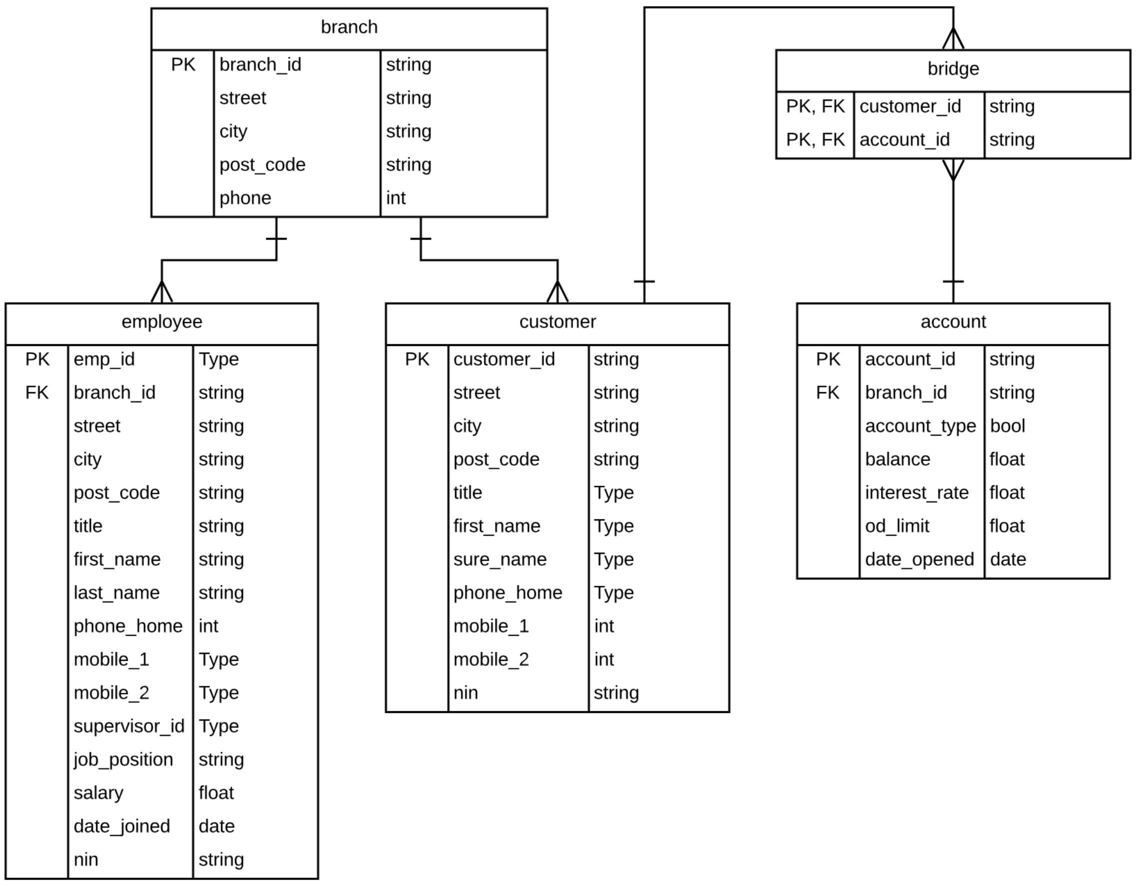
Database Report

Jostein Dyrseth

March 2018

1 Entity Relationship Diagram



branch

PK	branch_id	string
	street	string
	city	string
	post_code	string
	phone	int

bridge

PK, FK	customer_id	string
PK, FK	account_id	string

employee

PK	emp_id	Type
FK	branch_id	string
	street	string
	city	string
	post_code	string
	title	string
	first_name	string
	last_name	string
	phone_home	int
	mobile_1	Type
	mobile_2	Type
	supervisor_id	Type
	job_position	string
	salary	float
	date_joined	date
	nin	string

