Tips and Tricks for Project Task 2

Data Storage Paradigms, IV1351

1 PostgreSQL documentation

Here are two important sites with PostgreSQL documentation:

- The official PostgreSQL documentation can be found at `https://www.postgresql.org/docs/current/index.html`
- The PostgreSQL tutorial is found at `https://www.postgresqltutorial.com/`. It is a web site with extensive PostgreSQL documentation. It's easier to follow and has more examples than the official documentation, but doesn't cover all details.

2 The Database Used in the Examples

All examples in this document refer to the same example database, which is shown in Figure 1. This is a simple employee database, which contains employees working at departments. The employees have managers, which are also employees, and the employees fulfill one or more roles. There are also phones and laptops. A phone is either not in use or assigned to an employee. A laptop can be assigned to a department, apart from being assigned to an employee or not being in use. You're strongly advised to create this database and experiment with the queries below. There are Sql scripts for creating the database and filling it with data at `https://github.com/KTH-IV1351/proj-task-3.git`

3 How to Read Data Spread Over Multiple Tables

It's very common that the data looked for in a query isn't all found in the same table, but instead spread over multiple tables. Such a query normally contains `JOIN` clauses, but how to extract the sought data and join it? An often used approach is to first merge all required tables into one large table, and then extract the required rows and columns from that merged table. The merging is done with `LEFT JOIN` if the tables have some

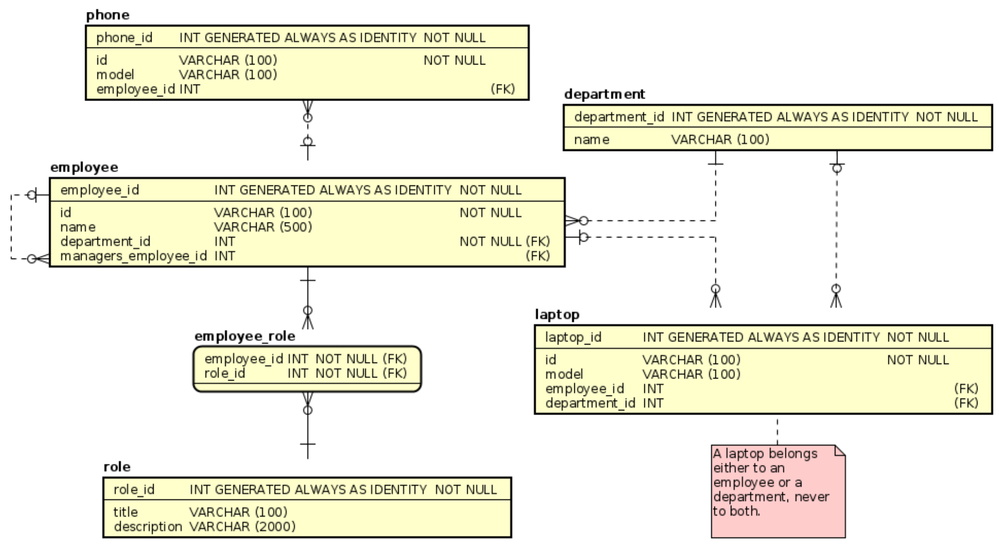
phone
phone_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
id VARCHAR (100) NOT NULL
model VARCHAR (100)
employee_id INT (FK)
department
department_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
name VARCHAR (100)
employee
employee_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
id VARCHAR (100) NOT NULL
name VARCHAR (500)
department_id INT NOT NULL (FK)
managers_employee_id INT (FK)
employee_role
employee_id INT NOT NULL (FK)
role_id INT NOT NULL (FK)
laptop
laptop_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
id VARCHAR (100) NOT NULL
model VARCHAR (100)
employee_id INT (FK)
department_id INT (FK)
role
role_id INT GENERATED ALWAYS AS IDENTITY NOT NULL
title VARCHAR (100)
description VARCHAR (2000)
A laptop belongs either to an employee or a department, never to both.

Figure 1: The employee database used for all Sql examples in this document

column in common, on which a join can be done. If there's no such common column, a `FULL JOIN` is done instead.

