**Database Design and Implementation for Interpreters**

Table of Contents

[Project Direction Overview 2](#_Toc102729960)

[Use Cases and Fields 2](#_Toc102729961)

[Structural Database Rules 7](#_Toc102729962)

[Conceptual Entity-Relationship Diagram 9](#_Toc102729963)

[Full DBMS Physical ERD 10](#_Toc102729964)

[History Table and Trigger 15](#_Toc102729965)

[Index Identification and Creations 19](#_Toc102729966)

[Stored Procedure Execution and Explanations 21](#_Toc102729967)

[Question Identification and Explanations 28](#_Toc102729968)

[Data Visualizations 30](#_Toc102729969)

# Project Direction Overview

Currently I work as a Chinese<>English interpreter for a large interpretation company in the US. My daily task is handling numerous interpretation requests (over the phone) in many fields, such as medical, insurance, bank, legal, etc. We can work from home; however, the company requires us to use a wired cord to connect the laptop for us to log into their platform, and a land line to answer calls, which is not very convenient. Therefore, I would like to create an app which could be downloaded on a smart phone to improve the flexibility and portability of our mode of work. This app would create the opportunity for interpreters to work in wireless fashion. For example, interpreters would be able to log into their own accounts using their unique interpreter ID as an account number. The app would provide the option on the phone screen for the interpreters to change the work status into either ‘active’ (ready to work) or ‘inactive’ (not ready to work). The company’s dispatcher would then transfer the client calls to our cell phones if our status were active. The app would be able to automatically record the details of each call, such as the client’s name, client ID, date and time, work duration, etc. Additionally, the app would also provide options for the interpreters to view many useful information on the app such as our profile, the call records, the wages, work schedule, etc.

# Use Cases and Fields

Use case 1 – Adding new interpreter:

1. The interpretation company sends the app download link to interpreters’ phones.
2. The interpreters click on the link and download the app on their smart phones.
3. When interpreters open the app for the first time, they would need to follow the instructions on the screen, create their own account, enter their basic information, enter their work status (part-time, full-time, or temporary) and set a password for their accounts.

This use case requires storing the account information. The significant fields include:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| InterpreterId | This field stores each interpreter’s unique identification ID that is assigned by the company. | This field would be set as a primary key because this number can uniquely identify each interpreter. Also, this number would be the interpreters’ account number. |
| InterpreterName | This field stores full name of each interpreter. | This field is needed because the interpreters’ names would be displayed on the interpreters’ profile pages. |
| WorkStatus | This field stores the interpreters’ work status -- full-time, part-time or temporary. | It would display on the profile page. |
| EmailAddress | This field stores the interpreters’ working email addresses. | The email addresses would be also displayed on the account holder’s profile page, and it could be used for contacting the interpreters. |
| PhoneNumber | This field stores the interpreters’ phone numbers. | Their phone numbers would be needed as well for their profile page, and for the future contact purposes. |
| MailingAddress | It stores the mailing address of the interpreter. | This would be displayed on the profile page. |
| TimeZone | It stores the time zone the interpreter is in. | It would be on the profile page, and it would help calculate the correct scheduled work time for interpreters who are in different time zones. |
| HourlyRate | This field stores each interpreters’ hourly rate. | This is necessary because it would be displayed on the profile page. |

Use case 2 – Creating Call log:

1. Once an interpreter is scheduled to work, he/she would log into the app.
2. The interpreter would change their status to ‘active’ meaning that they are ready to work.
3. The clients phone the company when they have interpretation needs.
4. The company’s dispatchers receive a client’s call and transfer the call to an interpreter (whose status is active) based on their language needs.
5. The interpreter would answer the calls from their phones.
6. The app would record each call’s details, including language, call type (phone or video), date, client info, etc.

