**ARTISTREE**

**Project Direction Overview:**

I would like to create a database for a webapp which will help hire creative freelancers and also help freelancers choose projects to work on. There are many freelancing apps out there which is a hub for all sorts of professions, but this app targets only those who work in the creative field. Such professions include musicians, sound designers, writers, lyricists, costume designers, makeup artists, jewelry makers etc. An employer can post a project online and put out RFPs on the messageboard that will contain certain skill names in the form of tags and freelancers will be suggested those based on the skills they have selected for themselves when they created their profile on the website. Freelancers can even sign contracts with their employers and keep those records on the website. Employers can also pay their recruits through the portal on the website and the website will keep track of when the payment is due and how much needs to be paid on a periodic basis based on what was agreed upon in the contract.

For example, Employer Joseph Calamaza is organizing a corporate party for an account firm and he is looking for a stand up comedian for the entertainment of the group. He posts a request for comedians who are eager to perform in front of a corporate audience at said place. He also puts down tags like comedian, corporate gala, and <name of place>. Comedian and freelancer Sandra Bullington responds to his request. Joseph looks into her profile to see her experiences and he seems impressed. He contacts her and lets her know about the details of the event and talks price with her. Once Sandra agrees, he creates a contract on the app and fills out a form where he has to mention when Sandra needs to be paid and how much amount. Those information, once signed by employing authorities, will be sent over to Sandra for her to go through and sign and send back. Both parties will have the same copy that will not be able to be edited further. Based on that, Sandra will be paid her due amount on the day it is due through the app which will already have had Sandra’s payment information. Sandra will also be rated based on her performances and Joseph can even write her a recommendation. Sandra can also rate Joseph as an employer so that more people can be helped in deciding whether to work with him or not.

This app will be called Artistree because it is like a tree with many branches of freelancing activities.

**Use Cases and Fields:**

1. **Creating an account as an employer or a freelancer - Use Cases:**

* A user enters the website for the webapp.
* They are asked to log in or sign up.
* They enter their credentials: Name, Address, Phone Number, Email, Employer/Freelancer etc.
* Account is created for them.

Significant fields for this use case are given as follows:

| **Field** | **What it Stores** | **Why it’s Needed** |
| --- | --- | --- |
| Username | This is the name using which user will login to the system every time. | Users need a credential to create their digital signature on the website. This is one of the values for that purpose. |
| Password | This is the password to secure their data. | Users need a credential to create their digital signature on the website. This is one of the values for that purpose. |
| FirstName | This is the first name of the user | This is needed to display the user’s name in any form or transaction. |
| LastName | This is the last name of the user | This is needed to display the user’s name in any form or transaction. |
| PhoneNumber | This is the user’s contact details. | For easier reachability to the user and for OTP purposes. |
| Email | This is the user’s contact details. | For easier reachability to the user and for OTP purposes. |
| Address | This is the user’s contact details. | Sending bills and storing basic contact information. |
| AccountType | Employer/Freelancer | This will denote what the dashboard will look like for the user. |
| Language | Language spoken by user | Users will need to be able to communicate, so showing what languages each user can speak is useful. |

1. **Creating skills - Use Cases:**

* A freelancer will get a link on their dashboard that says Skills
* They will get a form that will let them add a skill
* They will also be able to add a type to that skill. It will be like a genre. So if a person can play guitar, their skill will be Playing guitar and it will go under the type: Music. This will help garner more music playing related jobs for them.

Significant fields for this use case are given as follows:

| **Field** | **What it Stores** | **Why it’s Needed** |
| --- | --- | --- |
| SkillName | This will be the name of the skill | If the user wants to include a skill that is not already in the system, they will be able to add it. |
| Description | This is the description if the user wants to add any. | This will be used to further describe the skill. |
| Type | This is like a genre for skills like: music, literature, dancing etc. | This will be needed to provide relatable jobs to freelancers. |

1. **Creating Job/Project - Use Case:**

* Employer creates a job or a project.
* They will post it with the name, venue, other supporting URLs etc for the job.
* The creator of that project will be able to assign a freelancer to it and the project will change its status so that it is no more publicly visible.
* Now the Project is available to both the employer and the assigned freelancer.
* Once the job is done, the employer can mark it as completed.

Significant fields for this use case are given as follows:

| **Field** | **What it Stores** | **Why it’s Needed** |
| --- | --- | --- |
| ProjectTitle | This will be the title of the project | This will be needed to display the entire project on the message board. |
| Description | This will be additional descriptions of what the project/job is like. | One the project view is opened, the description will give more details about the job/project. |
| Venue | This will be where the job/project will take place. | This will be needed if the job is to be held somewhere physical and not online. |
| Creator | This will be the name of the user who created this project. | Employer/Freelancer who created the job. |
| Tags | These will be tags that define the project | Needed to seek the attention of appropriate freelancers. |
| Status | What point of the timeline the project is now standing | This will be needed to track the progress of the job and to make sure who is having access to it. |

1. **Create Contract - Use Case:**

* Once there is an agreement between an employer and freelancer, the employer will create a contract.
* They will be able to add the project title to the contract, enter the name of the employer, freelancer, enter details of the contract and the payment details on it.
* The contract, once confirmed, will be sent to the freelancer to read and sign.
* They will send it back to the employer and a copy of both the signed contract will remain on both profiles.

Significant fields for this use case are given as follows:

| **Field** | **What it Stores** | **Why it’s Needed** |
| --- | --- | --- |
| EmployerName | Name of employer | This is to document who the employer is. |
| FreelancerName | Name of freelancer | This is to document who the freelancer is. |
| ProjectTitle | Title of project | This is to document which project the contract is for. |
| ContractBody | Description of the contract and other important details | Documentation of the details of the contract. |
| CreateDate | Date of creation of the contract | This is needed to know when the contract was created. |
| SignedDate | Date of signing of the contract | This is needed to know when both parties signed the contract. |
| PayableAmount | Amount to be paid to the freelancer | This will be needed to document and track the payment amount. |
| PaymentDate | Due date of payment | This will be needed to remind the payer when to pay the payee. |

1. **Pay freelancer - Use Case:**

* On the due date of the payment, the employer will send the money to the freelancer.
* This amount will be tracked using the payment value input in the contract.
* The employer will use the billing details that will be associated with the freelancer’s profile and send the money over to them.
* The receipt will remain in the system and users can opt for email options as well

Significant fields for this use case are given as follows:

| **Field** | **What it Stores** | **Why it’s Needed** |
| --- | --- | --- |
| EmployerName | Name of employer | This is to document who the employer is. |
| FreelancerName | Name of freelancer | This is to document who the freelancer is. |
| PayableAmount | Amount to be paid | This is used to keep track of the payment. |
| PaidAmount | Amount paid | This is used to keep track of how much is paid in this iteration and how much is remaining |
| DueAmount | Amount remaining | This is used to keep track of how much money needs to be paid after this iteration |
| ContractId | Link to the contract | This is used to link the contract with the payment details. |
| BillingDetails | Billing details of freelancer | This is needed to send over the payment to the freelancer. |

**Structural Database Rules:**

To put down the structural rules for my database Artistree, let us go over my use cases from above and try to find out the entities and the relationships between them.

