**Advanced Database Systems Coursework Report**

# Task 1 ER Diagram of relational database Schema



# Task 2 Design undertaken And Reasons

The re-design of the database to a more object-relational approach:

To reduce the duplication of column names, several types were created:

* **ID\_Type** – this contained the following attributes:
  + All attributes, for this type, were set as VARCHAR2 as they will contain either letters or a combination of letters & numbers. This type is used over VARCHAR as this is reserved by oracle for distinction between NULL and an empty string. VARCHAR2 does not. VARCHAR2 can also store up to 4,000 characters. Varchar can only store up to 2,000
  + Title – this attribute will store the title of the person; max length is 4. This is due to the longest title allowed, ‘Miss’ being set as a constraint in the creation of customer and employee tables. <See here>
  + firsName – the length is set to max length 20. This is sufficient to store the majority of known first names
  + surName – the length is set to max 20 characters. This is sufficient to store most known surnames
  + niNum - this was set to max 12 characters in the following format – xx-xx-xx-xx. The standard format for UK national Insurance numbers. Setting to 15 is to even the number from 11
* **Address\_Type** – this type contains the following attributes:
  + All attributes were set to VARCHAR2
  + Street – max length is 30. To accommodate long named streets
  + City – max length is 20. This covers the ability to insert any city/town within the UK.
  + Postcode – max length is 10. This will cover the standard length of the UK postcode: xxxx-xxxx. Rounded to an even 10.
* **Phone\_Nos\_Type** – this type was created as an array.
  + Attributes for this type were set as VARCHAR2. As the home number, if present will be entered in the format xxxx-xxx-xxx. The mobiles numbers will also start with a 0, and will need to be entered as a string, rather than a number as the number type would remove this.
  + This type was used as there seemed to be no real reason why the three numbers should be kept as separate attributes. This way, the numbers entered would not need to be in a specific order, when they are entered. It is also restricted to only allow a max of three numbers, fulfilling the criteria for having the possibility of a home number and up to 2 mobile numbers.
* **Contact\_Details\_Type** – this type is a combining of the previous two types: Address\_Type and Phone\_Nos\_Type. This was done as both these types contain the two main options available for contact. It seemed appropriate.
* **Person\_Details\_Type** – This type gathers the ID\_Type and the Contact\_Details\_Type into this type. This was done so that the customer and employee types can inherit from this type, as they will both use the attributes that are present in the ID\_Type and Contact\_Details\_Type and removes the need to write, in each type, attributes that use these crated types.
* **Customer\_Type** –
  + this type inherits from Person\_Details\_Type
  + custID – max length 8. To simulate that the bank may have thousands/millions of customers, each with their own ID.
* **Employee\_Type** – this type, like customer, inherits from Person.
  + Attributes set as VARCHAR2 unless otherwise stated
  + empID – max length 8. Like custID of Customer\_Type, the max length was set to allow for thousands of employees to be added, if it is a multi-national bank, for future expansion. empID is inserted with the prefix ‘emp’ to prevent confusion with custID, if it was only numbers
  + position – max length 15. This was to accommodate the longest string entry for this attribute. Based on the constraint created for the table of this type. <see here>
  + salary – set as number. Max length 8. Due to this storing the employee’s annual salary, a whole number, setting the attribute as number was the most appropriate. As the inserted salary will not start with a 0, there would be no conflicts. And no need to store is as a VARCHAR2. Length 8 was chosen to allow large-salaried staff to be entered.
  + branch – This type references a tuple from the Branch table. This was chosen to include the branch details where the employee works, without out the need to add them separately. This also sets this attribute a s a foreign key to the Branch\_Table. As a tuple can only be referenced if it already exists. Removing the risk of a non-existent branch being added
  + join\_date – this attribute is set to Date. This is appropriate as this attribute will store an actual date.
  + supervisor – This attribute references a tuple from the Employee\_Table. This is to add the details of the employee’s supervisor to their record, without the need to re-write that employee’s information. By setting up this attribute as a recursive reference, it fulfils the need for this attribute to be a recursive foreign key. This attribute would be NULL, if the employee was head of the branch, or contain data relating to an employee in the next level up of the employee business structure.
* **Branch\_Type** – This type has the following attributes:
  + All attributes set as VARCHAR2 unless otherwise stated
  + bID – Max length of 6. This is to allow for hundreds, potentially thousands, of branches to be added. Each branch is prefixed with ‘B’.
  + branch\_address – this attribute is made up of the Address\_Type. This type already contains the attributes needed to insert the relevant data for each Branch address
  + branch\_phone – Max length 15. This type was chosen, instead of the previous Phone\_Nos\_Type, because only the phone number for a branch is being stored. No other number is required.
* **Account\_Type** – This type contains the following attributes:
  + All attributes set as VARCHAR2 unless otherwise stated
  + accNum – max length 10. This was to accommodate that there could be millions of accounts opened with this bank. accNum is prefixed with ‘Sav’ or ‘’cur’ depending on the type of account it is.
  + accType – max length 15. This to allow for either possible account type to be entered. The account available is constrained in the table for this object. <see here>. The length is also set to 15 to allow for possible changes to the types of account available to customers.
  + balance – this is set to Decimal. In the format 10,2. Decimal was used as a customer’s balance may not be whole number of ‘pounds’ but also contain ‘pence’. Therefore 2 of the 10 digits available are reserved for any possible numbers to the right of the decimal point. The remaining 8 are to accommodate for any size of balance, including millions. This format of xxxxxxxx.xx is typical of displaying the balance of a bank account.
  + bID – this references a tuple from the Branch\_Table. Same reason as this was chosen in the employee table: to store the information of the branch the account is associated with, without the need to repeat them.
  + interest\_rate – This attribute was set to decimal. In the format 3.2. This was chosen as most interest rates for bank accounts very rarely are in the double figures. So, a single digit was assigned to represent the whole number of the interest rate. The remaining 2 digits are to show the remaining interest rate that is to the right of the decimal point. To decimal places. This format of x.xx is typical representation of interest rates for bank accounts.
  + limitOfFreeOD – this was set as a number. Max length 8. Number was chosen as, typically, over draft limits are set as full numbers. If there an overdraft is set. Setting the max to 8 to represent an overdraft limit of millions, if the customer(s) are wealthy individuals.
  + openDate - this attribute is set to Date. This is appropriate as this attribute will store an actual date.
* **Cust\_Account** – This type contains the following attributes:
  + custID – references a tuple from the customer Table. This was chosen to add the customer details to the customer account table without the need to repeat entry of the data. Like previous ref’s
  + accNum – as above. But this attribute references a tuple from the account table.

