CS 699 - Database Design Document

# Project: T-Join

# Iteration: 5

# Author: Han Xu

# Email: [han10@bu.edu](mailto:han10@bu.edu)

# Date: October 11, 2020

## Project Direction Overview

At work I am improving the tracking processes of mandatory employee online training and onboarding activities. The progress of each employee is stored across multiple Learning Management Systems, making it hard for the data to be centralized and disseminated across other areas of the business. To solve this, I would like to design a database that houses all the training data, so that stakeholders such as team managers can look up the monthly learning progress of their team. By creating relationships from training data in disparate systems, I am essentially executing a big SQL join statement, hence the project name T-Join.

The primary client programs that will make use of this database are BI dashboards and web applications. For example, business analysts or managers should be able to connect to the database via Tableau or PowerBI, and extract data that they can visualize in a meaningful way. System administrators should be able to use a form (via the frontend of a web app) to insert new records into the database, when a new employee is onboarded for example. Given these business requirements, the database will need to serve both analytical and transactional purposes.

## Use Cases and Fields

Use Case 1: Adding An Employee

As a Systems Administrator, I would like to add a new record for an employee that recently joined the organization, so that their activities can be logged against their unique account.

User is Sys Admin:

1. User logs into the T-Join’s web portal.
2. User clicks the ‘create account’ button to be taken to a form.
3. User enters the information of the new employee into the form.
4. User clicks the ‘add record’ button to add the employee as a record into the database.

The following database fields are needed to store information about an employee. This information is needed to provide more context to their corresponding training data.

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| employee\_id | The unique identifier of an employee. | Used to differentiate between employees in the system, instead of relying on similar names/emails. |
| first\_name | The first name of an employee. | Used to display the first name of an employee on a web transcript or report. |
| last\_name | The last name of an employee. | Used to display the last name of an employee on a web transcript or report. |
| email | The email of an employee. | Used for automatic notifications to the employee from the web app, or manual outreach by managers. |
| title | The role of the employee within the organization. | Used to determine which training activities are needed for the employee. |
| joined\_date | The date on which the employee joined the organization. | Used for calculating the time it takes for the employee to complete onboarding training since they joined. |

Here are the fields for an employee’s team:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| team\_id | The unique identifier of a team. | Used to differentiate between teams in the system, instead of relying on similar names/emails. |
| manager\_first\_name | The first name of a team’s manager. | Used to display the first name of a manager that leads the team. |
| manager\_last\_name | The last name of a team’s manager. | Used to display the last name of a manager that leads the team. |
| manager\_email | The email of a team’s manager. | Allows business users to contact the manager regarding a team’s progress. |
| team\_name | The email of a manager. | Used for automatic notifications to the team manager from the web app, for updates on the employee. |
| employees | The list of team members stored in a related table. | Used to retrieve the list of employees that belongs to the team. |

Here are the fields for an employee’s department:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| department\_id | The unique identifier of a department | Used to differentiate between departments in the system, instead of relying on similar names. |
| department\_name | The name of a department. | Used to display the first name of a department in reports. |
| function | The primary goal of a department. | Used to display the departmental function in reports. |
| employees | The list of employees belonging to a department stored in a related table. | Used to display the list of employees that belong to a department. |

Use case 2: Reporting Training Progress

As a Team Manager, I would like to pull up a training report for the employees on my team, so that I can track their progress during the onboarding process.

User is Team Manager:

1. User logs into the T-Join’s web portal.
2. User clicks the ‘new report’ button to see a list of employees on his team.
3. User filters the list within certain parameters such as dates, completions, etc.
4. User clicks ‘download csv’ to export the final list into a csv file for offline viewing.
5. User presents the report to team members about which activities still need completing.

In addition to the employee personal information listed in the database fields of use case 1, the following fields related to enrollment data are also needed to realize use case 2. These fields essentially represent the transcript of an employee.

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| open\_tasks | Number of assigned training activities an employee still needs to complete. | Allows reporting mechanisms within a web app or BI tool to show how much onboarding work an employee still needs to do. |
| latest\_active\_date | The last day on which an employee consumed training on one of the training platforms. | Allows reporting mechanisms to show how long it’s been since an employee has taken training. |
| enrollments | The list of enrollments in training activities of an employee, stored in a related table. | Used to display the transcript to an employee or a report to a manager. |
| certifications | The list of certifications an employee has obtained, stored in a related table. | Used to display the certifications an employee has obtained. |
| session\_time | The total number of hours an employee has spent in any given training platform. | Used to show how many total session hours an employee has spent on training. |

Use case 3: Creating A Dashboard

As a Business Analyst, I want to create a Business Intelligence dashboard that visualizes how much training progress is made across multiple teams, and which training items are causing bottlenecks within the onboarding process.

User is Business Analyst:

1. User logs into the BI application.
2. User connects the BI app to the database, which is hosted on a cloud service.
3. User filters the data within the dashboard to only include the necessary fields.
4. User adds a chart to visualize the general progress of each team.
5. User adds a chart to visualize the time spent on each course.

In addition to use cases 1 and 2, this use case also needs database fields to store specific course and certification information. Only then can a BA say something like: “70% of users across all teams take on average twice as long to complete course XYZ, compared to other courses.”

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| course\_id | Unique identifier for a training course. | Used to differentiate between specific courses within the system, instead of relying on similar course titles. |
| course\_title | The name of a course that is referenced by managers and employees. | The main course reference for end-users (managers, analysts, etc.) in reporting. |
| course\_duration | The total estimated length of the course in minutes. | Used to compare expected durations versus the amount of time an employee actually spends in a course. |
| updated\_date | The date on which the course was added to the system. | Used to track when the last revision occurred for a course, which is needed to plan future updates. |

Please note the ‘course’ prefix is in the above field names are included to show context, but is not needed in a real database, as each fields will likely belong to a ‘courses’ table. As mentioned above, a similar table would be needed for certifications:

* cert\_id
* cert\_title
* cert\_duration
* updated\_date

Certifications are exam-based content as opposed to self-paced learning modules, but the fields store the same type of information for the same reasons.

Use Case 4: Viewing the Transcript

As an Employee, I want to visit my transcript page, to see all the training activities that I have completed across different platforms and save evidence of completion such as certificates.

User is Employee

1. User logs into T-Join’s web portal.
2. User clicks on the My Profile link.
3. User clicks on Transcript tab to view all training activities.
4. User clicks on a certification to download pdf copy of it to their local machine.

The fields needed for this use case are already covered by the employee, activities, course, and certification fields.

## Structural Database Rules

This section lists Structural Database Rules attached to each use case in the previous section. Entities within each structural rule are also title cased.

Use Case 1: Adding An Employee

User is Sys Admin:

1. User logs into the T-Join’s web portal.

2. User clicks the ‘create account’ button to be taken to a form.

3. User enters the information of the new employee into the form.

4. User clicks the ‘add record’ button to add the employee as a record into the database.

Based on feedback to iteration 1, the employee information entered will now include manager and department information as well. Their entities are reflected in the following rules:

**Structural database rule #1:**

An Employee belongs to a Team; a Team has one to many Employees.

At any given time, an Employee must belong to a Team, because all employees must report to someone, even the CEO (in which case it will be a single board entity). A Team must have at least one employee when it is first formed. This is why Employee-Team is a many-to-one relationship, and why participation is mandatory on both sides.

Use Case 2: Reporting Training Progress

User is Team Manager:

1. User logs into the T-Join’s web portal.
2. User clicks the ‘new report’ button to see a list of employees on his team.
3. User filters the list within certain parameters such as dates, completions, etc.
4. User clicks ‘download csv’ to export the final list into a csv file for offline viewing.
5. User presents the report to team members about which activities still need completing.

**Structural database rule #2:**

An Employee may have many Enrollments; an Enrollment is tied to one Employee.

Enrollment here defines a record of a particular training taken by an Employee, meaning each Enrollment is tied to the individual. This allows Enrollment to be the bridging entity between Employee and other entities that define activity types, such as courses and certifications. It also allows the database to remain flexible, so that later on we can add other entities to the relationship such as workshops or webinars. Furthermore, now the status of the Enrollment (session time, progress) can be in the Enrollment entity, separated from the person-related info in the Employee entity. This reduces dependence between them and allows the database to scale.

A good Employee may enroll in many training activities. A bad Employee may not enroll in any. An Enrollment must be tied to one Employee (a training record should not exist if no one took it). Therefore, this rule defines a one-to-many relationship between Employees and Enrollments. Participation is optional on the Employee side and mandatory on the Enrollment side.

Use Case 3: Creating a Dashboard

User is Business Analyst:

1. User logs into the BI application.
2. User connects the BI app to the database, which is hosted on a cloud service.
3. User filters the data within the dashboard to only include the necessary fields.
4. User adds a chart to visualize the general progress of each team.
5. User adds a chart to visualize the time spent on each course.

Since Dashboard reporting usually happens at the organization level, we need to define Departments as a separate entity.

**Structural database rule #3:**

An Employee belongs to a Department; a Department has many Employees.

An Employee has to belong to a single department, because employees are always tied to a particular business function such as finance or marketing. A Department also must have many employees, because a Department of 1 or 0 doesn’t make sense even when it’s first created. Therefore, participation is mandatory on both sides of this rule, and it is a one-to-many relationship between Employees and Departments.

I have updated Enrollment as the entity that bridges Employees and a training activity. The types of activities in my project are good candidates for a Specialization-Generalization relationship. Instead of one-to-many associative relationships between Enrollment-Course and Enrollment-Certification, I would like to make Course and Certification subtypes of the Activity supertype. This will allow Course and Certification entities to borrow the same common attributes such as completion\_status, session\_minutes, title, etc. from Activity.