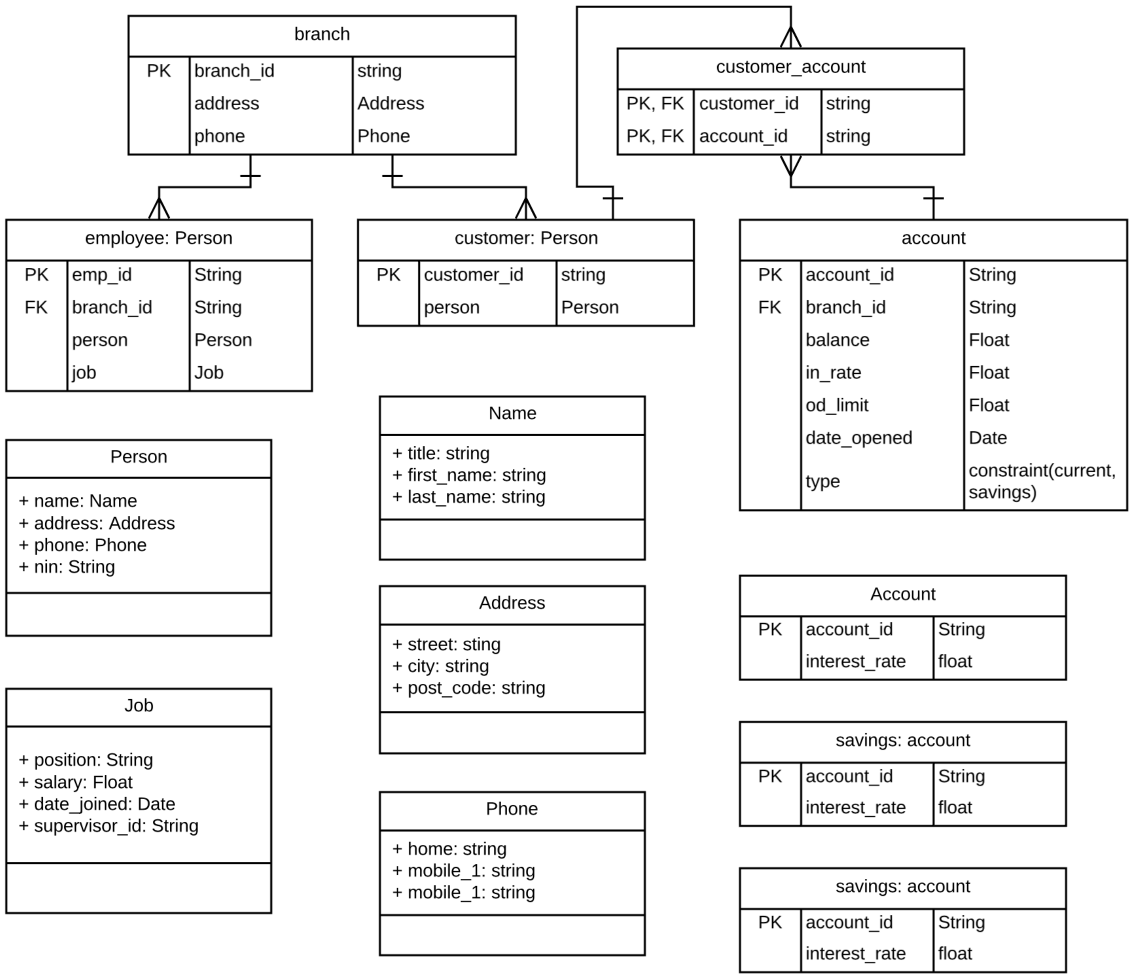
customer

PK	customer_id	string
	street	string
	city	string
	post_code	string
	title	Type
	first_name	Type
	sure_name	Type
	phone_home	Type
	mobile_1	int
	mobile_2	int
	nin	string

account

PK	account_id	string
FK	branch_id	string
	account_type	bool
	balance	float
	interest_rate	float
	od_limit	float
	date_opened	date

2 Redesign



2.1 Critical Discussion

2.1.1 Atomic Data-types & Sub-types

We start with creating the types: NameType, Address, Phone, Person and Job as objects. The Person type and Job type both hold nested types. Person holds name as a Name type, address as an Address, phone as Phone and a national insurance number or nin for short of type varchar2. The national insurance number should probably have been a number/integer. This is also unique. We will see this later while dealing with constraints.