As an example. let's say we want to list all phones and laptops used at department number two. This means all phones and laptops used by an employee working at department number two, and all laptops assigned directly to department two. This is a quite artificial query. In reality we would rather want to list phones and laptops in use at a department with a particular name, not a department with a particular primary key value. That will be done, but it will have to wait until Section 4, where subqueries are covered. Now the desired query will be constructed step by step, which is the best approach when writing a complex query. First, as can be seen in Figure 1 above, the phone id comes from the column `id` in the table `phone`, and the laptop id from the column `id` in the table `laptop`. This means we need data from both `phone` and `laptop` tables, which means we must join them. They do have one column in common, namely `employee_id`, but we can't join on that column since that would put phone and laptop for a particular employee on the same row, and an employee doesn't necessarily have the same number of phones and laptops. What we want is one single list, with just one phone *or* laptop on each row. Therefore we do the `FULL JOIN` in Listing 1. Since we consider the tables to be completely unrelated, we could have joined on any columns.

```sql
SELECT * FROM phone
FULL JOIN laptop ON laptop.id = phone.id;
```

Listing 1: A full join of two completely unrelated tables.

Next we want to join also the `employee` table, since we have to find at which department the employee using each device is working. This will be a `LEFT JOIN`, since each row in both `phone` and `laptop` is really related to an `employee`, namely the employee using that device. When this `LEFT JOIN` is added, we get the query in listing 2.

```
1  SELECT * FROM phone
2  FULL JOIN laptop ON laptop.id = phone.id
3  LEFT JOIN employee
4      ON laptop.employee_id = employee.employee_id
5          OR phone.employee_id = employee.employee_id;
```

Listing 2: A left join with the related `employee` *table.*

Before proceeding, a word of warning! *The* `OR` *clause on line 5 in Listing 2 turns that statement into a nested loop.* This happens since the only way to evaluate if either of the or conditions is true, is to compare all rows in the employee set with all rows in the laptop/phone set. How this can be avoided is explained in Section 8, but that speedup will also make the Sql more complicated, so for now let's live with this fact. Just be aware of the problem.

Now, to proceed with the query, our result already includes all required data. Since it also includes a lot of irrelevant data, it's time to extract the data we're interested in, which is done in Listing 3. The columns we are interested in are selected, and given distinctive names, on line one. On line seven we choose only phones and laptops used in department number two, since that's all we were interested in. Note that a laptop can be assigned directly to the department, not just to an employee working at the department.

```
1  SELECT phone.id AS phone_id, laptop.id AS laptop_id
2  FROM phone
3  FULL JOIN laptop ON laptop.id = phone.id
4  LEFT JOIN employee
5      ON laptop.employee_id = employee.employee_id
6          OR phone.employee_id = employee.employee_id
7  WHERE employee.department_id = 2 OR laptop.department_id = 2;
```

Listing 3: The relevant data is extracted from the joined tables

This gives us one column with all phones used at department two, and one column with all that department's laptops, which could very well be our complete query. It might, however, be clearer if we also add a column with the department number. The problem is that the department number might come either from the `employee` table or from the `department` table, but we want all department numbers in the same column. This can be solved with the `CONCAT` function, as on line 2 i Listing 4. The `CONCAT` function actually just concatenates strings, but what serves us here is that `NULL` values are simply ignored. What we want is to merge two columns, where one is NULL and the other is the department number. This means that in our example we will always concatenate the string '2' with `NULL`, which, since `NULL` is ignored, will always be '2'.

```sql
SELECT phone.id AS phone_id, laptop.id AS laptop_id,
    CONCAT(laptop.department_id, employee.department_id)
        AS department_id
FROM phone FULL JOIN laptop ON laptop.id = phone.id
LEFT JOIN employee
    ON laptop.employee_id = employee.employee_id
        OR phone.employee_id = employee.employee_id
WHERE employee.department_id = 2 or laptop.department_id = 2;
```

