate request (typically a SQL statement)



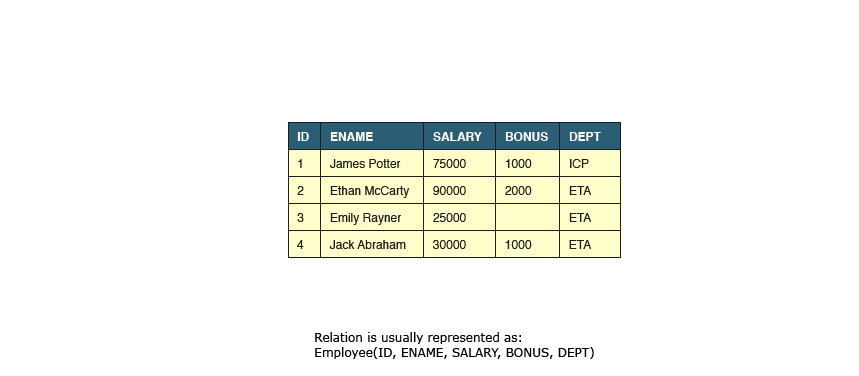
**DBMS Functions :**

Database Management Systems offer several functions that help us overcome problems associated with file based systems. We will focus on Data Management, Integrity, Transaction and Concurrency in this course. Security, Recovery and Utilities are out of scope for this introductory level course.

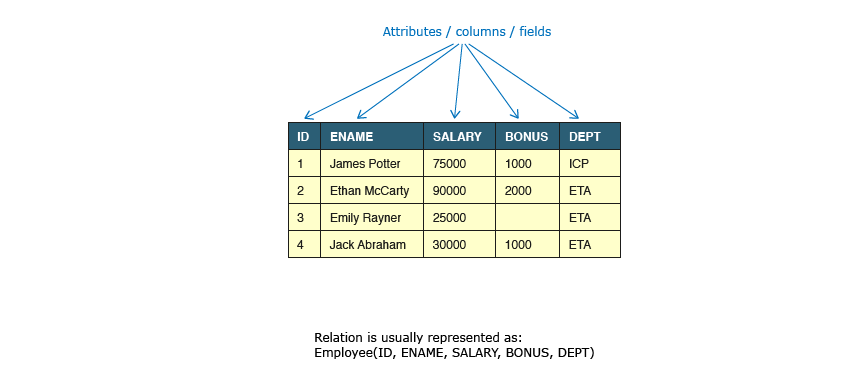


**Relational Database Management System :**

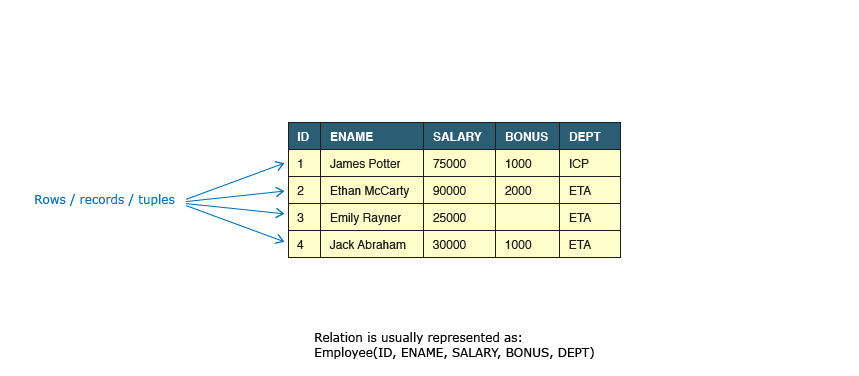
Relational Databases store data in relations i.e. tables. Each relation must have a name.



An attribute is a named column of a relation. It stores a specific information about an object e.g. salary.

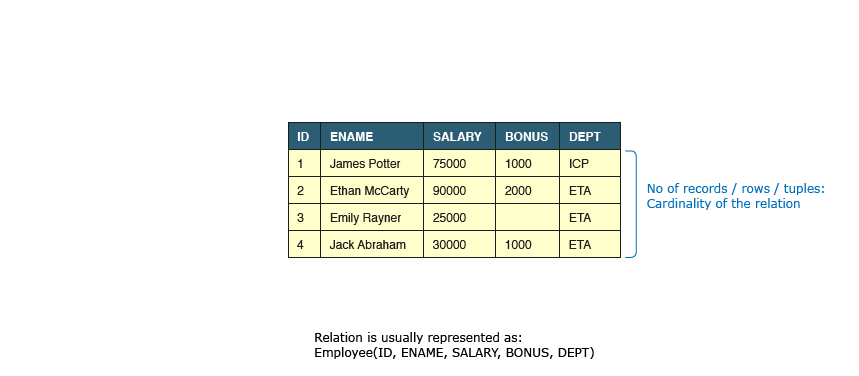


A tuple is a row in a relation. It represents relationship between attributes that can contain single value.

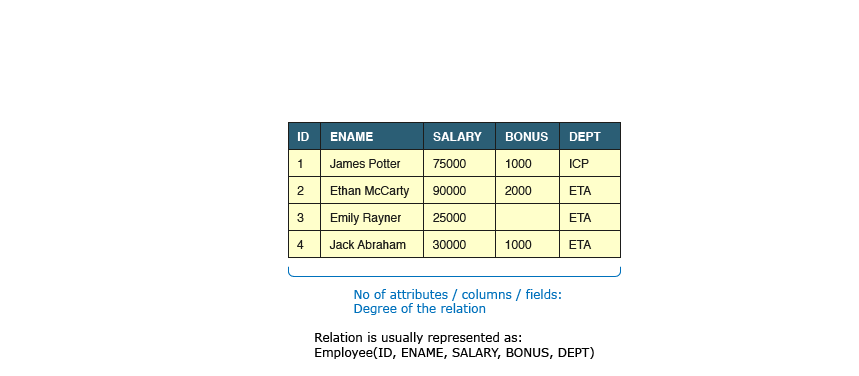


Prev

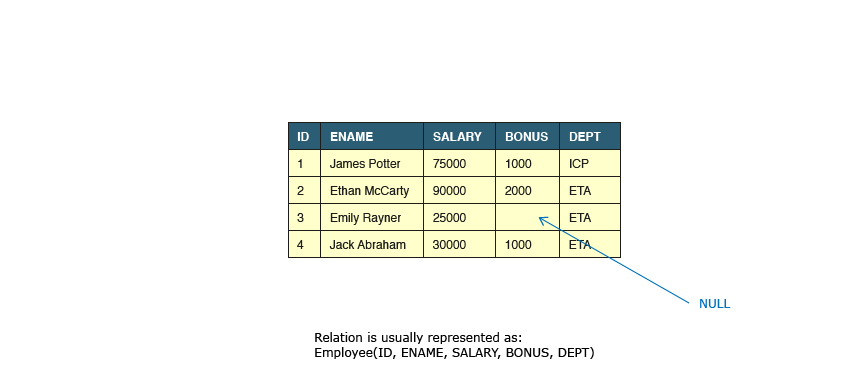
Cardinality of relation is the number of rows it contains. e.g. Cardinality of relation below is 4.



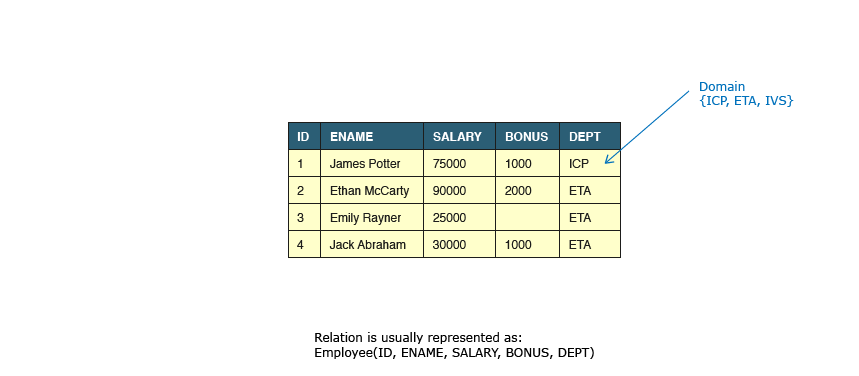
Degree of relation is the number of attributes it contains. e.g. Degree of relation below is 5.



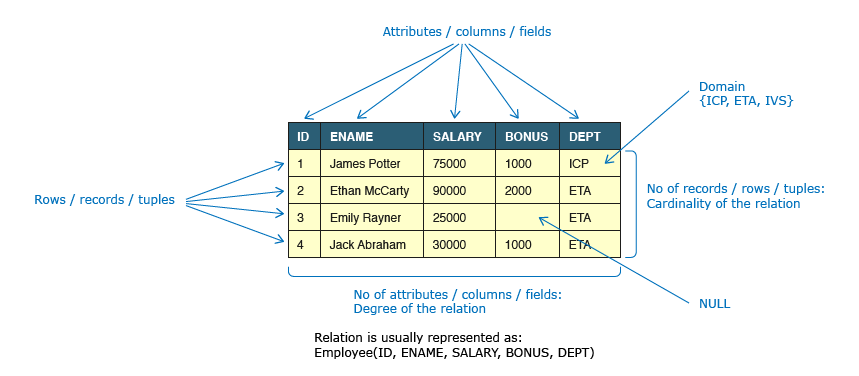
NULL represents the value of an attribute that is currently unknown or not applicable.



A domain is the set of allowable values for one or more attributes.



A collection of relations with distinct relation names is called as Relational Model.



Prev

**Flat Files :**

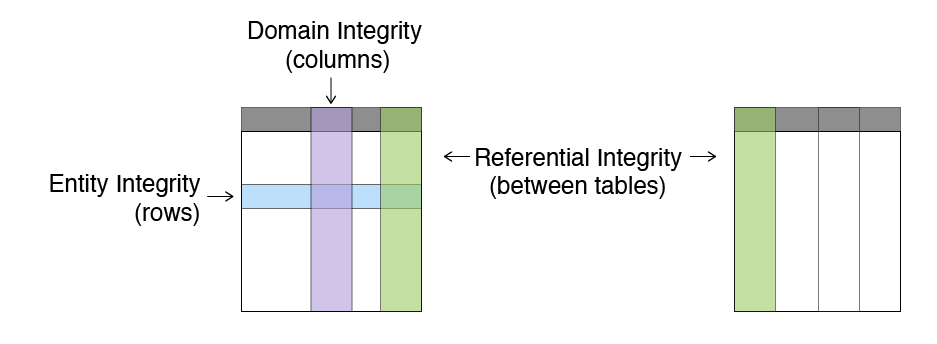
Data is stored in flat files and can be accessed using any programming language. The file based approach suffers following problems:

1. Dependency of program on physical structure of data
2. Complex process to retrieve data
3. Loss of data on concurrent access
4. Inability to give access based on record (Security)
5. Data redundancy

**Data Integrity :**

Data integrity refers to maintaining and assuring the accuracy and consistency of data over its entire life-cycle. Database Systems ensure data integrity through constraints which are used to restrict data that can be entered or modified in the database. Database Systems offer three types of integrity constraints:

| **Integrity Types** | **Definition** | **Enforced Through** |
| --- | --- | --- |
| Entity Integrity | Each table must have a column or a set of columns through which we can uniquely identify a row. These column(s) cannot have empty (null) values. | PRIMARY KEY |
| Domain Integrity | All attributes in a table must have a defined domain i.e. a finite set of values which have to be used. When we assign a data type to a column we limit the values that it can contain. In addition we can also have value restriction as per business rules e.g. Gender must be M or F. | DATA TYPES, CHECK CONSTRAINT |
| Referential Integrity | Every value of a column in a table must exist as a value of another column in a different (or the same) table. | FOREIGN KEY |



**Candidate Key :**

A **Candidate Key** is a minimal set of columns/attributes that can be used to uniquely identify a single tuple in a relation. Candidate Keys are determined during database design based on the underlying business rules of the database. Consider the following relation in context of a business firm:

**Employee(EmployeeNo, Name, AadharNo, Salary, DateOfBirth)**

Let us try to identify some candidate keys for this relation

| **Key** | **Rationale** |
| --- | --- |
| EmployeeNo | This seems to be a good candidate key as companies usually issue a unique number for each employee. |
| AadharNo | This seems to be a good candidate key for a company based in India. However we have assumed that every employee has an Aadhar number which may not be true. In addition for a multinational firm with employees across the globe this will not work at all. |
| Name, DateOfBirth | This might work for a small firm of 10 people as a combination of Name and Date of Birth is likely to be unique. |
| Salary | This is not a good candidate as salary is generally paid uniformly to people at same level. |
| EmployeeNo, DateOfBirth | It is not a candidate key as EmployeeNo alone is unique. By definition only minimal set of attributes can be candidate key. |

Thus the choice of candidate key depends upon the business context

**Primary Key :**

**Primary key** is the candidate key that is selected to uniquely identify a tuple in a relation. The mandatory and desired attributes for a primary key are:

| **Mandatory** | **Desired** |
| --- | --- |
| must uniquely identify a tuple | should not change with time |
| must not allow NULL values | should have short size e.g. numeric data types |

Let us identify the primary key from the previous example:

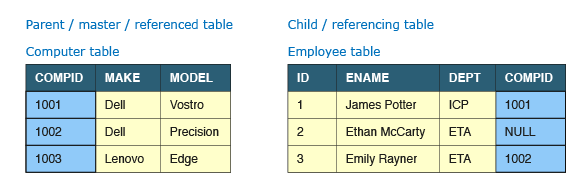
**Employee(EmployeeNo, Name, AadharNo, Salary, DateofBirth)**

| **Key** | **Rationale** |
| --- | --- |
| EmployeeNo | Good candidate as it is numeric, cannot be null and does not change with time. |
| AadharNo | It will be null for people who do not have Aadhar number. Hence it cannot be considered as primary key. |
| Name, DateOfBirth | Both Name and DateOfBirth cannot be null. However even if uniqueness is guranteed, it is not a good choice due to large size. |

When two or more columns together identify the unique row then it's referred to as **Composite Primary Key.** The combination of Name and DateOfBirth if selected as a primary key would be a composite primary key.

**Foreign Key :**

A foreign key is a set of one or more columns in the child table whose values are required to match with corresponding columns in the parent table. Foreign key establishes a relationship between these two tables. Foreign key columns on child tables must be primary key or unique on the parent table. The child table can contain NULL values. Let us take Employee and Computer tables as provided below:



Computer is the parent table with CompId as the primary key. Employee is the child table with Id as the primary key. If we want to allocate a maximum of one computer to an employee then CompId must be made the foreign key in the Employee table. It can only contain values that are present in Computer table or no values at all. We cannot allocate a computer that does not exist to an employee.

Additionally multiple rows in the child table can link to the same row of the parent table dependening upon the type of relationship.

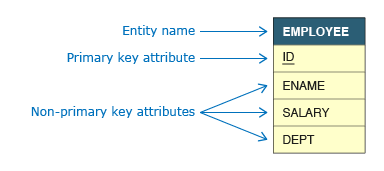
**E-R Diagram :**

Jack is part of database team and he needs to present the database design to business users. The business users are non-technical and it's difficult for them to read a verbose design document. What can Jack do? Jack needs to use an Entity Relation (ER) Model.