Step 4 would require the database to store relevant information. Significant fields include:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| CallRecordNum | An identification number of each call. | It can identify each single call. |
| StartTime | It stores the date and time of each call’s start time. | It is needed so that interpreters can search their call history based on date and time. |
| EndTime | It stores the date and time of each call’s end time. | It is needed so that interpreters can search their call history based on date. |
| ClientId | It stores the identification number for each client. | It will uniquely identify the clients by a sequence of numbers. |
| ClientName | It stores the name of each client that the interpreter works for. | When interpreters search the call records, they would be able to know which clients they have worked for. |
| WorkDuration | It stores the total amount of time (hours) for the client. | This is necessary because we need this information to show how long each call lasts, and it is also useful for calculating the earnings for each call. |
| CallDescription | It stores a brief description of each call, such as ‘filing an auto claim’, ‘a medical visit with a cardiologist’, etc. | It is needed so that when interpreters search for a specific call they can view that call’s description. |
| Language | It stores the language (English, Mandarin, etc.) for that call. | It is useful so that when interpreters check their call history, they would know what language they interpreted for (especially when they know more than one language). |
| CallType | It stores whether the call was made via either phone or video. | It is necessary to show the type of each call. |

Use case 3—generating earnings report:

1. The interpreters would log into their accounts.
2. The interpreters would select the option to view their wages.
3. The app would then provide the option to choose a date range.
4. The interpreters would be able to view the details of earnings for the chosen time period.
5. The app would also have the option to choose to generate a wage report, sending it to the interpreter’s email address.

The database would use the fields in first use case to identify the interpreter’s hourly rate (this rate applies to all calls from clients) and email address and make use of the fields of the second use case such as each call’s duration. In addition, other important fields would be listed as follows:

|  |  |  |
| --- | --- | --- |
| Fileds | What it stores | Why it’s needed |
| Date | It stores the date whenever the interpreter receives calls. | It is necessary for the search function described in the use case 3. |
|  |  |  |
| NumberOfCalls | It stores the total number of calls for that day. | It is necessary to produce a wage report for a specific day, and it can be used calculate the total earnings on that day. |
| ActualTimeWorked | It stores the total actual hours the interpreter has worked on that day. | It is necessary to produce a wage report for a specific day, and it can be used calculate the total earnings on that day because the earnings are based on your hourly rate and the actual call time (not scheduled work time). |
| Earnings | It stores the earnings for that day. | It can be displayed on the screen when the interpreter is asking for the earnings for a specific day. And it is necessary in order to calculate the earnings for a certain period. |

Use case 4 – searching call history:

1. The interpreters would log into their accounts.
2. The app would provide a search option for the interpreters to look up each single client they have worked for.
3. The interpreters would enter the client’s name or ID.
4. The app would generate a report showing the history (date, work duration, etc.) for this specific client.

The database would use the fields provided in the second case to search the requested clients.

Use case 5 – checking scheduled work time:

1. The interpreters would log into their accounts.
2. The app would provide an option to check each interpreter’s personal schedule with the company.
3. The interpreters would choose the time period they want to view.
4. The app would display their work schedule for the requested period.

The significant fields that would be needed for this use case are:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it needed |
| Date | It stores the date (mm/dd/yyyy) | The date is needed for searching the working schedule by date. |
| StartTime | It stores the scheduled start time on that day. | It is needed because interpreters need to view their work schedule. |
| EndTime | It stores the scheduled end time on that day. | It is needed because interpreters need to view their work schedule. |
| TotalScheduledHours | It stores the scheduled total hours for that day. | It is necessary because it would show how many hours interpreters are scheduled to work on that day. |

Use case 6 – Viewing departmental info:

1. The interpreters would log into the app.
2. The app would provide an option to contact the department.
3. Then the option would have a drop-down menu that lists different departments.
4. The interpreters would choose the desired department, then the next page would show that department’s information and the associated contact personnel(s)’s contact information.

The database would need to store the following relevant information in order to perform this use case:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| DepartmentName | It stores the name of each department (such as HR, IT, etc.) | The department name is necessary for the interpreters to search by department. |
| EmailAdress | The email address of that department. | The email address of the chosen department would be displayed on the screen. |
| Phone | Phone number of that department. | The phone number of the chosen department would be displayed on the screen. |
| ContactPersonnel(s) | The name(s) of the contact personnel(s). | Their names would be displayed on the screen so that we know how to address them when sending emails or making phone calls. |

Use case 7- Viewing professional development opportunities.

1. The Interpreters log into their accounts.
2. The app would provide an option to show information in regard to all the training programs and associated mentors that are available to the interpreters.