We start with looking at the old design, which already has been represented to us. Taking some design decisions regarding how to restructure the design from relational to object-relational design is essential. First we make object-types for the following: Branch, Employee, Costumer, Customer Account and Account, so that we later can implement the tables for them. Each of these types hold more specific data in regards to what they are supposed to represent. See the code provided for more detail. For example, the Branch type holds three data-types: branch_id, address and phone. The branch_id is an atomic varchar2 datastructure holding a string. Each the address- and phone-type holds two different sub-types respectively, "Address" and "Phone". The Address holds a street name, city and a postcode where as the Phone holds a home phone-number, as well as two other numbers, mobile1 and mobile2.

In general all of these types are created with atomic types and subtypes to again hold more sensible data. Some of them are inheriting from other supertypes. This structure is more sensible to hold data since we are redesigning this to an object-relational database.

```
create type jd_EmployeeType as object(
    emp_id varchar2(4),
    branch_id varchar2(4),
    person jd_Person,
    job jd_JobType
);
/
```

2.1.2 Inserting Values to Tables

Inserting values to the tables is the next thing we do. We make sure to first specify the syntax "insert into branch_name values ()". We then specify the name of the type we want to insert along with the specific attributes. For example when inserting into the Branch_Table we do: "Address('33 Banking Street', 'Edinburgh', 'EH52 1BR'), Phone('605512336', '200501952', '362008274')".

While inserting into the Employee_Table we introduce another data type, sequence. The Employer needs a unique identifier to serve as it's PK. We write "seq_emp_id.nextval" in the very first row.

2.1.3 Constraints

We also add a constraint to the "job_position" attribute to only accept either: 'head', 'manager', 'project leader', 'accountant', 'cashier'. We also add two constraints to the account table: to assure that the account_id is indeed unique, as well as a account_type_constraint to check that the type is either a 'current' or 'savings' account.

No other constraints were made although there should probably have been more of them to ensure the system don't accept invalid values.

2.1.4 Referencing

In the CustomerAccountType we add a reference 'ref' to the customer-attribute and a reference to the account-attribute to point to that particular object in the table. This is a powerful feature in object-relational databases that we should utilize.

In the CustomerAccount_Table we add a customer and account attribute and 'ref' them to the respective types that was just explained. We hence use "scope is" to restrict the references to the object tables.

We also use the 'ref()' method to get get the reference of the object we pass in. So when we insert into the CustomerAccountTable, we also insert the object as a "pointer" or rather reference in this case.

```
insert into CustomerAccount_Table values (
  (select ref(c) from jd_Customer_Table c where c.customer_id = '1000')
  ,(select ref(a) from jd_Account_Table a where a.account_id = '8000'))
;
/
```

Once we try to say "select * from CustomerAccount_Table;", we get out 20 rows similar to this one:"

```
SYSTEM.CUSTOMERTYPE(
    '1000'
    ,SYSTEM.JD_PERSON(
        SYSTEM.JD_NAMETYPE(
            'Mrs'
            ,'Britt'
            ,'Haugland'
        )
    ,SYSTEM.JD_ADDRESS(
        '2 Silly Road'
        ,'Edinburgh'
        ,'RS33 2EF'
    )
    ,SYSTEM.JD_PHONE(
        '844676677'
        ,'300341038'
        ,'378804980'
    )
    ,'SEB5JCKTP'))
```

This is the power of sql.

2.1.5 Formatting

Formatting the output can help us read all the values quicker, better and with more understanding. The first two lines before the select statement, we have added a specification of how we want it to be displayed. We see that the first column should have the variable "First:Name" (vertical line). The vertical line just gives a new line to the heading. We also see that we want 10 character spacing. The same goes for the Last Name.

Note: Being constructive here, we see that we should add more formatting to the database in general, but because of time being limited I had to move on, but I show that I understand it at least.

3 Implementing & Populating the Tables

See code script provided for this section.