**ER model** is a graphical representation of entities and their relationships which helps in understanding data independent of the actual database implementation. Let us understand some key terms used in ER Modelling.

| **Term** | **Definition** | **Examples** |
| --- | --- | --- |
| Entity | Real world objects which have an independent existence and about which we intend to collect data. | Employee, Computer |
| Attribute | A property that describes an entity. | Name, Salary |

A sample ER Diagram representing the Employee entity along with its attributes is presented below:



**What is Relationship?**

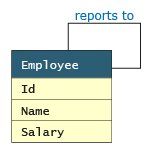
**Relationships** are association of one entity with another entity. Each relationship has a name e.g. a Computer **is allocated to** an Employee.



There can be more than one relationship between entities, e.g. an Employee **works in** a Department while the head of department (also an employee) **manages** a Department.

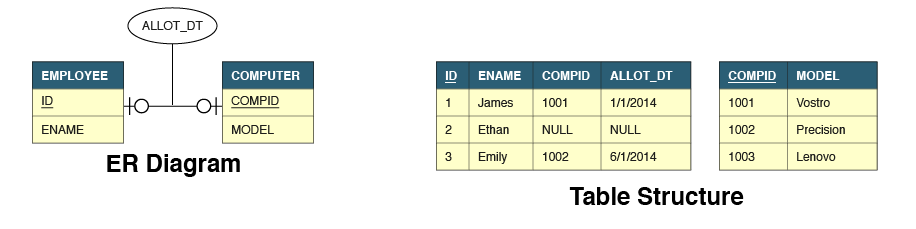


A relationship can also exist between instances of same entity, e.g. an Employee **reports to** another Employee.



**1 :1 Relationship :**

1:1 relationship represents association between single occurrence of one entity and a single occurrence of the second entity. For e.g. consider a company where each employee can be allocated a maximum of 1 computer and computers are not shared between employees.



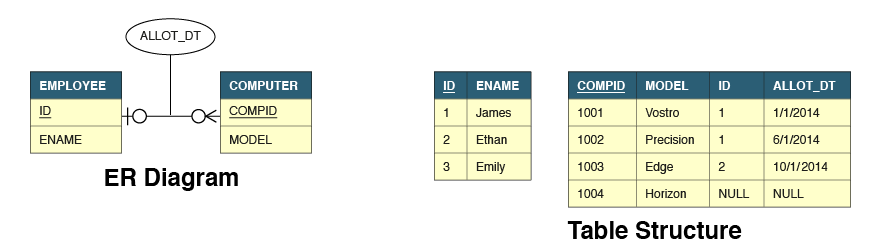
The Allot\_Dt attribute is not a property of employee or computer. It belongs to the relationship and is hence represented differently in the ER Model.

We can see that employee table has two additional attributes - CompId and Allot\_Dt. CompId is a foreign key to establish the link between these two tables. Allot\_Dt which is the attribute of the relationship is always stored in the table that has the foreign key.

Alternatively we could also have added Id and Allot\_Dt attributes in computer table to establish the link.

**1 : N Relationship :**

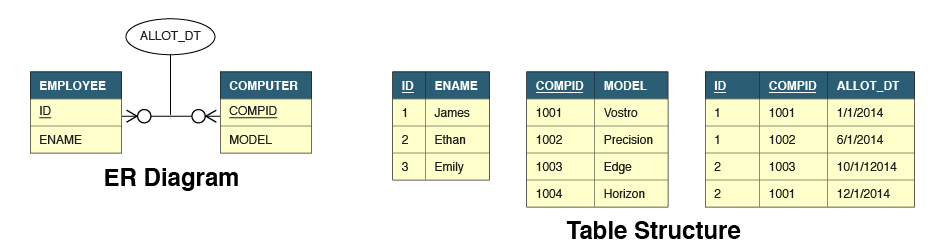
1 : N relationship represents association between single occurrence of one entity and multiple occurrences of second entity. For e.g. consider a company where each employee can be allocated many computers but still computers cannot be shared between employees.



In 1 : N relationships, the foreign key and relationship attributes are always added to the many (N) side of the relationship. Hence these attributes are added to Computer table. The reverse solution will not work.

**Many to Many :**

M:N relationship represents association between multiple occurrences of both entities. For e.g. consider a company where each employee can be allocated many computers and computers can be shared between employees.

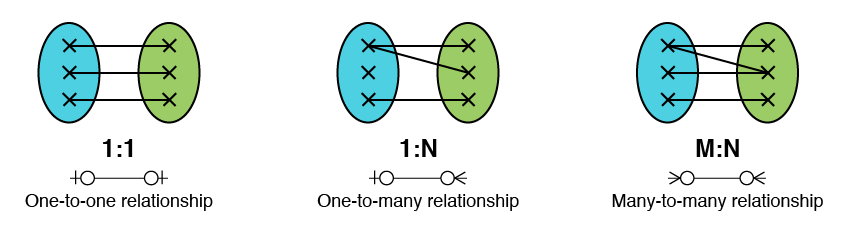


In M : N relationships, the relationship is represented by a completely new table that has a composite primary key. Such a structure requires two foreign keys on the new table linking to the primary keys of each of the parent tables. The attribute of the relationship resides on this new table.

**Cardinality Relationship :**

**Cardinality of relationship** is the number of instances in one entity which is associated to the number of instances in another. For the relationship between Employee and Computer, it helps us answer questions like how many computers can be allocated to an employee, can computers be shared between employees, can employees exist without being allocated a computer etc. e.g. if 0 or 1 computer can be allocated to 0 or 1 employee then the cardinality of relationship between these two entities will be 1:1.

Cardinality of relationships are of three types: 1:1, 1:N and M:N.

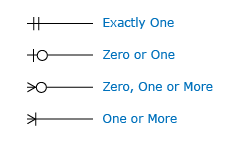


A relationship with cardinality 1:1 is also called as one-to-one relationship or 1:1 relationship.

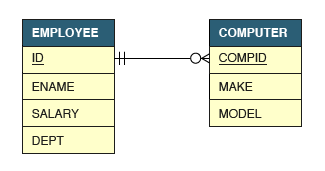
Some entites in the context of Infosys are Employee, Computer, Project, Salaried Account. Can you identify relationships and cardinality for these relationships?

**Crow Foot Relationship :**

**Crow foot notation** is one of the ways to represent cardinality of relationship in an ER Model. The notation comprises of four symbols and one of them need to be used for each entity in a relationship.

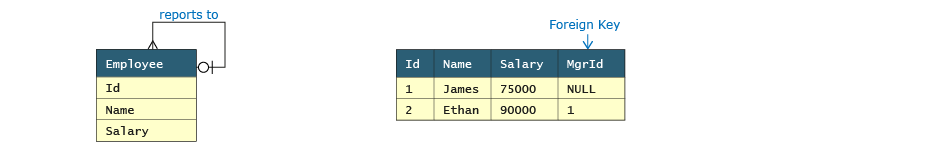
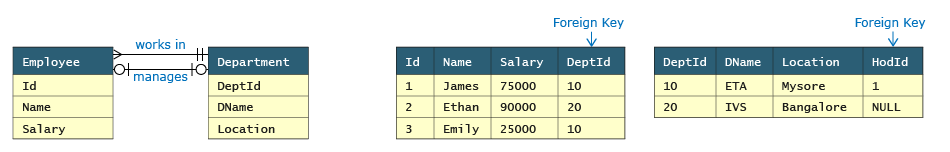
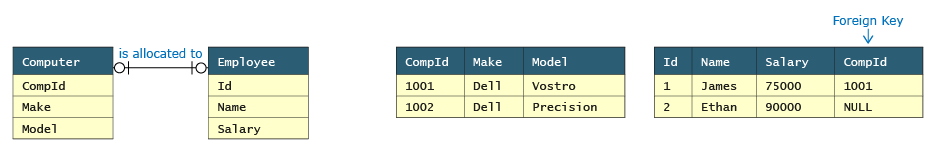


Let us say the relationship between employee and computer is such that a computer must be allocated to one and only one employee but an employee can be allocated zero or any number of computers. Such a relationship is represented by the diagram below:



**Foreign key Relationship :**

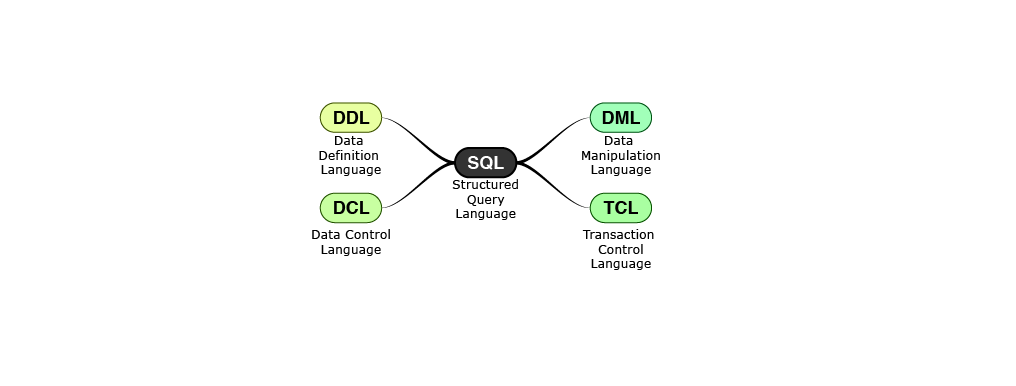
Foreign keys need to be created in tables in order to establish relationship between entities.



The table in which foreign key will be created depends upon the cardinality of relationship. Let us now discuss about types of cardinalities and how it impacts foreign key creation.

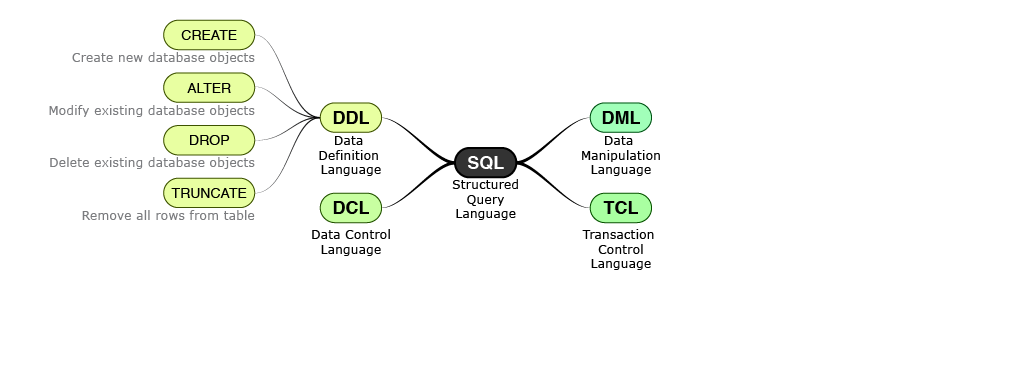
**SQL Query Language:**

Structured Query Language (SQL) is used to manage data in all relational databases like DB2, Oracle, SQL Server etc. SQL standards are maintained by ISO. While most database products comply with the ISO standard, they also offer additional proprietary features. In this course we will restrict ourselves to feature set offered by Oracle database.



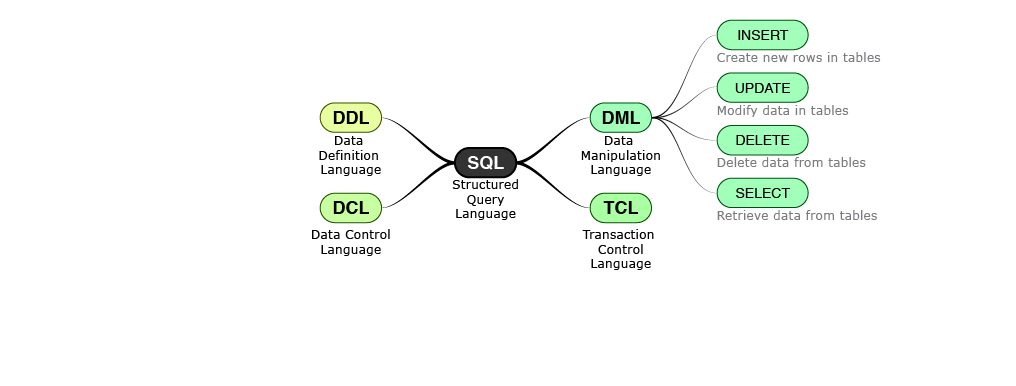
**Data Definition Language :**

Data Definition Language is used to specify the structure i.e. schema of a relational database. DDL provides commands for creation, modification and deletion of various database objects like tables, views, stored procedures, indexes, constraints etc. The output of DDL is placed in data dictionary which contains metadata i.e. data about data.



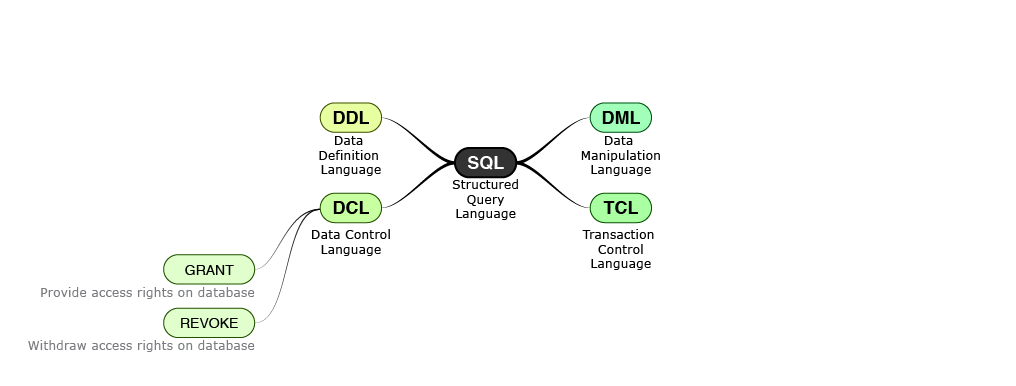
**Data Manipulation Language :**

Data Manipulation Language enables users to access or manipulate data in a relational database. DML provides commands for retrieval, creation, deletion and modification of information in a database. DML requires a user to specify what data is needed without specifying how to get it. The database engine is left to figure out effective means of retrieving data.



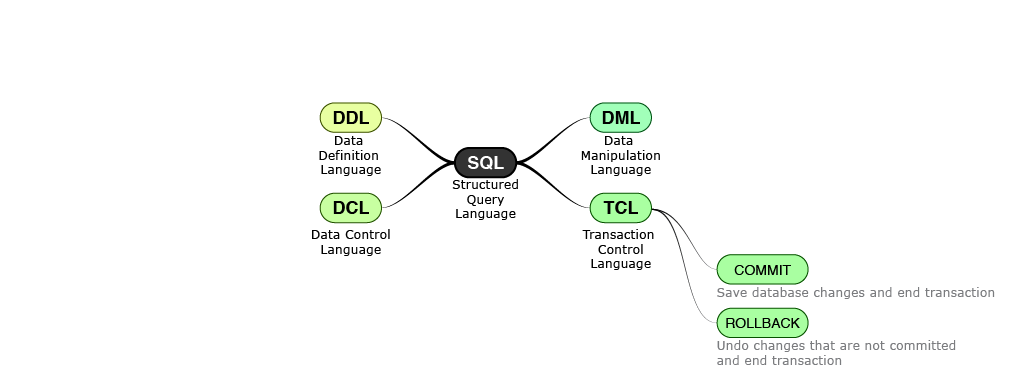
**Data Control Language :**

Data Control Language enables users to provide access to various database objects like views, tables, stored procedures etc. in a relational database. Typically only DBAs have access to grant and revoke privileges. Whenever a user submits a query, the database checks against the granted privileges and rejects the query if it is not authorized.