The significant fields for this use case are:

|  |  |  |
| --- | --- | --- |
| Training | It stores the name of the training | It is part of basic information about each training. |
| Description | It stores a brief description of each training. | It would show the interpreters what the training is about. |
| Mentor | It stores the mentor’s info(name and email address) for each training (if any). | It would allow the interpreters to contact their mentors if needed. |
| Source | It stores the source link of each training (all trainings would be held online) | It would provide the source of content regarding the training. |
| AvailableDate | It stores the available date of each training. | It would show the interpreters the available date. |
| CompletionDate | It stores the completion date of each training. | It would show interpreters the completion date. |
| TrainingStatus | It stores the status (completed or incomplete) of each training. | It would show the interpreters their status of each training. |

# Structural Database Rules

Replace this with a list of structural database rules for all significant entities and relationships, with the constraints defined, based upon the use cases you defined, along with supporting explanations.

**Structural Database Rules for Associative Relationships:**

Use case 1 – Adding new interpreter:

1. The interpretation company sends the app download link to interpreters’ phones.
2. The interpreters click on the link and download the app on their smart phones.
3. When interpreters open the app for the first time, they would need to follow the instructions on the screens, create their own account, enter their basic information, enter their work status (part-time, full-time, or temporary) and set a password for their accounts.

The first rule from this use case is:

An interpreter must have an account; an account is owned by an interpreter.

It indicates that the participation constraint of these two entities (interpreter and account) is mandatory, and the plurality constraint for both entities is singular.

Use case 2 – Creating Call log:

1. Once an interpreter is scheduled to work, he/she would log into the app.
2. The interpreter would change their status to ‘active’ meaning that they are ready to work.
3. The clients phone the company when they have interpretation needs.
4. The company’s dispatchers receive a client’s call and transfer the call to an interpreter (whose status is active) based on their language needs.
5. The interpreter would answer the calls from their phones.
6. The app would record each call’s details, including language, call type (phone or video), date, client info, etc.

The associative rules for this use case are:

The entities listed from this use case are account, interpreter, call, client and language.

Each account may associate with many calls; each call is associated with one account.

This rule indicates that one account may or may not have calls, so the relationship of Account to Call is optional, and it also indicates that the relationship is plural to Account. On the other hand, one call must associate with one account, so the relationship of Call to Account is mandatory, and the relationship is singular to Call.

Each interpreter may have many calls; each call is associated with one interpreter.

This indicates that Interpreter to Call is optional and plural. Call to Interpreter is mandatory and singular.

Each client may associate with many calls; each call is associated with one client.

This indicates that Client to Call is optional and plural. Call to Client is mandatory and singular.

One interpreter may associate with many clients; each client may associate with many interpreters.

This indicates that Interpreter to Client is optional and plural. Client to Interpreter is optional and plural.

One interpreter speaks at least one language; one language may be spoken by many interpreters.

This indicates that the relationship is mandatory and plural to Interpreter, and optional and plural to Language.

Use case 3—generating earnings report.

Use case 4 – searching call history.

Use case 5 – checking scheduled work time.

Use case 3,4, and 5 mainly described the functionalities of the application. There are no additional entities introduced in these use cases.

Use case 7- Viewing professional development opportunities.

1. The Interpreters log into their accounts.
2. The app would provide an option to show information in regard to all the training programs and associated mentors that are available to the interpreters.

There are three entities in this use case, Interpreter, Training and Mentor.

The rules are:

One interpreter may associate with many trainings, and one training is associated with many interpreters.

This indicates that the relationship is optional and plural to Interpreter. And the relationship is mandatory and plural to Training.

One mentor may associate with one training, one training may associate with many mentors.

This indicates that the relationship is optional and singular to Mentor. And the relationship is optional and plural to Training (because some trainings might be conducted via a pre-recorded video and may not need a mentor at all).

**Specialization-Generalization Structural Database Rules:**

Use case 1 – Adding new interpreter:

1. The interpretation company sends the app download link to interpreters’ phones.
2. The interpreters click on the link and download the app on their smart phones.
3. When interpreters open the app for the first time, they would need to follow the instructions on the screens, create their own account, enter their basic information, their work status (part-time, full-time, or temporary) and set a password for their accounts.

An interpreter is a full-time, part-time or temporary employee.

This indicates that one interpreter must be a full-time, part-time or temporary employee, so the relationship is complete and disjoint.