1. **Creating an account as an employer or a freelancer:**

Here, we can see that there is an entity called User. This entity can be divided into two other types of entities, Employer and Freelancer. Every user when creating an account must identify themselves as either an Employer or a Freelancer and cannot be otherwise. So we can summarise the rule as follows:

1. Each user is an Employer or a Freelancer.

Language is also an entity here where a user speaks many languages or none, and a language is spoken by multiple users or none. The business rule can be written down as follows:

1. A user **may** speak **zero to many** languages.
2. Each language **may** be spoken by **no to many** users.
3. **Creating skills:**

We can identify Skill and Operational as the 3rd and 4th entities respectively. The Skill entity is a large domain of skills that any user who is a freelancer may have. Skill has a relationship with User because a freelancer may have multiple skills and a skill may be had by multiple freelancers. It might happen that a skill can be created but a freelancer might not add it to their profile. By maintaining an entity called Operational, we will be able to break down the relationship described in the previous sentences. Here, in the entity Operational, we will find a set of data of skills that were added by freelancers to their profile to be identified by employers in the application. In this entity, a freelancer will always have at least one operational skill added to their profile and one operational skill has to be added (otherwise it will not be deemed operational) by one freelancer once only. On the other hand, a skill will be operational only when it is added by freelancers. So by that logic, each skill has to be operational for at least one freelancer to be operational. And each operational will always be addressing one skill only because a freelancer will only be able to choose that skill once. This can be summarised in the rule here:

1. A user **must** have **at least one** operational and each operational **must** be possessed by **one** user.
2. A skill **must** represent **at least one** operational and each operational **must** be representing **only one** skill.

No other relationships can be made between the 3 entities so we will move on to the 3rd use case:

1. **Creating Job/Project:**

From here, we can consider Project to be another entity. A user who is an employer will be able to create one or many projects and assign the same or different users who are freelancers to each of them. It may also happen that an employer may not have created any project. Similarly, a freelancer may choose to do one or many projects. They may also not have chosen to do any, at a given point of time. But a project will always have to be created by an employer and each project will have one employer only. On the other hand, a project will have to be assigned to one freelancer only. But there can be a point of time when a project is open and it does not have any freelancer assigned to complete it. We can, hence, write down the following rules:

1. Each user **may** create **zero to many** projects.
2. Each project **has to be** worked upon by **at least one** user.

We can move on to the following use case, which is:

1. **Create Contract:**

Here, we can identify Contract as another entity that seems to have a relation with the Project entity. A project can be the basis of zero to many contracts, in the event that a contract between an employer and freelancer expires and they might have to create a new one to establish the new agreement, or it can happen that a project might not have yet been made into a contract. But when a contract exists, it needs to have the project mentioned for which it is being made in the first place. And a contract will always identify one project. From this, we can come up with the following rules:

1. A project **may** be a part of **none to many** contracts.
2. Each contract **must** include **only one** project.

The final use case is given below:

1. **Pay freelancer - Use Case:**

The final entity is Payment, which has a relationship with Contract. A contract may have zero payments because there may be a time when no payment has yet been made for a contract, and the same contract may have multiple payments if the employer/freelancer have agreed to make transactions in installments. But a payment will always be under one contract. From here, we find the following rules:

1. A contract **may** generate **zero to many** payments.
2. A payment **must** always be enforced for **one** contract.

When an employer makes a payment, the payment will have to go to a freelancer. A user may have multiple billing information, and every billing information must be attributed to one to many users. So considering Billing Information as another entity, we have the following business rules:

1. A user **may** possess **zero to many** billing information.
2. Every billing information **must** be attributed to **at least one** user.

Again, a user may make no payment at all and then that same user may also make multiple payments. On the other hand, every user may or may not acknowledge payments. But every payment must be made by just one user, and every payment must be sent to one user. From this, we have the following business rules for the two relations:

1. A user **may** make **zero to many** payments.
2. Each payment **must** be made by **at most one** user.
3. A user **may** be sent **zero to many** payments.
4. Each payment **must** be sent to **one** user.

A billing information may be used by a payment, or multiple payments. It could also happen that a billing info has never been used by a payment. But every payment must use only one billing information. The following business rules demonstrate the paragraph above:

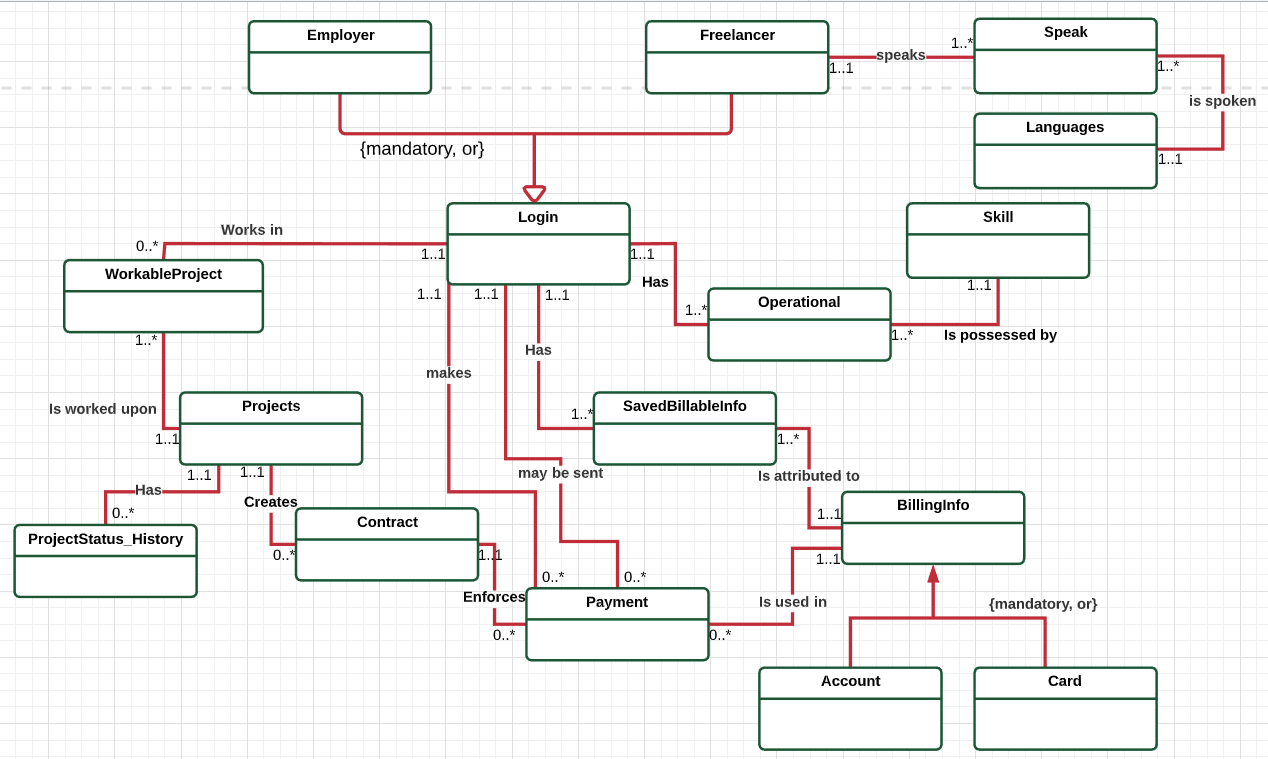
1. A billing information **may** be used in **zero to multiple** payments.
2. Each payment **must** use **only one** billing information.