4 Queries

We have to spend some time figuring out what and how we can extract out the information we want form our database. In exercise c we have to use nested queries to get the data we want. We also create something called a view, in task c to basically hold as a variable for that particular query. See code for the queries below:

```
-- Query a:
column first_name heading 'First|Name'
column first_name format A10
column last_name heading 'Last|Name'
column last_name format A10

select e.person.my_name.first_name as first_name,
e.person.my_name.last_name as last_name
from jd_Employee_Table e
where e.person.my_name.first_name like '%Jos%'; -- used 'Jos' instead of 'on'

-- Output:
-- First      Last
-- Name       Name
-- ---------- ----------
-- Jostein    Dyrseth


-- Query b:
column num_of_accounts heading 'Number of|Accounts', format A10
column branch_id heading 'Branch|ID', format A6

select count(*) as num_of_accounts, branch_id
  from jd_Account_Table a, jd_Branch_Table b
  where a.ref_to_branch.branch_id = b.branch_id
    and a.account = 'savings'
  group by b.branch_id;

-- Output:
--  Number of Branch
--   Accounts ID
-- ---------- ------
--          1 902
--          3 903
--          4 900
--          2 901
--          1 912
--          1 913
--
--  6 rows selected


-- Query c:
column full_name heading 'Full Name', format A20
column balance heading 'Balance', format A10
column branch_id heading 'Branch ID', format 120

with total_balance_per_customer (branch_id, customer_id, balance) as (
  select a.ref_to_branch.branch_id, c.customer.customer_id, sum(a.balance)
    from jd_CustomerAccount_Table c
      inner join jd_Account_Table a
```

```
        on a.account_id = c.account.account_id
    where a.account = 'savings'
    group by c.customer.customer_id, a.ref_to_branch
)

select c.person.my_name.last_name || ', ' || c.person.my_name.first_name as full_name
    ,t.balance as balance
    ,t.branch_id as branch_id
  from total_balance_per_customer t
    left join jd_Customer_Table c
      on c.customer_id = t.customer_id
  where balance = (select max(balance)
                      from total_balance_per_customer t_2
                      where t.branch_id = t_2.branch_id)

-- Output:
-- Full Name               Balance Branch ID
-- -------------------- ---------- ---------
-- Haugland, Britt          330.43 901
-- Dyrseth, Frode            80.97 900
```

5 Critical Discussion

Based on the design having been shown to us originally we see that there are quite a few things we as developers can improve upon to take advantage of object-relational features. With Object-Relational Databases there are plenty of benefits. The idea of making types are very powerful and that you can inherit from other types is beneficial too. This is something I should have implemented in this database. Referencing is very powerful and also the idea of having nested tables. This is giving the developer a lot of power, and it is quicker to create in general. Some disadvantages here could be cost and performance issues while doing queries, so having a clear goal for the design is essential before implementing it. Methods was unfortunately not implemented in this database. This is also a powerful advantage in the object-relational model. Some people argue that the methods should not be implemented on the database side, but for others it is used as a great feature. In oracle they support two types of collections, either VArrays or nested tables. This takes use of objects within objects, hence we call it object-relational databases. Constraints are also not implemented in the original design and have to been utilized so that we enforce data that have aggregate data or specific data to that object. This is a great feature. There is also another language that oracle is providing that is called "PL/SQL", but I did not have more time to use this feature.

6 Drop Statements DONE

In order to run the file as an SQL-script, we have to delete or drop, all objects at the very beginning of the file. Doing this we can begin from a clean slate. Running the script will execute and compile all types and tables for us. The order of deletion matters, because we can't drop any types or tables that are dependent on any other ones, unless we include the force keyword, which can be bad practise. However, doing these drop statements makes life much easier, not having to manually delete each at a time in the SQL Developer software.

```sql
-- Dropping Types and Tables
drop type jd_NameType force;
drop type jd_Address force;
drop type jd_Phone force;
drop type jd_Person force;
drop type jd_JobType force;
drop type jd_BranchType force;
drop type jd_EmployeeType force;
drop table jd_Employee_Table;
drop type jd_CustomerType force;
drop table jd_Customer_Table;
drop type jd_AccountType force;
drop table jd_Account_Table;
drop type jd_CustomerAccountType force;
drop table jd_CustomerAccount_Table;
drop table jd_Branch_Table;
```