The small start-up wants a plant distribution application that will help the user to upload different types (Air plant, Aquatic, Ferns, Bonsai, Creeper etc) of new plants and its details (Name, Description, Watering, Low or High Maintenance, Indoor or Outdoor, Quantity) including images for sale. Users can select the plants, add to cart and place the order. Before place the order application should show the total amount of the selected plants. Also, users can choose the payment option by cash or exchange of the plants they have uploaded. After placing the order, the user should be able to see the order status (New, Awaiting info, Accepted or Rejected). Users must login to the application to gain access. Application must track the user’s logins and logouts.

**1. Discover the entities.**

**First:**

Identify all the collective nouns and nouns in the statement of the problem that represent *objects of*

*interest* from the problem domain. These should **not** be descriptions or characteristics of the objects of

interest, but rather, the nouns or collective nouns representing the objects of interest.

**Second:**

List the discovered objects of interest. The convention used in this book is to use plural nouns for objects

of interest (entities). Using plural nouns makes it easier in the later stages of the database design process

when relationships are derived (step 2).

**First**: Identify all the collective nouns and nouns in the statement of the problem that represent *objects of*

*interest* from the problem domain:

The small start-up wants a plant distribution application that will help the user to upload different types (Air plant, Aquatic, Ferns, Bonsai, Creeper etc) of plants and its details (Name, Description, Watering, Low or High Maintenance, Indoor or Outdoor, Quantity) including images for sale. Users can select the plants, add to cart and place the order. Before place the order application should show the total amount of the selected plants. Also, users can choose the payment option by cash or exchange of the plants they have uploaded. After placing the order, the user should be able to see the order status (New, Awaiting info, Accepted or Rejected). Users must login to the application to gain access. Application must track the user’s logins and logouts.

**Second:**

List the discovered objects of interest *(entities)*:

1. Plants
2. Users
3. PlantTypes
4. Orders
5. PaymentOptions
6. OrderStatus
7. User Logins/Logouts (Log Entries)
8. WateringLevel
9. SunLightLevel
10. MaintenanceLevel

**2. Assign attributes to each entity discovered.**

**First:**

For each entity, list the possible characteristics and/or properties that are recorded in the problem domain

and are relevant to the client or end user. It is best to do this using a table with column headings for each

entity discovered, with the cells of that column containing the characteristics and/or properties for that

entity. A meeting with the client and relevant stakeholders is usually necessary to ensure that all attributes

are captured.

The convention used in this book is to combine attribute names that are more than one word into one

word, and capitalize the first letter of each word. For example, *First Name* becomes *FirstName* and *Last*

*Name* becomes *LastName*.

**Second:**

Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it

has been placed and not in any other entity or entities, and that it is not shared between or among entities.

For each entity, go through each attribute one at a time ensuring that it is where it belongs:

*Password* is for *Users.*

If one of the attributes does not belong in the entity, or if you are not sure it belongs with that entity, or if it

is shared between two entities, *remove it.* For example, it would be tempting to put *UserName* or *Login*

in *LogEntries*, but they do not belong there because they are for *Users* not *LogEntries*. Eventually the

primary key from *Users* would become part of the *LogEntries* table, but as a foreign key. This is explained in later of this document.

**First:**

For each entity, list the possible properties and/or characteristics recorded in the problem domain and those relevant to the client:

|  |  |  |
| --- | --- | --- |
| Plants | Users | PlantTypes |
| Name | Login | PlantType |
| Description | UserName |  |
| Height | Password |  |
| Quantity | Active |  |
| Price |  |  |
| Images |  |  |

|  |  |  |
| --- | --- | --- |
| Orders | PaymentOptions | OrderStatus |
| OrderId | PaymentType | Status |
| OrderDate |  |  |
| Quantity |  |  |
| OrderTotal |  |  |
|  |  |  |
|  |  |  |

|  |
| --- |
| LogEntries |
| LoggedOn |
| LoggedOut |
|  |
|  |
|  |
|  |

|  |  |  |
| --- | --- | --- |
| WaterLevel | SunLightLevel | MaintenanceLevel |
| WateringFrequency | LightingLevel | MaintenanceLevel |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Second:**

Ensure that every attribute is where it belongs, that is, that each attribute belongs within the entity that it

has been placed in and not in any other entity or entities, and that it is not shared between or among

entities.

**3. Select identifiers, keys and primary keys from attributes of each entity.**

**First:**

Go through each entity and list the possible identifiers and keys from the list of attributes.

**Second:**

Select the unique identifiers for each entity from the list of possible identifiers and keys.

**Third:**

From the list of unique identifiers, select one as the primary key. If there are no unique identifiers, then

create one and call it *ID* or a derivative of *ID*, such as, *UserID* or *UserId*. In most instances it may be better to create a primary key and call it *ID*, despite the fact that the entity has a unique identifier. Some designers create a field called *ID* for every entity and make it the primary key. There is nothing wrong with this because it creates a unique identifier for every entity, which is what we are striving for. It is a design decision.

**Fourth:**

Ensure that every other attribute in the entity depends wholly and solely on the primary key – does every attribute depend on the primary key, the whole primary key, and nothing but the primary key? If the answer to this question is *no* for any attribute, then that attribute does not belong in that entity, and you must remove it.

**First:**

The list overleaf shows the entities with primary keys (PK) and indexes or keys (bold) having completed

the first, second, third, and fourth steps outlined above:

|  |  |  |
| --- | --- | --- |
| Plants | Users | PlantTypes |
| (PK)PlantId | **(PK)**UserId | **(PK)**PlantTypeId |
| Name | Login | PlantType |
| Description | UserName |  |
| Height | Password |  |
| Quantity | Active |  |
| Price |  |  |

|  |  |  |
| --- | --- | --- |
| Orders | PaymentOptions | OrderStatus |
| (PK)OrderId | **(PK)**PaymentTypeId | **(PK)**OrderStatusCode |
| OrderDate | PaymentType | Status |
| Quantity |  |  |
| OrderTotal |  |  |
|  |  |  |
|  |  |  |

|  |
| --- |
| LogEntries |
| (PK)LogEntryId |
| LoggedOn |
| LoggedOut |
|  |
|  |
|  |

|  |  |  |
| --- | --- | --- |
| WaterLevel | SunLightLevel | MaintenanceLevel |
| (PK)WateringFrequencyId | **(PK)**LightingLevelId | **(PK)**MaintenanceLevelId |
| WateringFrequency | LightingLevel | MaintenanceLevel |
|  |  |  |

**4. Derived unary and binary relationship.**

A relationship is defined as something that *exists between entities*. An Entity-Entity Matrix (E-E Matrix) can be used to easily discover all the relationships that exist between any two entities discovered in Step 1 of the 6-step process. These relationships are either unary or binary relationships.

A unary relationship occurs when there are two entities involved in a relationship and they are the same entity. For example, *Courses* can be related to itself via the relationship ‘*prerequisite*’: *Courses* are *prerequisites* for *Courses*. A binary relationship occurs when there are two entities involved in a relationship and the entities are different. For example, *Lecturer’s lecture Courses*.

It must be noted that database theory allows for higher order relationships between three or more entities. However, this book ignores these types of relationships because unary and binary relationships are usually sufficient to answer any queries that will be made on the data in the database.