A billing information can be further divided into two other entities: Account and Card. Each billing info has to be either an account or a card, but cannot be both. The business rule can be written as follows:

1. Each billing information **must** be of an account **or** a card.

**Conceptual Entity‐relationship diagram (REVISED):**

The entity-relationship diagram that represents all of the rules above including entities: Login (User), Employer, Freelancer, Skill, Operational, Projects, Contract, BillingInfo, Account, Card, Payment and Languages is shown below:



This ERD was made using LucidChart. I have added a supertype Login which acts as a parent to subtypes Employer and Freelancer. The relationship, as described in rule#1, is representing a disjoint and totally complete constraint. This is shown by the {mandatory, or} text.

Another supertype is BillingInfo which acts as a parent to subtypes Account and Card. The relationship is disjoint and totally complete, so the text {mandatory, or} has been used.

Also, the relationship between Login and Projects is a M:N project so I have created a bridge entity called WorkableProject that has 1:M relationships with both Login and Projects.

Similarly, the relationship between Login and BillingInfo was also M:N, so I created a bridge entity called SavedBillingInfo which has 1:M relationships with both the entities.

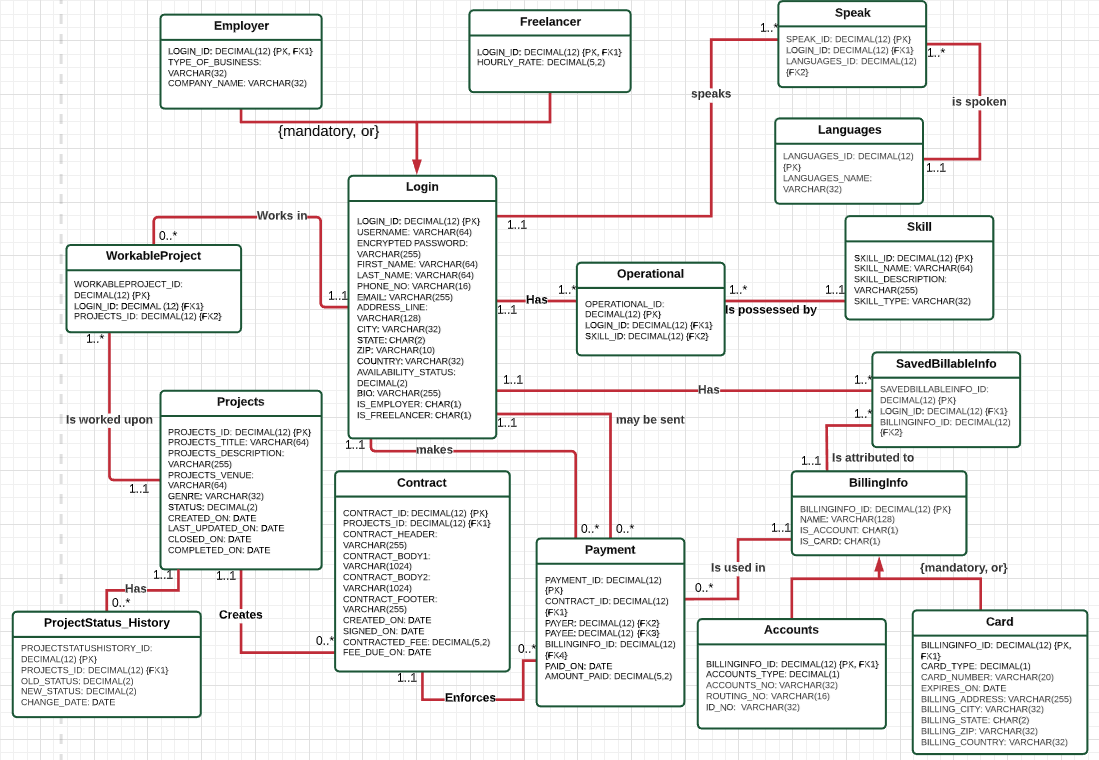
The final M:N relationship was between Login and Languages, which has been broken down using two 1:M between Login and Speak, and Languages and Speak.

I also want to point out that there are two relationships between Login and Payment. One refers to who sent the payment, while the other describes the recipient of the payment.

The ProjectStatus\_History table has been added to maintain an archive of statues that each project has been through throughout its lifetime.

**Full DBMS Physical ERD (REVISED):**

The full DBMS ERD is given below:



1. **Associative Relationships:**

All the relationships here are 1:M. So, I placed the Primary Key of the one side of the relationship as a foreign key in the many side of the relationship. I have used surrogate keys in all the tables. There was a M:N relationship between Login and Projects, and Login and BillingInfo, but I had disintegrated it into two 1:M relationships, one between Login and WorkableProject, and the second between Projects and WorkableProject. For the second pair, the 1:M relationships existed between Login and SavedBilliableInfo, and BillingInfo and SavedBillableInfo. Another M:N was broken down into two 1:M between entities User and Speak, and Speak and Languages. The primary keys from Login and Projects, Login and BillableInfo, and Login and Languages have been placed as foreign keys in the WorkableProject, SavedBillableInfo, and Speak tables respectively.