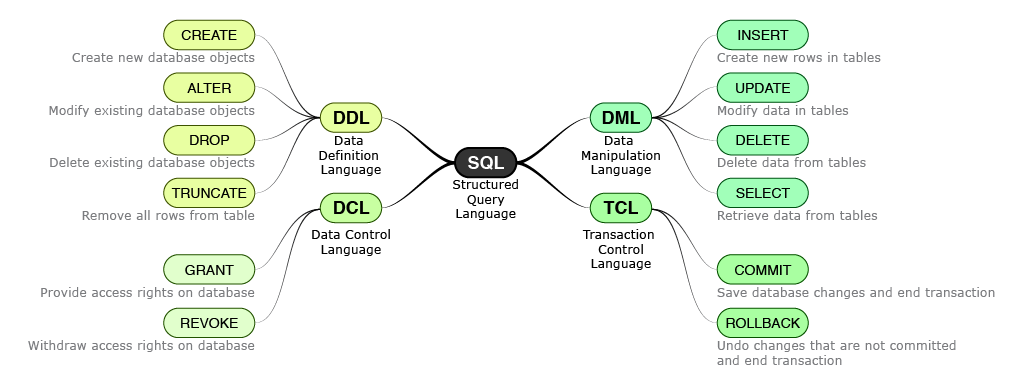
**Transactional Control Language :**

Transaction Control Language specifies commands for beginning and ending a transaction. A transaction consists of a sequence of SQL statements that are applied in an atomic (all or none) manner. A commit makes all the changes applied by the transaction permanent on the database while a rollback undoes all the changed applied by the transaction.



All SQL Basics :

To summarize, Structured Query Language (SQL) provides four types of languages based on type of operation to be performed on a database. These languages can be considered as subsets of SQL and logical groups only. Physically commands for all these languages are executed from the same interface provided by the database.



**What is difference Between Char and Varchar 2?**

SQL supports two character data types for storing printable and displayable characters. They are used for storing information like name, address, description etc.

|  | **CHAR(n)** | **VARCHAR2(n)** |
| --- | --- | --- |
| **Useful for** | Storing characters having pre-determined length | Storing characters whose length vary a lot |
| **Storage size** | size for n characters | size for actual no. of characters + fixed size to store length |
| **Storage Characteristic** | Trailing spaces are applied if data to be stored has smaller length than n. | Trailing spaces are not applied. |
| **Maximum size** | 2000 bytes | 4000 bytes |
| **Example** | A CHAR(10) field will store "Hello" as 10 bytes by appending 5 trailing spaces. | A VARCHAR2(10) field will store "Hello" as 7 bytes (assuming 2 bytes to store length). |
| **Alternate Name** | CHARACTER(n) | CHARACTER VARYING(n) |

**What is mean by Integral Datatypes?**

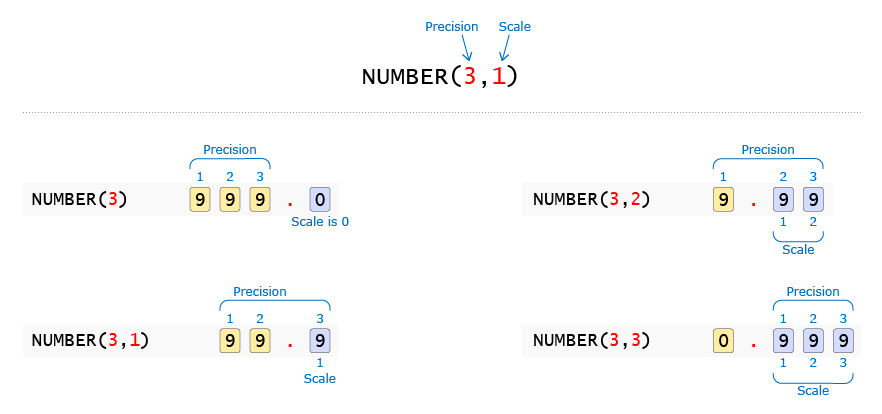
SQL supports SMALLINT, INTEGER and INT data types that are used for storing whole numbers. Unlike other databases Oracle does not define different size limits for them. They are all treated internally to have 38 digit of precision. Some real life examples of values are provided below:

| **Example** | **Value** |
| --- | --- |
| Height of Mount Everest in meters | 8848 |
| Length of Great Wall of China in meters | 885000 |
| Average distance of Earth from the Sun in meters | 150000000000 |

SQL, unlike programming languages, does not provide support for arbitrary length numbers i.e. numbers not bound by size limits. For e.g. Python supports bignum and Java supports BigInteger data types.

**What is mean by Non-Integral Datatypes?**

Nonintegral data types have an integer part and a fractional part. Either NUMERIC, DECIMAL or NUMBER data types can be used to store nonintegral numbers.



**Scale** is the number of digits allowed after the decimal point. **Precision** is the total number of significant digits i.e. digits both before and after the decimal point. If Scale is not provided then NUMBER datatype can be used to store integral values.

**What is mean by Miscellaneous data types?**

SQL supports following data types for representing date and large objects:

| **Data Type** | **Useful for** |
| --- | --- |
| DATE | Storing date data where time portion is not required. For e.g. Date of Birth, Date of Joining a Company etc. The default format in which date needs to be specified is DD-MON-YY. |
| TIMESTAMP | Storing date data with precision up-to 1 billionth (9 digits) of a second. Timestamps are typically used as audit fields in database to record the exact time when a transaction occurred. |
| CLOB (Character Large Object) | Storing large character based data which cannot be stored in VARCHAR2 due to its 4000 bytes size limit. |
| BLOB (Binary Large Object) | Storing large binary data like movies, images with size up to 4GB. |

Here are some key events and their dates from history:

| **Data Type** | **Example** | **Value** |
| --- | --- | --- |
| DATE | Date Infosys was founded | 02-JUL-81 |
| TIMESTAMP | Date Apollo 11 landed on the moon | 20-JUL-69 08:18:00.000000 PM |
|  |  |  |

**Different Operators of SQL?**

**Arithmetic Operators**

| **Operator** | **Symbol** | **Usage** | **Result** |
| --- | --- | --- | --- |
| Addition | + | 15 + 5 | 20 |
| Subtraction | - | 15 - 5 | 10 |
| Multiplication | \* | 15 \* 5 | 75 |
| Division | / | 15 / 5 | 3 |

**Comparison Operators**

| **Operator** | **Symbol** | **Usage** | **Result** |
| --- | --- | --- | --- |
| Equal to | = | 15 = 5 | false |
| Not equal to | <> | 15 <> 5 | true |
| Greater than | > | 15 > 5 | true |
| Greater than equal to | >= | 15 >= 5 | true |
| Less than | < | 15 < 5 | false |
| Less than equal to | <= | 15 <= 5 | false |

There is one important difference between Equal To comparison operator in programming languages and SQL. While SQL uses a single '=', programming languages typically use double '=' to distinguish it from the assignment operator.

| **Operation** | **Python Operator** | **SQL Operator** |
| --- | --- | --- |
| Assignment | = | = |
| Equality check | == | = |

**Logical Operators :**

**Other Comparison Operators**

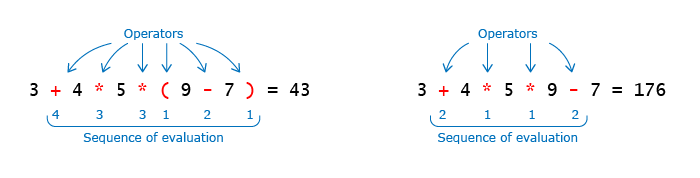
| **Operator** | **Symbol** | **Usage** | **Example** |
| --- | --- | --- | --- |
| Range | BETWEEN <*lower limit*> AND <*upper limit*> | Matches value between a range of values (Both inclusive) | Salary BETWEEN 2500 AND 3000 |
| List | IN (List of values) | Matches any of a list of values | Dept IN ('IVS', 'ETA', 'ICP') |
| String pattern matching | LIKE | Matches a character pattern | SupplierId LIKE 'S%' |
| NULL Test | IS NULL | Is a null value | Bonus IS NULL |

**Logical Operators**

| **Operator** | **Symbol** | **Usage** | **Example** |
| --- | --- | --- | --- |
| And | AND | Returns TRUE if both conditions are true | Salary >= 30000 AND Dept = 'ETA' |
| Or | OR | Returns TRUE if any one of the condition is true | Salary > 75000 OR Dept = 'ICP' |
| Not | NOT | Returns TRUE if following condition is false | Id NOT IN (2,3) |

**Expressions :**

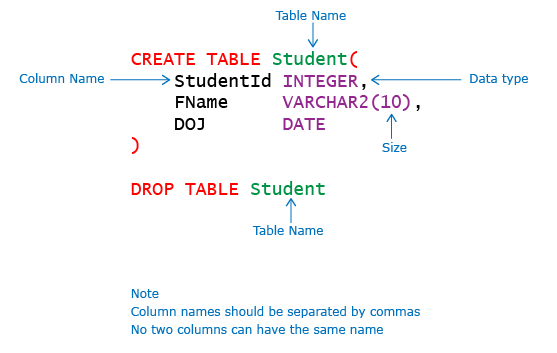
Similar to arithmetic expressions in programming languages, SQL expressions are created from constant values, operators and brackets. They evaluate to a single value and are used in SELECT and WHERE clauses. Some examples are provided below:



**Create Table and Drop Table in SQL**

**CREATE TABLE** statement is used to create a table in a database. Database tables are organized into rows and columns. Each table must have a name and can have any number of columns (minimum 1 column is required). Each column must have a data type which determines the type of values that can be stored. CREATE TABLE command will fail if a table is already existing in the database with same name. All tables must have a unique name.

**DROP TABLE** statement is used to remove an existing table from the database.



**Create table Example :**

We will now look at some common errors that occur while creating tables. The statement below has several syntax errors. Let us resolve these errors step by step:

1 CREATE TABLE Student (

2 StudentId INTEGER

3 Gender CHAR(1),

4 FName VARCHAR2,

5 DOJ,);

**Constraints of SQL:**

We have learnt that data integrity in database systems is enforced through constraints. These constraints are typically specified along with the CREATE TABLE statement. Constraints are classified into multiple types based on the number of columns they act upon as well as on the way they are specified.

| **Constraint Type** | **Applies On** |
| --- | --- |
| Single Column Constraint | Single Column |
| Composite Constraint | Multiple columns |
| **Constraint Type** | **Specified** | |
| Column Level Constraint | With Column definition | |
| Table Level Constraint | After Column definition | |

Table level constraint can be specified after all columns used in the constraint have been defined. It is not necessary to specify them after all columns in the table are defined. Composite constraints can only be specified as table level constraints.

Various constraints that can be created on database tables are:

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

We can also specify DEFAULT value for a column. Oracle database does not consider DEFAULT as a constraint.

**Not Null Constraint :**

NOT NULL Constraint prevents a column from accepting NULL values. NOT NULL can only be applied as a column level constraint. Constraint name is optional and it can be specified by using CONSTRAINT keyword.

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/84#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/84#Step2)

Let us now create NOT NULL constraint on StudentId and FName columns.

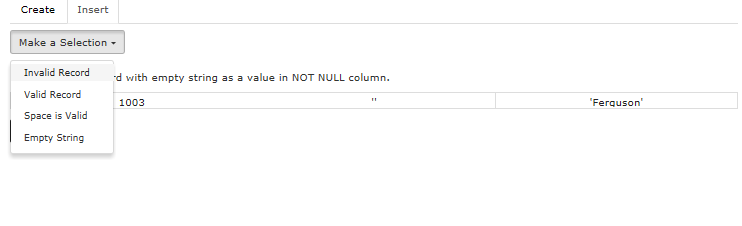
CREATE TABLE Student (

StudentId INTEGER CONSTRAINT Stud\_SId\_nn NOT NULL,

FName VARCHAR2(10) NOT NULL,

LName VARCHAR2(10));

**Errors While Inserting in Not Null :**



**Default Constraint :**

A column can be given the default value by using DEFAULT option. The data type of column and default expression must be the same. DEFAULT option can be provided for nullable as well as NOT NULL attributes. Oracle database does not consider DEFAULT as a constraint.

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/85#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/85#Step2)

Let us create Student table with current date as default for date of joining.

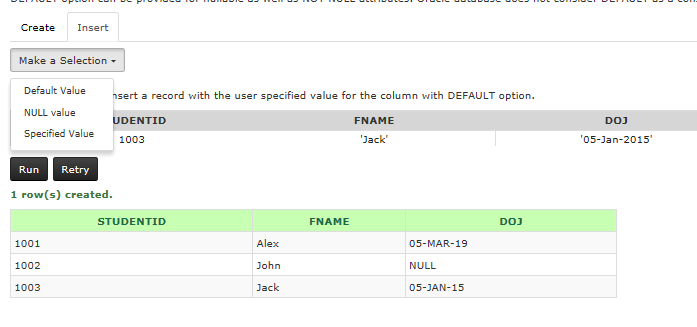
CREATE TABLE Student (

StudentId INTEGER,

FName VARCHAR2(10),

DOJ DATE DEFAULT SYSDATE);

**Different Examples :**



**Primary Key Constraint :**

PRIMARY KEY constraint on a column ensures that the column cannot contain NULL and duplicate values. We can have only one PRIMARY KEY in a table.

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/86#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/86#Step2)

We will now create a Student table with primary key constraint on StudentId column.

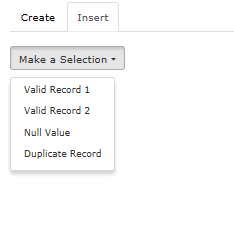
CREATE TABLE Student (

StudentId INTEGER CONSTRAINT stud\_sid\_pk PRIMARY KEY,

FName VARCHAR2(10),

ContactNo NUMBER(10));

**Example :**



**Check Constraint Examples :**

CHECK constraint is used to limit the values that can be specified for a column.

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/89#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/89#Step2)

Let us create a CHECK constraint on Gender column to only allow 'M' (Male) and 'F' (Female) values.

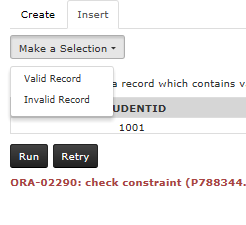
CREATE TABLE Student (

StudentId INTEGER,

FName VARCHAR2(10),

Gender CHAR(1) CONSTRAINT Stud\_gender\_ck1 CHECK(Gender IN('M', 'F')));

**Examples :**



**Unique Constraint Examples :**

UNIQUE constraint on a column ensures that two rows in a table cannnot have same value in that column. Unlike Primary Key, UNIQUE constraint allows NULL values. A table can have many UNIQUE constraints.