The same requirements will be used in this section to illustrate how to use the E-E Matrix to derive all of the unary and binary relationships that exist between entities discovered during Step 1 of the 6-step process.

**First: Build the matrix.**

The E-E Matrix is built using entities discovered in Step 1 of the six-step process. It is a table consisting of an equal number of rows and columns, with each entity discovered heading a row and a column. The intersection of the rows and columns represents the relationships that may exist between the entities. While not essential, it is strongly recommended that the order of the entities on the row and column headings is the same, which makes filling in the relationships easier in Second step described later in this section.

List of discovered entities:

1. Plants
2. Plant Types (Type)
3. Water Level (WL)
4. Sunlight Level (SL)
5. Maintenance Level (ML)
6. Orders (Order)
7. Order Status (Status)
8. Payment Options (Pay)
9. Users (Users)
10. LogEntries (LogEntry)

Construct the Matrix using the entities:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Plant | Type | WL | SL | ML | Order | Status | Pay | User | LogEntry |
| Plant |  |  |  |  |  |  |  |  |  |  |
| Type |  |  |  |  |  |  |  |  |  |  |
| WL |  |  |  |  |  |  |  |  |  |  |
| SL |  |  |  |  |  |  |  |  |  |  |
| ML |  |  |  |  |  |  |  |  |  |  |
| Order |  |  |  |  |  |  |  |  |  |  |
| Status |  |  |  |  |  |  |  |  |  |  |
| Pay |  |  |  |  |  |  |  |  |  |  |
| User |  |  |  |  |  |  |  |  |  |  |
| LogEntry |  |  |  |  |  |  |  |  |  |  |

**Second: Fill in the matrix.**

Each cell in the E-E Matrix represents a potential relationship that can exist between the entity on the row heading and the entity on the column heading. There is the possibility that no relationship exists between the two entities; that there is one relationship between the two entities; and that there is more than one relationship between the entities. There is also the possibility that relationships can exist when the entity on the heading row is the same as the entity that is on the heading column. Relationships in cells where this is the case are called *unary* relationships. In order to determine what relationships, exist, it is necessary to revisit the problem domain and

thoroughly question the client and end users about relationships and potential relationships. You may need to go through the Matrix cell by cell, questioning the client and end users about each potential relationship.

Go through each cell in the Matrix, asking the question *‘is [Entity in Row Heading] related to [Entity in Column Heading]?’* For example, *‘is Lecturers related to Courses?’* If a relationship [or relationships] exists, place a verb in the cell for each relationship. Each verb that is placed in the cell represents a relationship that exists between the entity on the row heading and the entity on the column heading. It is possible that the same verb can be used to represent different relationships but in different contexts depending on the entities that are in the relationship, thus creating different relationships with the same verb. You will notice that the relationships are duplicated for half of the Matrix, the difference being that the relationships are in reverse. This reversal does not change the fact that a relationship exists, nor does it

change the relationship – it is the same relationship, only in reverse.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Plant | Type | WL | SL | ML | Order | Status | Pay | User | LogEntry |
| Plant |  | belongs | has | has | has | purchase |  |  | sell |  |
| Type | belongs |  |  |  |  |  |  |  |  |  |
| WL | has |  |  |  |  |  |  |  |  |  |
| SL | has |  |  |  |  |  |  |  |  |  |
| ML | has |  |  |  |  |  |  |  |  |  |
| Order | purchase |  |  |  |  |  | belongs | use | place |  |
| Status |  |  |  |  |  | belongs |  |  |  |  |
| Pay |  |  |  |  |  | use |  |  |  |  |
| User | sell |  |  |  |  | place |  |  | create | create |
| LogEntry |  |  |  |  |  |  |  |  | create |  |

Ignore the top half of the Matrix drawn down the diagonal from the top left of the Matrix to the bottom right of the Matrix, since it is a mirror image of the bottom half. You will need to examine each relationship closely to ensure that it is a relationship that you want to capture in your database, or to ensure that it is a relationship that is not captured through another pair or other pairs of relationships. Remember to verify that each relationship captured in the E-E Matrix is a valid relationship to the client. You may need to go through the Matrix cell by cell with the client, verifying that each relationship is valid and relevant.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Plant | Type | WL | SL | ML | Order | Status | Pay | User | LogEntry |
| Plant |  |  |  |  |  |  |  |  |  |  |
| Type | belongs |  |  |  |  |  |  |  |  |  |
| WL | has |  |  |  |  |  |  |  |  |  |
| SL | has |  |  |  |  |  |  |  |  |  |
| ML | has |  |  |  |  |  |  |  |  |  |
| Order | purchase |  |  |  |  |  |  |  |  |  |
| Status |  |  |  |  |  | belongs |  |  |  |  |
| Pay |  |  |  |  |  | use |  |  |  |  |
| User | sell |  |  |  |  | place |  |  | create |  |
| LogEntry |  |  |  |  |  |  |  |  | create |  |

Notice that *Water Level, Sunlight Level and Maintenance Level* are only related to *Plants*, and this is because only this relationship is necessary. There will be indirect relationship also existing between two entities. relationship between *Plants* and *Users* via *Orders*.

The importance of eliminating the redundant relationships cannot be over-emphasised. In many instances you may have to re-do the matrix several times, collaborating with the client each time, confirming and eliminating relationships. Remember that you are designing the database for your client and not for yourself, so you want to capture the relationships as the client sees them and not as you see them. it is so important to collaborate with the client and the end user to correctly identify all of the relationships.

**4. Create simplified Entity – Relationship diagram.**

The notation used in this book follows most conventions, in which a rectangle is used to represent an entity, and a diamond is used to represent a relationship. The name of the entity is printed inside the rectangle at the top, with the primary key (PK), indexes and other important attributes printed below it. The diamond contains the name of the relationship.

You do not need to include all the attributes for all of the entities on the simplified E-R diagram because doing so can clutter the diagram by making the entities unusually long. Therefore, in the interest of space, only the primary key and the most important and relevant attributes should be included in the entities on the simplified E-R diagram. Candidates for indexes or keys are represented in bold.

Each of the entities discovered in Step 1 of the six-step process is represented by a rectangle, clearly indicating the primary key and other important attributes. Each of the relationships derived in Step 2 of the

six-step process is represented by a diamond with the name of the relationship in the diamond.

The entities (rectangles) are then connected to the relationships (diamonds) according to the E-E Matrix derived in Step 2 of the six-step process. If there are many entities and many relationships, the diagram may span multiple pages.