Listing 4: A column with the department number has been added

4 Make Use of Subqueries

Subqueries are a very valuable tool for writing queries. However, this section is far from a complete subquery tutorial. It's just a reminder to use them, along with a short example. For the subquery example, we'll consider the query in Listing 4 above, which listed laptops and phones used at department number two. This is a quite artificial example, since in reality we would most likely want to list devices used at a department with a specified name, not a department with a certain PK value. To achieve this, the PK criteria in the `WHERE` clause can be replaced with a subquery, which returns the PK value of a department with a specified name. Listing 5 shows such a query returning the devices in use at the "admin" department.

```sql
 1  SELECT phone.id AS phone_id, laptop.id AS laptop_id,
 2      CONCAT(laptop.department_id, employee.department_id)
 3          AS department_id
 4  FROM phone FULL JOIN laptop ON laptop.id = phone.id
 5  LEFT JOIN employee
 6      ON laptop.employee_id = employee.employee_id
 7          OR phone.employee_id = employee.employee_id
 8  WHERE employee.department_id = (
 9      SELECT department_id
10      FROM department
11      WHERE name = 'admin'
12  ) OR laptop.department_id = (
13      SELECT department_id
14      FROM department
15      WHERE name = 'admin'
16  );
```

Listing 5: A subquery is used to get the PK of a department with a specified name, on lines 9 and 13

Subqueries can be placed both in a `SELECT` clause, a `FROM` clause and a `WHERE` clause. To learn more about subqueries, see for example the postgres documentation, `https://www.postgresql.org/docs/current/index.html` or the postgres tutorial, `https://www.postgresqltutorial.com/`. Below are some specific suggested pages with information about subqueries.

- The subquery section in the postgres tutorial, covers subqueries in a `WHERE` clause, `https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-subquery/`.
- The postgres documentation page about subqueries in a `WHERE` clause, `https://www.postgresql.org/docs/current/functions-subquery.html`.
- The postgres documentation about the `FROM` clause, where section 7.2.1.3 covers subqueries, `https://www.postgresql.org/docs/15/queries-table-expressions.html#QUERIES-FROM`.

5 Remember Views and Materialized Views

Views are very useful when writing database queries. This section first explains the advantage of using a view at all, then there's a discussion of when a materialized view can be appropriate.

5.1 Non-Materialized View

The purpose of a view that's not materialized is to reuse a query, which can serve two purposes. First, to avoid rewriting the same query many times, which means to avoid duplicated code. Second, to give a name to a complex query, in order to explain what it does. As can be seen from this, using views when writing Sql has the same importance as using private methods in an object-oriented program. As an example, let's again consider the search for laptops and phones, used above in Sections 3 and 4. By now, see Listing 5, we've put some effort into this query, and it would be nice to be able to reuse it. To achieve that, we'll store the query in a view. It's a good practice to not include the `WHERE` clause in the view, in order to make it a bit more general. This means our view will look like Listing 6. When this view exists in the database, the query to list phones and laptops in use at the "admin" department is simplified to Listing 7. Note that the same view can be used also to list devices used in another department, all that has to be changed is the search criteria in Listing 7, not the view itself in Listing 6.

```
CREATE VIEW phone_and_laptop AS
    SELECT phone.id AS phone_id, laptop.id AS laptop_id,
        CONCAT(laptop.department_id, employee.department_id)
            AS department_id
    FROM phone FULL JOIN laptop ON laptop.id = phone.id
    LEFT JOIN employee
```

```
        ON laptop.employee_id = employee.employee_id
            OR phone.employee_id = employee.employee_id;
```

Listing 6: An example of a view

```
SELECT phone_id, laptop_id, department_id
FROM phone_and_laptop
WHERE department_id = (
    SELECT CAST(department_id AS TEXT)
    FROM department
    WHERE name = 'admin'
);
```

Listing 7: A query using the view in Listing 6

To further emphasize the utility of views, let's use the view in Listing 7 for some other purpose than listing phones and laptops in a particular department. That view can also be used by a query telling how many phones and how many laptops are used at each department, and how many are not used at all, as can be seen in Listing 8.

```
SELECT COUNT(phone_id) AS phones, COUNT(laptop_id) AS laptops,
    (CASE WHEN department_id = ''
          THEN 'not in use'
          ELSE department_id
    END) AS department_id
FROM phone_and_laptop
GROUP BY department_id
ORDER BY department_id;
```