To support this, I have revised Use Case 4 to support both Activity subtypes, and updated structural database rules #4 and #5.

Use Case 4: Viewing courses in the transcript

User is Employee

1. User logs into T-Join’s web portal.
2. User clicks on the My Profile link.
3. User clicks on Transcript tab to view all training activities.
4. User clicks on the courses tab to see which courses are completed or still in progress.
5. User clicks on the certification tab to view all obtained certifications.
6. User clicks the download button next to a certification to save an offline copy.

**Structural database rule #4:**

An Enrollment is tied to an Activity; An Activity can have many Enrollments.

Enrollment is now the bridging entity between Employee and Activity. An Enrollment must always be a record of some Activity, or it won’t exist. An Activity may not always have anyone enrolled in it. Therefore, it is a one-to-many relationship between Activity-Enrollment. The relationship is mandatory on the Enrollment side and optional on the Activity side.

**Structural database rule #5:**

An Activity is a Course, a Certification, or none of these.

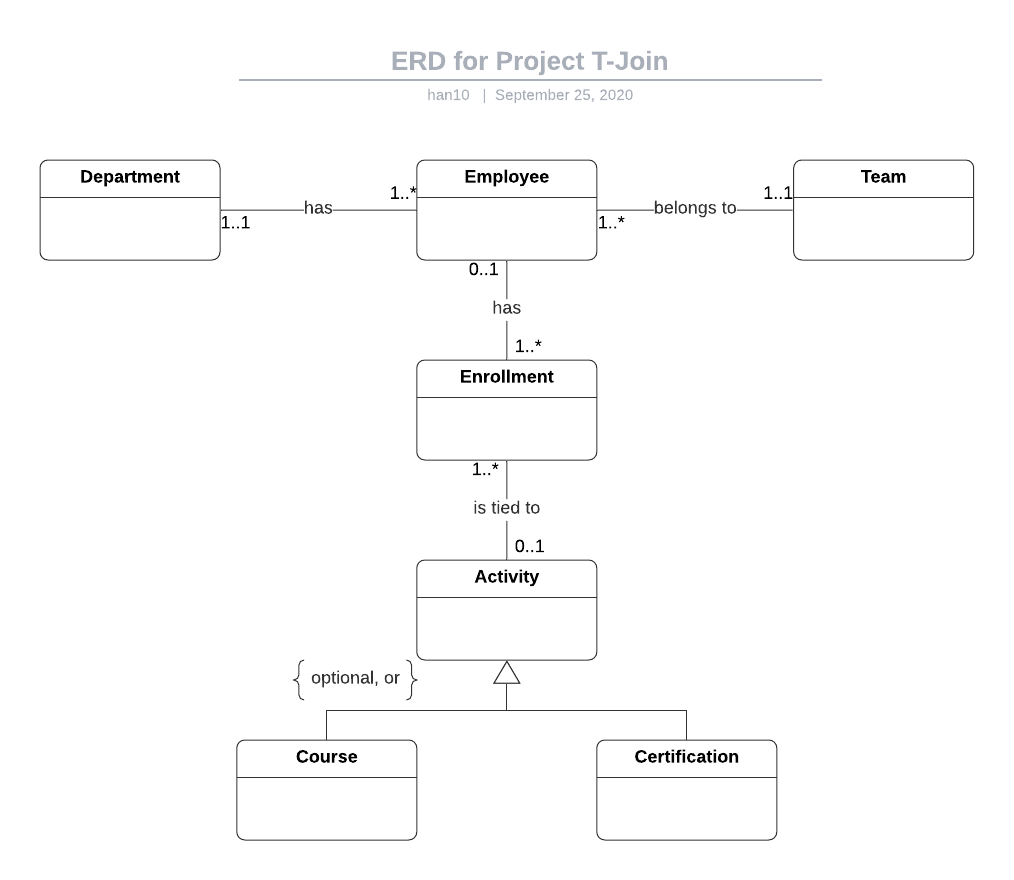
A training activity can only be of one type because it cannot be a Course and a Certification at the same time. It can also be something else entirely, because there are other types of training that employees are taking (workshops, webinars, etc.), but we are not yet tracking. Therefore, this relationship is partially complete and disjoint.

## Conceptual Entity Relationship Diagram (ERD)

Structural database rules revisited:

1. An Employee belongs to a Team; a Team has one to many Employees.
2. An Employee may have many Enrollments; an Enrollment is tied to one Employee.
3. An Employee belongs to a Department; a Department has many Employees.
4. An Enrollment is tied to an Activity; An Activity can have many Enrollments.
5. An Activity is a Course, a Certification, or none of these.

The following ERD has been updated with the specialization-generalization relationship.



This diagram shows that Course and Certification now are now subtypes of Activity, which may have Employees enrolled in it via the Enrollment bridging entity. The ‘optional’ constraint indicates partial completeness, and the ‘or’ constraint indicates a disjoint relationship as explained above.

## DBMS Physical Attributes

The following table includes the attributes for each entity. Following the example given in the instructions, I will not include the primary and foreign keys that were already defined in the previous iteration. Note that some of the previously defined fields in the first iteration are not listed here because they can be derived, such as total session time from time spent on each course for example.

**Employee table**

|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| first\_name | VARCHAR(255) | This is the first name of an employee, up to 255 characters. |
| last\_name | VARCHAR(255) | This is the last name of an employee, up to 255 characters. |
| email | VARCHAR(255) | This is the company email of an employee, up to 255 characters. |
| title | VARCHAR(255) | This is the job title of an employee, up to 255 characters. |
| employee\_type | CHAR(1) | The systems that use this database need to differentiate between fulltime employees (F), part-time (P) and contractors (C). |
| joined\_on | DATE | The date on which the employee joined is needed when deriving the time spent on training activities. |

**Department table**

|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| dep\_name | VARCHAR(255) | This is the name of the department, up to 255 characters. |
| dep\_descrip | VARCHAR(1024) | This is a short description of the department’s function, which should fit within 1024 characters. |
| dep\_email | VARCHAR(255) | This is the email alias for the department, which its employees may subscribe to for training updates. |

**Team table**

|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| team\_name | VARCHAR(255) | This is the name of the team, up to 255 characters. |
| team\_descrip | VARCHAR(1024) | This is a short description of the team’s function, which should fit within 1024 characters. |
| target\_percent | DECIMAL(5,2) | This is the target completion percentage of how many members should complete their training activities on a team. It will be used in conjunction with the actual completion percentage, which can be derived from existing attributes in other tables. Percentages are displayed using face value, which can go up to 100.00. |

**Enrollment table**

|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| last\_active | DATE | This tracks the last time the employee engaged with the training activity. |
| session\_mins | DECIMAL(6) | This tracks the total number of minutes the employee spent within the training activity. Decimal is the better type here because a) seconds are not needed in my use cases, and b) total minutes make application-level computations easier than working with datetime objects. |
| completed\* | BOOLEAN | True or False value for whether an employee has completed the activity. |
| progress\_percent\*\* | DECIMAL(3) | This tracks the percentage progress an employee has made throughout a training activity, measured in whole percentage face values up to 100. |
| cert\_code | VARCHAR(255) | This unique string is used to certify the completion of an activity up to 255 characters. Useful for validation of certificates, but optional because it does not apply to some activities. |

\*No need for equivalent started field, because there won’t be an enrollment record if not started.

\*\* Technically completed field can be replaced by when progress\_percent = 100. However, some LMS platforms do not report on course-level progress, and it also depends on whether the author has enabled it. Therefore, we must allow both simple completion Boolean (NOT NULL) as well as more granular progress percentage (optional/NULL).

**Activity table**

|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| title | VARCHAR(255) | This is the title of the activity, up to 255 characters. |
| activity\_domain | VARCHAR(255) | This is optionally used to indicate which product, service, or specialization the activity is for, up to 64 characters. |
| activity\_descrip | VARCHAR(1024) | A description of the activity up to 1024 characters. |
| format | CHAR(1) | This indicates the format of the course, such as ‘S’ for self-paced, or ‘I’ for in-person, or ‘V’ for virtual class. |
| offered\_on | TIMESTAMP | This is the exact time and date on which the activity is offered, if it has a live component (such as virtual or in-person sessions). |
| location | VARCHAR(1024) | This is either the address of the physical location where an employee may take a course/exam, or a link to an online session, up to 1024 characters. |
| price | DECIMAL(7,2) | This is the price of activities up to 7 digits with 2 decimals for standard display of currencies. For example, price of most courses are free and will be stored as 0.00. |
| duration\_mins\* | DECIMAL(5) | This is the duration in minutes for a particular activity, which can be estimated for courses or set in stone for certifications. It is used when calculating the difference between it and actual time spent. 5 digits will be enough to accommodate activities that last up to 2 months (most activities are a couple hours). |
| updated\_on\*\* | DATE | This is used to track when the activity was last updated, in order to determine content relevance. |

\*Instead of a general duration field, I could add an optional estimated\_duration field to the course table and a NOT NULL duration field to the certification table to ensure that each certification record stores an allotted time for the exam. However, we would have to repeat this for each new activity in the future.

\*\*Currently, no use case warrants the need for a similar created\_on field.