**First:**

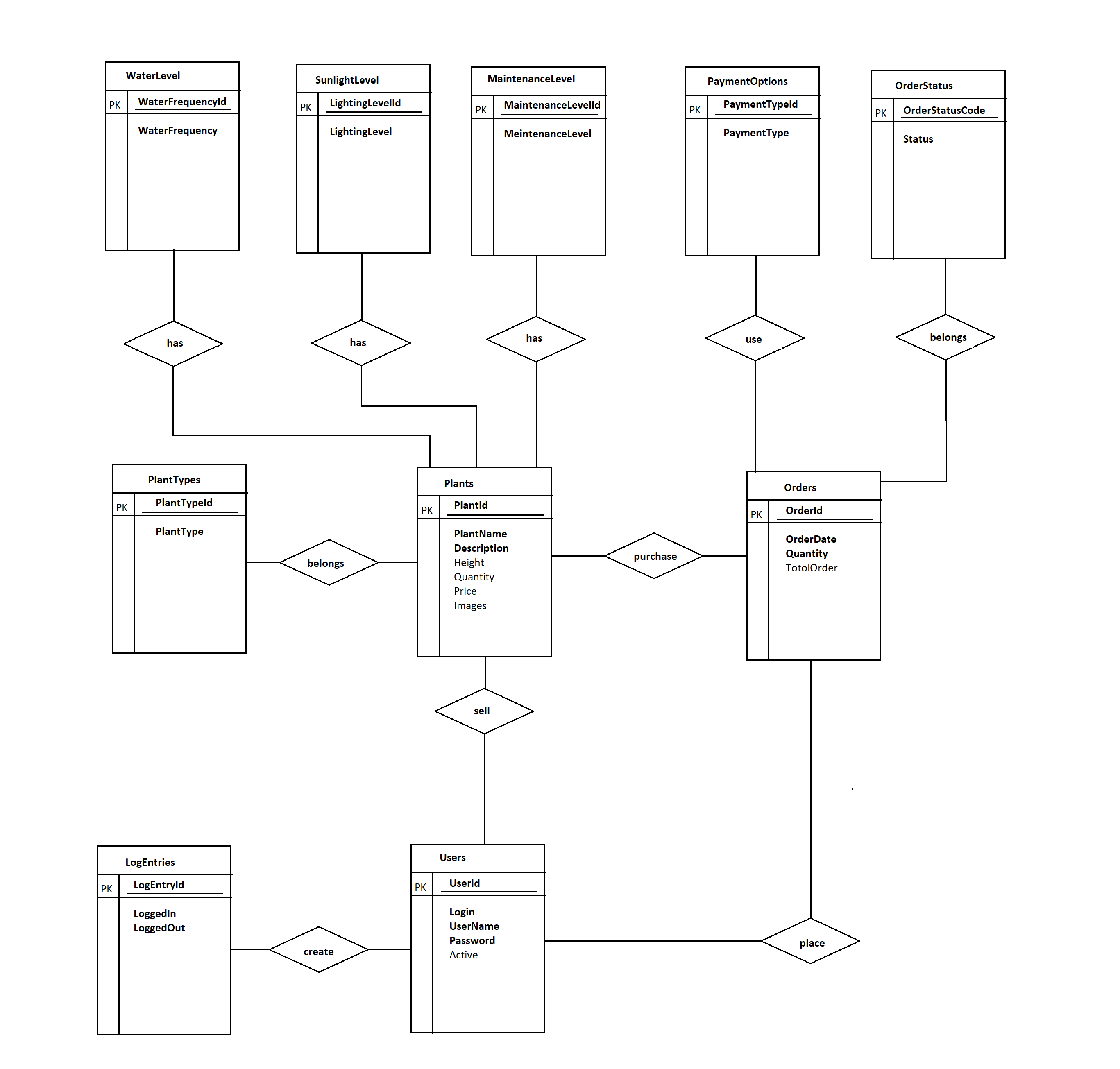
Create the entities (rectangles), clearly indicating the primary key and other important attributes.

**Second:**

Create the relationships (diamonds) and join them to the entities according to what was derived in the E-E Matrix. You may need to re-organize the diagram so that it is not confusing. Entities that have relationships with

themselves can be represented on the diagram more than once, or by simply having the diamond clearly feed to and from the same entity. In our case has examples of this, with the *Plants* and *Orders* entities.

Create the entities (rectangles), clearly indicating the primary key and other important attributes.



**5. List assertions for all relationships.**

Assertions are necessary to iron out the details about the relationships between the entities in the database. The relationships between the entities must satisfy the assertions made about them. Further, the client must verify that every assertion made is true and correct within the context of the database that is being modelled because the database is going to be modelled assuming that the assertions made at this stage are true and correct.

In order to obtain all of the assertions for the database that is being modelled, each relationship must be looked at individually. In addition, each relationship must be looked at from two directions: from *Entity A* to *Entity B* and then from *Entity B* to *Entity A*. At the end of this process, each relationship yields two assertions.

Optionality, cardinality, entity class, and entity occurrences help to record the assertions that are made about the entities and their relationship to each other. They tell us what **can** and **must** happen in a relationship and the number of occurrences an entity may have in a relationship.

*Optionality* says what **can** and **must** happen in a relationship. It can take one of two values, 0 or 1:

Optionality ‘0’ – Can (optional)

Optionality ‘1’ – Must (obligatory)

A hidden nuance about having an optionality of ‘0’ (can) is that it infers the possibility that something **cannot** happen in a relationship. Be mindful of this and always verify with the client what can and what cannot happen in a relationship.

*Cardinality* indicates the **number** of entity occurrences in a relationship, which can be any number from zero to infinity. However, in this book, cardinality will have one of two values, 1 and N:

Cardinality ‘1’ – Only one

Cardinality ‘N’ – Many or At least one

Each relationship on the simplified E-R diagram will yield two assertions because the relationship is looked at from two directions: from *Entity A* to *Entity B*, then from *Entity B* to *Entity A*. Therefore, there will be twice as many assertions as there are relationships.

Here are the steps for listing the assertions using the simplified E-R diagram:

**First:**

Look at each relationship from *Entity A* to *Entity B* and write out the relationship in words, using the entities involved in the relationship, the optionality, and cardinalities.

Here is an example:

*“Each Order must be placed by only one User”.*

The word **a** (**A**) is used to specify that what is being referred to is the entity occurrence and not the entity class.

The assertion always starts with an entity occurrence and always ends with an entity occurrence or an entity class, and the relationship is the verb in the middle of the two. The optionality comes after the first entity occurrence, the cardinality comes before the second entity occurrence or entity class, and the relationship separates the two entities. This is how assertions should be written:

***Entity Occurrence optionality relationship cardinality Entity Occurrence or Entity Class***

**Second:**

Look at each relationship in reverse, from *Entity B* to *Entity A*, and write out the relationship in words, using the entities involved in the relationship, the optionalities, and the cardinalities. Here is the same example in reverse:

*“One User can place many Orders”.*

The word *each* is used to specify that what is being referred to is the entity occurrence and not the entity class. The client must verify that every assertion made is true and correct within the context of the database that is being modelled.

**Assertions for all relationships of Plants Distribution Application:**

* A Plant must belong to many PlantTypes.
* A Plant must have only one WaterLevel.
* A Plant must have only one SunlightLevel.
* A Plant must have only one MeintainanceLevel.
* A Plant must be sold by only one User.
* A Plant must be purchased by only one Order.
* Each Order must use only one PaymentOption.
* Each Order must belong to only one OrderStatus.
* Each Order must place by only one User.
* A User can create many LogEntries.

**Reverse assertions for all relationships of Plants Distribution Application:**

* A PlantType can be used for many Plants.
* A WaterLevel can be required for many Plants.
* A SunlightLevel can be required for many Plants.
* A MeintananceLevel can be required for many Plants.
* A User can sell many Plants.
* An Order must be used to purchase many Plants or at least one Plant.
* A PaymentOption can used by many Orders.
* A OrderStatus can be used by many Orders.
* A User can place many Orders.
* Each LogEntry must be created by only one User.