The number of tables used was not changed from the number in the relational database scenario. All primary key constraints were set, as they are in the relational database. Except for the Customer\_Account\_Table as the inserts for this table are referenced tuples from other tables, were the necessary primary keys are created.

Foreign key constraints were not needed, since the foreign key requirement is fulfilled with referencing the relevant tuple from another table. Where the related attribute is set as the primary key of that table.

In addition, all remaining attributes were constrained to be not null. As this is data being help by a bank, it was deemed appropriate that all data for attributes will be needed to simulate the real-life scenario. The only exception to this was the Phone\_Nos\_Type where only a single entry is required to be considered not null, there was no specific constraint for this type besides this. They were created with the following additions:

* **Customer\_Table**
  + The custID was set as Primary Key
  + Insurance number was constrained to be only unique – it cannot be repeated. As this is a unique identifier to every person over the age of 16.
  + Title also was constrained to be only one of the following: Mr, Mrs, Miss, Ms or Dr. This was chosen as these are the common titles for individuals.
* **Employee\_Table**
  + empID was set as Primary Key
  + insurance number and title were constrained with the above mentions.
* **Branch\_Table**
  + bID was set as the primary key
* **Bank\_Account\_Table**
  + accNum set as Primary Key
  + The balance attribute was constrained to only accept either current or savings as a value. As per scenario requirements.
* **Customer\_Account\_Table**
  + Only thing of note here is there was no need to set the constraints, primary/foreign key, for either attribute as they are references to other tuples. This referencing fulfils both these scenario requirements.

# Possible Alternatives

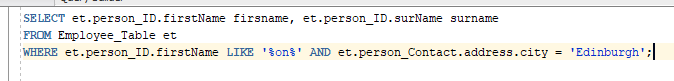
For created types:

* Phone\_Nos – this type could be designed as a nested table with three separate attributes, instead of being grouped into an array.
* Database creation inserts – PL/SQL can be used as an alternative to insert tables/types. Doing this would allow the creation of triggers that can be used to check the inserted data for the following attributes:
  + niNum attribute – using a trigger to check the niNum entered for each tuple is not already present a tuple. This way it can be guaranteed that each niNum entered will be unique, across customers and employees. Unless an employee is also a customer of the bank they work at.
  + An employee with an account, with the bank, is not at the same branch they work at – this can be achieved by using a trigger to check, when an employee is added, if they are also a customer – compare the niNum attribute. And checking if any customer account, associated with the employee, as a customer, is not at the same branch as the branch entered for the place of work for the employee.

# Task 4 SQL Statement

## 4.a

### SQL query



### Screen shot of Result

### Screen Clipping

### Comments

This query was simple. Only using two attributes from a single table to perform the required query. A slight issue arose when the entered city for some employees did not start with a capital letter. This was remedied in the inserts.

## 4.b

SQL query

### Screen Clipping

### Screen shot of Result

### Screen Clipping

### Comments

For this query to work. I had to get the branch address from the Bank\_Branch\_Table even though each tuple in the Bank\_Account\_Table does contain a reference to the tuple of the branch the account is at. By placing the count of savings account at each distinct branch, in a nested switch, and comparing the branchID to the those in the Bank\_Branch\_Table will prevent each branch appearing more than once. Also observed is the branches that do not have savings account. This shows that even with further accounts, branches etc added they will appear successfully.

