



In this lab, you will again use the PHPMyAdmin web interface to your database. You will work with the tables from the airline database you installed for lab sheet 8 and modified during this lab 10. You will create a procedure step by step and then call it. As an exercise for yourself, you will then modify the procedure a little. Note that the setup on the student MySQL server does not currently permit students to create functions, so these will not be used here. However, we will put a prepared statement together and execute it, and you get to extend this a bit, too (as an optional challenge).

Note again that these lab sheets are not assessed, but they may be discussed in tutorials and their content is examinable!

Estimated time to complete this lab: 60 minutes.

A) Finding a plane for a route

The operations manager of our airline is looking for a simple way to assign a suitable plane to new routes. Given the **route_id**, she wants a procedure that comes up with the **aircraft_id** of the aircraft that has the lowest workload (distance flown per week) and is suitable for the route category – long or short haul. A route is considered short haul if it is 3000 km or less one way and long haul if it is over 3000 km. The A380 and B747 are long haul aircraft whereas the B737 and A320 are short haul. The **aircraft_id** found is then to be inserted into the **Route** table by the procedure.

Let's start by getting the SQL queries for this together. For now, we'll ignore the long haul/short haul requirement and just look for the aircraft with the lowest workload. This query from lab sheet 8 has come in handy before - it computed the return flight distance for each aircraft in our fleet:

```
SELECT aircraft_type, registration, 2*sum(distance)
FROM Aircraft
LEFT JOIN Route
ON Route.aircraft_id = Aircraft.aircraft_id
GROUP BY registration;
```

If we look for the smallest one-way distance, we can leave out the multiplication by 2. We may be tempted to wrap a min() aggregate function around the distance sum, and of course we want the aircraft id as well:

```
SELECT Aircraft.aircraft_id, aircraft_type, registration, min(sum(distance))
FROM Aircraft
LEFT JOIN Route
ON Route.aircraft_id = Aircraft.aircraft_id
GROUP BY registration;
```

Try this and see what happens. Doesn't work? Illegal use of aggregate function? Indeed: The sum() function is already applied to the groups, and MySQL simply isn't smart enough that you want the min() to then work across groups. But wait - help is at hand: Let's drop the min() and let's sort the result set:

```
SELECT Aircraft.aircraft_id, aircraft_type, registration, sum(distance)
FROM Aircraft
LEFT JOIN Route
ON Route.aircraft_id = Aircraft.aircraft_id
GROUP BY registration ORDER BY sum(distance);
```

Now this works, except that we get a whole result set, not just the top row, which is the one with the plane we're actually after. MySQL along with a number of other SQL RDMBS knows a nonstandard extension to SQL, the LIMIT clause, which just so happens to come in handy here:

```
SELECT Aircraft.aircraft_id, aircraft_type, registration, sum(distance)
FROM Aircraft
LEFT JOIN Route
ON Route.aircraft_id = Aircraft.aircraft_id
GROUP BY registration ORDER BY sum(distance) LIMIT 0, 1;
```

This limits the maximum number of records in the result set, starting at record 0, to 1. So we now have our least busy plane. Try this out to ensure it works for you.

Next, we'll need to find a way of accommodating the long/short haul requirement, presuming that we'll have the distance of the new route available as a parameter. Easy, e.g., as part of an additional WHERE clause, with the whole query for now preceded by a SET for the distance parameter:

```
SET @distance = 4000; # parameter variable for the distance
SELECT Aircraft.aircraft_id, aircraft_type, registration, sum(distance)
FROM Aircraft
LEFT JOIN Route
ON Route.aircraft_id = Aircraft.aircraft_id
WHERE
    (
        (aircraft_type = 'A380'
        OR
        aircraft_type = 'B747')
        AND @distance > 3000
    )
OR
    (
        (aircraft_type = 'A320'
        OR
        aircraft_type = 'B737')
        AND @distance <= 3000
    )
GROUP BY registration ORDER BY sum(distance) LIMIT 0, 1;
```

Try this out and convince yourself that it works. What we now need is the result in a variable, which we can later use in the UPDATE query to the Route table. Enter another clause of the SELECT statement – the INTO clause. This clause can be used to store the result fields from a SELECT into a SQL variable. Since we only want one value, and the sum of the distances is now mentioned in the ORDER BY clause, we can drop **aircraft_type** and **registration** as well as **sum(distance)** from the SELECT list, and one variable will suffice:

```
SET @distance = 4000; # parameter variable for the distance
SELECT Aircraft.aircraft_id
FROM Aircraft
LEFT JOIN Route
ON Route.aircraft_id = Aircraft.aircraft_id
WHERE
    (
        (aircraft_type = 'A380'
        OR
        aircraft_type = 'B747')
        AND @distance > 3000
    )
OR
    (
        (aircraft_type = 'A320'
        OR
        aircraft_type = 'B737')
        AND @distance <= 3000
    )
GROUP BY registration ORDER BY sum(distance) LIMIT 0, 1
INTO @aircraft_id;
SELECT @aircraft_id; # check that it worked
```

Now we're ready to do design the UPDATE, assuming that the **route_id** of the new route will be in a parameter named **new_route_id**:

```
UPDATE Route SET aircraft_id = @aircraft_id WHERE route_id = new_route_id;
```

Given the **new_route_id**, we can also find the distance quite easily:

```
SELECT distance FROM Route WHERE route_id = new_route_id INTO @distance;
```