**Remember: Verify with the client that each assertion is true and correct within the context of the database that is being modelled.**

**6. Create detailed E-R diagram using assertions.**

This step is the easiest of the six steps in the six-step process. All that must be done in this step is to place the assertions on the simplified E-R diagram, thus creating the detailed E-R diagram. The assertions are placed on the detailed E-R diagram between the entity and the relationship using the following convention:

**Optionality: cardinality**

where:

Optionality ‘0’ – Can (optional)

Optionality ‘1’ – Must (obligatory)

Cardinality ‘1’ – Only one

Cardinality ‘N’ – Many or At least one

Here is a list of all the possible combinations of **optionality: cardinality** that can appear on the detailed

E-R diagram:

0:1 – [Entity] **can** [*relationship*] **only one** [Entity]

0: N – [Entity] **can** [*relationship*] **many** [Entity]; or [Entity] **can** [*relationship*] **at least one** [Entity]

1:1 – [Entity] **must** [*relationship*] **only one** [Entity]

1: N – [Entity] **must** [*relationship*] **many** [Entity]; or [Entity] **must** [*relationship*] **at least one** [Entity]

The assertions are always positioned between the entity (rectangle) and the relationship (diamond) on the simplified E-R diagram, and they have a different meaning depending on which side of the relationship they appear on. However, the assertion is always written and interpreted as **optionality: cardinality**. The examples below show how to correctly insert assertions on simplified E-R diagrams.

**First:**

List of assertions with **(optionality: cardinality)**:

**Assertions for all relationships of Plants Distribution Application:**

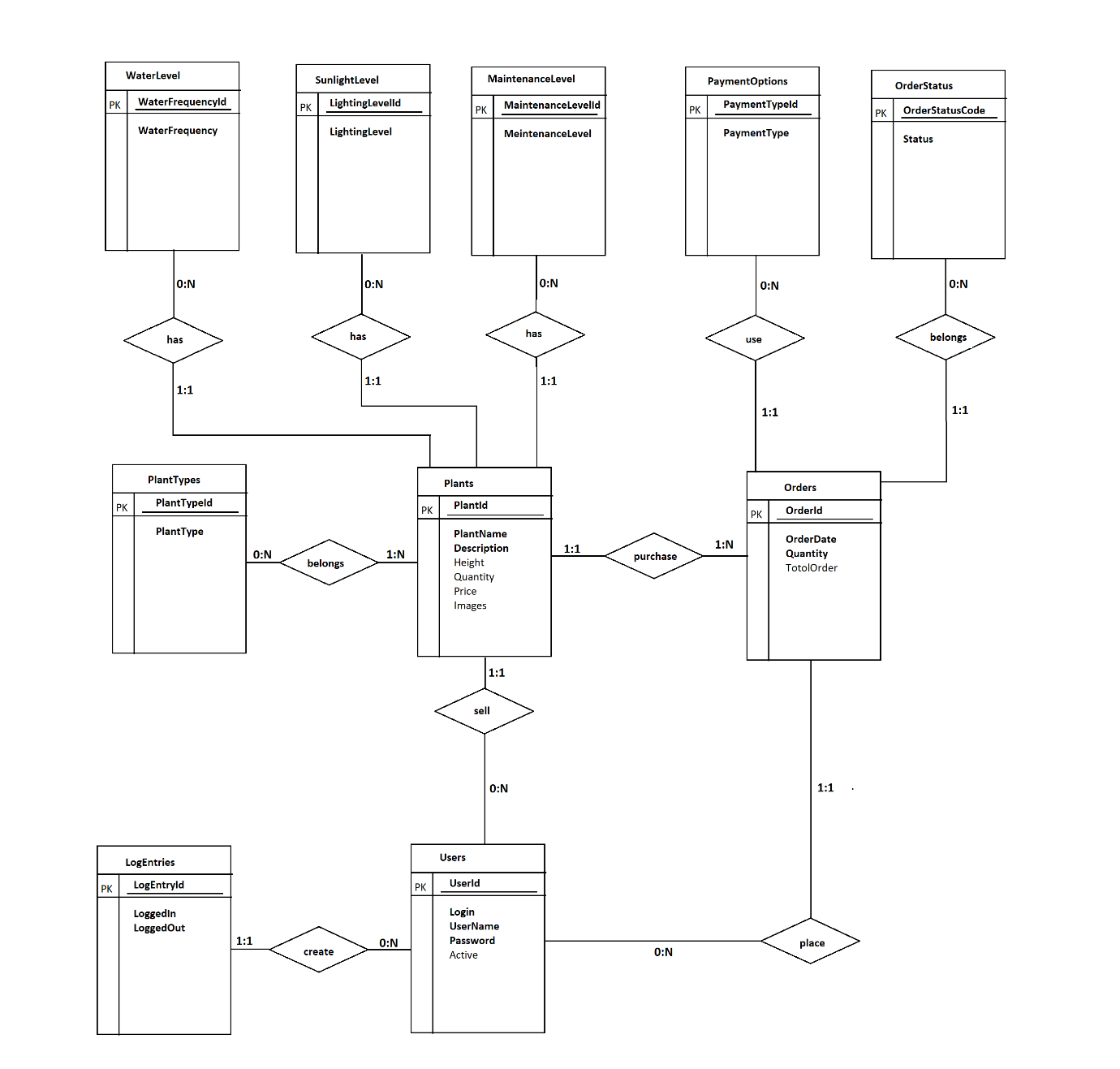
* A Plant must belong to many PlantTypes. (1: N)
* A Plant must have only one WaterLevel. (1:1)
* A Plant must have only one SunlightLevel. (1:1)
* A Plant must have only one MeintainanceLevel. (1:1)
* A Plant must be sold by only one User. (1:1)
* A Plant must be purchased by only one Order. (1:1)
* Each Order must use only one PaymentOptions. (1:1)
* Each Order must belong to only one OrderStatus. (1:1)
* Each Order must place by only one User. (1:1)
* A User can create many LogEntries. (0: N)

**Reverse assertions for all relationships of Plants Distribution Application:**

* A PlantType can be used for many Plants. (0: N)
* A WaterLevel can be required for many Plants. (0: N)
* A SunlightLevel can be required for many Plants. (0: N)
* A MeintananceLevel can be required for many Plants. (0: N)
* A User can sell many Plants. (0: N)
* An Order must be used to purchase many Plants or at least one Plant. (1: N)
* A PaymentOptions can used by many Orders. (0: N)
* A OrderStatus can be used by many Orders. (0: N)
* A User can place many Orders. (0: N)
* Each LogEntry must be created by only one User. (1:1)

**Second:**

Insert the generated assertions as **optionality: cardinality** one at a time on the simplified E-R diagram. The resulting detailed E-R diagram for Plants Distribution Application can be seen below.

****

**7. Transform the detailed E-R diagram to an implementable R-M diagram.**

This is the final step of the six-step process. However, it is the most complicated. It involves transforming the detailed E-R diagram created in the previous step into a Crow’s Foot Relational Model diagram.

