

CS252 - Fundamentals of Relational Databases - Module Assignment 2007

Introduction

Your task is to design a relational database to record information concerning personal bank accounts: details about the customers holding these accounts and details of individual payments of money into and out of these accounts.

You must do all the Exercises below. This assignment will count for 30% of the module and must be all your own work.

Marking

Syntax will be taken into consideration when marking, so is it highly recommended that you check your SQL statements syntax in Oracle on mimosa and check your **Tutorial D** syntax in Rel before submission. You may use screenshots, if you wish, to present your solutions.

Because we do expect you to check your syntax, we are likely to penalise minor syntax errors, such as unbalanced or misplaced parentheses and missing commas.

Late submission will incur a penalty of 5% per day.

Deadline: 12:00 (Midday), Friday, 30th November, 2007. Submission should be on paper, to cabinet room 006.

Scenario and Requirements

The bank has **customers**. Each new customer is assigned a unique **customer number**. Each customer's **name** and **address** (in free form—no need to split into street, town, post code, etc.) must be recorded. Optionally, one **e-mail address** for a customer can be recorded, as well as up to three **phone numbers**—but no more than one phone number of each type (home, work, cell). It is possible for the same phone number or e-mail address to be used by more than one customer. *Every phone number is of a unique type, regardless of the customer(s) using it.* Thus, for example, it is not possible for customer A's work number to be the same as Customer B's home number, even if A and B are the same customer.

To become a customer of the bank, a customer must open at least one **account**. It is possible, of course, for one customer to have several accounts (of types: current, savings or mortgage only). Each account is uniquely identified by an **account number**. For each account, the account holder's **customer number** must be recorded and also the **account type**, and the **date** on which the account was opened. The same customer is permitted to hold several accounts of the same type.

The recording of details of payments into and out of an account is complicated by the various different methods of payment. Each transaction against a particular account is automatically assigned a **transaction number** upon reception by the bank's computer. Transaction numbers

are unique within an account and you may assume they start at 1 for each account and thereafter increase sequentially by 1.

The information recorded for each transaction varies according to the kind of transaction, as follows.

Payment in:

For a payment into an account, the **account number**, **date of receipt** (by the bank), **time of receipt**, **source**, and **amount** (negative) are recorded. It is not possible for any payment to be made into an account before it has been opened.

Payment by cheque:

A cheque on a particular account is uniquely identified *within that account* by its cheque number. For a payment by cheque, the **account number**, **cheque number**, **date written** (as shown on the cheque), **date of receipt**, **time of receipt**, **payee**, and **amount** are recorded. It is not possible for a cheque to be written before the account has been opened. It is not possible for any payment to be made from an account, by this or any other means, before the account has been opened. It is possible for more than one payment to be made by cheque to the same payee with the same date written and date processed.

Payment by direct debit:

For a payment by direct debit, the **account number**, **date of receipt**, **time of receipt**, **payee**, and **amount** are recorded.

Payment by debit card:

This includes cash withdrawals from ATMs. A customer can be issued with any number of debit cards. Each debit card is for a particular account and several debit cards can be issued for the same account, perhaps for use by various family members. Each debit card is identified by a **card number**. For every debit card, the relevant **account number**, **cardholder's name**, and **expiry date** are recorded. For a payment by debit card, the **card number**, **date of receipt**, **time of receipt**, **payee**, and **amount** are recorded. It is not possible to use a debit card for any payment on a date after the expiry date.

Every transaction is of *exactly* one of the above four kinds. (And of course every payment in, payment by cheque, payment by direct debit, and payment by debit card is indeed a transaction.)

Questions

(Total: 100 marks; 1 marks for overall syntax)

1. (37 marks)

Design a database to meet the above requirements. Your design should be expressed in the form of **Tutorial D** VAR declarations and CONSTRAINT declarations as needed to meet the requirements. Do *not* include any data that does not relate directly to the stated requirements. Present your definitions in three sections, as follows:

1. Customers and their contact info;
2. Accounts and cards;
3. Transactions.

For each relvar in your design, give its predicate, in the form used in lecture HACD.4. For each constraint declaration, state which requirement(s) it addresses. If you find some requirement too difficult, you will get some credit for explaining why.