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/90#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/90#Step2)

Let us create Unique constraint on ContactNo so that two student cannot have the same contact details.

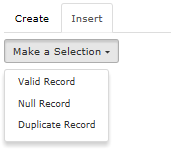
CREATE TABLE Student (

StudentId INTEGER,

FName VARCHAR2(10),

ContactNo NUMBER(10) CONSTRAINT Stud\_cno\_uk UNIQUE);

**Examples :**



**Foreign Key :**

**Student Table** has already been created and inserted with few records in the database.

| **STUDENTID** | **FNAME** | **CONTACTNO** |
| --- | --- | --- |
| 1001 | John | 8754212356 |
| 1002 | Jack | 7456878956 |

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/91#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/91#Step2)

Let us now create a table Marks which stores marks scored by each Student in each Course

CREATE TABLE Marks(

CourseId INTEGER,

StudentId INTEGER,

MarksScored DECIMAL(5,2));



Run

**Table created.**

| **NAME** | **Null?** | **TYPE** |
| --- | --- | --- |
| COURSEID |  | NUMBER |
| STUDENTID |  | NUMBER |
| MARKSSCORED |  | NUMBER(5,2) |

**Need for Composite Primary Key :**

**Student Table** has already been created and inserted with few records in the database.

| **STUDENTID** | **FNAME** | **CONTACTNO** |
| --- | --- | --- |
| 1001 | John | 8754212356 |
| 1002 | Jack | 7456878956 |

* [Create](https://lex.infosysapps.com/fastrack/generic/dbms/page/96#Step1)
* [Insert](https://lex.infosysapps.com/fastrack/generic/dbms/page/96#Step2)

Let us now create a table Marks which stores MarksScored by each Student in each Course

CREATE TABLE Marks (

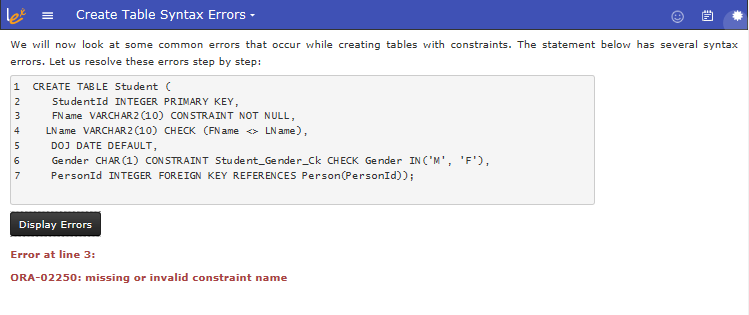
CourseId INTEGER CONSTRAINT marks\_cid\_pk PRIMARY KEY,

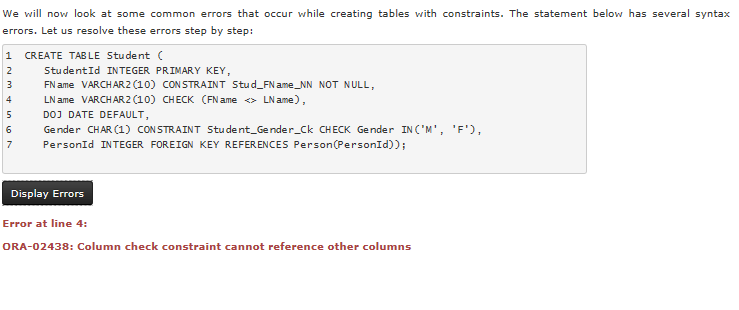
StudentId INTEGER CONSTRAINT marks\_sid\_fk REFERENCES Student(StudentId),

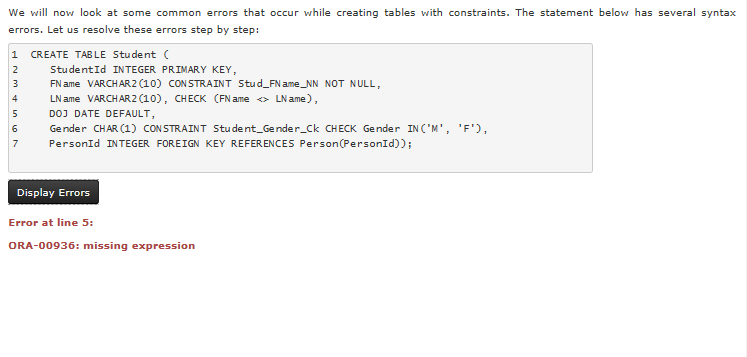
MarksScored DECIMAL(5,2));

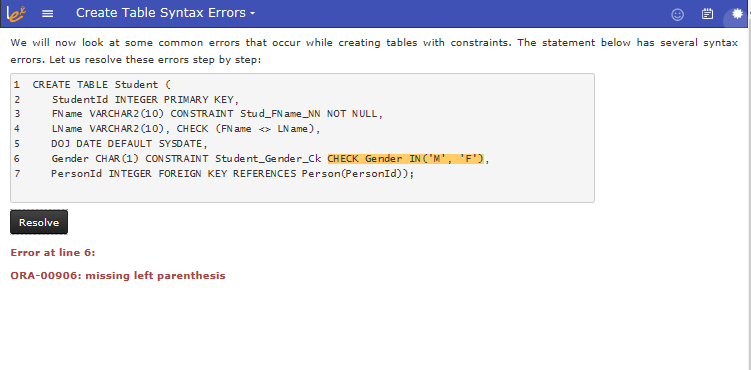
**Different Errors OF Create Table :**

**Error 1 : Missing Right Paranthesys**

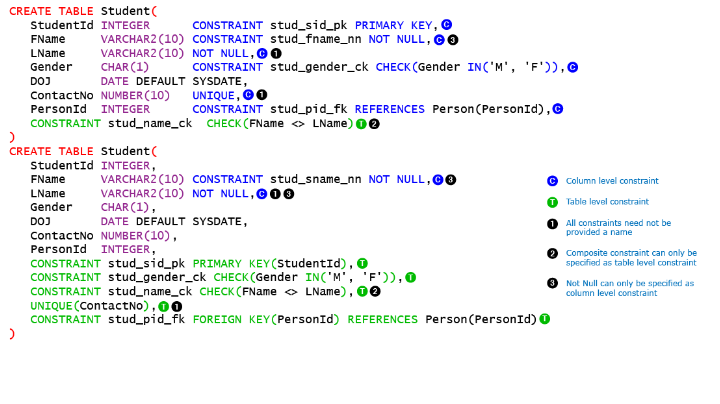








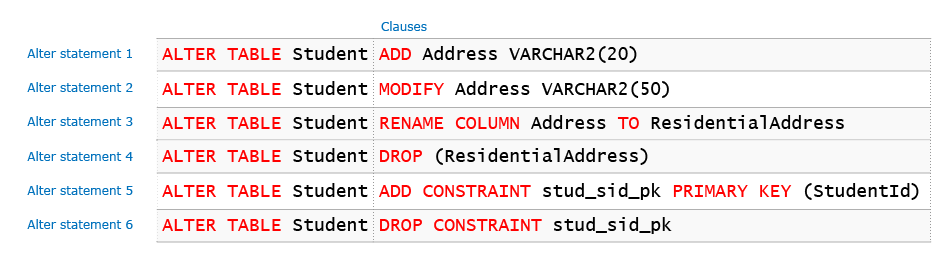
**Constraint Summary :**



**Alter Table Statements :**

How do we add a new column to a table that already exists. One option is to drop the table and create it again. What if the table has lot of data and we do not want to lose existing data? We cannot drop the table now. One option is to take a backup of data, then drop and recreate the table and finally load the data from backup into the modified table. This approach might work but if number of rows are large then this process will take lot of time. Is there an easier way to avoid all these troubles?

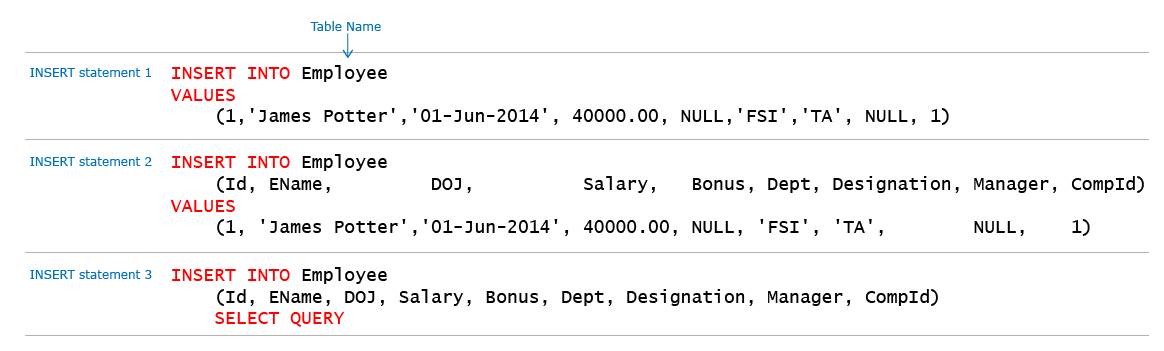
We need to use **ALTER TABLE** command through which the structure of existing table can be changed without any loss of data. ALTER table can also be used to rename a column, change data type of a column and add or remove constraints. Syntax for ALTER TABLE command is provided below:



We will now understand the first four syntax of Alter Command. Adding and dropping constraints is out of scope for the current course.

**Insert Statements :**

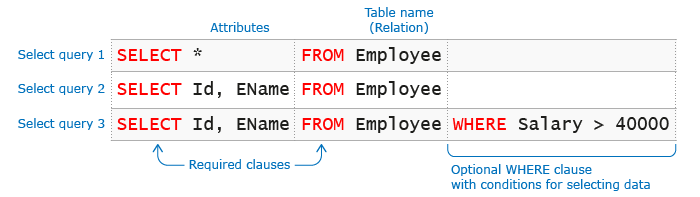
Insert statement is used to add tuples (records) to table. It supports three alternate syntax as shown below:



If column names are not used then values must be provided for all columns in the order of their specification during table creation. If Column names are used then the data provided in values clause must have same data type of column at same position. Multiple rows can be inserted through a single INSERT statement only when it is used with SELECT statement.

**Select Statements :**

SELECT query allows us to retrieve data from one or more tables in a relational database. It can be represented as:



Let us understand some illustrative algorithms for fetching data from the database.

**Query**

SELECT \*

FROM Employee

**Illustrative Algorithm to fetch data**

for each tuple t in Employee

Add t to the result

SELECT \*

FROM Employee

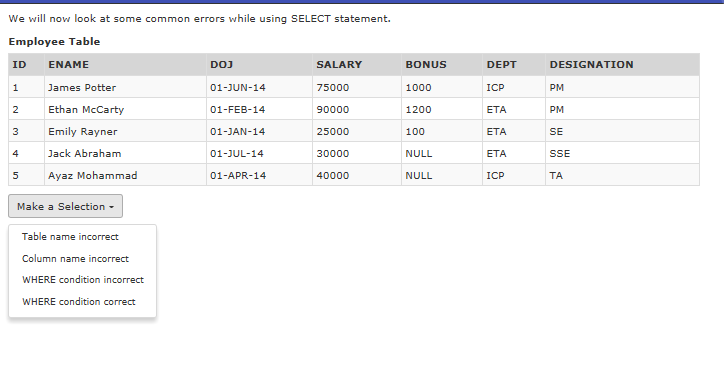
WHERE Salary > 40000;

for each tuple t in Employee

if Salary > 40000

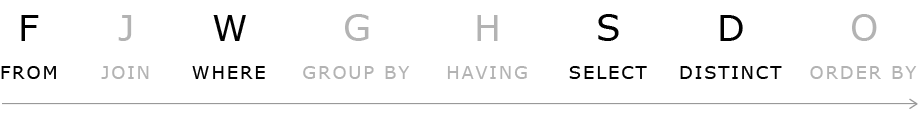
Add t to the result

**Errors Of Select Statement :**



**Query Execution Order :**

A SELECT statement can have many clauses so it is important to understand the order in which these are executed to provide the result. However, for ease of understanding we can refer to the execution order by FJWGHSDO.



A quick way to remember this is to use the mnemonic "**F**rank **J**ohn's **W**icked **G**rave **H**aunts **S**everal **D**ull **O**wls". In this section we will focus on FROM, WHERE, SELECT and DISTINCT keywords.

The first step is always the FROM clause as we need to identify the tables from which data has to be fetched.

SELECT must be always be executed after the WHERE clause, e.g. we can have a query

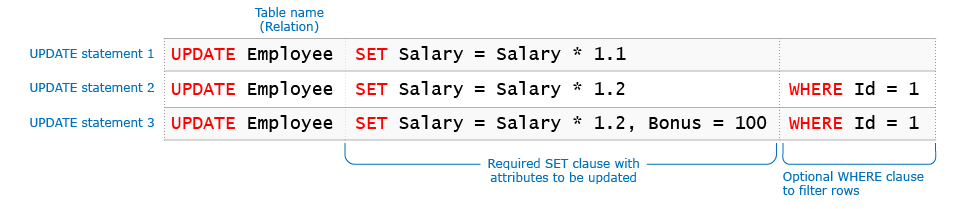
SELECT EName FROM Employee WHERE Id = 1.

Here the filtering needs to happen on a Id column which is not included in the SELECT clause. Unless SELECT executes after WHERE, this functionality cannot be supported.

DISTINCT removes duplicates based on all columns of the SELECT clause. These columns could be a subset of all columns of the table OR may even contain derived columns through use of expression. Thus DISTINCT is dependent on SELECT clause and it's execution must happen after SELECT clause.

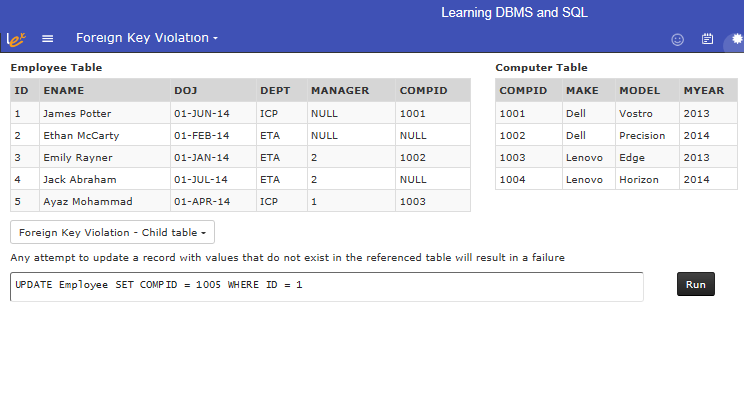
**Updating the data :**

Update statement is used to modify existing records in a single table in a relational database. Update statement can be represented as:

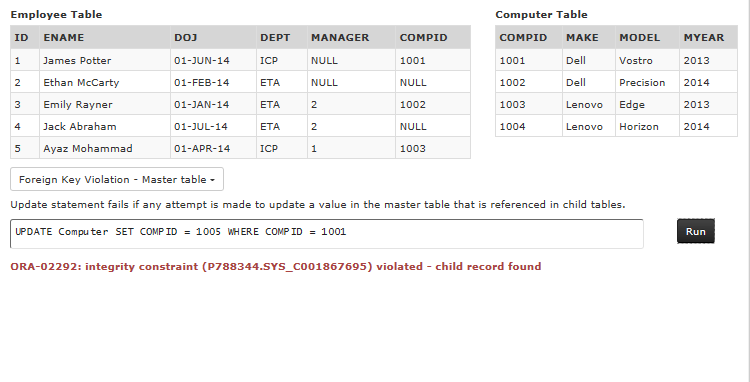


The database system ensures that no constraints are violated during execution of an update statement. Any violation of constraints results in failure of the statement.

