

Introduction to Databases

Basic SQL

Dr Paolo Guagliardo

(v26.0)

CHANGELOG

v26.0 Initial version.

Database schema. Consider the following schema:

CUSTOMER : *ID, Name, City*

ACCOUNT : *Number, Branch, CustID, Balance*

and assume that:

- the values for *ID* in CUSTOMER are all distinct (i.e., no two rows in CUSTOMER can have the same value for *ID*);
- the values for *Number* in ACCOUNT are all distinct (i.e., no two rows in ACCOUNT can have the same value for *Number*).

We will see how to write the above constraints in SQL later on in the course.

Problem 1. Write query the following queries in SQL:

- (1) Return the value of attribute *Number* of all accounts owned by customers called “John Doe”.
- (2) Return *Number* and *Branch* of all accounts owned by a customer with *ID* “xyz123”, only if there is such a customer in the CUSTOMER table.
- (3) Return *Number* and *Balance* of all overdrawn accounts in the “London” branch.
- (4) Return all pairs (*Name, Number*) where *Name* is the name of a customer and *Number* is the number of an account owned by that customer, such that the branch of the account is in a different city than the one where the customer lives.

Solution.

```
(1) SELECT A.number  
FROM Customer C, Account A  
WHERE C.id = A.custid AND C.name = 'John Doe' ;
```

or, using the explicit join syntax for readability:

```
SELECT A.number  
FROM Customer C JOIN Account A ON C.id = A.custid  
WHERE C.name = 'John Doe' ;
```

- (2) To make sure that customers owning the account do indeed exist in the CUSTOMER table, we need to join ACCOUNT and CUSTOMER:

```
SELECT A.number, A.branch
FROM   Customer C JOIN Account A ON C.id = A.custid
WHERE  C.id = 'xyz123' ;
```

Observe that it is *not* enough to select only from ACCOUNT. In other words, the following query

```
SELECT A.number, A.branch
FROM   Account A
WHERE  A.custid = 'xyz123' ;
```

does not comply with the requirements, because it does not guarantee the existence of a customer with ID “xyz123” in the CUSTOMER table.

- (3) This is just an easy filter on ACCOUNT:

```
SELECT A.number, A.balance
FROM   Account A
WHERE  A.branch = 'London' AND A.balance < 0 ;
```

- (4)

```
SELECT C.name, A.number
FROM   Account A JOIN Customer C ON A.custid = C.id
WHERE  A.branch <> C.city ;
```

In the above formulation, we (but not necessarily SQL or the DBMS) see the condition in the WHERE clause as a filter on the result of the join rather than as a part of the join condition itself. The reason for this is that common join conditions typically match *for equality* the values of one or more columns, as these are the kind of joins (called *equi-joins*) that DBMSs are good at optimising.

Of course, we could also equivalently write:

```
SELECT C.name, A.number
FROM   Account A JOIN Customer C ON A.custid = C.id AND A.branch <> C.city ;
```

or

```
SELECT C.name, A.number
FROM   Account A, Customer C
WHERE  A.custid = C.id AND A.branch <> C.city ;
```

The last version is the most agnostic with respect to what is a filter and what is a join condition.

Problem 2. Given an instance of the schema above, what is the answer to the following query?

```
SELECT C1.id, C1.name, C1.city
FROM   Customer C1, Customer C2
WHERE  C1.id = C2.id AND ( C1.name <> C2.name OR C1.city <> C2.city ) ;
```

Solution. First, notice how the use of range variables in the FROM clause makes it possible to join a table with itself, which is called a *self-join*. The above query looks for rows in CUSTOMER for which there is another, distinct row in CUSTOMER with the same value for *ID*. Since this is forbidden by the constraints/assumptions we have in the schema, the answer to the query is empty for every instance of that schema.

Problem 3. Describe a situation in which a query keeps working without errors after a change of the given schema, but produces a different answer than what was originally intended, and discuss how the situation could be avoided with an appropriate use of table aliases and qualification of attributes.

Solution. Consider the following query:

```
SELECT Name
FROM   Customer, Account
WHERE  ID = CustID AND City = 'Edinburgh' ;
```

which returns the *name of all customers from Edinburgh who have an account*. If we change the schema as follows:

```
ALTER TABLE Customer DROP COLUMN City ;
ALTER TABLE Account RENAME COLUMN Branch TO City ;
```

the above query keeps working without errors, but it no longer has the same intended semantics, as it will now return the *name of all customers who have an account in Edinburgh*.¹ This kind of problems can be quite hard to track down, because everything seems to work normally. In such situations, it would be preferable that the query threw an error to signal an anomaly, which can usually be achieved with an appropriate use of range variables and full qualification of attributes. For the case at hand, the following query gives an error after the schema change:

```
SELECT C.Name
FROM   Customer C, Account A
WHERE  C.ID = A.CustID AND C.City = 'Edinburgh' ;
```

because table CUSTOMER no longer has a column named *City*.

¹The same would apply also if the original column *City* in CUSTOMER were renamed to something else, say *Residence*, instead of being dropped.