Transforming an E-R diagram into an R-M diagram is not just a change of diagrams, but also a change of *view*. The E-R diagram is a **conceptual model** and represents the ***user’s view***of the data and the logical structure of the database, whereas the R-M diagram is an **implementation model** and represents the ***developer’s view***of the data and the physical structure of the database.

There are two types of models in relational database design, ***conceptual***

***models***, and ***implementation******models***. ***Conceptual models***are concerned with the logical nature of the data and what is being represented, and ***implementation models***are concerned with the physical nature of the data and with how the data will be represented in the database. The E-R diagram is a *conceptual model,* and the R-M diagram is an *implementation model*.

The diagram above is an example of a Crow’s Foot Relational Model diagram and shows the Relations, relationships, and how Relations are connected via relationships. In the Relations, *FK1* and *FK2* refer to foreign keys, defined in previous section as *“a primary key in one Relation (table) that appears as a column in another Relation (table) and is used to join the two Relations (tables) together in a relationship.”*

The diagram below gives a breakdown of the Crow’s Foot notation for relationships as they will be used in this book.

****

Here we have used an online website to create R-M diagram for our Plant Distribution application.

Created the diagram using <https://app.gleek.io/diagrams>

**Script:**

WaterLevel {1}-has-{0..n} Plants

SunlightLevel {1}-has-{0..n} Plants

MaintenanceLevel {1}-required-{0..n} Plants

Plants {1}-purchase-{1} Orders

OrderStatus {1}-belongs-{0..n} Orders

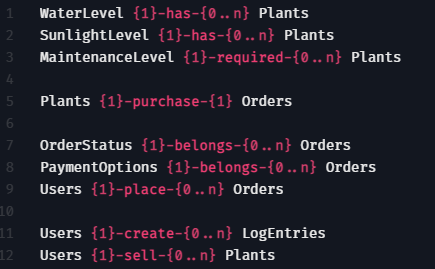
PaymentOptions {1}-belongs-{0..n} Orders