**Foreign Key Violation :**

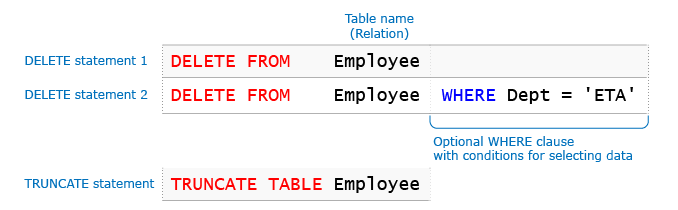






**Delete Statement :**

DELETE statement is used to delete records from a single table in a relational database. The database system ensures that no constraints are violated during execution of a delete statement. Any violation of constraints results in failure of the statement.



TRUNCATE statement can also be used to delete data from tables. TRUNCATE statement deletes all rows from the table as it does not support WHERE clause. TRUNCATE statement is a faster option compared to DELETE when you have to delete all rows from the table.

**Error Codes :**

Whenever you perform a database operation for e.g. insert, update etc. you get a status code from the server indicating whether the operation was successful or not and the reason for the failure if any. In addition a descriptive message is also provided. Let us quickly recap on some of the codes you have encountered so far.

| **Status Code** | **Message** | **Cause** |
| --- | --- | --- |
| ORA-00000 | Successful Completion | The statement was executed successfully. |
| ORA-00001 | Unique constraint violation | Trying to insert a duplicate value or Updating the key value column violating unique constraint. |
| ORA-00904 | Invalid Identifier | The sql statement might be executed on an invalid column or missing column. |
| ORA-0913 | Too Many Values | More values might be passed than expected. Usually happens in insert statement. |

These status codes and messages are published in a reference document. You can refer the link below for Oracle database.

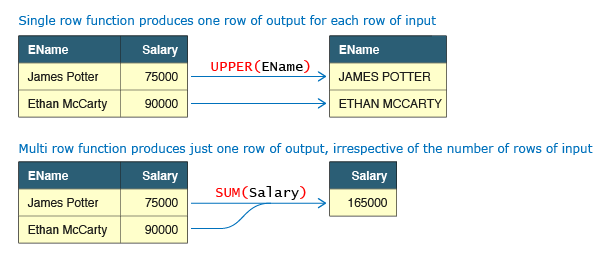
**Difference Between Single Row and Multiple Row Functions :**

Select both Batch and Section

SQL functions are built in modules provided by a database. You can use them in data manipulation statements to perform calculations on data.

All functions return a single value. They are categorized into two types based on number of rows they operate upon.

|  | **Single Row Function** | **Multi Row Function** |
| --- | --- | --- |
| Returns | Single Row | Single Row |
| Operates On | Single Row | Multiple Rows |
| Used in Clauses | SELECT, WHERE, ORDER BY and HAVING | SELECT, ORDER BY and HAVING clauses |



**Numeric Functions :**

Numeric functions are single row functions that accept a numeric value and return numeric output.

| **Name** | **Syntax** | **Function** |
| --- | --- | --- |
| ABS | ABS(value) | Returns absolute value of a number |
| ROUND | ROUND(value, digits) | Rounds the value to specified decimal digits |
| CEIL | CEIL(value) | Rounds up the fractional value to next integer |
| FLOOR | FLOOR(value) | Rounds down the fractional value to the lower integer |

Make a Selection

* [Ceil, Floor and Abs](https://lex.infosysapps.com/fastrack/generic/dbms/page/219)
* [Round](https://lex.infosysapps.com/fastrack/generic/dbms/page/219)



**Character Functions :**

Character functions work on character strings and can return a character string or a numeric value.

| **Name** | **Syntax** | **Function** |
| --- | --- | --- |
| UPPER | UPPER(value) | Converts value to upper case |
| LOWER | LOWER(value) | Converts value in lower case |
| CONCAT | CONCAT(value1, value2) | Concatenates value1 and value2 |
| LENGTH | LENGTH(value) | Returns the number of characters in value. |

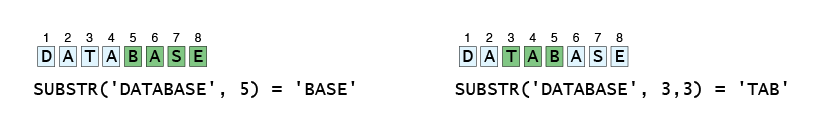
Length, Upper and Lower

* [Length, Upper and Lower](https://lex.infosysapps.com/fastrack/generic/dbms/page/220)
* [Concatenating String data](https://lex.infosysapps.com/fastrack/generic/dbms/page/220)



**Substring Function :**

Substring function is used to extract part of a string. It has the following syntax SUBSTR(value, start\_position, length)



Substring 1

* [Substring 1](https://lex.infosysapps.com/fastrack/generic/dbms/page/223)
* [Substring 2](https://lex.infosysapps.com/fastrack/generic/dbms/page/223)



**Conversion Functions :**

Use conversion functions to convert data from one format to another.

| **Name** | **Syntax** | **Function** |
| --- | --- | --- |
| TO\_CHAR | TO\_CHAR(value, format) | Converts a number or a date to a string. Use this function for formatting dates and numbers. |
| TO\_DATE | TO\_DATE (value, format) | Converts a string to a date. |
| TO\_NUMBER | TO\_NUMBER (value, format) | Converts a string to a number. |

**SELECT '01-Jan-2014' DATE\_STRING, TO\_DATE('01-Jan-2014') CONV\_NOFORMAT, TO\_DATE('01-Jan-2014', 'DD-Mon-YYYY') CONV\_FORMAT FROM DUAL;**

**Date Functions :**

Database provides functions to determine the current time and to perform date operations like adding a specific duration to a date, finding time difference between two dates etc.

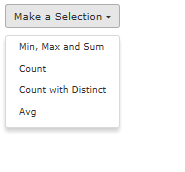
| **Name** | **Syntax** | **Function** |
| --- | --- | --- |
| SYSDATE | SYSDATE | Returns current date of System i.e. the host on which database server is installed. |
| SYSTIMESTAMP | SYSTIMESTAMP | Returns current timestamp of the System. |
| ADD\_MONTHS | ADD\_MONTHS(date, n) | Adds n months to the given date. |
| MONTHS\_BETWEEN | MONTHS\_BETWEEN(date1,date2) | Finds difference between two dates in months. |

**Aggregate Functions :**

Aggregate functions operate on multiple rows to return a single row. Some aggregate functions like SUM (total), AVG (average) operates only on numeric columns while others like MIN (lowest value), MAX (highest value) and COUNT (number of rows) operate on all data types. All aggregate functions ignore NULL values except COUNT(\*).

Employee Table

| **ID** | **ENAME** | **SALARY** | **BONUS** | **DEPT** |
| --- | --- | --- | --- | --- |
| 1 | James Potter | 75000 | 1000 | ICP |
| 2 | Ethan McCarty | 90000 | 1200 | ETA |
| 3 | Emily Rayner | 25000 | 100 | ETA |
| 4 | Jack Abraham | 30000 | NULL | ETA |
| 5 | Ayaz Mohammad | 40000 | NULL | ICP |

*  [Min, Max and Sum](https://lex.infosysapps.com/fastrack/generic/dbms/page/228)
* [Count](https://lex.infosysapps.com/fastrack/generic/dbms/page/228)
* [Count with Distinct](https://lex.infosysapps.com/fastrack/generic/dbms/page/228)
* [Avg](https://lex.infosysapps.com/fastrack/generic/dbms/page/228)

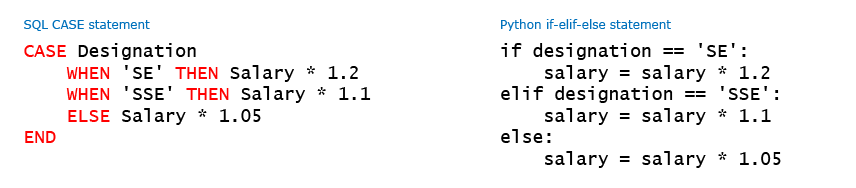
**Other Functions :**

| **Name** | **Syntax** | **Function** |
| --- | --- | --- |
| NVL | NVL(value1, value2) | Substitutes value1 by value2 if value1 is NULL. The data type of value1 and value2 must be same. |
| USER | USER | Returns the current logged in user |

**Case Statement :**

**CASE** statement can be used in SELECT clause to conditionally assign values to a computed attribute. It can also be used with WHERE, GROUP BY etc. CASE statement has two different syntax styles: Simple CASE expression and Searched CASE expression. Both of them are similar to **if else** statement in Python.

(1) Simple CASE expression :

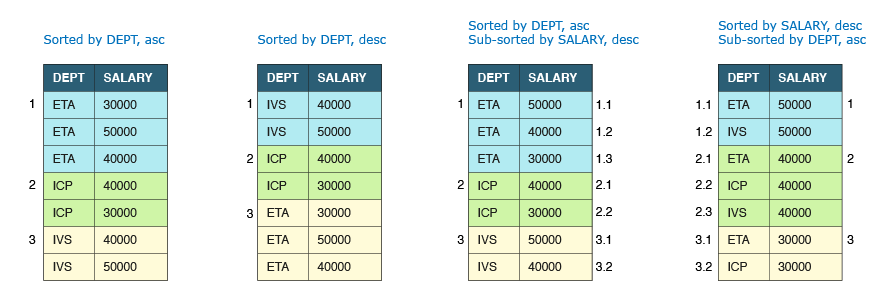


We can use the first syntax form when all conditions check for equality against a single column.

**Order By Clause :**

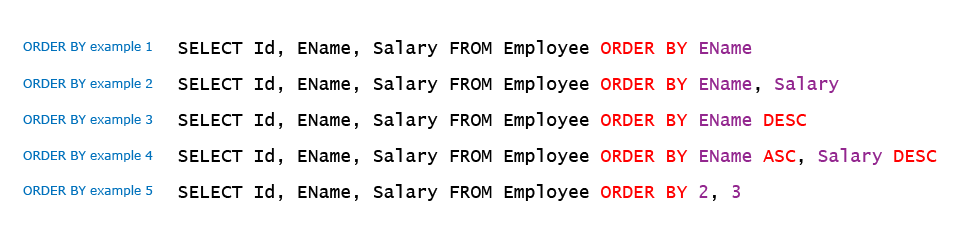
Order By clause is used to sort the result of a query in a particular order. Before we understand the syntax of ORDER BY, let us understand the sorting process on tabular data. We all know that data in a single column can be sorted in ascending or descending order.

We can also sort data by multiple columns. In such a case data is sorted on primary (first) column first. Sorting on secondary column happens only when multiple rows have the same value in the primary column. The sort order can be different for the two columns i.e. primary can be sorted in ascending and secondary in descending and vice-versa. This two column sorting mechanism can be extended to any number of columns.



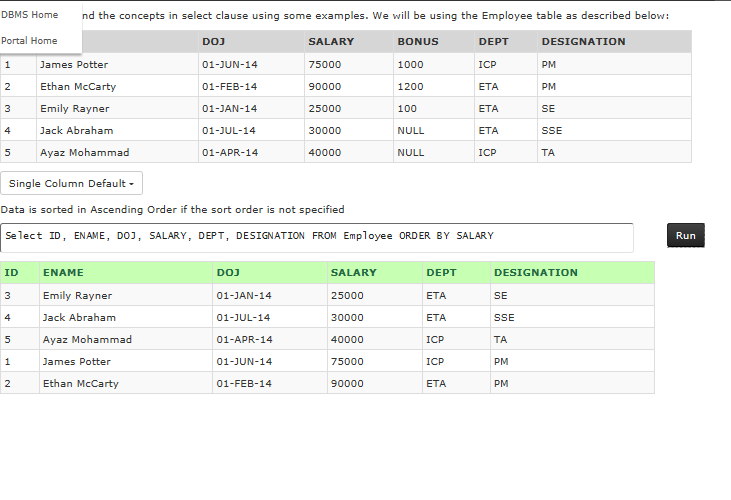
**Order By Syntax :**

ORDER BY must be used to specify the columns on which data has to be sorted and the sort order i.e. ascending or descending. Rows are sorted in ascending order if sort order is not specified. DESC should be used to sort the rows in descending order. ORDER BY clause must be the last clause and can be used only in SELECT statement. The sort order only applies to the query result i.e. the underlying data in the table is not sorted.

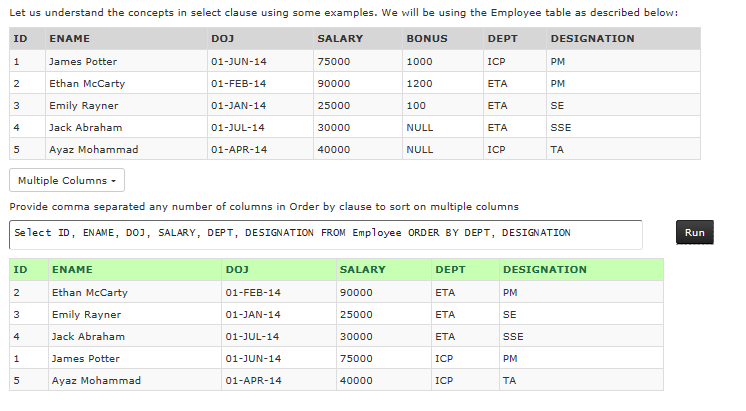


An important point to note is that sorting of rows in the result is guranteed only on columns specified in the ORDER BY clause. If multiple rows have same value for columns in ORDER BY clause then they can appear in any order.

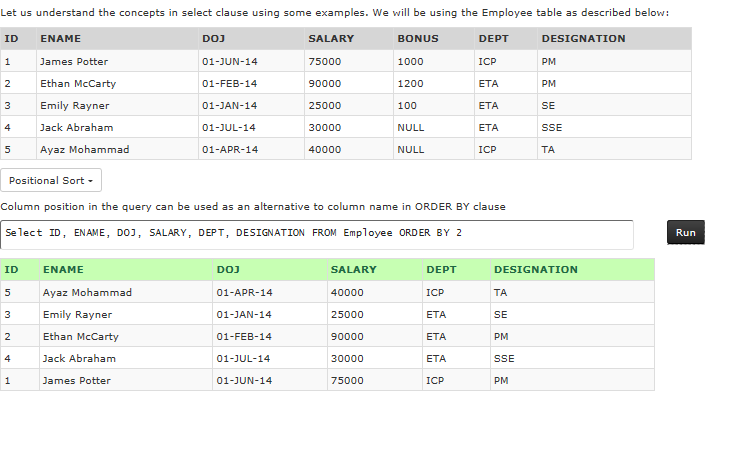
**Example 1 :**



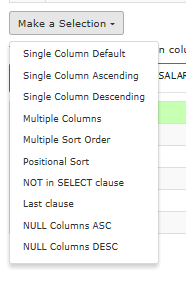
**Example 2 :**



**Example 3 :**



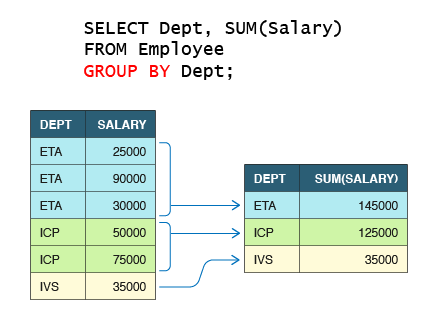
**Sorting Techniques :**



**Group By Clause:**

Earlier we have seen aggregate functions being used to calculate min, max, avg etc. for all records of the query. What if the requirement is to calculate subtotals at Department level? In that case we will have to run the query once for every department. Is there a better way to achieve this functionality?