# Conceptual Entity-Relationship Diagram

The associative structural database rules I came up are:

1. An interpreter must have an account; an account is owned by an interpreter.
2. Each account may associate with many calls; each call is associated with one account.
3. Each interpreter may have many calls; each call is associated with one interpreter.
4. Each client may associate with many calls; each call is associated with one client.
5. One interpreter may associate with many clients; each client may associate with many interpreters.
6. One interpreter speaks at least one language; one language may be spoken by many interpreters.
7. One interpreter may associate with many trainings, and one training is associate with many interpreters.
8. One mentor may associate with one training, one training may associate with many mentors.
9. One interpreter may have many pay changes, one pay change is associated with one interpreter.

Specialization-Generalization structural database rules I came up with are:

1. An interpreter is a full-time, part-time, or temporary employee.

Based on these rules, the ERD is demonstrated as follows:

Diagram

Description automatically generated

# Full DBMS Physical ERD

Based on the business rules and the conceptual ERD I created last week, the relationship classifications for all of the associative relationships in my design are:

**The Interpreter/Language relationship is M:N**, so I created a bridging entity (Speaks) to support the relationship. The bridging entity has foreign keys to both Interpreter and Language, resulting in two 1:M relationships between Speaks and Interpreter and Language.

**The Interpreter/Account relationship is 1:1**; one interpreter must associate with only one account.

**The Interpreter/Call relationship is 1:M**; one interpreter can associate with many calls.

**The Account/Call relationship is 1:M**; one account can associate with many calls.

**The Call/Client relationship is 1:M**; one client may associate with many calls.

**The Interpreter/Client relationship is M:N**, so a bridging entity(Service) is created to support the relationship. The bridging entity has foreign keys to both Interpreter and Client, resulting in two 1:M relationships between Service and Interpreter and Call.

**The Interpreter/Department relationship is M:N**, so a bridging entity (Communicates) is created to support the relationship. The bridging entity has foreign keys to both Interpreter and Department, resulting in two 1:M relationships between Communicates and Interpreter and Department.

**The Interpreter/Training relationship is M:N**, so a bridging entity (Participation) is created to support the relationship. The bridging entity has foreign keys to both Interpreter and Department, resulting in two 1:M relationships between Participation and Interpreter and Training.

**The Interpreter/Paychange relationship is 1:M**; one interpreter can associate with many pay changes.

I also have one specialization-generalization relationship. The subtypes Full-time, Part-time and Temporary are under the Interpreter entity. Each of these subtypes have a primary and foreign key of interpreter\_id which reference the primary key of Interpreter.

My DBMS physical ERD with these relationships is demonstrated below:

Diagram, schematic

Description automatically generated

Adding attributes to the DBMS Physical ERD:

|  |  |  |  |
| --- | --- | --- | --- |
| Table | Attribute | Datatype | Reasoning |
| Interpreter | interpreter\_id | DECIMAL (5) | Each interpreter is assigned with a 9-character unique synthetic identification number. |
| Interpreter\_name | VARCHAR (255) | This stores the name of each interpreter; up to 255 characters are allowed in case any super long names. |
| work\_status | CHAR (1) | This is the subtype discriminator to indicate the work status for the interpreter. |
| email\_address | VARCHAR (255) | Every interpreter must have a working email address. 255 characters would be enough for an email address. |
| phone\_number | DECIMAL (20) | Every interpreter needs to provide their contact phone number. 20 digits would be safe even for international numbers. |
| mailing\_address | VARCHAR (500) | Interpreters should provide their mailing address, because the company might need to mail some documents or equipment to them. I decided to allow up to 500 characters because some of the international addresses are relatively long. |
| time\_zone | VARCHAR (50) | Interpreters need to report their time zone to coordinate the proper work time. |
| Full-time | Monthly\_salary | DECIMAL (5) | Full time employees are paid by the month. 5 digits provides a safe upper bound for their salary. |
| start\_date | DATE | This is to show the start date of their employment. |
| Part\_time | hourly\_rate | DECIMAL (4) | Part-time employees are paid by the hour. |
| start\_date | DATE | Their employment start date is also necessary information to show. |
| Temporary | hourly\_rate | DECIMAL (4) | Temporary employees are also paid by the hour. |
| start\_date |  | Temporary employees would have a scheduled start and end date. |
| end\_date |  |  |
| Call | call\_id | DECIMAL (10) | Each call must have a unique identification number. |
| client\_id | DECIMAL (5) | Each call is associated with one client. Their ID is 5 digit long. |
| call\_type | CHAR (3) | There would be 2 call types: interpretation over the phone would be represented as ‘OPI’; video remote interpretation would be ‘VRI’. So 3 characters would be required. |
| call\_date | DATE | Each call is associated with a specific date. |
| start\_time | Time | Each call has a start time. 10 characters would be a safe upper bound for time. |
| end\_time | Time | Each call has an end time. |
| work\_duration | DECIMAL (4,2) | Each call’s duration (hours) would be recorded. 4 digits are safe upper bound, and 2 decimal digits would provide a pretty accurate time count. |
| call\_description | VARCHAR (500) | This would be a brief description of each call. No more than 500 characters are allowed. |
| Account | Interpreter\_id | CHAR (9) | The interpreter ID is considered as the account number. |
| password | VARCHAR (20) | The password of each account should not be longer than 20 characters. |
| status | VARCHAR (8) | The status of each account can be ‘active’ or ‘inactive’, so 8 characters would be the max number it would need. |
| Department | department\_id | DECIMAL (5) | Each department is assigned with a unique 5-digit synthetic number. |
| department\_name | VARCHAR (50) | Each department has a name. 50 characters should be a safe upper bound for their names. |
| email\_address | VARCHAR (255) | Each department needs to provide an email address for the purpose of communicating with the interpreters. |
| phone\_number | DECIMAL (20) | Each department has a contact phone number. Up to 20 digits is allowed in case an extension is needed. |
| contact\_personnel | VARCHAR(255) | Each department needs to have a contact person listed. 255 characters would be a safe upper bound for their names. |
| Training | training\_id | DECIMAL (7) | Each training is assigned with a 7-digit identification number. |
| description | VARCHAR (1000) | This would show the interpreters what the training is about. |
| available\_date | DATE | This would show the interpreters available dates for training. |
| completion\_date | DATE | This would show interpreters the completion date. |
| training\_status | VARCHAR (15) | This would show the interpreters their status of each training. |
| mentor\_name | VARCHAR(255) | Each training may have a mentor; therefore, their names are needed. |
| mentor\_id | DECIMAL (5) | Each mentor is assigned with a 5-digit identification number. |
| Language | language\_id | DECIMAL (4) | Each id represents one language. |
| language\_name | VARCHAR (255) | 255 characters would be a safe upper bound for the names of languages. |

Normalizing my DBMS Physical ERD

I reviewed the ERD and realized that I can normalize the training table by creating a ‘Mentor’ table. This is because mentor-id determines mentor\_name, but mentor\_id is not the primary key of the ‘Training’ table. The associative relationship between the entities ‘Training’ and ‘Mentor’ is identified as 1:M. Each training may have one mentor; each mentor may have many trainings.

I also move the addresses for the interpreters and clients to an address entity. I did not normalize the address table further to state, because interpreters and clients might from many countries, and it would be very complicated to do so, and I do not think it is necessary for my database.

My updated ERD is demonstrated below:

Diagram, schematic

Description automatically generated

# History Table and Trigger

Upon reviewing my DBMS physical ERD, a history table could be added to record pay changes for full-time, part-time, and temporary employees. Such a history table would help the company keep track of the interpreters’ pay changes. The relationship between Interpreter and Paychange is that one interpreter may have many pay changes; each pay change is for one interpreter.

My updated physical ERD is demonstrated below:

Diagram, schematic

Description automatically generated

The Paychange entity is linked to Interpreter. The attributes I added for Paychange are:

|  |  |
| --- | --- |
| Attribute | Description |
| Pay\_change\_id | This is the primary key for this history table. Its datatype is DECIMAL (5). |
| Old\_pay | This is the salary or rate before the change. |
| New\_pay | This is the salary or rate after the change. |
| Interpreter\_id | This is the foreign key to the Interpreter table, a reference to the interpreter who had the pay change. |
| Change\_date | This is the date the pay change occurred. |

Here is the screenshot of the table and sequence creation:

Graphical user interface, text, application

Description automatically generated

I created three triggers (pay\_change\_trg\_f, pay\_change\_trg\_p, pay\_change\_trg\_t) for the entities (full-time, part-time and temporary respectively). Here are the screenshots of my trigger creations:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, Word

Description automatically generated  
To test if the trigger works, I created one full-time interpreter.