**Course table**

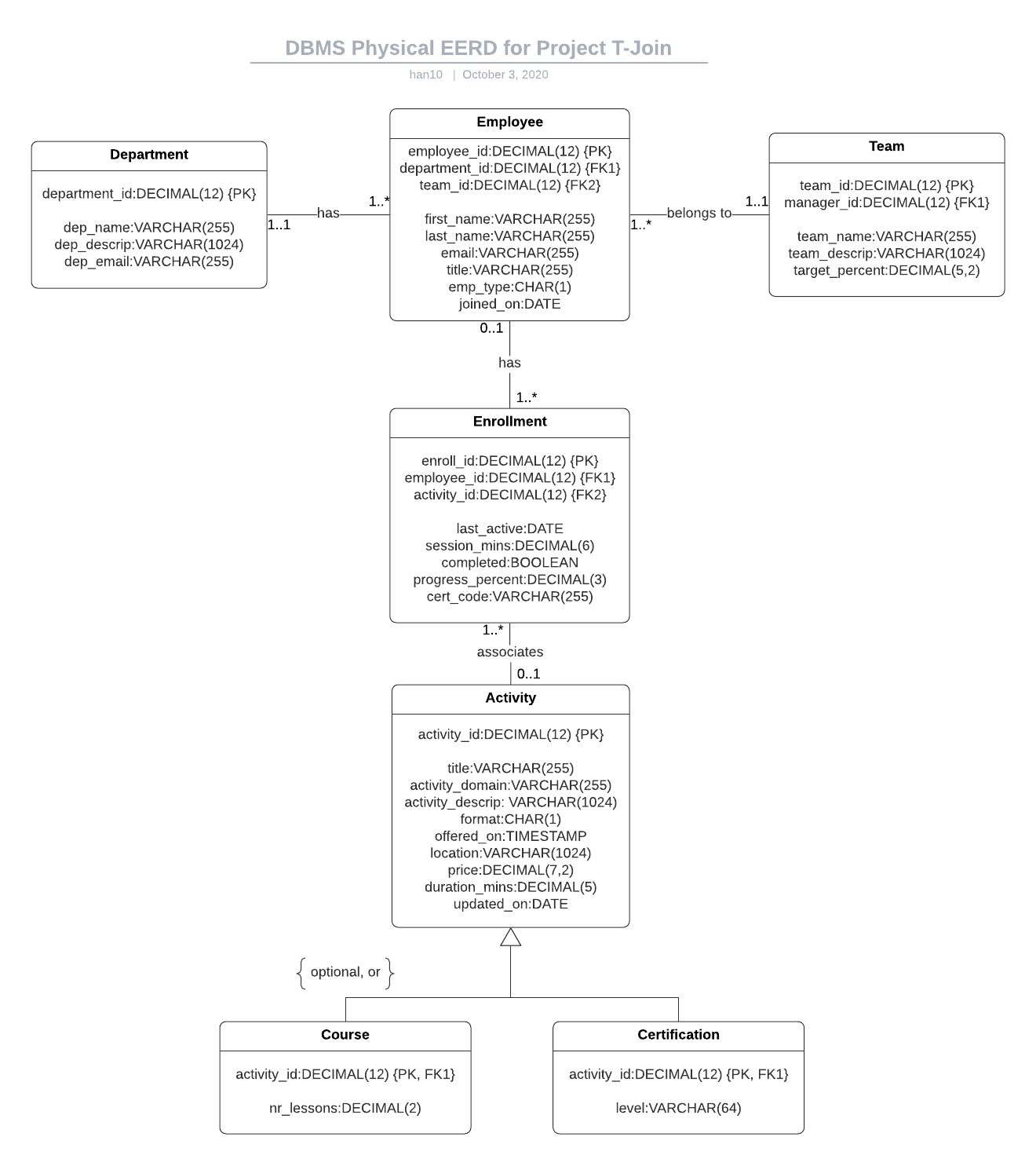
|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| nr\_lessons | DECIMAL(2) | This is the number of modules/lessons contained within a course, up to 99. Single digits for most courses, but I’ll allow some breathing room. |

**Certification table**

|  |  |  |
| --- | --- | --- |
| Attribute | Datatype | Reasoning |
| level | VARCHAR(64) | This is the level of expertise the employee is certifying for, which can be traditional names such as ‘Beginner’ or modern ones such as ‘Practitioner’ up to 64 characters. This is determined by the exam author. |

## DBMS Physical EERD

Here is the updated diagram with the above attributes included.



## Normalization

There are 3 areas with risk of redundancies:

* job\_title will be repeated in the Employee table, because many employees have the same job title. Creating a separate Job entity solves this and allows other fields to be added in the future, such as role\_description.
* Many teams will share the same target\_percent, because there are standard goals for most teams. Moving goals into its own table also allows other attributes to be added later such as plan\_name.
* In the Activity table, address will be repeated across many records because there are set locations and URLs at which courses or exams are held.

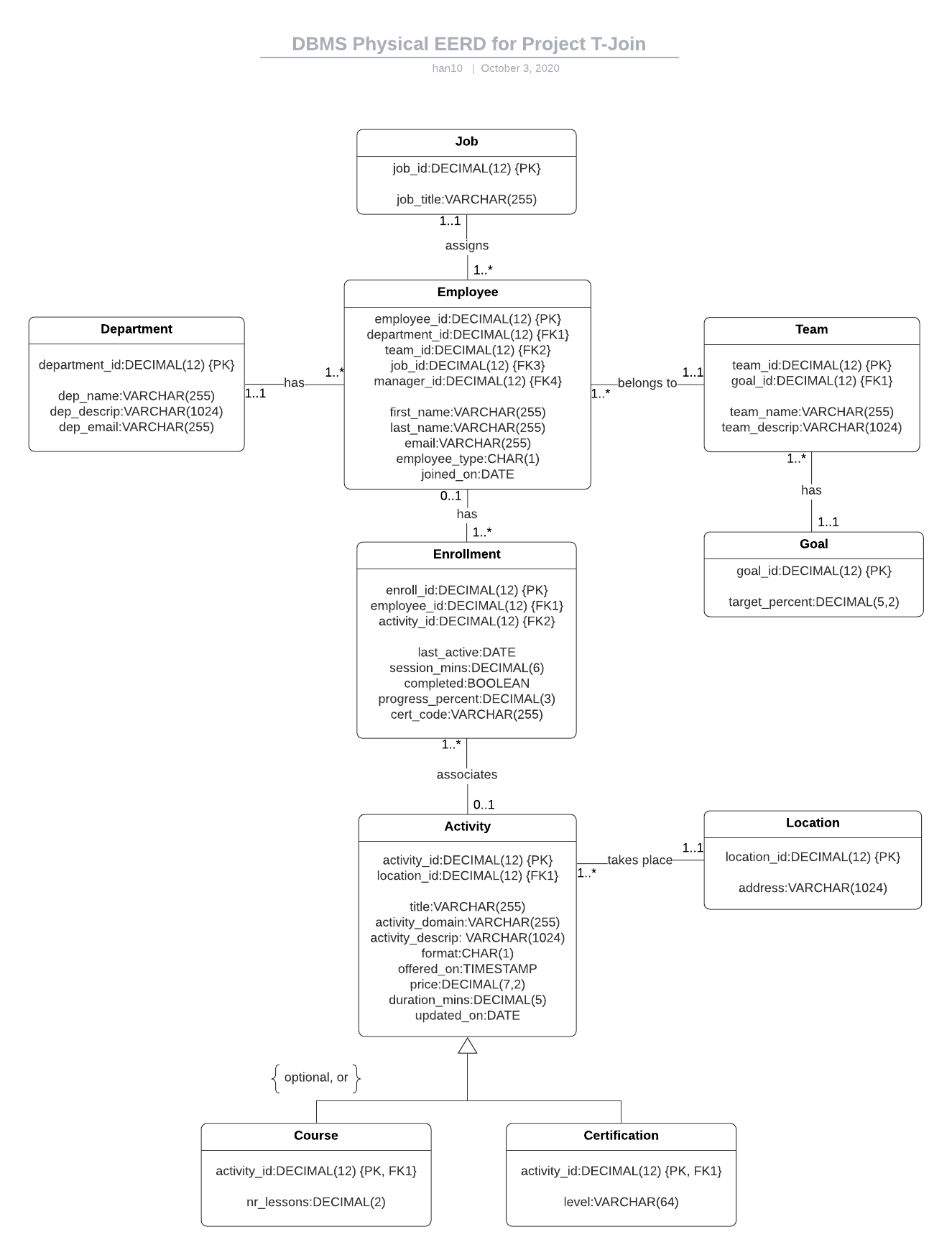
The next page contains the updated Physical DBMS EERD with entities normalized to BCNF. Here are the updated structural database rules with the new ones italicized:

Structural database rules revisited:

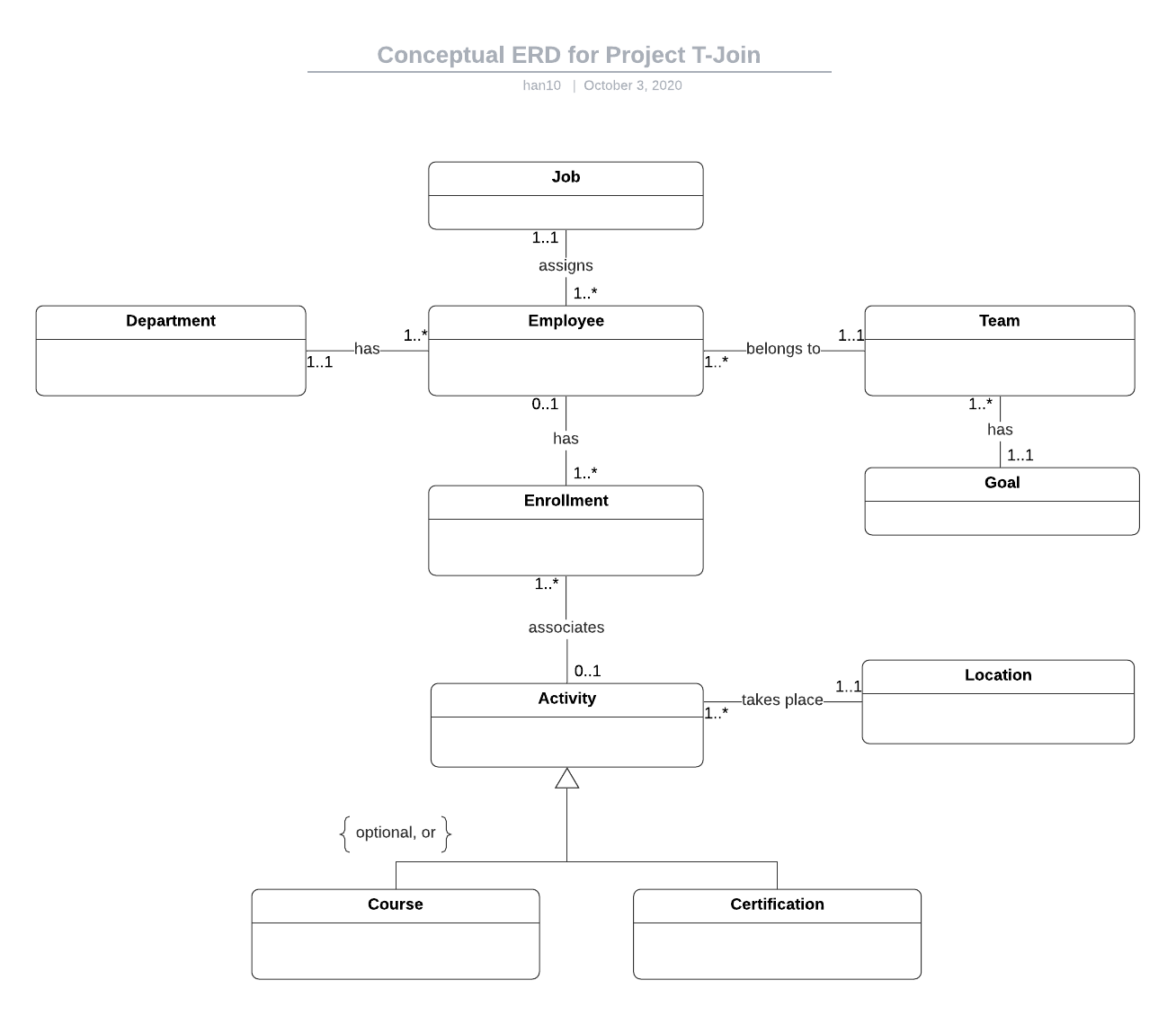
1. An Employee belongs to a Team; a Team has one to many Employees.
2. An Employee may have many Enrollments; an Enrollment is tied to one Employee.
3. An Employee belongs to a Department; a Department has many Employees.
4. An Enrollment is tied to an Activity; An Activity can have many Enrollments.
5. An Activity is a Course, a Certification, or none of these.
6. *An Employee has a Job; a Job is assigned to many Employees.*
7. *A Team has a Goal; a Goal is set by many Teams.*
8. *An Activity takes place at a Location; a Location hosts many activities.*

Also note a change made based on previous iteration feedback: manager\_id has been moved into the Employee table as a foreign key that references its own employee\_id. This prevents further redundancies since managers can be employees also. This field may be left null, because some employees do not have managers. I will also have to rely on self-joins to query manager-employee relational data.

Here is the updated DBMS Physical EERD



Here is the updated conceptual ERD



## Table Creation

A functioning script is provided in cs669\_project\_tables\_han10.sql. Here are the screenshots of successfully executing the CREATE commands.



## 

## 

## 

## Indexing

Here’s the list of primary key columns that are already indexed:

* Job.job\_id
* Department.department\_id
* Employee.employee\_id
* Team.team\_id
* Goal.goal\_id
* Enrollment.enroll\_id
* Activity.activity\_id
* Location.location\_id
* Course.activity\_id
* Certification.activity\_id

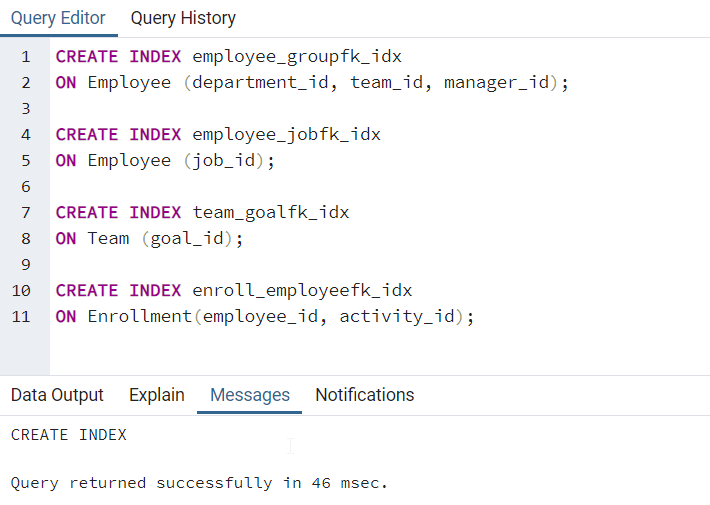
Here are the indexes for foreign key columns that are not listed above:

|  |  |  |
| --- | --- | --- |
| Column | Unique? | Description |
| Employee.department\_id | Not unique | There can be many Employees belonging to the same Department. |
| Employee.team\_id | Not unique | There can be many Employees on the same Team. |
| Employee.job\_id | Not unique | There can be many Employees with the same Job. |
| Employee.manager\_id | Not unique | There can be many Employees reporting to the same Manager. |
| Team.goal\_id | Not unique | There can be many Teams with the same Goal. |
| Enrollment.employee\_id | Not unique | There can be many Enrollments for the same Employee. |
| Enrollment.activity\_id | Not unique | There can be many Enrollments for the same activity. |

Three indexes based on commonly queried columns (in bold):

1. It is very likely that the **Employee.joined\_on** column will be in a WHERE clause, because most reports will look at a certain time range of when new employees joined the company. For example, employees who joined last year would not be relevant in a monthly new hire report. This should therefore be a non-unique index because many employees would join on the same date.
2. **Enrollment.progress\_percent** should be indexed, because it is also likely to be in a WHERE clause. Many uses cases will require queries above or below a certain progress threshold. For example, ‘how many members from team X have completed 50% of course Y’? Therefore, this should also be a non-unique index because many employees will have made the same progress on a given activity. Also, it may be tempting to index Enrollment.completed for the same reason, but since it’s a Boolean column of True/False values, there aren’t enough distinct values to warrant it.
3. Finally, **Activity.duration\_mins** should be indexed, because there will be many queries filtering on the duration of an Activity within a WHERE clause. This will enable reports that look at percentage completions of longer courses vs shorter ones for example. This is definitely a non-unique index due to many activities having the same duration.

Here I’m creating the first set of indexes pertaining to the foreign key columns:



The first one is a composite index because those columns are often queried together, which can be thought of as the overall group an Employee belongs to. There are also many instances where the job\_id column will be queried without the other foreign key columns, because querying related Job data is more relevant to reporting on an individual Employee. Similar reasoning applies to the remaining 2 indexes. Finally, here are the query-driven indexes:



## Latest iteration (5) >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

## Notes About Iteration 4

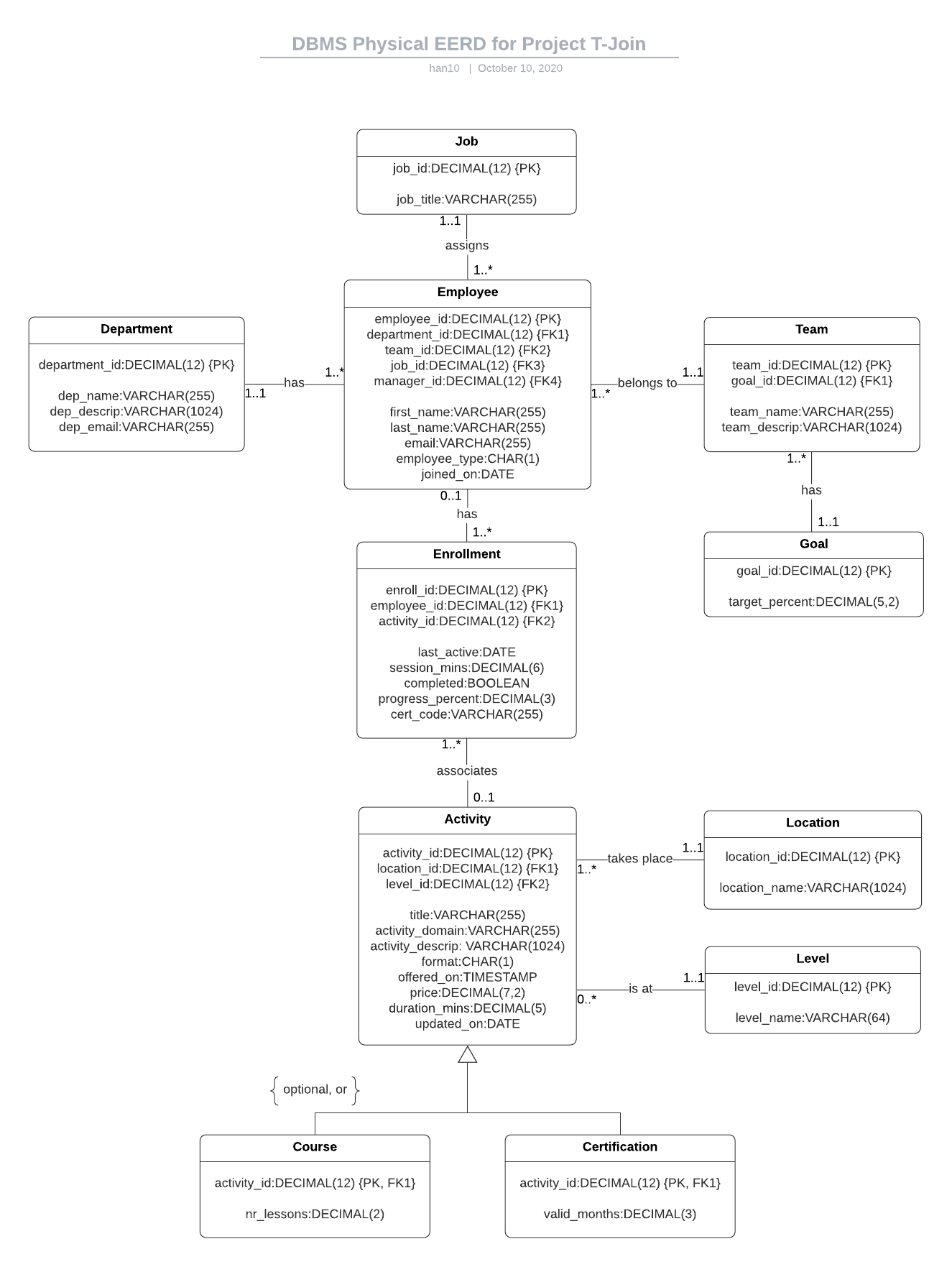
Based on the feedback for iteration 4, one could make the argument for normalizing Location.address, Certification.level, and Enrollment.cert\_code. Here are the reasons why it doesn’t always apply to my particular business use case:

* As mentioned before, Location.address is not always referring to a physical address. It could also contain a unique URL that’s auto generated for each training/exam session. This is similar to scheduling a meeting on outlook for example, where either physical or virtual locations can be entered as free text. Most values will be unique and unformatted. Also, because the application interacting with the database needs to accept either an URL or partial address within the same field, there’s not much benefit to breaking the text down into further components (similar to the TrackMyBuys example).
* Certification.Level will be repeated across many activities and should indeed be normalized in the updated design.
* Enrollment.cert\_code is auto generated and always unique for each Enrollment if applicable to the Activity. For the same reason as Location.address, there’s no benefit to normalizing it.

Here are some updates to the current design:

* I have moved Level into its own entity that has the following relationship with Activity, so that it may pertain to many different Activity subtypes. Here’s the new structural database rule:
  + **Many Activities may be at a certain Level; A Level represents the difficulty of an Activity.**
* Certification.valid\_months is a new optional attribute that determines the number of months a certification is valid for until it expires (at which point it needs to be taken again). Null values means no expiration date.
* Location.address has been changed to Location.location\_name to avoid confusion.

The updated physical ERD is on the next page. Table creation scripts are also updated in *cs669\_project\_tables\_han10.sql.*



## Implementing Transactions

Before executing the script in *cs669\_project\_procedures\_han10.sql*, please run the table creation script again in *cs669\_project\_tables\_han10.sql.* In this section I will define and execute Stored Procedures that are related to 3 data insertion use cases.

Adding An Employee

User is Sys Admin:

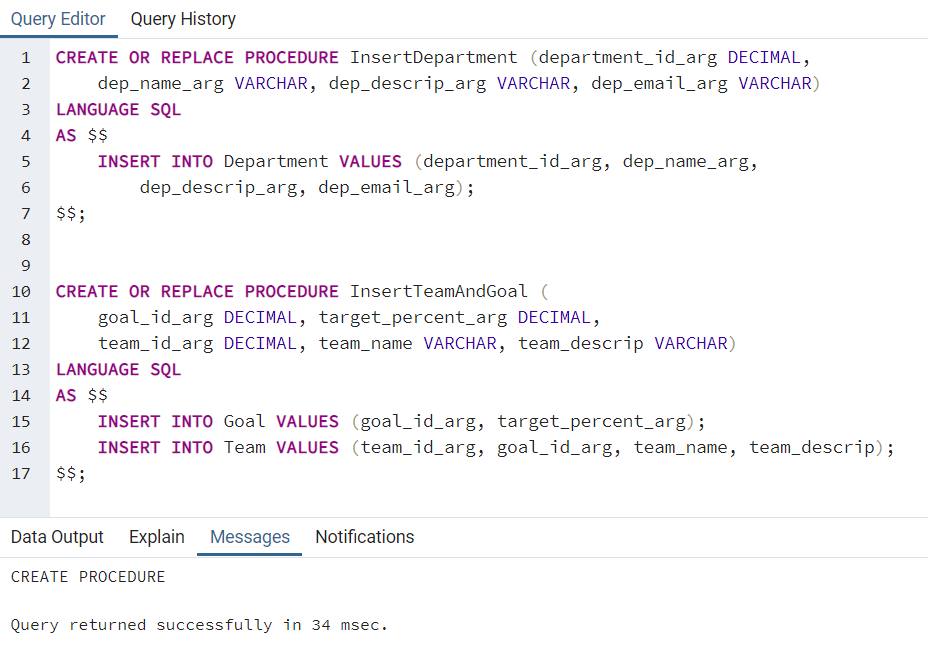
1. User logs into the T-Join’s web portal.

2. User clicks the ‘create account’ button to be taken to a form.

3. User enters the information of the new employee into the form.

4. User clicks the ‘add record’ button to add the employee as a record into the database.

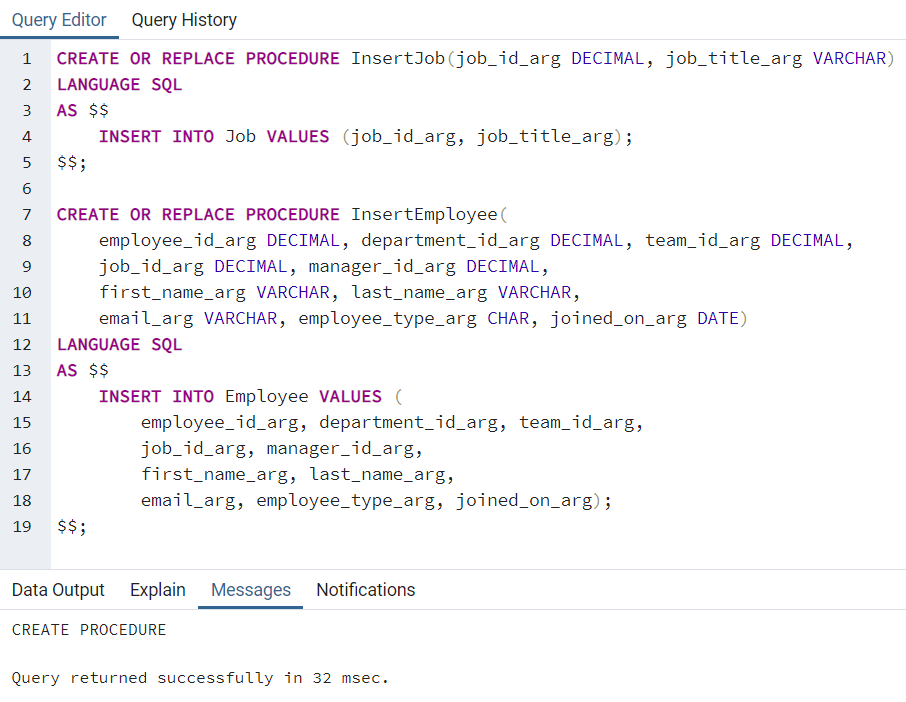
For this use case, I need to insert a new Employee into the database, which would require the insertion of a new Job, Department, Team and Goal as well. In real-world situations, the Team, Department, and Goal records should exist first, and then new Employee records with related Job records are added as new employees are hired. Therefore, my first stored procedures will be for inserting a Department, Team, and Goal:



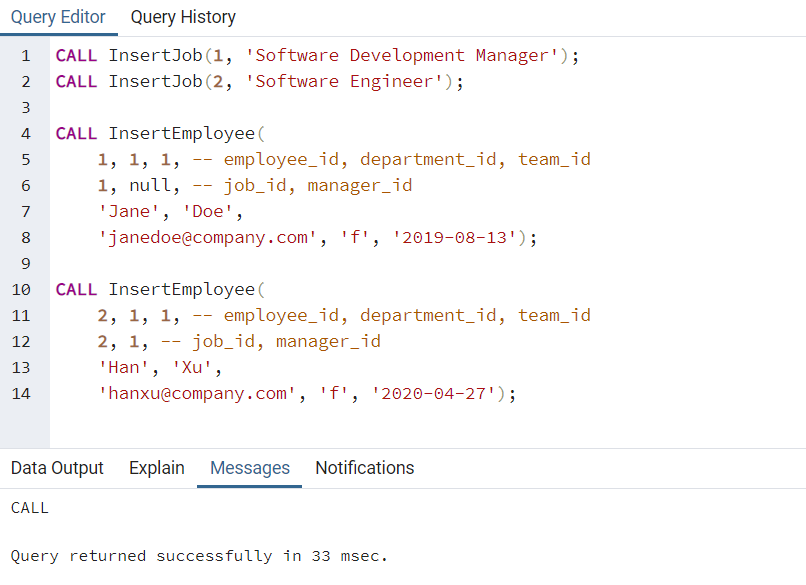
I have defined two functions so that Department can be created by itself, and Team and Goal are always created together. An added benefit here is that the goal\_id\_arg argument can be reused as the primary key for Goal and foreign key for Team within the same procedure. Then I call both procedures with the following arguments. Please note the long descriptions are truncated (full versions are in the .sql file).