We can use **GROUP BY** to achieve such results using a single query. **GROUP BY** groups the data from the table into different groups based on criteria provided and calculates the aggregate function for each group. Thus the result has 1 row for each group.



**Example of Group By Clause :**

Let us try to write a query to display the department and the total salary paid in each department.

SELECT Dept, SUM(Salary) FROM Employee GROUP BY Dept;

**Employee Table**

| **ID** | **ENAME** | **SALARY** | **DEPT** | **DESIGNATION** |
| --- | --- | --- | --- | --- |
| 1 | James Potter | 75000.00 | ICP | PM |
| 2 | Ethan McCarty | 90000.00 | ETA | PM |
| 3 | Abhinav Mittal | 35000.00 | IVS | SSE |
| 4 | Jack Abraham | 30000.00 | ETA | SSE |
| 5 | Ayaz Mohammad | 40000.00 | ICP | TA |
| 6 | Alice Jackson | 50000.00 | ICP | TA |

| **ID** | **ENAME** | **SALARY** | **DEPT** | **DESIGNATION** |
| --- | --- | --- | --- | --- |
| 2 | Ethan McCarty | 90000.00 | ETA | PM |
| 4 | Jack Abraham | 30000.00 | ETA | SSE |
| 1 | James Potter | 75000.00 | ICP | PM |
| 5 | Ayaz Mohammad | 40000.00 | ICP | TA |
| 6 | Alice Jackson | 50000.00 | ICP | TA |
| 3 | Abhinav Mittal | 35000.00 | IVS | SSE |

**Aggregation**

| **SUM(Salary)** |
| --- |
| **120000.00** |

| **SUM(Salary)** |
| --- |
| **165000.00** |

| **SUM(Salary)** |
| --- |
| **35000.00** |

**Illustrative Algorithm**

Sort on Grouped Columns  
For each distinct Grouped Columns  
 Calculate aggregate function  
 Add row to result

Prev

Step 11 of 11

Next

**Result**

| **DEPT** | **SUM(Salary)** |
| --- | --- |
| ETA | 120000.00 |
| ICP | 165000.00 |
| IVS | 35000.00 |

**Having Clause :**

We have now seen how to use GROUP BY in conjunction with aggregate functions to get summary of data category wise. What if we want to filter this summary? For e.g. if we want to fetch only those departments whose average salary of their employees is greater than a specific value.

This can be achieved using HAVING clause. Having allows aggregate functions to be used as filter criteria which cannot be done using WHERE clause.

**Having Clause Example :**

Let us write a query to display the department and the total salary for those departments whose total salary is more than 90000.

SELECT DEPT, SUM(Salary) FROM Employee GROUP BY DEPT HAVING SUM(Salary) > 90000;

**Employee Table**

| **ID** | **ENAME** | **SALARY** | **DEPT** | **DESIGNATION** |
| --- | --- | --- | --- | --- |
| 1 | James Potter | 75000.00 | ICP | PM |
| 2 | Ethan McCarty | 90000.00 | ETA | PM |
| 3 | Abhinav Mittal | 35000.00 | IVS | SSE |
| 4 | Jack Abraham | 30000.00 | ETA | SSE |
| 5 | Ayaz Mohammad | 40000.00 | ICP | TA |
| 6 | Alice Jackson | 50000.00 | ICP | TA |

| **ID** | **ENAME** | **SALARY** | **DEPT** | **DESIGNATION** |
| --- | --- | --- | --- | --- |
| 2 | Ethan McCarty | 90000.00 | ETA | PM |
| 4 | Jack Abraham | 30000.00 | ETA | SSE |
| 1 | James Potter | 75000.00 | ICP | PM |
| 5 | Ayaz Mohammad | 40000.00 | ICP | TA |
| 6 | Alice Jackson | 50000.00 | ICP | TA |
| 3 | Abhinav Mittal | 35000.00 | IVS | SSE |

**Aggregation**

| **SUM(Salary)** |
| --- |
| **120000.00** |

| **SUM(Salary)** |
| --- |
| **165000.00** |

| **SUM(Salary)** |
| --- |
| **35000.00** |

**Illustrative Algorithm**

Sort on Grouped Columns  
For each distinct Grouped Columns  
 Calculate aggregate function  
 If aggregate value meets having condition  
 Add row to result

Prev

Step 13 of 13

Next

**Result**

| **DEPT** | **SUM(Salary)** |
| --- | --- |
| ETA | 120000.00 |
| ICP | 165000.00 |

**Order Of Query Execution :**

We have now learnt some additional clauses in SQL statements like ORDER By, GROUP BY, HAVING etc. Let us look what would be the execution order for these clauses.



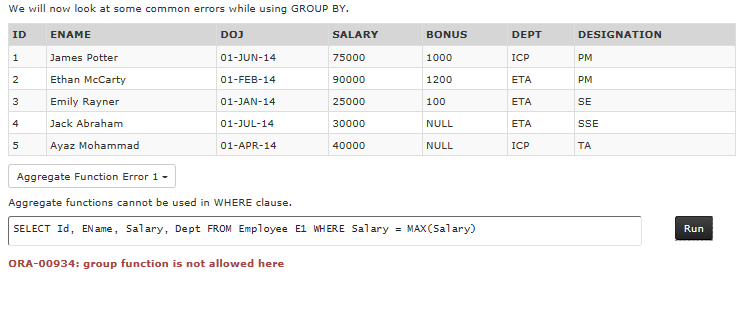
GROUP BY must always be after the WHERE clause otherwise aggregate functions will be calculated wrongly.

Having must always be after Group by as it filters records based on aggregate functions calculated during GROUP BY evaluation

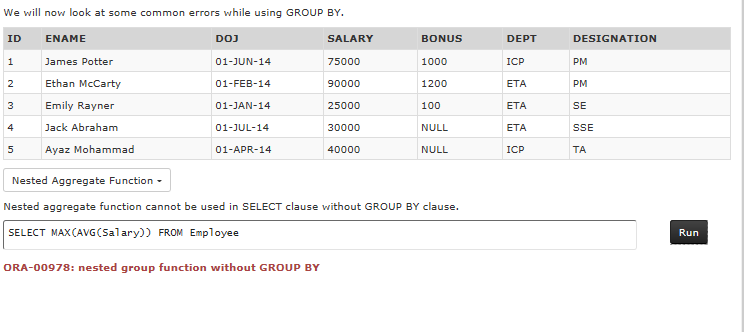
SELECT clause must be evaluated after Group By and Having because displaying attributes not used in GROUP BY are not allowed in SELECT clause. It can only filter columns from the grouped resultset

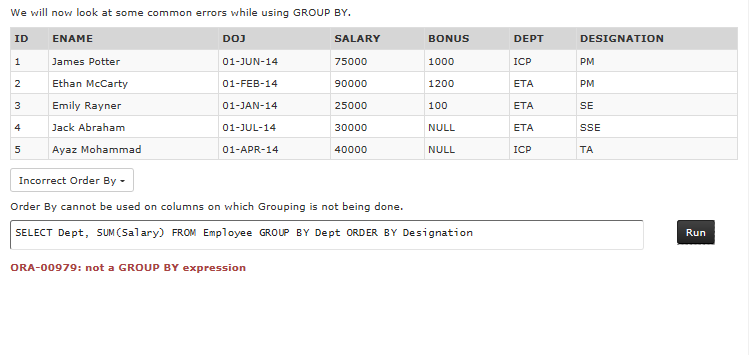
ORDER BY must always be the last step in query execution as it depends on input from other clauses

**Group By Errors :**



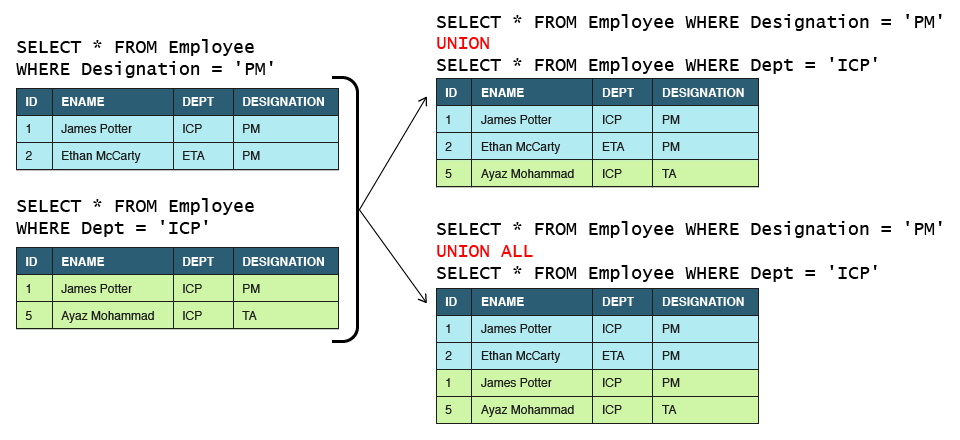






**Union and Union ALL :**

Use UNION and UNION ALL clause to combine results from two or more SELECT statements. The select statements may be from same or different tables.They must have same number of columns and their data types at same position in both the query must be compatible (either same or convertible through automatic conversion).



UNION removes all duplicates from the result. Two records are considered duplicates if values at corresponding positions of all their columns match.

**Introduction to Joins :**

How do we fetch data from multiple tables in a single query? Let us say we want to display employee id, employee name along with computer id, model of the computer allocated to the employee in a single tabular format. The input tables and sample output is provided below:

Employee Table

| **ID** | **ENAME** | **COMPID** |
| --- | --- | --- |
| 1 | James Potter | 1001 |
| 2 | Ethan McCarty | NULL |

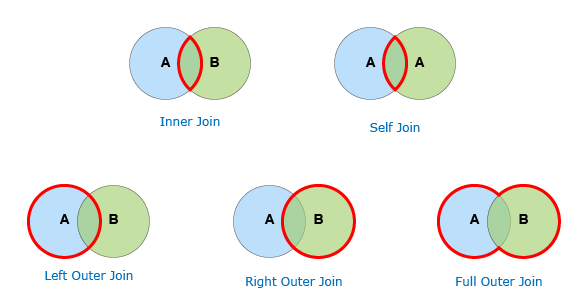
Computer Table

| **COMPID** | **MODEL** |
| --- | --- |
| 1001 | Vostro |
| 1002 | Precision |

Result Table

| **ID** | **ENAME** | **COMPID** | **MODEL** |
| --- | --- | --- | --- |
| 1 | James Potter | 1001 | Vostro |

We can meet such requirements by using JOINS which can combine data from multiple tables. JOINs are of multiple types:



In addition we also have Cross Join also called Cartesian product which is of academic interest only and is rarely used.

**Cross Join :**

CROSS Join is also referred to as Cartesian Product. A CROSS join with m rows in table A and n rows in table B will always produce m \* n rows. Essentially it combines each row from the first table with each row of the second table. A cross join is rarely used as it mostly produces lot of meaningless data. However it is useful to understand the concept of other joins. Let us understand this join using Employee and Computer tables.

**SELECT E.ID, E.ENAME, E.COMPID AS E\_COMPID, C.COMPID, C.Model**

**FROM Employee E CROSS JOIN Computer C;**

**Inner Join :**

INNER Join is the most frequently used JOIN. It matches the records from both tables based on the join predicate and returns only the matched rows. For ease of understanding one can think that first a Cartesian Product is created and then all the rows that do not meet the join condition are dropped from the result. Inner join also has a short hand syntax given its wide use. Let us understand this join using tables in Employee database:

**SELECT ID, ENAME, E.COMPID AS E\_COMPID, C.COMPID AS C\_COMPID, MODEL**

**FROM Employee E INNER JOIN Computer C ON E.COMPID = C.COMPID;**

While using Inner Joins there can be situation where you want to filter rows based on some criteria e.g. a need to fetch all employees from ETA who are allocated a computer. The filter condition can be supplied in two ways when using ANSI syntax ( t1 INNER JOIN t2 ON condition).

**Option 1:** Using a WHERE clause

SELECT Id, EName, E.CompId AS E\_CompId, C. CompId AS C\_CompId, Model

FROM Employee E INNER JOIN Computer C ON E.CompId = C.CompId WHERE Dept='ETA'

The query is evaluated using a two step process:

Step 1. Two tables are joined using join condition and resultset is evaluated

Step 2. Filter condition in WHERE clause is applied on all the rows of the resultset to give the final result

**Option 2:** Combining with the join condition using AND operator

SELECT Id, EName, E.CompId AS E\_CompId, C.CompId AS C\_CompId, Model

FROM Employee E INNER JOIN Computer C ON E.CompId = C.CompId AND Dept = 'ETA'

Here the query is evaluated in a single step as the filter condition is applied right at the time of join condition evaluation.

**Left Outer Join :**

LEFT OUTER Join for tables A and B will always return all records from table A even if matching record is not found in table B as per the join condition. For records where match is found the result set is exactly same as the inner join result. However for non matching records all columns from table B appear as NULL in the result. Let us understand this join using Employee and Computer tables:

**SELECT ID, ENAME, E.COMPID AS E\_COMPID, C.COMPID AS C\_COMPID, MODEL**

**FROM Employee E LEFT OUTER JOIN Computer C ON E.COMPID = C.COMPID;**

Left Outer Join is used to fetch all rows from a main table and some additional information from a lookup table using join condition. Unlike INNER JOINs additional conditions have to be supplied carefully depending upon the business requirement.

If the objective is to filter all records from the resultset then the filter condition must be supplied through the WHERE clause, e.g. a need to show employee details and model of their allocated computer for all employees who belong to 'ETA'.

SELECT Id, EName, E.CompId AS E\_CompId, C.CompId AS C\_CompId, Model

FROM Employee E LEFT OUTER JOIN Computer C ON E.CompId = C.CompId WHERE Dept = 'ETA'

However care must be taken that this filter condition is using an attribute from the main table. Any attempt to filter (except check for NULL) using attribute from lookup table will result in wrong output as all NULL rows will get filtered and the purpose of using OUTER join will get defeated.

If we want to conditionally fetch values from the lookup table then the additional criteria must be combined with the join condition using AND operator, e.g. a need to show details of all employees and in addition model of allocated computer for only those employees who are allocated a computer manufactured in '2014'.