Graphical user interface, text, application

Description automatically generated

Then I updated the monthly salary to 5500.

Graphical user interface, application, table

Description automatically generated with medium confidence

Lastly, I verified that the Paychange table has the record of the pay change I just made:

Graphical user interface, application

Description automatically generated

# Index Identification and Creations

Here is the list of the primary keys which are already indexed:

Account.account\_id  
Address.address\_id

Call. call\_id

Client.client\_id

Department.department\_id

Full\_time.interpreter\_id

Part\_time.interpreter\_id

Temporary.interpreter\_id

Interpreter.interpreter\_id

Language.language\_id

Mentor.mentor\_id

Training.training\_id

Paychange.pay\_change\_id

Adding foreign key indexes:

|  |  |  |
| --- | --- | --- |
| Column | Unique? | Description |
| Account.interpreter\_id | unique | The foreign key in Account referencing Interpreter is unique because one account is only linked with one interpreter. |
| Call.interpreter\_id | Not unique | The foreign key in Call referencing Interpreter is not unique because there can be many calls from the same interpreter. |
| Call.client\_id | Not unique | The foreign key in Call referencing Client is not unique because there can be many calls from the same client. |
| Call.account\_id | Not unique | The foreign key in Call referencing Account is not unique because one account can associate with many calls. |
| Interpreter.address\_id | Not unique | The foreign key in Interpreter referencing Address is not unique because several interpreters may have the same address. |
| Client.address\_id | Not unique | The foreign key in Client referencing Interpreter is not unique because more than one client may have the same address. |
| Service.interpreter\_id | Not unique | The foreign key in Service referencing Interpreter is not unique because many services can be provided by the same interpreter. |
| Service.client\_id | Not unique | The foreign key in Service referencing Client is not unique because many services can be provided to the same client. |
| Speaks.interpreter\_id | Not unique | The foreign key in Speaks referencing Interpreter is not unique because many ‘speaks’ can be done by the same interpreter. |
| Speaks.language\_id | Not unique | The foreign key in Speaks referencing language is not unique because many ‘speaks’ can be related to the same language. |
| Communicates.interpreter\_id | Not unique | The foreign key in Communicates referencing Interpreter is not unique because many ‘communicates’ can be done by the same interpreter. |
| Communicates.department\_id | Not unique | The foreign key in Communicates referencing Department is not unique because many ‘communicates’ can be done by the same department. |
| Participation.interpreter\_id | Not unique | The foreign key in Participation referencing Interpreter is not unique because many ‘participations’ can be done by the same interpreter. |
| Participation.training\_id | Not unique | The foreign key in Participation referencing Interpreter is not unique because many ‘participations’ may be associated with the same training. |
| Training.mentor\_id | Not unique | The foreign key in Training referencing Mentor is not unique because many trainings may be done by the same mentor. |
| Paychange.interpreter\_id | Not unique | The foreign key in Paychange referencing Interpreter is not unique because many pay changes may be associated with the same interpreter. |

Besides the primary and foreign keys, I would select Call.call\_date to index. This is because a date range would be usually used to retrieve call records for a certain period. This index would be a non-unique index because many calls happen on the same day.

I would also select Training.available\_date to index. This is because participating in regular trainings is very important to the interpreters. Searching for trainings by available date would be performed frequently in a real-life scenario. This index would be a non-unique index because many trainings may happen on the same day.

Another one would be Client.client\_name. Unlike the interpreters who use their interpreter ids many times a day, client ids may not be handy when searching for a client’s information because reports and analysis would be retrieved by client (company) name more often than client id. This index is a non-unique index because it is possible that a company name can repeat.

Here is a screen shot of creating a foreign key index (for completed SQL statements, please see my sql script):

Graphical user interface, text

Description automatically generated

Here is a screen shot of creating query-driven indexes:

Text, letter

Description automatically generated

# Stored Procedure Execution and Explanations

Use case 1 – Adding new interpreter:

1. The interpretation company sends the app download link to interpreters’ phones.
2. The interpreters click on the link and download the app on their smart phones.
3. When interpreters open the app for the first time, they would need to follow the instructions on the screen, create their own account.
4. They would need to enter their basic information, enter their work status (part-time, full-time, or temporary).