1. **Specialization-Generalization Relationships:**

There exists 2 relationships like that and those totally complete disjoint relationships exist between supertype Login and subtypes Employer and Freelancer, and supertype BillingInfo and subtypes Account and Card. As the rules go, the primary keys of Login and BillingInfo are inherited respectively by each of their subtypes as both Primary and Foreign keys.

1. **Normalization:**

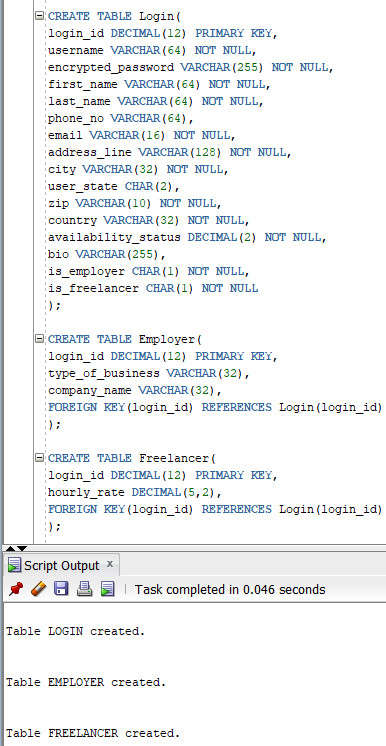
I did not find any repeating groups in the entities, nor did there seem to be any partial or transitive dependencies between the attributes. Although the same City, ZIP, State and Country might be used by multiple rows in the User and Card tables and the zips might be dependent on the cities and states, it seemed feasible to me to rather keep them as denormalized because the address fields are kept for documentation. Besides, the intention of this application is to connect employers and freelancers from across the world, so there will be a lot more data than can be thought of. It seemed more practical to me to keep them in the User/Card entities.

**Creating Tables, Attributes, Constraints (REVISED):**

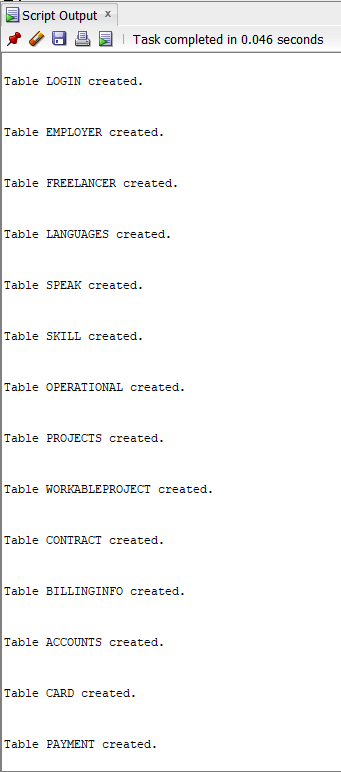
Here are my attributes and the reasonings behind each of them:

| **Table** | **Attribute** | **Datatype** | **Reasoning** |
| --- | --- | --- | --- |
| Login | username | VARCHAR(64) | There needs to be credentials for users to login with that is easy to remember for them. |
| Login | encrypted\_password | VARCHAR(255) | This will be needed to secure the account. |
| Login | first\_name | VARCHAR(64) | This will be the first name of the account holder. |
| Login | last\_name | VARCHAR(64) | This will be the last name of the account holder. |
| Login | phone\_no | VARCHAR(64) | This will be a contact number of the account holder. |
| Login | email | VARCHAR(16) | This is the email so that they can get codes, newsletter, password resets etc. |
| Login | address\_line | VARCHAR(128) | This is their street address so that it is easier to reach them by freelancers in case of emergency. |
| Login | city | VARCHAR(32) | This is the name of the city. |
| Login | user\_state | CHAR(2) | Since all states are represented by two characters, I decided to keep this as CHAR(2) |
| Login | zip | VARCHAR(10) | Most zip codes are within 10 characters. |
| Login | country | VARCHAR(32) | This is the name of the country since there might be account holders from across the world. |
| Login | availability\_status | DECIMAL(2) | By default, the availability\_status of the user will be available. Each status will be represented by a number: 1, 2, 3...and so on. |
| Login | bio | VARCHAR(255) | This will be a short bio about the user where they may introduce themselves. |
| Login | is\_employer | CHAR(1) | This will be a Y if the user is an employer, N if freelancer. |
| Login | is\_freelancer | CHAR(1) | This will be a Y if the user is a freelancer, N if employer. |
| Employer | type\_of\_business | VARCHAR(32) | This is a string using which the employer will specify what business they are in. |
| Employer | company\_name | VARCHAR(32) | This is the name of the company the employer works in. |
| Freelancer | hourly\_rate | DECIMAL(5,2) | This is the hourly rate the freelancer asks for their service. |
| Languages | languages\_name | VARCHAR(32) | This will be the name of the language, for example, Bengali, English etc. |
| Skill | skill\_name | VARCHAR(32) | This is the name of the skill, for example, play guitar, teach music etc. |
| Skill | skill\_description | VARCHAR(255) | This is a description of the skill should the user want one. |
| Skill | skill\_type | VARCHAR(32) | This is the genre of the skill. So if the skill is playing guitar, the type would be music. |
| Projects | projects\_title | VARCHAR(32) | This is the title of the project. |
| Projects | projects\_description | VARCHAR(255) | This is a description of the project which will let the freelancer know what the project is about. |
| Projects | projects\_venue | VARCHAR(64) | This is the place where the project will need to be executed. |
| Projects | genre | VARCHAR(32) | This is the genre of the project, for example, if it is a project for a music gig, the genre will be music. |
| Projects | status | DECIMAL(2) | This is the status of the project, ex - Newly Created, Closed, Completed etc. These status will be represented by numbers in the database. |
| Projects | created\_on | DATE | This is the date the project was created on. |
| Projects | last\_updated\_on | DATE | This is the date the project was last updated on. |
| Projects | closed\_on | DATE | This is the date the project was closed, as in, when the project was assigned to someone. |
| Projects | completed\_on | DATE | This is the date the project was marked as done. |
| Contract | contract\_header | VARCHAR(255) | This is the header of the contract, since most contracts are divided into paragraphs. |
| Contract | contract\_body1 | VARCHAR(1024) | This is the body of the contract. |
| Contract | contract\_body2 | VARCHAR(1024) | This is another field that will document the contract body should the total characters in the previous body overflow. |
| Contract | contract\_footer | VARCHAR(255) | This is the footer of the contract. |
| Contract | created\_on | DATE | This is the date the contract was created on. |
| Contract | signed\_on | DATE | This is the date the contract was signed on. |
| Contract | contracted\_fee | DECIMAL(5,2) | This is the fee that has been decided in the contract. |
| Contract | fee\_due\_on | DATE | This is the date by which the fee needs to be paid by the employer. |
| BillingInfo | name | VARCHAR(128) | This is the name of the account/card holder |
| BillingInfo | is\_account | CHAR(1) | If the billing info is of an account, the value will be Y for this field, otherwise no. |
| BillingInfo | is\_card | CHAR(1) | If the billing info is of a card, the value will be Y for this field, otherwise no |
| Accounts | accounts\_type | DECIMAL(1) | This will denote if the account is checking or savings |
| Accounts | accounts\_no | VARCHAR(32) | This is the account number |
| Accounts | routing\_no | VARCHAR(16) | This is the routing number |
| Accounts | id\_no | VARCHAR(32) | This is an identification number to verify if the credentials provided are accurate. |
| Card | card\_type | DECIMAL(1) | This will denote if the card is credit or debit |
| Card | card\_no | VARCHAR(32) | This is the card number, it is usually 16 digits but I have kept a longer value just in case some other country’s card number is longer. |
| Card | expires\_on | DATE | This is the date the card expires on. |
| Card | billing\_address | VARCHAR(128) | This is the street address on the card. |
| Card | billing\_city | VARCHAR(32) | This is the city on the card. |
| Card | billing\_state | CHAR(2) | This is the state (if applicable) on the card. |
| Card | billing\_zip | VARCHAR(10) | This is the zip on the card. |
| Card | billing\_country | VARCHAR(32) | This is the country on the card. |
| Payment | paid\_on | DATE | This is the date on which the payment was made. |
| Payment | amount\_paid | DECIMAL(5,2) | This is the amount paid in one particular instance of payment. |
| ProjectStatus\_History | old\_status | DECIMAL(2) | This is the value to hold the old status of the project that has been just changed. The datatype is exactly the same as the one for the status field in the Projects table. |
| ProjectStatus\_History | new\_status | DECIMAL(2) | This is the value to hold the new status of the project that has been just changed to. The datatype is exactly the same as the one for the status field in the Projects table. |
| ProjectStatus\_History | change\_date | DATE | This is the date on which the status is changed. |