SELECT Id, EName, E.CompId AS E\_CompId, C.CompId AS C\_CompId, Model

FROM Employee E LEFT OUTER JOIN Computer C ON E.CompId = C.CompId AND MYear = '2014'

**Right Outer Join :**

RIGHT OUTER Join for tables A and B will always return all records from table B even if matching record is not found in table A as per the join condition. Right outer join is the mirror image of left join. In fact it is rarely used because the same resultset can be obtained by using a left join and reversing the order of the tables in the query. Let us understand this join using tables in Employee database:

**SELECT ID, ENAME, E.COMPID AS E\_COMPID, C.COMPID AS C\_COMPID, MODEL**

**FROM Employee E RIGHT OUTER JOIN Computer C ON E.COMPID = C.COMPID;**

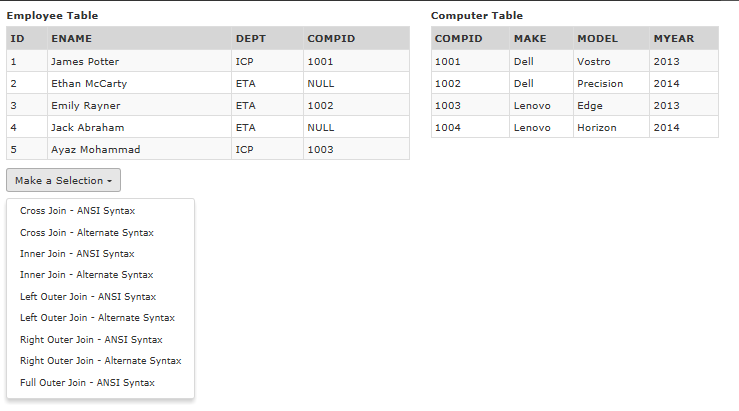
**Full Outer Join :**

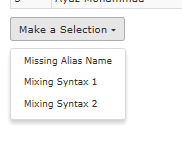
FULL OUTER Join combines the effect of both LEFT OUTER JOIN and the RIGHT OUTER JOIN. Full Outer Join between table A and table B returns matched as well as unmatched rows from both tables. For two tables with p and q rows, a 1:1 relationship and m matched rows the total number of rows in the resultset is m + (p - m) + (q - m) = p + q - m. Let us understand this join using tables in Employee database:

**SELECT ID, ENAME, E.COMPID AS E\_COMPID, C.COMPID AS C\_COMPID, MODEL**

**FROM Employee E FULL OUTER JOIN Computer C ON E.COMPID = C.COMPID;**

**Different Syntax of Join :**





**Self Join :**

SELF Join represents join of a table with itself. In this example we use inner self join to retrive employee's manager name. The Cartesian product of Employee table with itself will contain 5 x 5 = 25 rows. However only three rows have manager matching id and these appears on the result.

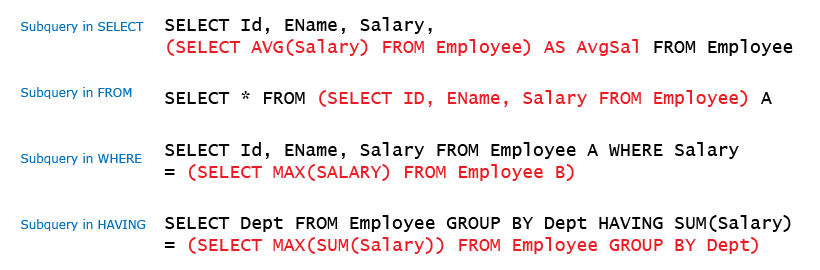
**Example :**

SELECT EMP.ID EMPID, EMP.ENAME EMPNAME, MGR.ID MGRID, MGR.ENAME MGRNAME

FROM Employee EMP INNER JOIN Employee MGR ON EMP.MANAGER = MGR.ID;

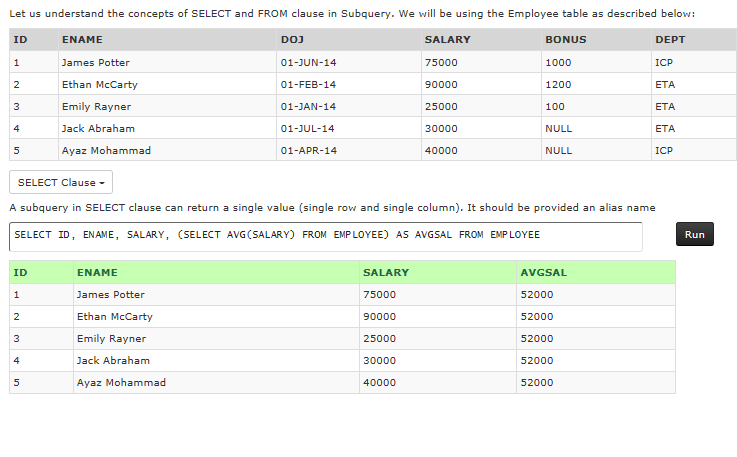
**Subqueries :**

Subquery is a query within a query. A subquery must be enclosed in brackets and can be used in SELECT, FROM, WHERE and HAVING clauses.

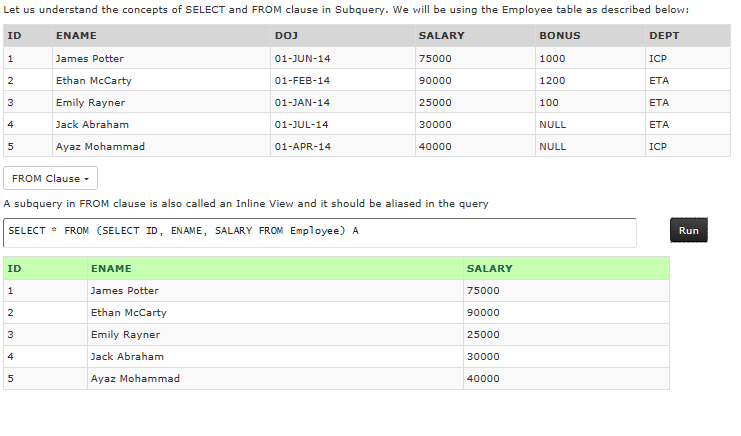


Subquery in SELECT and FROM clause are rarely used. Subqueries in WHERE and HAVING clauses are classified into Independent and Correlated subqueries.

**Select Clause Subqueries :**

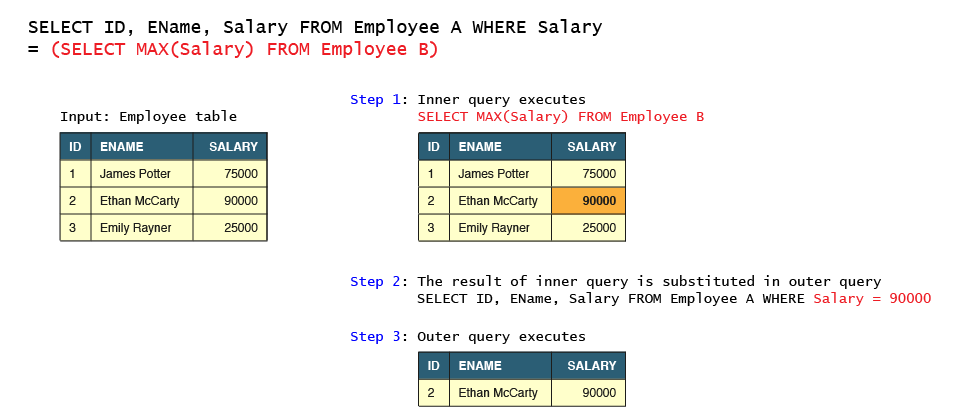


**From Clause Subqueries :**

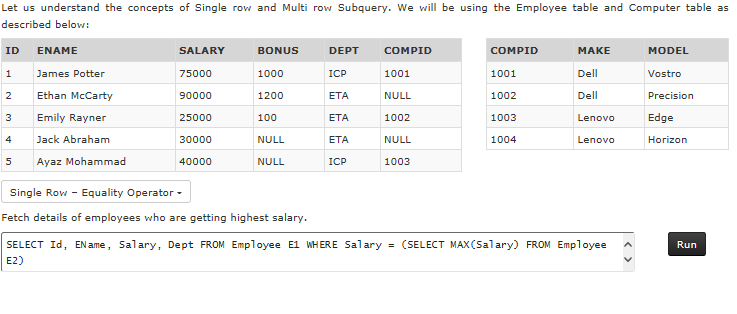


**Independent Subqueries :**

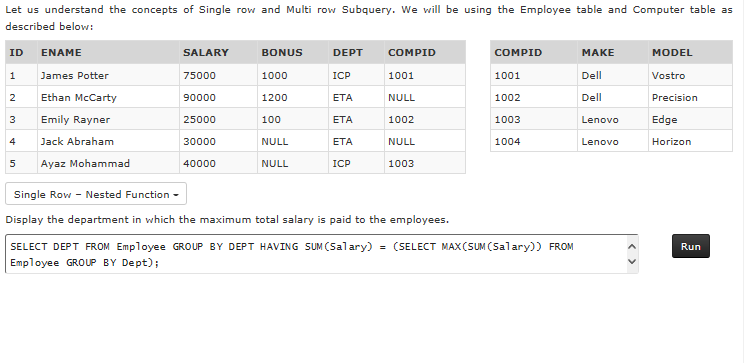
In an independent subquery, the inner and outer query are independent of each other. You can run an inner query and inspect its result independent of the outer query. Independent subquery are further classified into single row and multiple row types depending upon the number of rows returned.



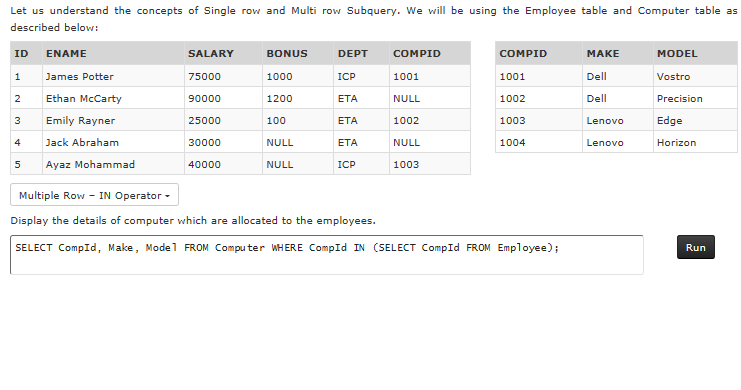
**Example 1 :**



**Example 2 :**



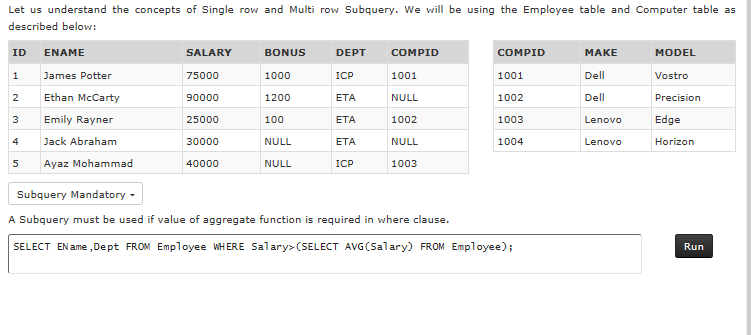
**Example 3 :**



Example 4 :

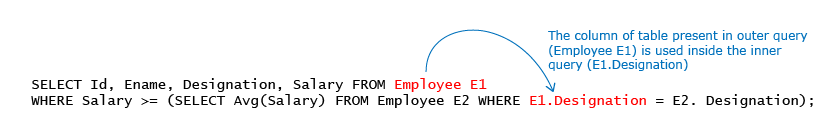


Example 5:

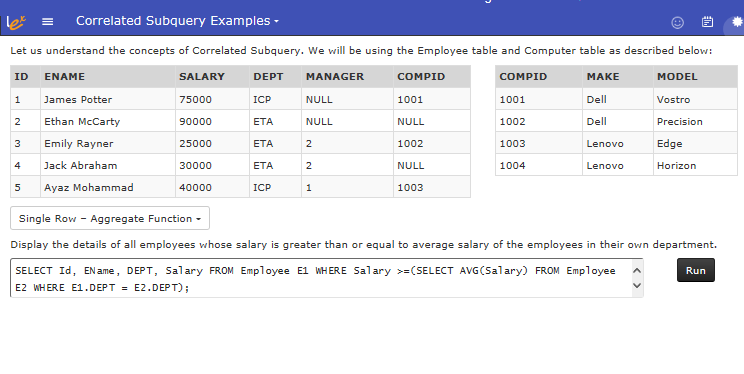


**Correlated Subqueries :**

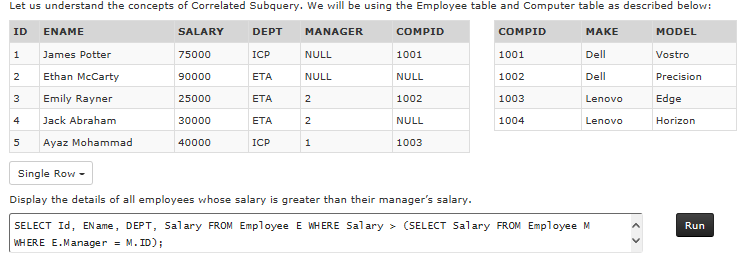
A Correlated subquery is one in which the inner query that depends upon the outer query for it's execution. Specifically it uses a column from one of the tables in the outer query. The inner query is executed iteratively for each selected row of the outer query. In case of independent subquery, the inner query just executes once.



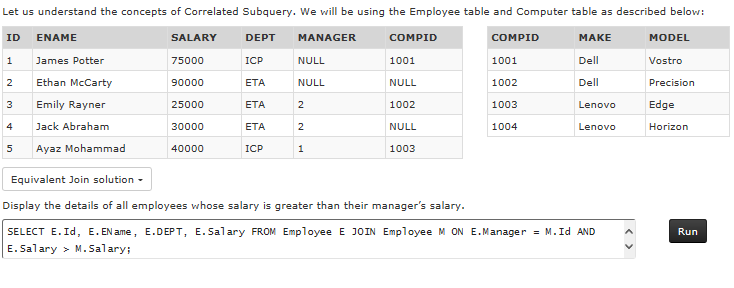
Example 1 :



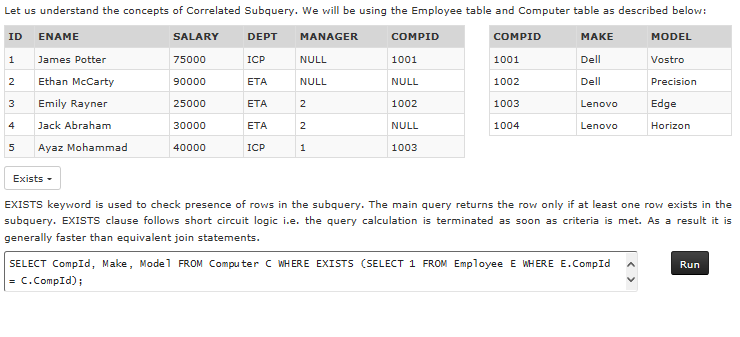
Example 2 :



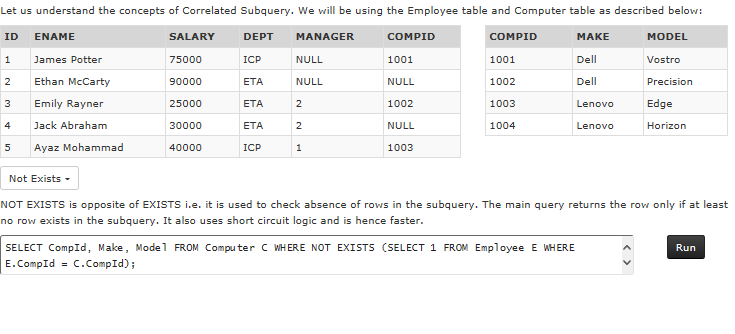
Example 3 :



Example 4 :

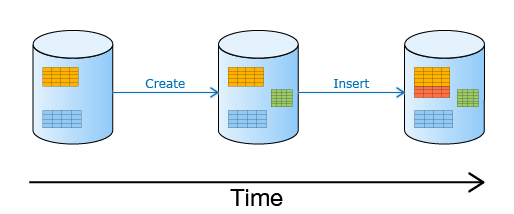


Example 5 :



Database State and Operations

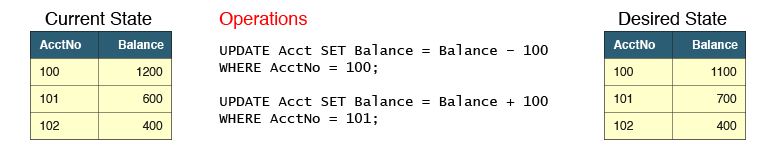
So far we have learnt various database operations like CREATE, DROP, UPDATE, DELETE etc. These operations alter the state of a database and move it forward in time. There are other operations like SELECT, DESCRIBE which does not alter the state at all.