For this use case, I will implement a transaction that adds interpreters’ information. I named the procedure ‘AddFulltimeInterpreter’, and the parameters are corresponded to the Address, Interpreter and Full-time tables. Since this procedure is always for a full-time interpreter, I did not include a parameter for work status but hardcoded work status as ‘F’ (i.e., full-time).

CREATE PROCEDURE AddFulltimeInterpreter

@first\_name VARCHAR (100) ,

@last\_name VARCHAR (100) ,

@email\_address VARCHAR (255),

@phone\_number DECIMAL(20) ,

@time\_zone VARCHAR (50),

@start\_date DATE,

@monthly\_salary DECIMAL(5),

@street VARCHAR (255),

@city VARCHAR (255),

@state VARCHAR (255),

@post\_code VARCHAR (65)

AS

BEGIN

DECLARE @current\_interpreter\_id INT = NEXT VALUE FOR interpreter\_seq;

DECLARE @current\_address\_id INT = NEXT VALUE FOR address\_seq;

INSERT INTO Address (address\_id,street, city, state, post\_code)

VALUES ( @current\_address\_id,@street, @city, @state, @post\_code);

INSERT INTO Interpreter(interpreter\_id,first\_name, last\_name,

work\_status, email\_address, phone\_number,

address\_id, time\_zone)

VALUES (@current\_interpreter\_id,@first\_name, @last\_name,

'F', @email\_address, @phone\_number,

@current\_address\_id,@time\_zone);

INSERT INTO Full\_time (interpreter\_id,first\_name, last\_name,start\_date,monthly\_salary)

VALUES (@current\_interpreter\_id,@first\_name, @last\_name,@start\_date,@monthly\_salary)

END;

Graphical user interface, text, application

Description automatically generated

Here is the screenshot of my stored procedure execution:

BEGIN TRANSACTION AddFulltimeInterpreter;

EXECUTE AddFulltimeInterpreter 'Alex','Johnson', 'alej@gmail.com', 6176541234,'EST', '10/12/2016', 5000,'1356 Harrison Blvd','Overland', 'KS', '66589';

COMMIT TRANSACTION AddFulltimeInterpreter;

Graphical user interface, text, application, email

Description automatically generated

Use case 6 – Viewing departmental info:

1. The interpreters would log into the app.
2. The app would provide an option to contact the department.
3. Then the option would have a drop-down menu that lists different departments.
4. The interpreters would choose the desired department, then the next page would show that department’s information and the associated contact personnel(s)’s contact information.

For this use case, I will implement a transaction that adds the department information. I named the procedure ‘AddDepartment’, and the parameters are corresponded to the Department table.

CREATE PROCEDURE AddDepartment

@department\_name VARCHAR (50),

@email\_address VARCHAR (255) ,

@phone\_number DECIMAL (20),

@contact\_personnel VARCHAR (255)

AS

BEGIN

INSERT INTO Department (department\_id, department\_name, email\_address, phone\_number, contact\_personnel)

VALUES (NEXT VALUE FOR department\_seq,@department\_name, @email\_address, @phone\_number, @contact\_personnel);

END;

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Use case 7- Viewing professional development opportunities.

1. The Interpreters log into their accounts.
2. The app would provide an option to show information in regard to all the training programs and associated mentors that are available to the interpreters.

For this use case, I will firstly implement a transaction that adds the mentors’ information, then another transaction that adds the training information. I named the procedure ‘AddMentor’ and ‘AddTraining’. ‘AddTraining’ uses a subquery to look up the associated mentor\_id.

AddMentor stored procedure:

CREATE PROCEDURE AddMentor

@first\_name VARCHAR (100),

@last\_name VARCHAR (100)

AS

BEGIN

INSERT INTO Mentor (mentor\_id, first\_name,last\_name)

VALUES (NEXT VALUE FOR mentor\_seq, @first\_name, @last\_name);

END;

BEGIN TRANSACTION AddMentor;

EXECUTE AddMentor 'Xiaoying', 'Wang';

COMMIT TRANSACTION AddMentor;

Graphical user interface, text, application

Description automatically generated

‘AddTraining’ stored procedure:

CREATE PROCEDURE AddTraining

@description VARCHAR (1000),

@available\_date DATE,

@completion\_date DATE,

@training\_status VARCHAR(15),

@first\_name VARCHAR (100),

@last\_name VARCHAR (100)