And now I can define stored procedures for inserting Employee and related Job:



These procedures could technically be combined into one, but most of the time a Job definition already exists, and new employees are hired and given an pre-defined job. So it’s more realistic to allow insertions of Job and Employee at separate times. Now I call each procedure twice to demonstrate the self-join attribute of an Employee that references a manager\_id:



Here, the manager\_id for Jane Doe is null because we will assume that she does not report to anyone. The manager\_id for Han is Jane. This is how we can avoid the need for another Manager table, which leads to duplicate records.

Adding A Course

User is Course Author:

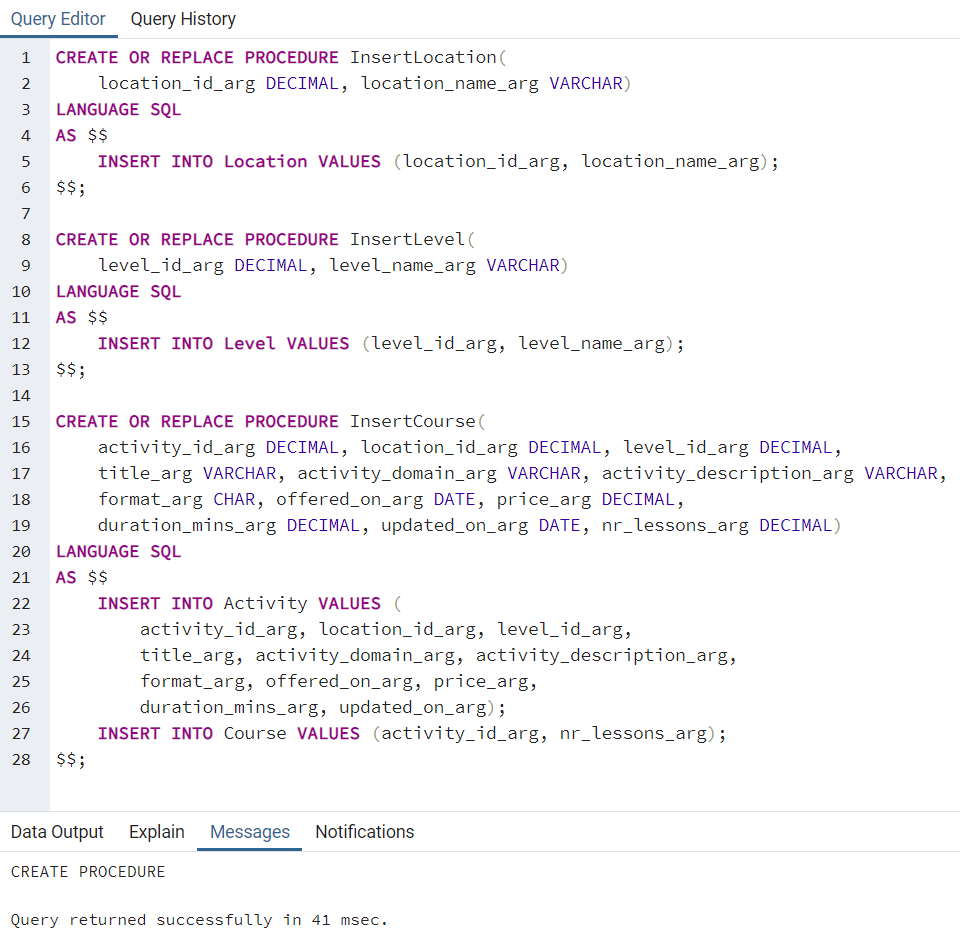
1. User logs into the T-Join’s web portal.

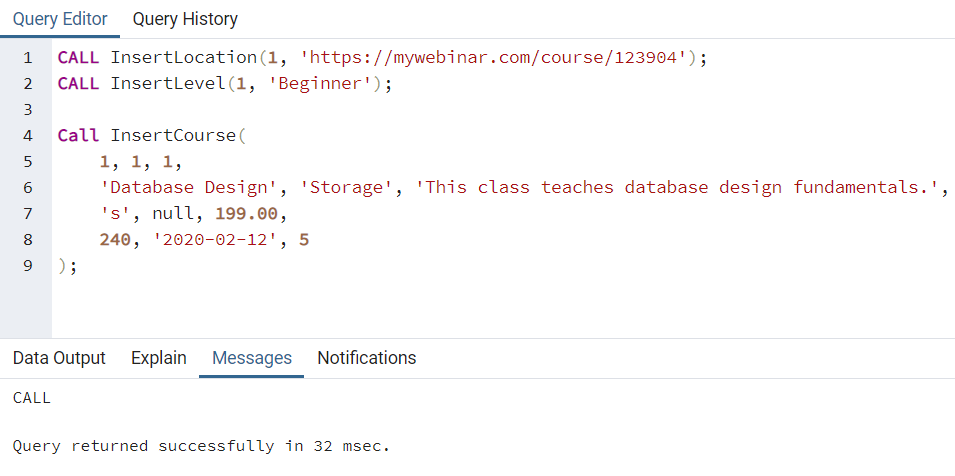
2. User clicks the ‘Add Activity’ button to be taken to a form.

3. User enters the information of the new Activity (such as location and level).

4. User clicks the ‘submit’ button to add the Activity to the application.

For this use case I will need to insert data into Activity, Location, Level, and Course. Location and Level are inserted separately, but Activity and Course are inserted within the same stored procedure because they share the same identity. Here are my stored procedure definitions, followed by their executions:





Logging An Enrollment

User is Employee:

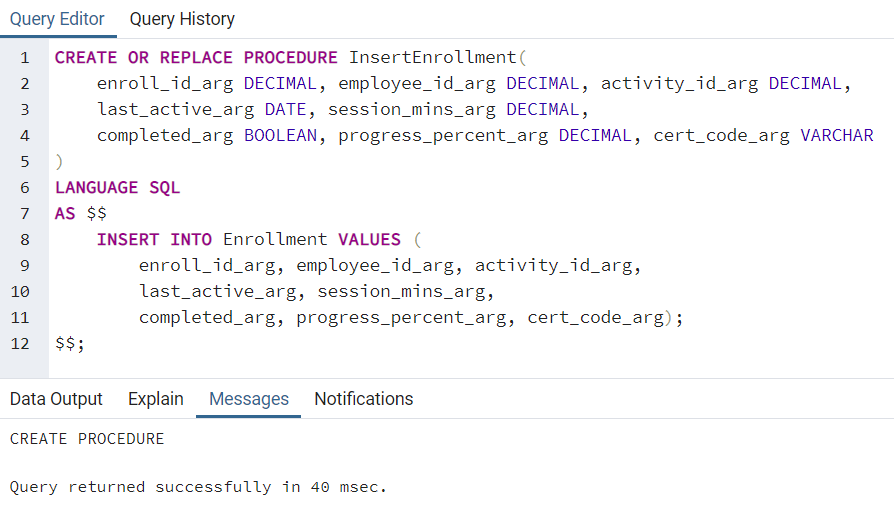
1. User completes a training activity.

2. User logs into T-Join’s web portal.

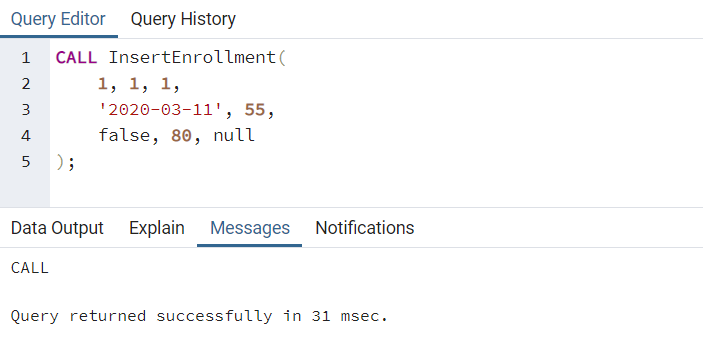
3. User click the profile button to be taken to the transcript page.