Each of these operations are atomic by default i.e. they are applied in all or none manner. For e.g. a table cannot be created partially with less columns, an insert statement cannot insert a partial record etc.

**Need For Transactions :**

There are many situations when we need multiple database operations to be performed in atomic manner. The simplest example is money transfer from one account to another. It is evident that any credit without a debit or vice versa would be unacceptable. This business transaction has to be done using two operations; an update to increase the balance to the receiving account and another update to decrease the balance from the sending account.



A database operation can fail due to several reasons like connectivity failures, integrity constraints violation etc. What would happen if the first update succeeds and the second one fails? It leaves the database in an inconsistent state from business perspective.

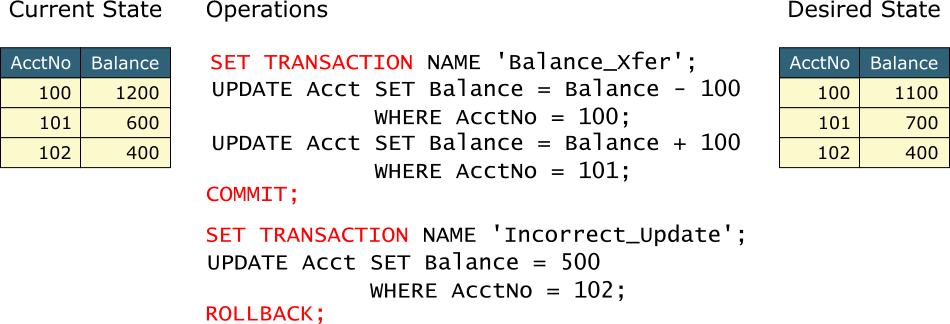
Introduction to Transactions :

Transaction is a logical unit of work containing one or more operations on a database. A transaction provides two important functions:

* Ensures that all operations within a transaction happens in an atomic manner
* Provides capability to undo the partial processing in the event of failure at any step

Database provides three statements for transactions

| **Statement** | **Description** |
| --- | --- |
| Set transaction | Initiates the transaction |
| Commit | Successfully completes the transaction. Actions of a transaction cannot be rolled back after commit has been executed. |
| Rollback | Ends the transaction after undoing all the work performed after begin transaction statement. |



Autocommit Transaction :

All operations that alter the state of database always happen in a transaction. So till now all the database operations you have performed was actually being run in a transaction. But how is it possible? We have never used set transaction, commit and rollback commands so far.

Well, database servers automatically start a transaction whenever they encounter the first executable SQL statement. An easy way to understand is to imagine that all SQL statements were surrounded by set transaction and commit statements.

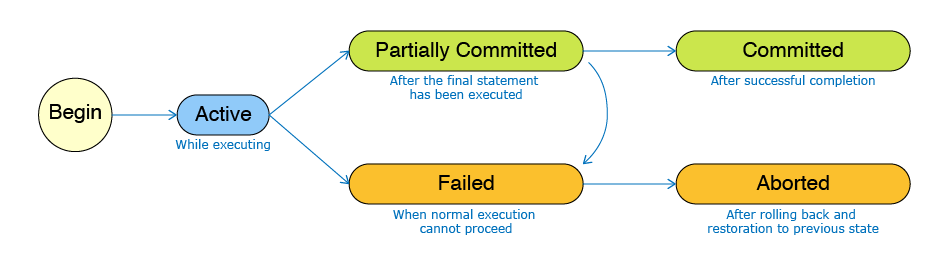
SET TRANSACTION;

UPDATE Acct SET Balance = Balance – 100 WHERE AcctNo = 100;

COMMIT;

The **AUTOCOMMIT** property of a connection controls automatic issue of commit after the operation. AUTOCOMMIT can have ON or OFF values and it depends on the default setting of the client you are using to connect to the database.

| **Command** | **Description** |
| --- | --- |
| SET AUTOCOMMIT ON | Changes the mode of connection to ON. In this mode COMMIT command is automatically issued after every SQL statement that alters the state of a database. |
| SET AUTOCOMMIT OFF | Changes the mode of connection to OFF. In this mode user is expected to provide an explicit COMMIT or ROLLBACK command to complete the transaction. |
| SHOW AUTOCOMMIT | Display the current state of AUTOCOMMIT property. |



**ACID Properties :**

All transactions exhibit ACID properties

| **Property** | **Description** |
| --- | --- |
| Atomicity | All operations within the transaction must all succeed or fail. |
| Consistency | A transaction always moves the database from one consistent state to another. Hence all integrity and data constraints must be satisfied. |
| Isolation | Transactions execute in isolation of each other. In other words partial execution of one transaction is not visible to other transactions. Only committed data is visible to other transactions. |
| Durability | Once a transaction is committed, it is permanently saved, the data is preserved even in the case of power failure, hardware failure etc. |

**Modular Query Writing :**

SQL queries can get quite complex. It is often easier to write them in a modular way. We'll learn how to do that for a business scenario.

We'll do this in two steps.

**Step 1 - Design:** We determine the SQL concepts that are relevant to the given problem.

**Step 2 - Coding:** We write and test the query in incremental steps.

Next we'll use a questionnaire to determine the SQL concepts that are relevant to the given problem.

* You do not need to memorize this questionnaire. It will be available to you during your exams also.
* There are many ways to meet a business requirement. This questionnaire shows you only one possible way to think through SQL concepts. It is not a complete set of rules, and it will not help you solve all possible situations. If you run into any issues, contact your educator.
* The questionnaire focuses on advanced concepts like INNER JOIN, OUTER JOIN, GROUP BY, HAVING, independent subquery and correlated subquery. To keep things simple, we have omitted concepts like functions (NVL, LOWER etc.), ORDER BY, UNION and logical operators (BETWEEN, LIKE, etc.), but if you need them, please feel free to use them.
* The questionnaire assumes a two table scenario. When you need more tables, use relevant questions repeatedly as required. For e.g you need to ask JOIN questions repeatedly for each pair of tables to be joined

We will now demonstrate modular query writing by solving some questions.

**Business Scenario**

Toys corner’ is a famous toy shop that sells varieties of toys under different categories and for different age limits. To make the business more reachable, they maintain a database and an application. The table structures and sample data of the database are as given below.

The **Customers table** provides the details of all the customers who visit the shop to make purchase.

| Column Name | Data Type | Constraints | Description |
| --- | --- | --- | --- |
| CustId | NUMBER | PRIMARY KEY | Id of the customer |
| CustName | VARCHAR2(10) | NOT NULL | Name of the customer |
| CustType | CHAR(1) |  | Type of customer |

The **Category table** provides the details of different categories of toys available in the shop.

| Column Name | Data Type | Constraints | Description |
| --- | --- | --- | --- |
| CId | CHAR(4) | PRIMARY KEY | Id of the category. Should start with 'C' |
| CName | VARCHAR2(5) |  | Name of the category |

The **Toys table** provides the details of toys that are available under different categories.

| Column Name | Data Type | Constraints | Description |
| --- | --- | --- | --- |
| ToyId | CHAR(5) | PRIMARY KEY, CHECK | Id of the toy. Should start with 'T' |
| ToyName | VARCHAR2(50) | UNIQUE, NOT NULL | Name of the toy |
| CId | CHAR(4) | FOREIGN KEY, NOT NULL | Refers to the cId in category table |
| Price | NUMBER | NOT NULL, CHECK | Cost of the toy. Toy cost must be greater than 0 |
| Stock | NUMBER | NOT NULL | Number of toys available |

The **Transactions table** provides the transactions made by the customer.

| Column Name | Data Type | Constraints | Description |
| --- | --- | --- | --- |
| TxnId | NUMBER | PRIMARY KEY | Id of the transaction |
| CustId | NUMBER | FOREIGN KEY | Refers to the custId in customer table |
| ToyId | CHAR(5) | FOREIGN KEY | Refers to the toyId in toy table |
| Quantity | NUMBER |  | Number of toys purchased |
| TxnCost | NUMBER |  | Cost of the transaction |

[Download](https://lex.infosysapps.com/fastrack/generic/dbms/docs/Table_Creation_Script.txt) table creation script for Modular Query Writing

**Examples :**

DROP TABLE Transactions CASCADE CONSTRAINTS PURGE;

DROP TABLE toys CASCADE CONSTRAINTS PURGE;

DROP TABLE Category CASCADE CONSTRAINTS PURGE;

DROP TABLE Customers CASCADE CONSTRAINTS PURGE;

CREATE TABLE customers(

CustId NUMBER PRIMARY KEY,

CustName VARCHAR2(10) NOT NULL,

CustType CHAR(1)

);

CREATE TABLE Category(

CId CHAR(4) PRIMARY KEY CHECK (CId Like 'C%'),

CName VARCHAR2(15)

);

CREATE TABLE toys(

ToyId CHAR(5) PRIMARY KEY CHECK (ToyId Like 'T%'),

ToyName VARCHAR2(50) UNIQUE NOT NULL,

CId CHAR(4) NOT NULL REFERENCES Category(CId),

Price NUMBER NOT NULL CHECK (Price > 0),

Stock NUMBER NOT NULL);

CREATE TABLE Transactions

(

TxnId NUMBER PRIMARY KEY,

CustId NUMBER REFERENCES Customers(CustId),

ToyId CHAR(5) REFERENCES Toys(ToyId),

Quantity NUMBER ,

TxnCost NUMBER);

COMMIT;

Toys table

| ToyId | ToyName | CId | Price | Stock |
| --- | --- | --- | --- | --- |
| T1001 | GT Racing Car | C101 | 500 | 40 |
| T1002 | Hummer Monster Car | C101 | 600 | 20 |
| T1003 | ThunderBot Car | C101 | 700 | 15 |
| T1004 | Ken Beat | C102 | 150 | 20 |
| T1005 | Drummer | C102 | 200 | 10 |
| T1006 | Kelly | C103 | 150 | 13 |
| T1007 | Barbie | C103 | 550 | 40 |

Category table

| CId | CName |
| --- | --- |
| C101 | Vehicles |
| C102 | Musical |
| C103 | Dolls |
| C104 | Craft |

Transactions table

| TxnID | CustId | ToyId | Quantity | TxnCost |
| --- | --- | --- | --- | --- |
| 1000 | 103 | T1006 | 5 | 2750 |
| 1001 | 104 | T1002 | 2 | 1200 |
| 1002 | 103 | T1005 | 3 | 600 |
| 1003 | 101 | T1001 | 1 | 500 |
| 1004 | 101 | T1004 | 3 | 450 |
| 1005 | 103 | T1003 | 3 | 2100 |
| 1006 | 104 | T1003 | 4 | 2400 |

Customers table

| CustId | CustName | CustType |
| --- | --- | --- |
| 101 | Tom | R |
| 102 | Harry |  |
| 103 | Dick | P |
| 104 | JoY | P |

**Example :**

DELETE FROM Transactions;

DELETE FROM toys;

DELETE FROM Category;

DELETE FROM Customers;

INSERT INTO Customers VALUES (101,'Tom','R');

INSERT INTO Customers VALUES (102,'Harry',NULL);

INSERT INTO Customers VALUES (103,'Dick','P');

INSERT INTO Customers VALUES (104,'JoY','P');

INSERT INTO Category VALUES ('C101','Vehicles');

INSERT INTO Category VALUES ('C102','Musical');

INSERT INTO Category VALUES ('C103','Dolls');

INSERT INTO Category VALUES ('C104','Craft');

INSERT INTO Toys VALUES ('T1001','GT Racing Car','C101',500,40);

INSERT INTO Toys VALUES ('T1002','Hummer Monster Car','C101',600,20);

INSERT INTO Toys VALUES ('T1003','ThunderBot Car','C101',700,15);

INSERT INTO Toys VALUES ('T1004','Ken Beat','C102',150,20);

INSERT INTO Toys VALUES ('T1005','Drummer','C102',200,10);

INSERT INTO Toys VALUES ('T1006','Kelly','C103',150,13);

INSERT INTO Toys VALUES ('T1007','Barbie','C103',550,40);

INSERT INTO Transactions VALUES (1000,103,'T1006',5,2750);

INSERT INTO Transactions VALUES (1001,104,'T1002',2,1200);

INSERT INTO Transactions VALUES (1002,103,'T1005',3,600);

INSERT INTO Transactions VALUES (1003,101,'T1001',1,500);

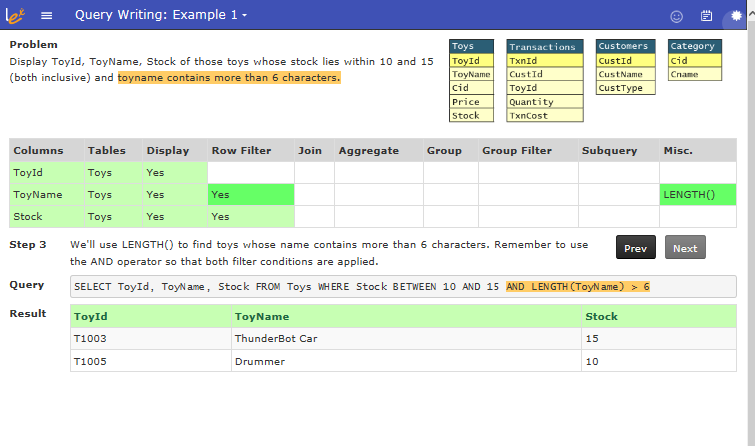
INSERT INTO Transactions VALUES (1004,101,'T1004',3,450);

INSERT INTO Transactions VALUES (1005,103,'T1003',3,2100);

INSERT INTO Transactions VALUES (1006,104,'T1003',4,2400);

COMMIT;

Example 1 :



Example 2 :