## 4.c

### SQL query

### Screen Clipping

### Screen shot of Result

### Screen Clipping

### Comments

This query was a little complex. As it uses to inner join to compare data for tuples across multiple tables. As data, for tuples, is referenced from other tables that they require and ensuring the referenced data is correct. This query is very unlikely to go wrong. As the data from each tuple, in the relevant table is, basically, referring to itself this ensures there is no chance of tuples being mixed with other tuples containing different data.

The only issue with this query was the difficulty in only displaying the highest value of balance for each branch. As can been seen this was not solved.

## 4.d

### SQL query

### Screen Clipping

### Screen shot of Result

### Screen Clipping

### Comments

This query would be difficult to do without using references, as using references helps to select the required information. Using INNER JOIN allows the comparison of the niNum attribute to ensure only employees that are customers are selected. The query result shows that the employees who are customers do not have accounts at the same branch they work at. Fulfilling a requirement for the scenario.

## 4.e

### SQL query

### Screen Clipping

### Screen shot of Result

### Screen Clipping

### Comments

Unfortunately, I was unable to fully answer the task for this query – I was unable to select only the highest overdraft limits for each account. Other than this, the query performed successfully.

## 4.f

### SQL query

### Screen Clipping

### Screen shot of Result (Excerpt)

### Screen Clipping

### Comments

This query was difficult to execute. The only real successful execution was displaying the numbers for each customer. The following commented out WHERE clauses:

Screen Clipping

Allow to change the display to only show values that are NOT NULL and/or values that begin with ‘077’. The query may be fully possibly using a nested table to store values. But as only a varray was used. This, it would appear, would be difficult to perform.

## 4.g

### SQL query

### Screen Clipping

### Screen shot of Result

### Screen Clipping

### Comments

This was a relatively easy query to perform as references are used throughout the structure of the object-Relational database. This query will only work if Mr. Jones supervises Mrs. Smith who supervises employees. If either of these people change position/leave then the query would need to be changed. It is a very specific query. But the names can be interchanged with other staff, as long as they are employees at the bank.

## 4.h

### SQL query

### Screen Clipping

### Screen shot of Result (Excerpt)

### Screen Clipping

### Comments

This query has been partially answered. Medals have been awarded to individuals based on their years of service. But, no method was implemented to count the employee supervised. And, all employees are displayed that were awarded a medal. As the returned value from the method is a string. This can be compared with the WHERE clause to filter out employees who were not awarded a medal.

# Task 5 Critical Discussion of Object-Relational Model (ORM )against Relational Model (RM)

## Advantages

* Using references for table inserts allow the data to be accessed by only calling on the one table. No need for multiple table joins to display information. Whereas using a RM would require inner joins to be used. This advantage can be seen in the employee Table where selecting the branch an employee works at can be done through this employee table, where the relevant branch details have been referenced.
* Creating user-defined types and then using them for inheritance reduces the duplication of attributes – as seen with customer and employee types. These types both use the same attributes but as they inherit from another type, where these attributes are created, there is no need to define them in these two types.
* Using ORM allows for ‘real world’ relations to be created. RM does not. This can be seen with the use of referencing an employee’s supervisor, a recursive foreign key on the table itself. This would be difficult to perform on a RM schema, but easily achieved with an ORM schema. This is exemplified, In the provided ORM schema, where each employee’s supervisor is a reference to the supervisor’s tuple from the Employee\_Table. This ability with ORM allows for recursive queries.
* Another advantage for ORM is the use of constraints. Something that is not supported in RM and would need to be built into the DBMS that uses RM. Use of constraints can be seen, in the ORM schema, where the insert for title, position and account type all have been restricted to the stated acceptable inputs. The RM schema does not support this and would need to be down to user-insert discretion or, as stated, built into the DBMS.
* ORM schemas support the use of methods to allow for the behaviour of real-world objects. An example of this, in the scenario schema, is the method to provide employees with ‘medals’ when they reach certain milestones in their career with the bank. In the Rm schema, it would not be possible to do this, as it is not supported. The alternative would be to perform multiple queries, based on the milestone requirements, to display the employees eligible.

## Disadvantages

* Queries for selecting the required information can be complex as dot notation is needed to select the required attributes. If the user does not know the structure of the database, this can make it difficult to correctly form queries for database interrogation. Whereas RM does not use dot notation selecting attributes is far easier.
* Inserting data into an ORM can be complicated, especially if there is inheritance for the types used. As knowing the user-defined types the attributes are originally created in is essential for insert. Any errors can be lengthy to solve. RM does not have this issue as the attributes, for each table, are created with the table. There is no nesting. This can be seen with creating the inserts for the Employee\_Table: inserting the employee’s name, address and contact number(s) the object types the attributes were originally created need to be referenced during insert, else the inserts will not work.
* Another disadvantage would be that to develop an ORM schema is much costlier than a RM schema. This causes resistance among business’ to change over to the ORM.

# Task 6 Drop Statements