AS

BEGIN

DECLARE @current\_training\_id INT = NEXT VALUE FOR training\_seq;

INSERT INTO Training

(training\_id,description, available\_date, completion\_date, training\_status, mentor\_id)

VALUES (@current\_training\_id, @description,@available\_date, @completion\_date, @training\_status,

(SELECT mentor\_id from Mentor where first\_name=@first\_name AND last\_name = @last\_name));

END;

BEGIN TRANSACTION AddTraining;

EXECUTE AddTraining 'Medical terminology', '10/15/2020','11/15/2020','Completed','Xiaoying','Wang';

COMMIT TRANSACTION AddTraining;

Graphical user interface, text, application, email

Description automatically generated

# Question Identification and Explanations

First Query: Paychange

Paychange is the history table in my database. It tracks the pay changes for the interpreters. One example of using the Paychange table to retrieve useful information would be retrieving all the interpreters who have received over $ 500 pay raise for the year of 2021. This would be useful because my company has a rule that the interpreters who get a greater than $500 pay raise once will not be approved for another raise during that year. So, this information would be used by the HR department to make decisions regarding pay raises for the interpreters.

Graphical user interface, application

Description automatically generated

From this table, we can see these interpreters would not be able to have another raise in the year of 2021.

Second Query:

One example of retrieving from the supertype (Interpreter) and one of the subtypes (Full\_time) in my database would be retrieving contact info for all full-time interpreters who started working since the year 2016. The reason why I believe this information would be useful is that our company provides different training opportunities based on work status (full-time, part\_time or temporary) and the year of hiring. If there is a training only available for full-time interpreters who were hired after a certain year, we would use this query to retrieve all eligible interpreters’ contact info so that notifications could be sent out to them.

Graphical user interface, text, application

Description automatically generated

Third Query:

The third query would be an extension of the second query. Some trainings not only limit the interpreters’ work status and year of hiring, but also the language they speak.

One example would be retrieving contact info for all full-time interpreters who started working in 2016 and speak English. This query would involve four tables (Full\_time, Interpreter, Speaks and Language):

Graphical user interface, text, application, email

Description automatically generated

Fourth query:

I would like to create a view to retrieve the number of each type (full-time, part-time and temporary) of Mandarin interpreter currently active. This information would help the company’s Mandarin team know the number of Mandarin interpreters actively working at the moment in order to make sure that there are enough staff to meet the Mandarin interpretation needs.

This query would need four tables to retrieve the desired information. The SQL execution screenshot is demonstrated below:

Graphical user interface, text, application

Description automatically generated

From the result, we know that there are 2 full-time Mandarin interpreters, 1 part-time Mandarin interpreter and 1 temporary Mandarin interpreter currently working.

# Data Visualizations

I would like to create a query that shows how many interpreters the company has for each language. As a company providing interpretation services, the company first needs to know what languages they can provide service for, as well as the number of interpreters for that language. The company would also be able to use this information to promote their services to their clients.

The query that is used to retrieve this information is:

Graphical user interface, application

Description automatically generated

Chart, bar chart

Description automatically generated

In a real-life scenario, this bar chart would be much larger than this. From a bar chart like this, we would know what languages we have for now, and the number of interpreters for each available language. We can also see which language has the most interpreters (this bar chart shows the language Mandarin has the most interpreters), and also what languages have the least amount of interpreters (again this bar chart shows Korean, Russian, and Vietnamese have one interpreter).

I also would like to know the number of calls the company received each day for a certain period. This information would help the company know the call volume each day and analyze the data to make sure they have enough interpreters, especially for a typical high call volume day.

The query that is used to retrieve such information is:

Graphical user interface, text, application, email

Description automatically generated

Chart, bar chart

Description automatically generated

From this bar chart, we can tell that there are more calls on 2/2/2010, and fewer on the other two days. In a real-life scenario, we would have a much greater number of calls each day. From a chart like this, the company would be able to compare the call volumes in a certain period. Analyzing such data would help them find a call volume pattern to anticipate future needs. For example, based on my own experience, the call volume is usually higher on Mondays and Fridays, and right after holidays. The company would also have a better idea when they make interpreter schedules so that there would be enough interpreters especially for busy days.