Your design must adhere as far as possible to each of the following conditions:

- (a) Each relvar in your database must be in **fifth normal form**. If you include a relvar that is not in 5NF you must give an explanation for this.
- (b) KEY constraints must be used wherever possible to implement uniqueness requirements. *Note:* There is a known bug in Rel when more than one KEY constraint is declared for the same relvar. The declarations are accepted but only the first KEY constraint is actually enforced. *You are not required to provide workarounds for this bug.*
- (c) Special values, such as blanks, zero, or the empty character string '', must *not* be used to represent information that is inapplicable, such as e-mail address for a customer who does not have one.
- (d) For the declared types of attributes, use CHAR for dates and times, RATIONAL for amounts, INTEGER for transaction numbers, and CHAR for everything else. For the dates and times, use character strings of the form 'yyyy-mm-dd' and 'hh:mm' (using the 24-hour clock), respectively. Use negative values for the amounts of payments in, positive for payments out.

Your design should use no more than four relvars for all types of transactions.

Using **Tutorial D** relvar update operators such as INSERT or assignment (:=), enter some sample data into your database. You can use your own discretion as to how much data to enter but it should minimally include:

- Customer 11111 with an e-mail address and customer 22222 without one.
- At least two phone numbers.
- An account 12345678 for customer 11111.

- A debit card with card number 987654321 for account 12345678.
- At least eight transactions of at least three different kinds against account 12345678.

The data you enter should be sufficient for you to give a reasonable test of the next exercise.

2. (12 marks)

Write a **Tutorial D** query to produce a single relation representing a bank statement for account 12345678. The result should contain a tuple for each transaction against the account, giving:

- transaction number
- date
- type (CR=payment in, CH=payment by cheque, DD=payment by direct debit, DC=payment by debit card)
- description (cheque number if available, otherwise payee for payments out or source for payments in)
- amount paid in or out (negative for payments in)
- balance (the sum of all payments for all transactions with transaction numbers up to and including this transaction's number)

Hint: Devise a query *q1* to meet all of the above requirements except the last one, the balance. You may want to break this query down into steps, using WITH (see the Notes for Lecture 5, Slide 4). Now define a virtual relvar, or use WITH again, to give a name such as `transcommon` for *q1*. (The syntax for defining a virtual relvar is given in Worksheet 4, Exercise 3.) You can now compute the balances by replacing the ...'s in the following expression by something appropriate (you have to work out what to write in their place):

```
EXTEND common ADD (
    SUM ( common RENAME ( ... AS ... )
        WHERE ... <= ... , Amount ) AS balance
```

3. (38 marks)

Using Oracle's SQL, repeat Exercise 1 with the conditions revised as follows:

- Condition (c) does not apply. Instead, you should aim to minimize the number of base tables (and that means *no more than one table for all types of transactions*) in your design while remaining in fifth normal form, using NULL to represent the absence of data that is optional.
- SQL has a much greater variety of built-in data types than **Tutorial D**. For each column you should choose the one you think is the most appropriate. Allow up to 50 characters for names and 100 for addresses.

- (c) You do not have to give predicates for your base tables.
- (d) You do not have to enforce any constraints concerning dates, as these tend to require subqueries that Oracle doesn't support. (You may be aware that commercial applications typically use SQL triggers in an attempt to work around such restrictions but you are not asked to attempt such solutions here.)

Hints:

Some of the requirements are made more difficult than they should be by Oracle's refusal to allow subqueries to be used in table constraints. A workaround that sometimes helps (but not always!) is to declare a redundant UNIQUE constraint so that FOREIGN KEY can be used to reference a set of columns that really constitute a proper superkey. Example:

```
CREATE TABLE T1 ( A INTEGER,
                  B INTEGER NOT NULL,
                  PRIMARY KEY ( A ) ;
                  UNIQUE ( A, B ) ;

CREATE TABLE T2 ( A INTEGER,
                  B INTEGER,
                  C INTEGER NOT NULL,
                  PRIMARY KEY ( A, C ),
                  UNIQUE ( B, C ),
                  FOREIGN KEY ( A, B ) REFERENCES T1;
```

We include the B column in T2 so that we can enforce a certain uniqueness constraint. But that column is redundant because T1 tells us which single B value goes with any given A value. The redundant UNIQUE constraint on T1 allows us to declare that "foreign superkey" in T2, making sure that the correct B value is indeed given in each row of T2.

4. (12 marks)

Using the database you set up in Exercise 3, repeat Exercise 2 in Oracle SQL.

Hint: The SQL counterpart of the hint given in Exercise 2 for computing the balance is likely to involve the use of a subquery in a SELECT clause of the following form:

```
SELECT common.*, ( SELECT SUM ( amount )
                  FROM    ...
                  WHERE   ... <= ... ) AS balance
FROM ( SELECT ... ) AS common
```