Here is a screenshot of creating the first 3 tables:

****

Here is a screenshot of the execution of all the tables.

****

****

**Index Identification and Creations (REVISED):**

1. **Identifying all primary keys:**

Across the 15 entities, I have listed below the primary keys that have already been indexed:

* Login.login\_id
* Employer.login\_id
* Freelancer.login\_id
* Languages.languages\_id
* Speak.speak\_id
* Skill.skill\_id
* Operational.operational\_id
* Projects.projects\_id
* WorkableProject.workableproject\_id
* Contract.contract\_id
* BillingInfo.billinginfo\_id
* Accounts.billinginfo\_id
* Card.billinginfo\_id
* SavedBillableInfo.savedbillableinfo\_id
* Payment.payment\_id
* ProjectStatus\_History.projectstatushistory\_id

1. **Identify all foreign keys:**

Foreign keys should always be indexed. All my foreign keys are listed below along with whether they are unique or not and why so:

| **Column** | **Unique?** | **Description** |
| --- | --- | --- |
| Speak.login\_id | Not unique | A user can speak many languages, so the index on this foreign key is not unique. |
| Speak.languages\_id | Not unique | A language can be spoken by multiple users, so the index on this key is not unique. |
| Operational.login\_id | Not unique | A user can have multiple operational skills, so this key is not unique. |
| Operational.skill\_id | Not unique | The same skill can be operational for multiple users, so this key is not unique. |
| WorkableProject.login\_id | Not unique | A user can be working in multiple projects, so this key is not unique. |
| WorkableProject.projects\_id | Not unique | The same project can be worked upon by multiple users, so this key is not unique. |
| Contract.projects\_id | Not unique | The same project can be used to create multiple contracts, so this key is not unique. |
| SavedBillableInfo.login\_id | Not unique | A user may have multiple billable info. This key is hence not unique. |
| SavedBillableInfo.billinginfo\_id | Not unique | The same billing info might be used by multiple users who use the same account/credit card for example, so this key is not unique. |
| Payment.contract\_id | Not unique | The same contract might enforce multiple payments, so this key is not unique. |
| Payment.payer | Not unique | The same payer might make multiple payments, so this key is not unique. |
| Payment.payee | Not unique | The same payee might receive multiple payments, so this key is not unique. |
| Payment.billinginfo\_id | Not unique | The same billing info might be used in multiple payments, so this key is not unique. |
| ProjectStatus\_History | Not unique | The same project’s status might be changed multiple times, so this key is not unique. |

1. **Identify three query driven indexes:**

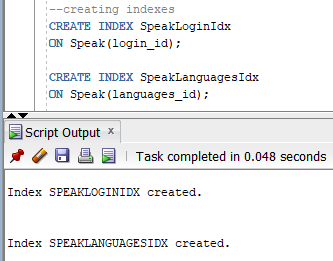
1. The first query driven index I have thought of is **Projects.projects\_title**. An employer may want to know how much amount remains to be paid for a particular project. The index will be non-unique because multiple projects may have the same title.

2. The second query driven index is **Projects.created\_on**. Analysts may want to see the trajectory of the number of projects being created on Artistree. This will also be non-unique because multiple projects may be created on the same day.

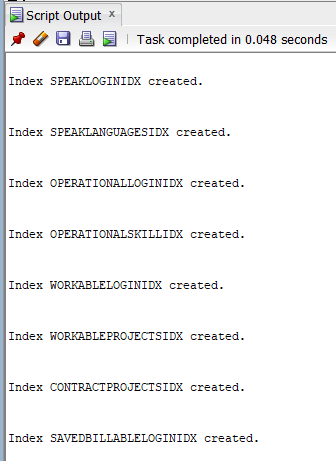
3. Finally, the third query driven index is **Projects.Completed\_on**. An employer may want to know how many projects they have completed within a certain date range. This will be non-unique too because multiple projects could have been completed on the same day.