Users {1}-place-{0..n} Orders

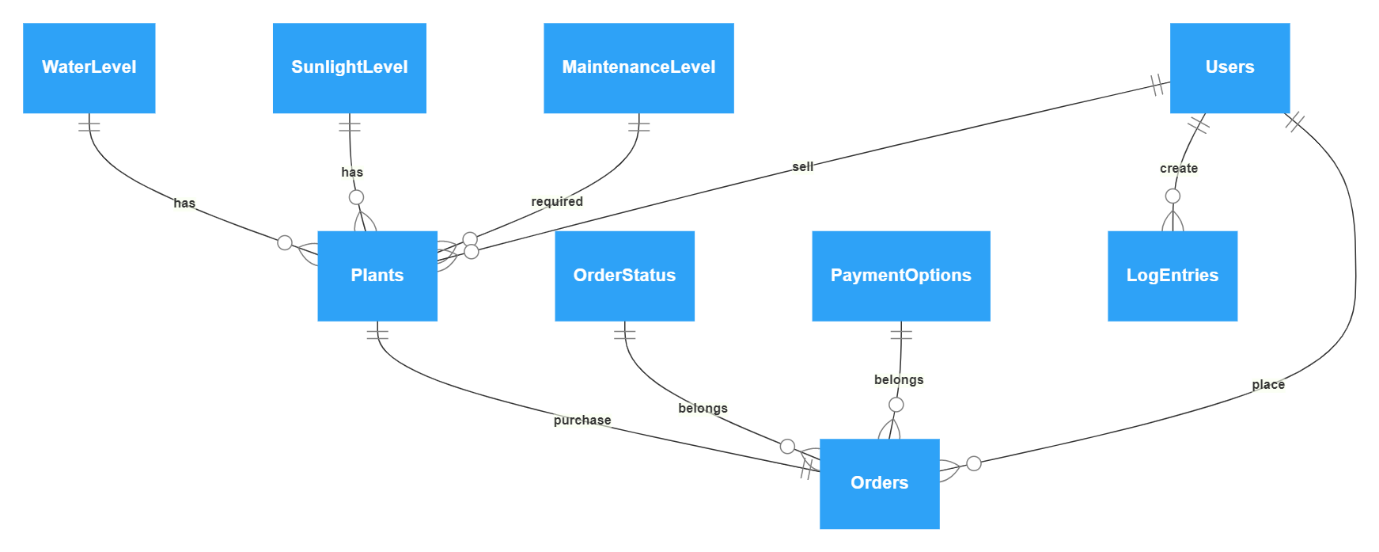
Users {1}-create-{0..n} LogEntries

Users {1}-sell-{0..n} Plants

**Script Screenshot:**

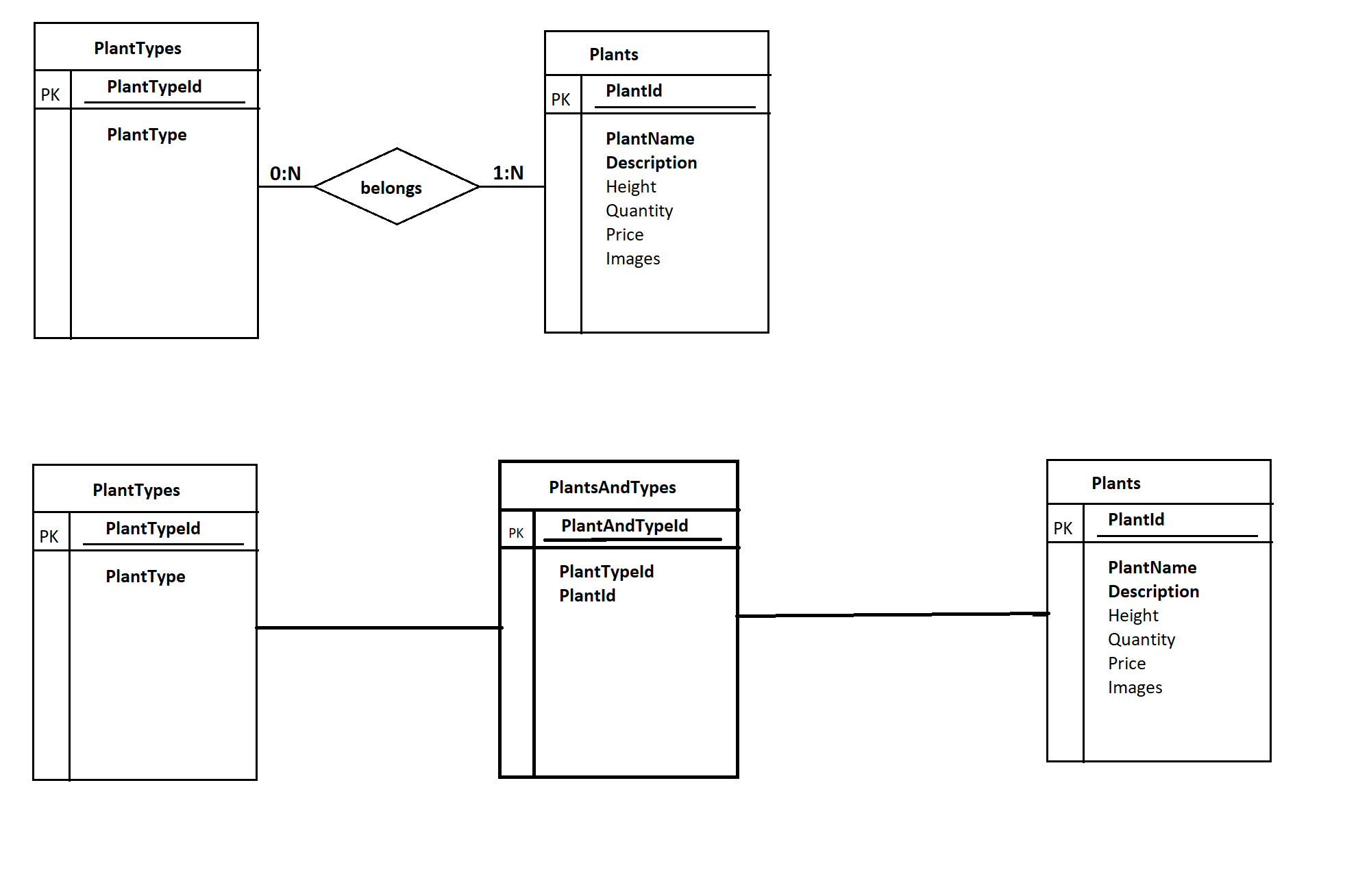


**Relationship Model:**

****

But still, we have not cover one more important relation in our R-M diagram. We have one many to many relationships in your E-R diagram. Refer the below screenshot for details.

**Many to many relationships Screenshot:**



Here is what to do to transform these relationships into many-to-many relationships on an R-M diagram:

**First:**

Identify all many-to-many relationships. These relationships are identified by a cardinality of *N (many or at least one)* on both sides of the relationship.

**Second:**

Remove both the relationship and the connectors to that relationship. Replace the relationship with a new *Relation*. The name of this new Relation should be a combination of the names of the two Relations that are on either side of the removed relationship. Combine these names in a manner that makes sense to form the name of the new Relation.

**Third:**

Create new *1 to 0 or more*, or *1 to 1 or more* relationships that connect the two existing Relations to the new Relation. Use a *1 to 0 or more* relationship if the optionality of the old relationship was *0 (can)*, and a *1 to 1 or more* if the optionality was *1 (must)*. The many side (Crow’s Foot) of the two new relationships should be on the new Relation. Ensure that the primary key for each of the two existing Relations becomes a foreign key in the new Relation and create a new and separate primary key for the new Relation, such as a unique identifier (Id). In some instances, the two *foreign keys* can combine to form a *composite primary key*, but that depends on

the data – whether there can be more than one instance of the composite primary key. If you are not sure, just create a new and separate primary key.

**Fourth:**

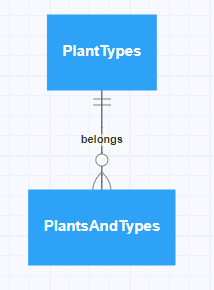
Explore whether attributes exist for the newly created Relation. In most cases they will not, but in some cases they will. These attributes must be valid for both foreign keys and not for any one foreign key individually; otherwise, that attribute belongs in one of the relations for the foreign keys and not in this new relation.

Many to many relations between Plants and PlantTypes.