4. The status is shown to be ‘completed’ next to the training activity.

For this use case I will insert a record into Enrollment, which is the bridging entity for Employee and Activity. Here is the stored procedure and its execution:



This insertion is for Jane Doe:

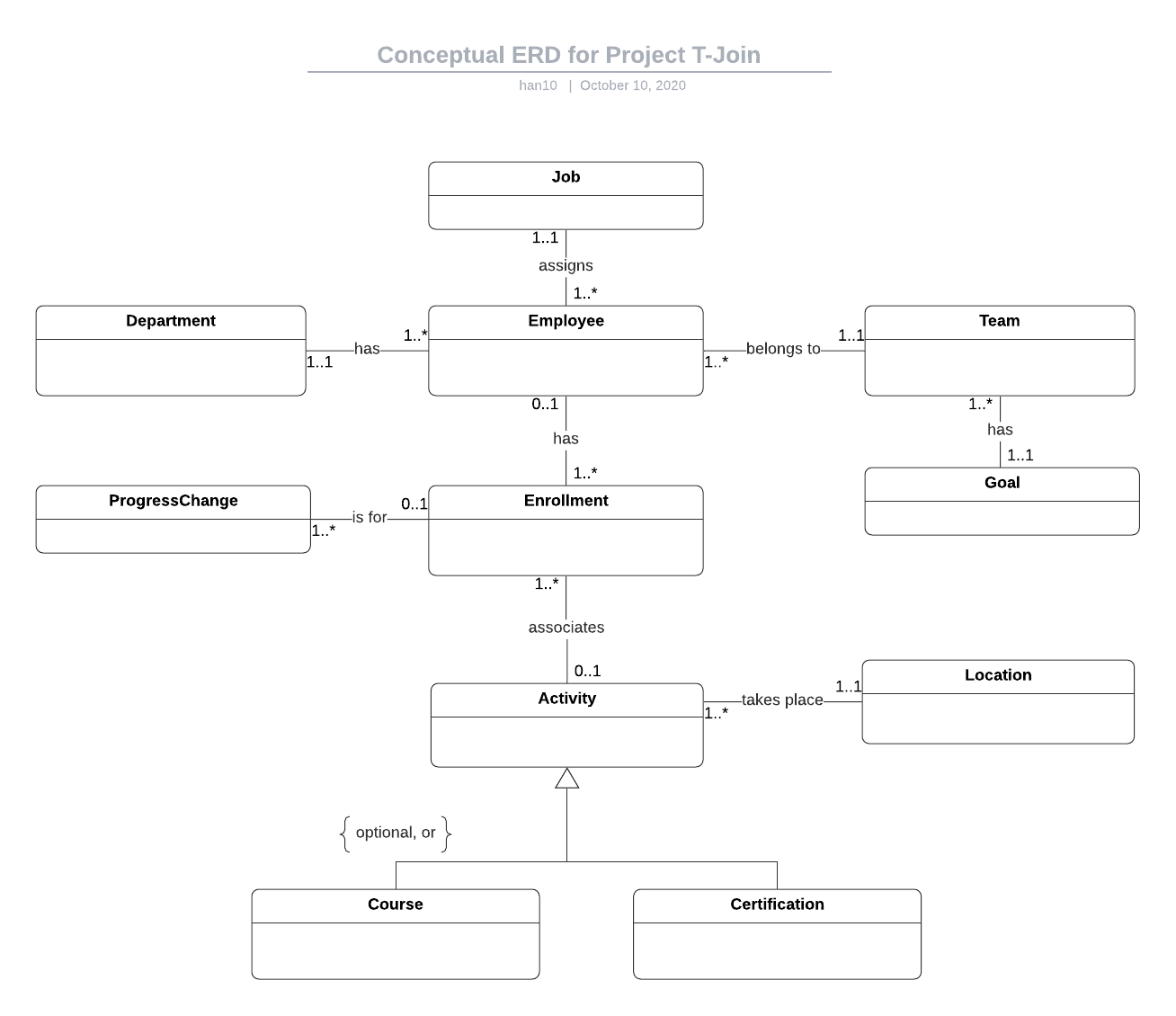


## Maintaining History Tables With Triggers

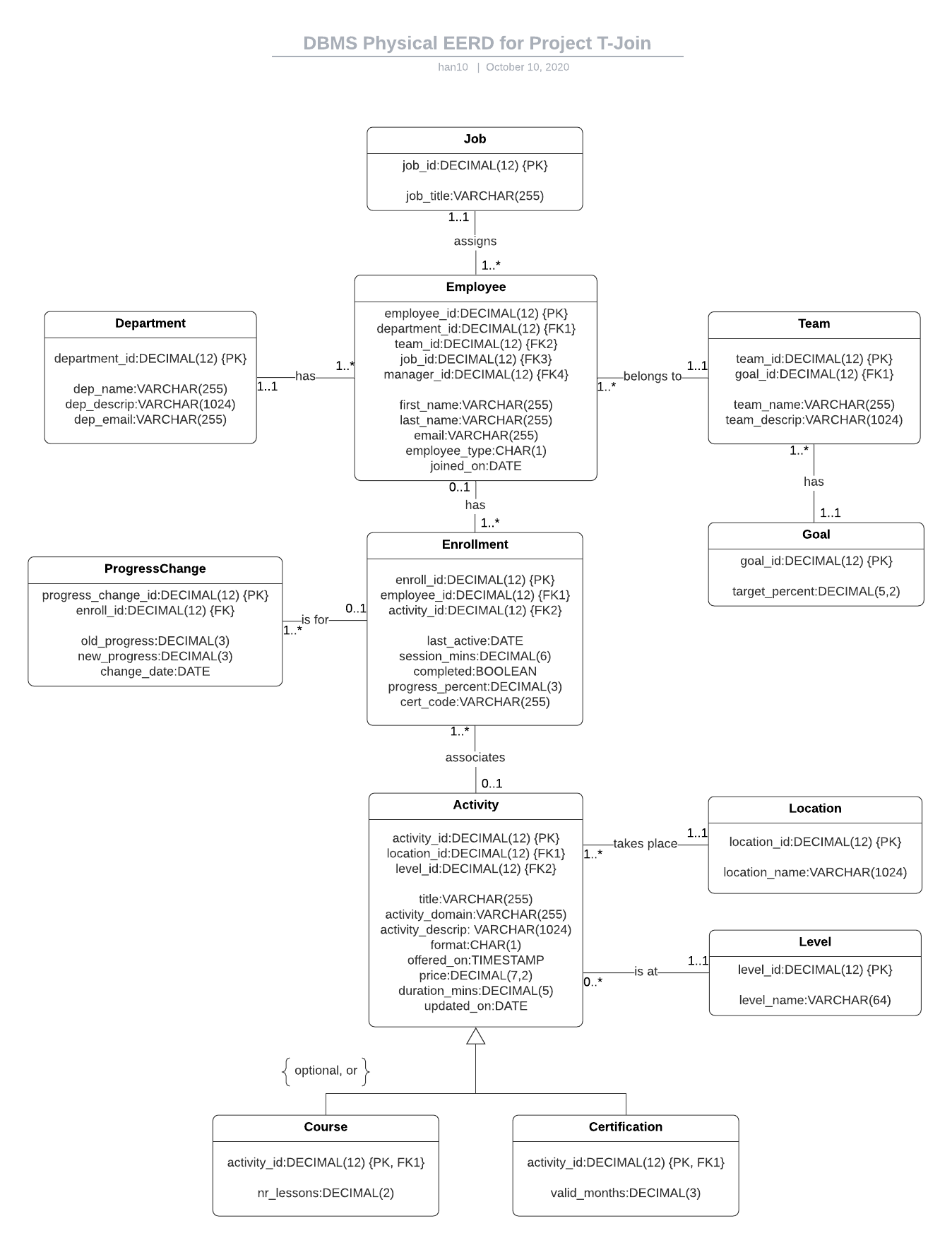
The history table I will create is for tracking changes to the Enrollment table. Since Enrollment is related to an Employee and an Activity, I can use its history data to determine how many times an Employee has accessed a course for example, and what the intervals are between each access in terms of days elapsed and progress made. This will then help determine insights such as onboarding velocity. To do this I will track specifically track the changes to progress\_percent in a ProgressChange table.

Here’s the new structural database rule and updated conceptual ERD:

**An Enrollment can have zero to many Progress Changes; a Progress Change is for one Enrollment.**

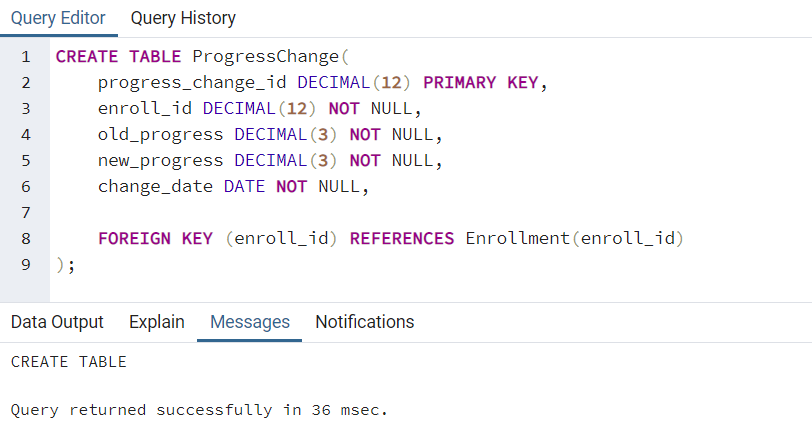
****

Here’s the updated physical ERD:

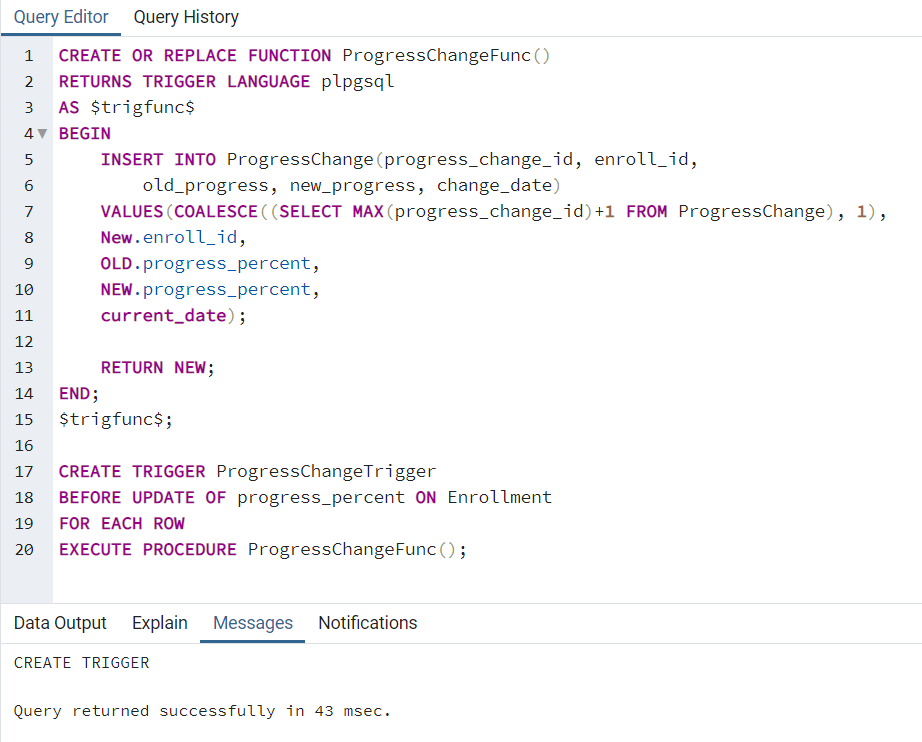


|  |  |
| --- | --- |
| **Attribute** | **Description** |
| progress\_change\_id | This is the primary key of the history table, up to 12 digits. |
| enroll\_id | This is the foreign key to the Enrollment table, which has the progress\_percent attribute being tracked. |
| old\_progress | This is the progress\_percent value before the change, which mirrors the progress\_percent datatype in the Enrollment table. |
| new\_progress | This is the progress\_percent value after the change, which mirrors the progress\_percent datatype in the Enrollment table. |
| change\_date | This is the date the change occurred with a DATE datatype. |

Here’s the table creation SQL code for this history table:



Here’s the procedure and trigger for this history table:



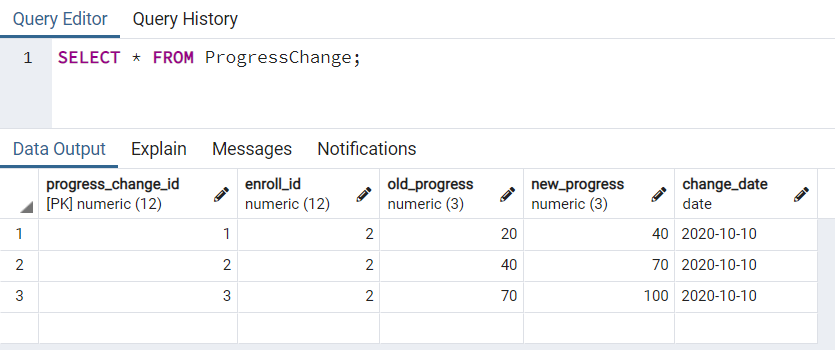
Lines 5-11 is where the main block of the function is defined. It starts by specifying which fields to insert into the ProgressChange history table. Then on line 7 it grabs the highest primary key id for the history table and increments it by 1, else starts with 1. Lines 8-10 inserts the new enroll\_id from the new record in the Enrollment table, as well as the old and new progress\_percent values. Line 11 then inserts the current date via the built-in variable.

Lines 17-20 creates the trigger that activates whenever an update is made on the progress\_percent field in the Enrollment table. This will be a row trigger that executes the ProgressChangeFunc function.

Now I will insert a new record into the Enrollment table and update it several times to test the changes to the ProgressChange table.



Note that I’m updating fields based on realistic scenarios where the session\_mins and last\_active fields are also updated. Only progress\_percent should be tracked in the history table:



## Organization-Driven Queries

Queries are defined in *cs\_669\_project\_queries\_han10.sql*. Before running this script, please ensure that:

1. *cs669\_project\_procedures\_han10.sql* is run already to run procedures with initial data
2. run *cs669\_project\_data\_han10.sql* to insert more test data.

**Question 1: Which employees that joined in the past month did not complete onboarding training?**

This question is important to my company, because one of the problems we’re trying to solve is to see which new hires are not engaged, so then we can send automated notifications to follow up with them. If they still have not completed the beginner level training after 1 month, we can assume that they intentionally skipped it.

I use the following query to retrieve this data:

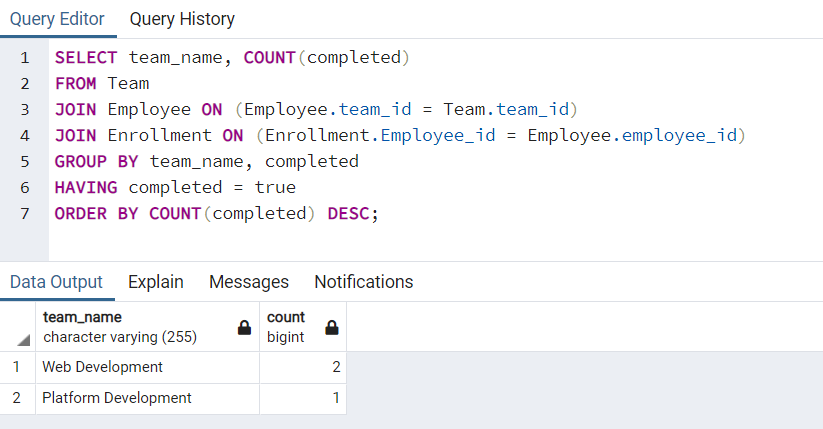


I first specify the columns I want in line 1, then specify the subset of Employees I’m interested in via a subquery in lines 2-4, which are those that joined within the last month. For performance reasons, I do this before joining the other 3 tables in lines 5-7 so that less rows are joined in total. Then in lines 8-9 I specify to filter by rows where the enrollment is not completed and the level is beginner, as well as the remaining rows where there are no enrollments in beginner-level activities at all.

The result returned is accurate, since Sarah did join within the last month, but did not enroll in activity\_id = 1, which is the only beginner activity added to this database.

**Question 2: Which teams have the most trained employees by number of completions?**

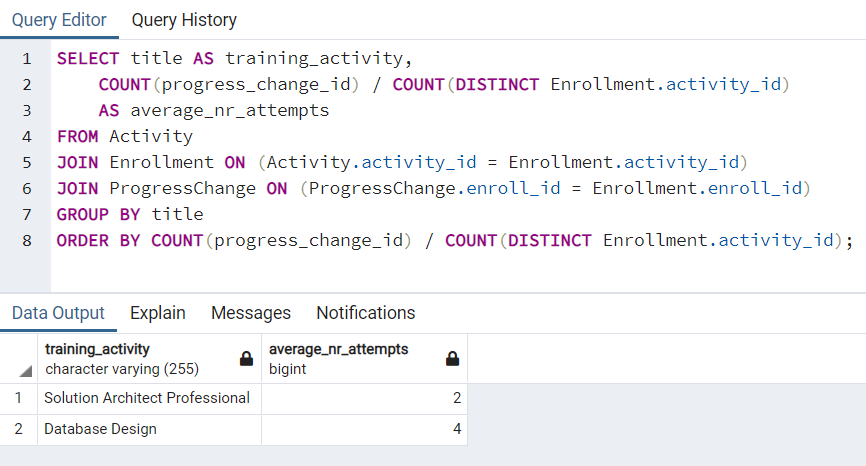
This question is important, because at work we try to motivate teams and individuals by ranking them based on their activities. Creating leaderboards is one of the ways we can highlight over-achievers and under-achievers at the same time. To answer this question, I need a query that can give me an ordered/ranked list of teams and the total completed training enrollments of each individual on those teams.



This query returns the team name and the count of all completions for each team. After specifying the columns I want in line 1, I join the Team table with Employee and Enrollment on lines 3-4, and group the data by team\_name and completed in line 6. Finally, I arrange the rows in descending order so the team with the most completions is at the top.

**Question 3: Which activities have on average the most number of attempts by employees?**

This question is important, because it provides the company with feedback on which training content to improve for future new hires. For example, if an activity requires very few attempts than perhaps it’s too easy. If an activity requires many attempts, then it could suggest the content is either too difficult or too boring. Either way, an ordered list from least attempts to most will provide the Learning team a queue to work from when it’s time to update their content. A good way to measure attempt is to look at how many times the employees has made progress on their enrolled activity. In other words, my query involves the history table:



On line 1, I select the training title for the first column. Then for the 2nd column, I divide the number of progress changes by the number of unique activities in order to get the average number of attempts for each activity. On lines 5-6, I join the other tables I need. Then on line 7, I group the data by the Activity title so that the report can match the calculated average with each Activity. Finally on line 8, I order it by the same calculated value.

## Summary and Reflection

I am overall satisfied with the end state of this database, as it really shows the work I’ve put into the design stage. By addressing some of the issues such as the self-join manager field and bridging the Activity subtypes, I found that the queries are much easier to construct. The 3 organizational queries above cover multiple points in both complexity groups, but they were not very hard to write thanks to the design.

This project has helped me get over the fear of creating more tables as a result of normalization or other methods of eliminating redundancies. After becoming more proficient in joins and subqueries, I feel more comfortable spreading data across multiple entities and decomposing existing entities into multiple components.

One area I was struggling with in this final iteration is to decide which insertions to group together within the same stored procedure. Combining insertions into an inheritance hierarchy is obvious, but doing the same for other types of relationships could cause issues because the id’s may exist already in the related table. In the future, I could research ways to generate a unique key and save it to a variable, so that the same id can be used for multiple related tables.