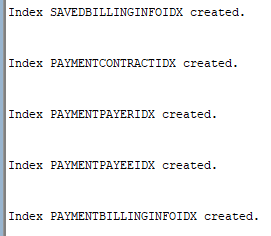
1. **Creating the indexes:**

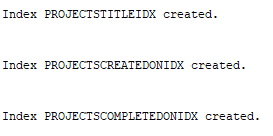
Here is a screenshot of the index creation query and output:



Here is a screenshot of the execution of all the indexes created:



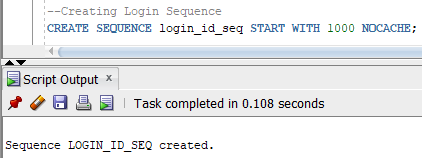




**Stored Procedure Execution and Explanations:**

Creating an account as an employer or a freelancer (Use Case # 1):

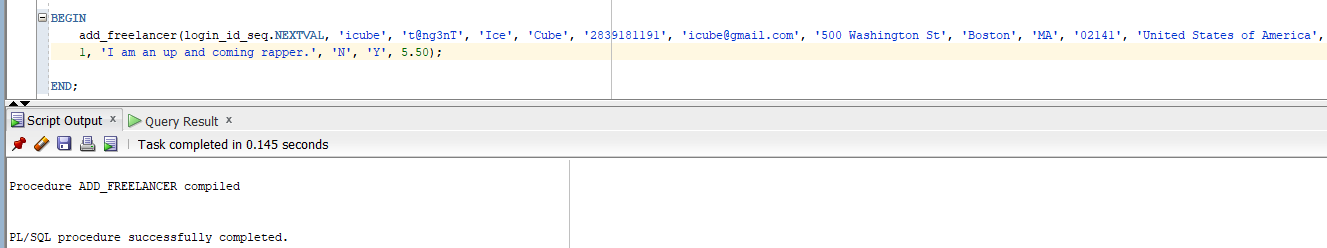
At first, I created a sequence to maintain login\_id’s of the Login, Employer, Freelancer tables:



Next, I created the procedure to create a login entry and associated Freelancer entry too. Here, I have listed all attributes of Login and Freelancer table so that by calling one procedure, I will have data populated in both the Login and Freelancer table:



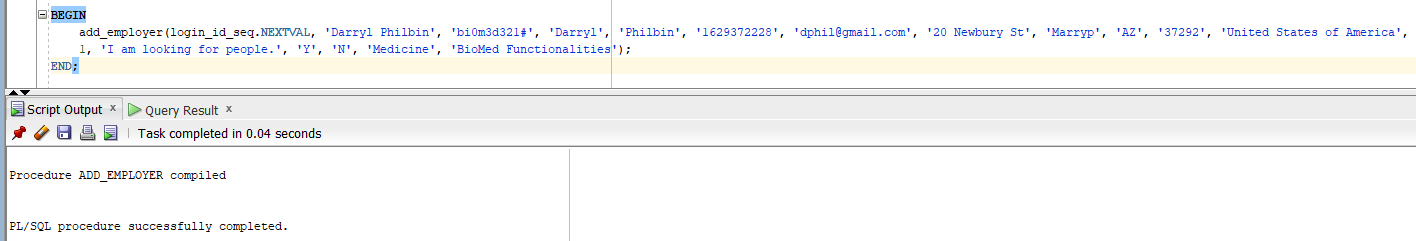
Finally, I executed the procedure adding an entry to the login table and the corresponding entry for the Freelancer table. The data are dummy but they have been made to look realistic and they fall within the datatypes they have been defined in:



Similarly, I have also created a procedure for adding employer, replacing the attributes for Freelancer table with those of Employer’s:



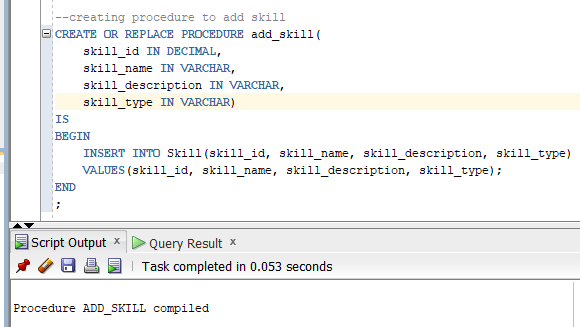
Executing it:



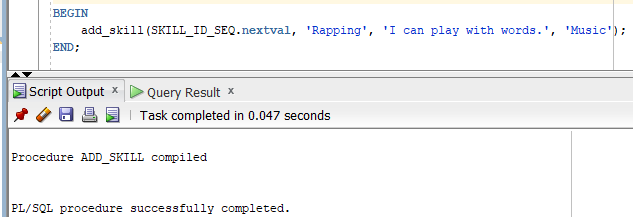
Now that I have inserted rows in the Login table, I can move forward with the next use case.

Creating skills (Use Case # 2):

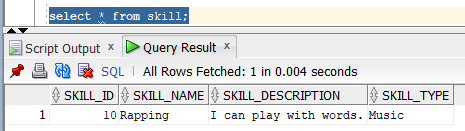
For the second use case, I have created a procedure that will add skills:



This is the execution of the add\_skill procedure. Using the sequence.nextval keyword, I will get a unique value for the primary key of this table and all the tables that I have added sequences for:

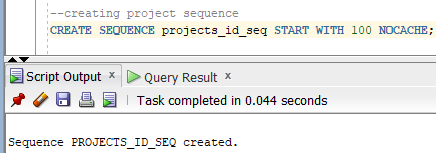


This is the result of adding a row in skills table:

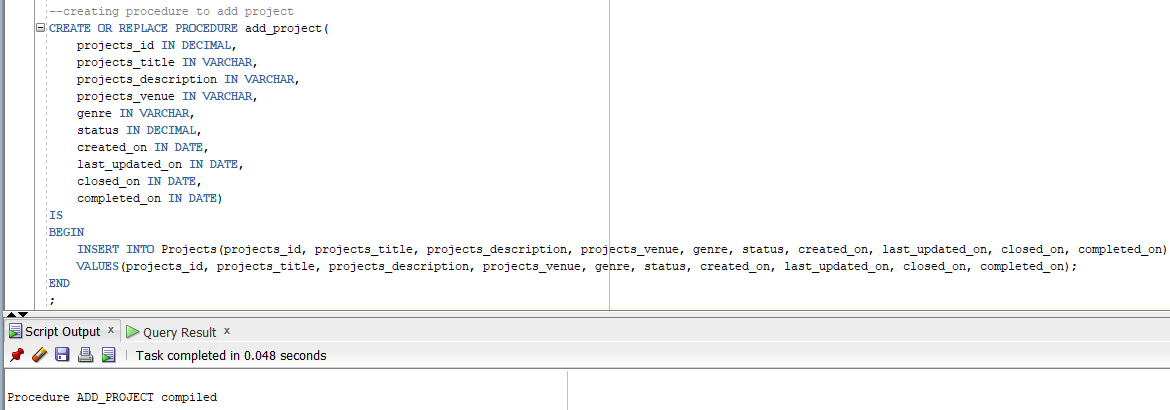


Creating Job/Project (Use Case # 3):