**Previous Script( partially correct relation ):**

Plants {1..n}-belongs-{0..n} PlantType

**Diagram:**

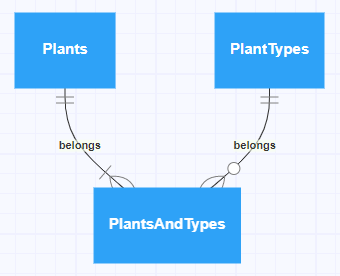


**Previous Script (corrected version):**

Plants {1}-belongs-{1..n} PlantsAndTypes

PlantTypes {1}-belongs-{0..n} PlantsAndTypes

**Diagram:**



**The complete R-M diagram is :**

**Script :**

WaterLevel {1}-has-{0..n} Plants

SunlightLevel {1}-has-{0..n} Plants

MaintenanceLevel {1}-required-{0..n} Plants

Plants {1}-purchase-{1} Orders

OrderStatus {1}-belongs-{0..n} Orders

PaymentOptions {1}-belongs-{0..n} Orders

Users {1}-place-{0..n} Orders

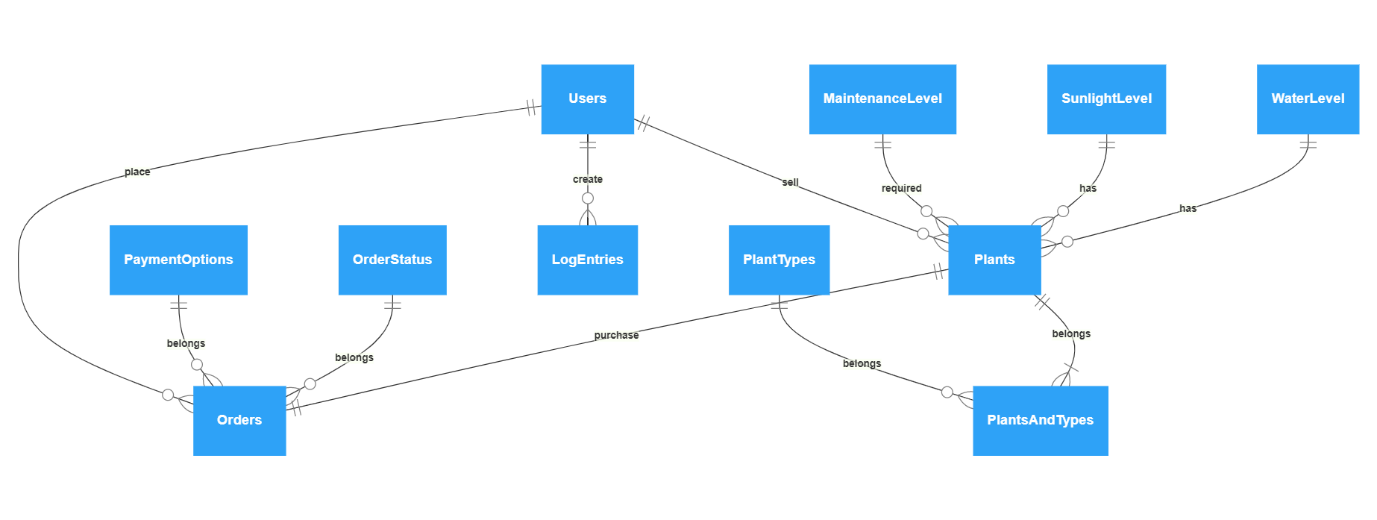
Users {1}-create-{0..n} LogEntries

Users {1}-sell-{0..n} Plants

Plants {1}-belongs-{1..n} PlantsAndTypes

PlantTypes {1}-belongs-{0..n} PlantsAndTypes

**Diagram:**

****

**The complete R-M diagram with attributes is :**

**Script :**

WaterLevel

string WaterFrequencyId\_PK

string WaterFrequency

SunlightLevel

string SunlightLevelId\_PK

string SunlightLevel

MaintenanceLevel

string MaintenanceLevelId\_PK

string MaintenanceLevel

PlantTypes

string PlantTypeId\_PK

string PlantType

Plants

string PlantId\_PK

string PlantName

string PlantDescription

string height

string Quantity

string Price

string Images

string WaterFrequencyId\_FK

string SunlightLevelId\_FK

string MaintenanceLevelId\_FK

PaymentOptions

string PaymentTypeId\_PK

string PaymentType

OrderStatus

string OrderStatusCode\_PK

string Status

Orders

string OrderId\_PK

string OrderDate

string Quantity

string TotalOrder

string UserId\_FK

string PaymentTypeId\_FK

string OrderStatusCode\_FK

Users

string UserId\_PK

string Login

string UserName

string Password

string Active

LogEntries

string LogEntryId\_PK

string LoggedIn

string LoggedOut

string UserId\_FK

PlantsAndTypes

string PlantAndTypeId\_PK

string PlantTypeId\_FK

string PlantId\_FK

WaterLevel {1}-has-{0..n} Plants

SunlightLevel {1}-has-{0..n} Plants

MaintenanceLevel {1}-required-{0..n} Plants

Plants {1}-purchase-{1} Orders

OrderStatus {1}-belongs-{0..n} Orders

PaymentOptions {1}-belongs-{0..n} Orders

Users {1}-place-{0..n} Orders

Users {1}-create-{0..n} LogEntries

Users {1}-sell-{0..n} Plants

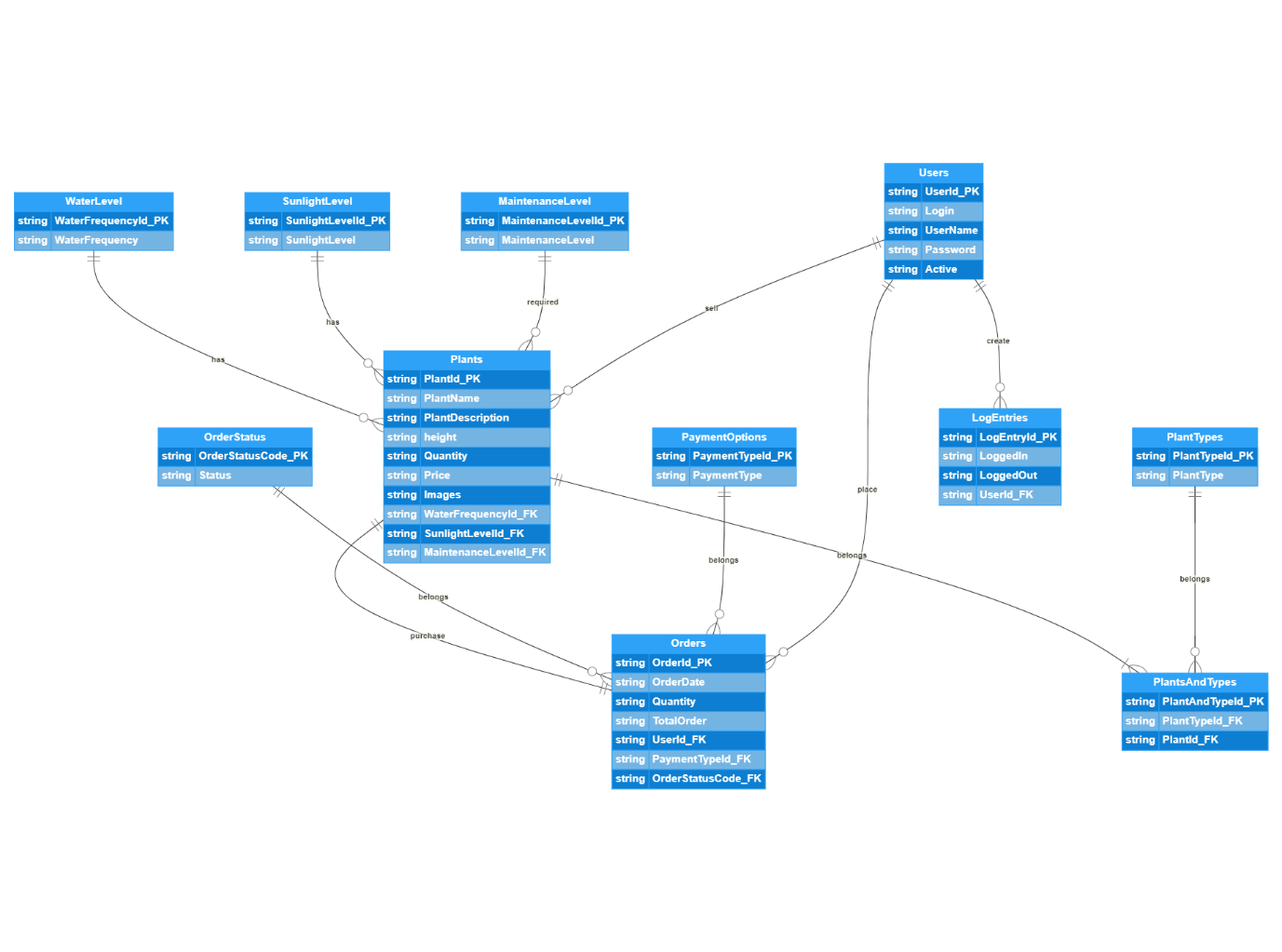
Plants {1}-belongs-{1..n} PlantsAndTypes

PlantTypes {1}-belongs-{0..n} PlantsAndTypes

PlantTypes {1}-belongs-{0..n} PlantsAndTypes

**Note:** this website does not support the PK and FK relation. So, I manually entered their name followed by ‘\_’(under score). Also, without data type the portal does not allow to enter any attribute. So, just to full fill the systax requirement I have declared all the arrtibutes as String. Later section I will discuss each attribute all of the Entities and define the appropriate Data Type.

**Diagram:**

****

**7. Create Data Dictionary.**

The table below gives a breakdown of the possible data types along with the other important considerations that need to be taken into account when implementing the columns of the tables for Plant Distribution Application.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| Plants | **(PK)PlantId** | Number |  |  | Yes |
|  | PlantName | Character |  | Yes | Yes |
|  | PlantDescription | Character |  |  | Yes |
|  | Height | Number |  |  | Yes |
|  | Quantity | Number |  |  | Yes |
|  | Price | Number |  |  | Yes |
|  | Images | ??????? |  |  |  |
|  | **(FK)WaterFrequencyId** | Number |  |  | Yes |
|  | **(FK)SunlightLevelId** | Number |  |  | Yes |
|  | **(FK)MaintenanceLevelId** | Number |  |  | Yes |
|  | **(FK)UserId** | Number |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| WaterLevel | **(PK)** **WaterFrequencyId** | Number |  |  | Yes |
|  | WaterFrequency | Character |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| SunlightLevel | **(PK)** **SunlightLevelId** | Number |  |  | Yes |
|  | SunlightLevel | Character |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| MaintenanceLevel | **(PK)** **MaintenanceLevelId** | Number |  |  | Yes |
|  | MaintenanceLevel | Character |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| PaymentOptions | **(PK)** **PaymentTypeId** | Number |  |  | Yes |
|  | PaymentType | Character |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| OrderStatus | **(PK)** **OrderStatusCode** | Number |  |  | Yes |
|  | Status | Character |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| Users | **(PK)UserId** | Number |  |  | Yes |
|  | Login | Character |  |  | Yes |
|  | UserName | Character |  | Yes | Yes |
|  | Password | ??????? |  |  | Yes |
|  | Active | ??????? |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| LogEntries | **(PK)LogEntryId** | Number |  |  | Yes |
|  | LoggedIn | Date |  | Yes | Yes |
|  | LoggedOut | Date |  | Yes | Yes |
|  | **(FK)UserId** | Number |  |  | Yes |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| PlantType | **(PK)** **PlantTypeId** | Number |  |  |  |
|  | PlantType | Character |  | Yes |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| PlantAndType | **(PK)** **PlantAndTypeId** | Number |  |  |  |
|  | **(FK) PlantTypeId** | Number |  |  |  |
|  | **(FK) PlantId** | Number |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Table | Column | Recommended Data Type | Length | Indexed | Required  (default) |
| Orders | **(PK)OrderId** | Number |  |  | Yes |
|  | OrderDate | Date |  | Yes | Yes |
|  | Quantity | Number |  |  | Yes |
|  | TotalOrder | Number |  |  | Yes |
|  | **(FK)UserId** | Number |  |  | Yes |
|  | **(FK)PaymentTypeId** | Number |  |  | Yes |
|  | **(FK)OrderStatusCodeId** | Number |  |  | Yes |