So far, so good. If we put it all together and strip the initial SET and the SELECT for the aircraft id at the end, we get the body of our procedure:

```
SELECT distance FROM Route WHERE route_id = new_route_id INTO @distance;
SELECT Aircraft.aircraft_id
  FROM Aircraft
  LEFT JOIN Route
  ON Route.aircraft_id = Aircraft.aircraft_id
  WHERE
    (
      (aircraft_type = 'A380'
      OR
      aircraft_type = 'B747')
      AND @distance > 3000
    )
  OR
  (
    (aircraft_type = 'A320'
    OR
    aircraft_type = 'B737')
    AND @distance <= 3000
  )
  GROUP BY registration ORDER BY sum(distance) LIMIT 0, 1
  INTO @aircraft_id;
UPDATE Route SET aircraft_id = @aircraft_id WHERE route_id = new_route_id;
```

To declare the actual procedure, we need to note that the body above contains semicolons, which are default delimiters in SQL. This is a problem, because the CREATE PROCEDURE is a single statement that needs to finish at the first delimiter it encounters – which would be the one after the INTO clause. So in order to put the whole body into a procedure, we need to change the delimiter temporarily:

```
delimiter //

CREATE PROCEDURE proc_plane_for_route(new_route_id INT UNSIGNED)
BEGIN
  SELECT distance FROM Route WHERE route_id = new_route_id INTO @distance;
  SELECT Aircraft.aircraft_id
    FROM Aircraft
    LEFT JOIN Route
    ON Route.aircraft_id = Aircraft.aircraft_id
    WHERE
      (
        (aircraft_type = 'A380'
        OR
        aircraft_type = 'B747')
        AND @distance > 3000
      )
    OR
    (
      (aircraft_type = 'A320'
      OR
      aircraft_type = 'B737')
      AND @distance <= 3000
    )
    GROUP BY registration ORDER BY sum(distance) LIMIT 0, 1
    INTO @aircraft_id;
  UPDATE Route SET aircraft_id = @aircraft_id WHERE route_id = new_route_id;
END
//

delimiter ;
```

Great. Now we have a method. Let's find a plane for the Auckland-Honolulu run (which isn't that new but currently has no valid plane assigned), with **route_id** 9 (before you do, check out the aircraft id on that route – it should still be 7 unless you have changed it):

```
CALL proc_plane_for_route(9);
COMPSCI 280 S1 2011
```

Now have a look at the **Route** table to convince yourself that the Honolulu run has a new **aircraft_id**. Hey, well done, this was a biggie!

Exercise: Modify the procedure such that you can pass in a route name (such as “Auckland-Apia”) and a distance instead of the new route id and the route record in the **Route** table is inserted automatically before the aircraft is assigned to the route.

B) Prepared statements

Prepared statements are useful if we want to run the same query several times with only a small change in parameters. For example, consider the following query, which lists all routes that can be flown by a particular plane in the right haul category:

```
SELECT route_description, distance
FROM Aircraft, Route
WHERE (
    (
        (
            aircraft_type = 'A380'
            OR
            aircraft_type = 'B747'
        )
        AND distance >3000
    )
    OR
    (
        (
            aircraft_type = 'A320'
            OR
            aircraft_type = 'B737'
        )
        AND distance <=3000
    )
)
AND registration = 'ZK-HUGE';
```

So what if we wanted to pose this query multiple times for different registrations? Then we can allocate a prepared statement:

```
PREPARE stmt_possible_aircraft_routes
FROM 'SELECT route_description, distance
FROM Aircraft, Route
WHERE (
    (
        (
            aircraft_type = \'A380\'
            OR
            aircraft_type = \'B747\'
        )
        AND distance >3000
    )
    OR
    (
        (
            aircraft_type = \'A320\'
            OR
            aircraft_type = \'B737\'
        )
        AND distance <=3000
    )
)
AND registration = ?';
```

Note that our query is now a string, and quotes around literal strings inside that string have to be backslash-escaped in MySQL. Note also that the registration string has been replaced by a placeholder, and that there are no quotes around the placeholder (except for the one that ends the query string). This is because prepared statements expect us to supply the right data type here, i.e., a string rather than just the content of a string.

Copy this query into a text editor such as Notepad++ for future use. If you enter the query as is into PHPMysqlAdmin, it will prepare the statement – and immediately deallocate it again (dispose of it) as each query you enter into PHPMysqlAdmin is a single session, and prepared statements don't persist between sessions.

Enter it again, but don't hit "Go" yet. Now we can call the query like this, in the same query window as the PREPARE statement:

```
SET @registration = 'ZK-DUCK';  
EXECUTE stmt_possible_aircraft_routes USING @registration;
```

Now hit "Go". This should work.

You might think that the simplified version:

```
EXECUTE stmt_possible_aircraft_routes USING 'ZK-DUCK';
```

would work, but it doesn't and you must use a variable here in MySQL.

Exercise: Modify the PREPARE statement so that we can also set the threshold at which a flight becomes long haul, so we can execute the statement like this:

```
SET @distance = 6000;  
SET @registration = 'ZK-GIGA';  
EXECUTE stmt_possible_aircraft_routes USING @distance, @registration;
```

Hint: This is a little challenge because the distance turns up twice in our where clause. I solved the problem with a control flow function, by the way.