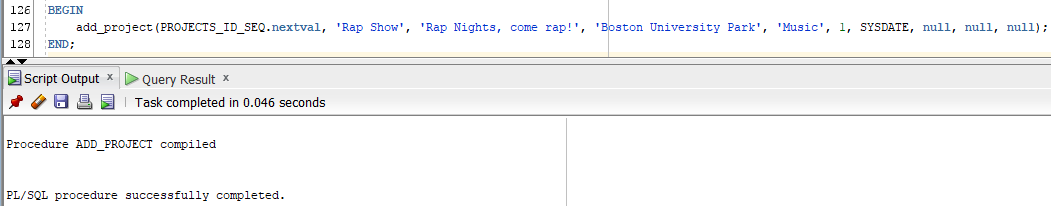
First, I have created a sequence to address the project\_id:

****

Next, the procedure to create new project is created:



Finally, the procedure is executed. I am inserting SYSDATE in the created\_on field because the project will be created on whichever day is current. So instead of manually putting down today’s date, I have put down whatever date it is in the system:

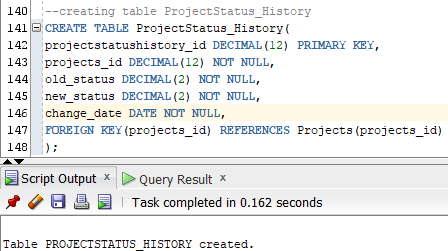


**Trigger Creation and Use:**

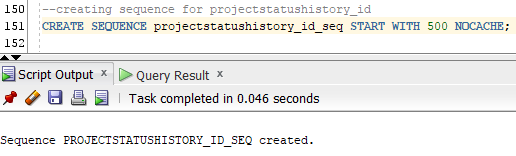
The ProjectStatus\_History table has been added in the conceptual diagram above and as well as in the logical ERD. The table consists of a primary key, a foreign key that is attributed to the Projects table, a field to hold the old status of the project, one to hold the new, and finally the date to track when the status was changed.

The relationship between the tables Projects and ProjectStatus\_History is 1:M where each project may or may not have changes in statuses but each ProjectStatus\_History will have to be attributed to at most one project.

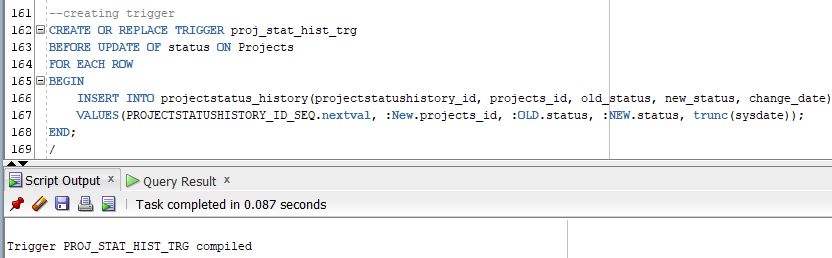
Below is a screenshot of creating the table with all its attributes and constraints:



I have created a sequence for this table to properly maintain the uniqueness of the primary key:



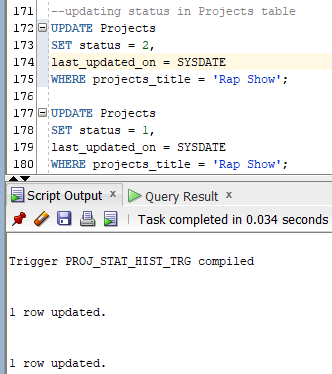
The trigger is created as follows:



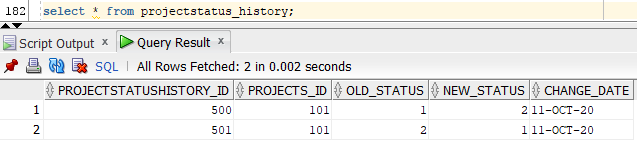
Here, before the update query on the status field in Projects table is executed, the trigger will be executed and a corresponding entry will be inserted in the ProjectStatus\_History table for that project. I have specifically mentioned the status column in the trigger because unless there is an update on that field, there will not be any entry in the history table.

The trigger will enter the next unique value that the sequence generates as the primary key, the corresponding projects\_id, the old status, the new status and the system date truncated to show only the date field.

The update queries are given below:



I have updated the same project twice to show some data volume in the history table. The history table now shows the following values:



The first update changes the status from 1 to 2, the second update from 2 to 1. Both updates have been documented correctly in the table.

**Question Identification and Explanations:**

1. Users using the Projects module are from which countries and what is their number?

The data analysts of the application might want to know where their main demographic is from who are actively using the application. The main objective of this application is to give and get jobs, and that is basically done using the Projects module of the application. Moreover, that the Projects module is being utilised correctly can be identified if the status of the projects is 3 = Closed.

1. Which project was most worked on a year?

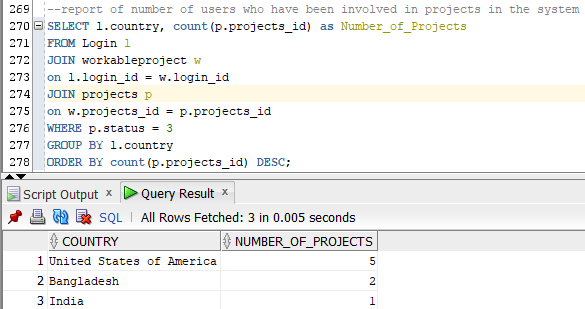
Data analysts may also want to know which projects were most worked on in a given year and what their genres were. This will give them an idea about what type of projects are mostly worked on in the application. The count of project status update histories will tell them how many times the project was worked on, as in, how many times the projects status changed will give an idea that the project was closed, published, and maybe completed as well because there were changes in the status. Since the employer is the one changing the statuses, their names and companies will also be listed for clarity.

1. How much was the amount of transactions done in a given year?

Data analysts may want to know how much their payment module is being used. If the application decides they want to have a Business Intelligence dashboard that will show them how users are using the utilities provided to them, they will most definitely want to see the amount of transactions made. So, in the application, if they select the year, the query will show them the total amount of the transactions grouped by month.