**8. SQL Commands to create table.**

The SQL commands to implement the design (create the tables) in a MySQL database are given below:

CREATE TABLE Plants (

PlantId int(11) NOT NULL AUTO\_INCREMENT,

PlantName varchar(50) NOT NULL,

PlantDescription varchar(200) NOT NULL,

Height int(5) NOT NULL,

Quantity int(5) NOT NULL,

Price int(5) NOT NULL,

Images BLOB,

WaterFrequencyId int(1) NOT NULL,

SunlightLevelId int(1) NOT NULL,

MaintenanceLevelId int(1) NOT NULL,

UserId int(11) NOT NULL,

PRIMARY KEY (PlantId),

FOREIGN KEY UserId (UserId)

REFERENCES Users (UserId)

ON UPDATE CASCADE ON DELETE RESTRICT,

FOREIGN KEY SunlightLevelId (SunlightLevelId)

REFERENCES SunlightLevel (SunlightLevelId)

ON UPDATE CASCADE ON DELETE RESTRICT,

FOREIGN KEY WaterFrequencyId (WaterFrequencyId)

REFERENCES WaterLevel (WaterFrequencyId)

ON UPDATE CASCADE ON DELETE RESTRICT,

FOREIGN KEY MaintenanceLevelId (MaintenanceLevelId)

REFERENCES MaintenanceLevel (MaintenanceLevelId)

ON UPDATE CASCADE ON DELETE RESTRICT

INDEX PlantName (PlantName)

);

**8. Other Details of RDBMS.**

The below properties are also important to check while using any RDBMS.

1. **Transactions and ACID Compliance.**
2. **Implementation considerations.**

The selected RDBMS would need to support transactions and be ACID compliant. Most RDBMSs would claim to support transactions and be ACID compliant, but as a developer you would need to investigate the levels of transaction isolation that are possible, and the level ACID compliance:

1. Are transactions processed concurrently?
2. Is transaction serialization possible?
3. What types of locks are available to be used on tables and rows?
4. Are constraints on columns possible?
5. Are *dirty reads* possible?
6. Are *phantom reads* possible?
7. Are *repeatable reads* possible?

To solve the problem outlined in the scenario above, your RDBMS should be able to do the following:

Prevent erroneous values from being entered into columns in tables. For example, negative values should not be allowed to be entered in the *NumInStock* column.

Process transactions. You should be able to create a transaction that could read the value of *NumInStock* and update its value if necessary (part of an all or none proposition).

Isolate transactions appropriately. You should be able to prevent transactions from reading the value of *NumInStock* while it is being used by another transaction in which it may be updated, has been marked to be updated, or has been updated and not committed.

RDBMSs would offer many flavours of the tools and techniques needed to provide a solution to the above scenario. However, there are trade-offs that may have to be made to preserve the integrity of the data in the tables of your database.

1. **Normalization.**

Normalization is the process of organizing the Relations (tables) in a database so that they reduce data redundancy and prevent inconsistent data dependencies. It is a very important part of database design, and RDBMSs must provide tools and techniques for implementing normalized databases.

There are at least three normal forms associated with normalization: first normal form (1NF), second normal form (2NF), and third normal form (3NF). Other higher forms, such as Boyce-Codd Normal Form

(BCNF), fourth, and fifth normal forms do exist, but 1NF, 2NF, and 3NF are usually sufficient to “reduce data redundancy and prevent inconsistent data dependencies.”

1. **Cascade Update and Delete.**
2. **Indexes.**

An index (key) is a data structure that is created for a database table but is external to that table, and if well selected and implemented can result in faster searches for data in that table. However, indexes do

not guarantee that data in a table will be accessed faster or that it would speed up searches for data in that table. If poorly implemented, indexes can significantly reduce database performance and take up as much or even more space than the data in the tables themselves.

Indexes have two main architectures: clustered and non-clustered. Clustered indexes store the data in the database file(s) in the same order as the index, while non-clustered indexes do not. This means that each table can only have one clustered index, which is usually the primary key. Indexes are usually implemented as binary trees (B trees), B+ trees, and hashes. B trees and B+ trees are used to implement clustered indexes, and hashes are used to implement non-clustered indexes. Most RDBMSs will allow you to select which columns in a table you want to use as indexes, but will not

allow you to select the architecture and implementation of those indexes. ANSI SQL provides the CREATE INDEX statement to create indexes on table columns, but the RDBMS usually determines the

flexibility given to developers to choose the type and architecture of the index created.

1. **Primary Keys and Foreign Keys.**

As mentioned in Chapter 1, the primary key for a table “is a unique identifier that is used to distinguish or identify rows in a Relation (table)”, and a foreign key is “a primary key in one Relation (table) that appears as a column in another Relation (table) and is used to join the two Relations (tables) together in a relationship.” Every table in a normalized relational database must have a primary key. Primary keys and foreign keys play an important role in storing, organizing, and finding data stored in these databases. As a result, primary keys and foreign keys should always be indexed. Once a column is defined as the primary key of a table, the RDBMS will automatically create an index for this column. However, this may not be the case for foreign keys, and you should take care to always clearly define indexes for foreign keys in tables, if not automatically done by the RDBMS.

1. **General Implementation Considerations**

Indexes speed up searches but slow down inserts, updates, and deletes. So, as a rule of thumb, if a table will be updated more than it will be searched, keep the number of indexes to a minimum. Computers work better with whole numbers than with characters, so indexes on numbers (integers) will always be faster than indexes on characters. As a rule of thumb, try to make the primary key, which is often implemented as a clustered index, an automatically generated sequential whole number – autonumber or auto-increment. Indexes govern how the data in a table is accessed and searched, so as a database designer and developer you need to be aware of how the data will be accessed and searched by the client. You need to know what kind of queries will be conducted on the data to know what columns to create indexes on.

Here are some general guidelines to follow when creating indexes:

Index all primary keys (if not already done by the RDBMS)

Index all foreign keys

Index columns used in JOINs. These will most likely be foreign keys.

Index columns used in WHERE clauses of queries.

Index columns used in ORDER BY clauses of queries.

Index columns used in GROUP BY clauses of queries.