**Query Executions and Explanations:**

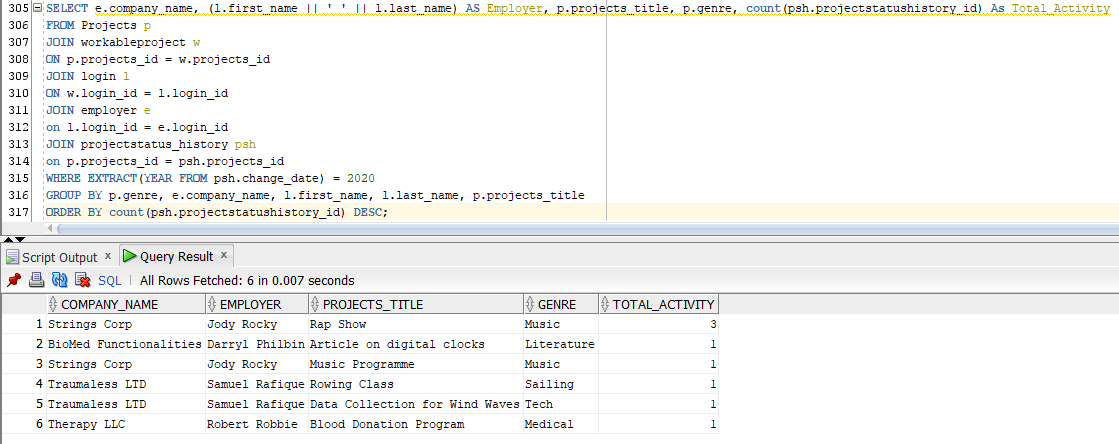
1. Report of the countries users in closed projects are from:



Here is the explanation of the query:

| **Code** | **Explanation** |
| --- | --- |
| SELECT l.country, count(p.projects\_id) as Number\_of\_Projects | Here, I am selecting countries to show from the login table and the number of projects as an aggregated number against each country. |
| FROM Login l  JOIN workableproject w  on l.login\_id = w.login\_id  JOIN projects p  on w.projects\_id = p.projects\_id | Here, I am joining tables Login, workableProject and Projects so that I am able to choose attributes from Login and Projects tables. |
| WHERE p.status = 3 | This is the where clause restriction where I am restricting the rows to only those projects which have been completed. |
| GROUP BY l.country | The same countries are grouped together in order to show the counts of projects against each group of country. |
| ORDER BY count(p.projects\_id) DESC; | The countries are finally ordered in the order of the number of projects they are doing starting from the most number of projects to the least. |

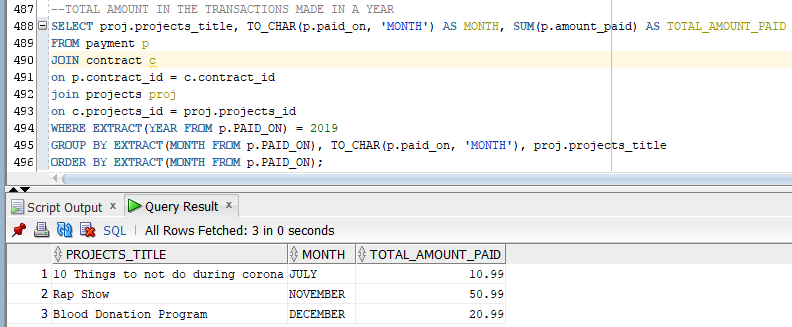
1. Report of which projects have been worked on in 2020:



Here is the explanation of the query:

| **Code** | **Explanation** |
| --- | --- |
| SELECT e.company\_name, (l.first\_name || ' ' || l.last\_name) AS Employer, p.projects\_title, p.genre, count(psh.projectstatushistory\_id) As Total\_Activity | Here the company’s name, the name of the employer concatenated as one attribute, the project title, the genre, and the count of the total number of times the status changed is selected. |
| FROM Projects p  JOIN workableproject w  ON p.projects\_id = w.projects\_id  JOIN login l  ON w.login\_id = l.login\_id  JOIN employer e  on l.login\_id = e.login\_id  JOIN projectstatus\_history psh  on p.projects\_id = psh.projects\_id | Here, 5 tables are joined to accumulate data from all the tables. |
| WHERE EXTRACT(YEAR FROM psh.change\_date) = 2020 | The year is extracted from the change\_date to check for the changes made in the entire year of 2020. |
| GROUP BY p.genre, e.company\_name, l.first\_name, l.last\_name, p.projects\_title | The counts are grouped by the attributes in the select statement |
| ORDER BY count(psh.projectstatushistory\_id) DESC; | The result set is ordered in the descending order of number of changes. |

1. Report of how much fees were paid in the year 2019:



Here is the explanation of the query:

| **Code** | **Explanation** |
| --- | --- |
| SELECT proj.projects\_title, TO\_CHAR(p.paid\_on, 'MONTH') AS MONTH, SUM(p.amount\_paid) AS TOTAL\_AMOUNT\_PAID | Here, we are showing the name of the project for which some fees have been paid off in the grouped months and we are also showing how much has been paid in those grouped months. |
| FROM payment p  JOIN contract c  on p.contract\_id = c.contract\_id  join projects proj  on c.projects\_id = proj.projects\_id | Here, we are joining the tables which will be used to show data in the report. For example, we are showing the title of the project, so we have joined the Projects table. |
| WHERE EXTRACT(YEAR FROM p.PAID\_ON) = 2019 | Here, we are extracting the year from the date the fees were paid in order to restrict the resultset by the year specified. (2019 in this case) |
| GROUP BY EXTRACT(MONTH FROM p.PAID\_ON), TO\_CHAR(p.paid\_on, 'MONTH'), proj.projects\_title | We are grouping by the attributes mentioned in the SELECT clause. We are also grouping by the month extracted from the date field, because it returns the row. This will be useful when we order by months. |
| ORDER BY EXTRACT(MONTH FROM p.PAID\_ON); | When we use TO\_CHAR, the months in order by will be ordered by the first letter of the name of the month. Rather, we want to show in order of the number of the month. This is why we used the EXTRACT function in the group by clause to use it in the order by clause as well. |

**Summary and Reflection:**

In this iteration, I have revised my conceptual ERD to accommodate a history table that will track the change of project statuses. Consequently, I have also changed the Logical ERD. I also created the table in sql and added an index to the foreign key projects\_id.

I created stored procedures to insert data to the tables. I also created a trigger that would be executed every time there is a change in the status table. This trigger will insert a row in the status history table with corresponding values. I identified three questions that data analysts would be able to use to analyze long term data. As this application grows bigger and more widespread, I hope to have a business intelligence team where these queries and the idea behind those will come in use.

This application will be a hub for job finders and recruiters alike. Here, starting from recruiting to paying the respective fees to the job-doer, I have tried to maintain all the steps and bound them together using entity relationships. This is not the final design but I feel like I have built an almost concrete foundation to build something on.