Listing 8: Another example of a query using the view in Listing 6

5.2 Materialized Views

Now that the usage of views is clear, it's time to see when a materialized view is preferred. Both kinds of views are stored in the database schema, and can be used just like tables by other queries. The difference is that a non-materialized view is just like an alias for some Sql code, the result of executing it isn't stored in any way. A materialized view, on the other hand, is executed when it's created, and the result is stored on disk. This means that reading will be faster, since all searching, sorting, etc is already done, and the result is ready to retrieve. On the other hand, the result of a materialized view isn't updated when tables involved in the underlying query are updated. It's necessary to explicitly use `REFRESH` to update a materialized view, which of course also takes time. This means that the decision on whether to use a materialized view depends on read vs write performance. Below are some factors to consider, but also remember the importance of measurements. Don't optimize if it's not needed.

- How often is data read and how often is it written?
- Is read (or write) performance currently a problem?
- Must reads return the current state of the data? Is it perhaps OK to return data that's one minute old? One hour old? One day old?

6 How to Analyze Queries With `EXPLAIN`

The `EXPLAIN` command shows how a query is actually performed, and is very useful to understand which parts of a query are more time-consuming. Note that this coverage of `EXPLAIN` is postgres specific. There are similar commands in other database management systems as well, in for example MySQL it's also called `EXPLAIN`, but it doesn't work exactly as in postgres. Also, this section is far from a complete coverage of `EXPLAIN`, more information can be found in the postgres documentation. There is a good explanation of how to interpret the output at `https://www.postgresql.org/docs/current/using-explain.html`. There is also an introduction to the command and its parameters, with some examples, in the postgres tutorial, `https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-explain/`, and a more thorough coverage of the command in the postgres documentation, `https://www.postgresql.org/docs/current/sql-explain.html`.

As an introduction to `EXPLAIN`, let's look at the result of running it on the query in Listing 4, by giving the command in Listing 9. As can be seen, all that's needed to analyze a query is to write `EXPLAIN` before the query being analyzed. However, it's a good practice to first run the command `VACUUM ANALYZE`, as is done in Listing 9. This command garbage collects deleted rows and collects statistics about the database. The result of the command in Listing 9 can be seen in Listing 10 (formatting has been changed to make everything fit on the page).

```sql
VACUUM ANALYZE;

EXPLAIN SELECT phone.id AS phone_id, laptop.id AS laptop_id,
    CONCAT(laptop.department_id, employee.department_id)
        AS department_id
FROM phone FULL JOIN laptop ON laptop.id = phone.id
LEFT JOIN employee
    ON laptop.employee_id = employee.employee_id
        OR phone.employee_id = employee.employee_id
WHERE employee.department_id = 2 or laptop.department_id = 2;
```

Listing 9: Analyzing the query in Listing 4

```
 1                           QUERY PLAN
 2 -------------------------------------------------------------
 3  Nested Loop Left Join  (cost=6.50..492.81 rows=93 width=54)
 4    Join Filter: ((laptop.employee_id = employee.employee_id)
 5               OR (phone.employee_id = employee.employee_id))
 6    Filter: ((employee.department_id = 2)
 7          OR (laptop.department_id = 2))
 8    ->  Hash Full Join  (cost=6.50..13.25 rows=200 width=34)
 9          Hash Cond: ((phone.id)::text = (laptop.id)::text)
10          ->  Seq Scan on phone (cost=0.00..4.00 rows=200
11                                              width=15)
12          ->  Hash  (cost=4.00..4.00 rows=200 width=19)
13                ->  Seq Scan on laptop  (cost=0.00..4.00
14                                      rows=200 width=19)
15    ->  Materialize  (cost=0.00..2.59 rows=106 width=8)
16          ->  Seq Scan on employee  (cost=0.00..2.06 rows=106
17                                                  width=8)
```

Listing 10: The result of executing the command in Listing 9.

The output has a tree structure, with the root node at the first line and child nodes indicated with `->`. When reading the output we start at the bottom, indicated by the rightmost arrow (`->`), in Listing 10 that node is found on line 13. There, the query planner says it will do a sequential scan of the `laptop` table, which it specified with the text `Seq Scan on laptop`. Why this sequential scan? Looking at Listing 9 we see that we have requested a `FULL JOIN` with the `laptop` table on line six. That means the node defined on line 13 in Listing 10 is the start of the right side of that `FULL JOIN`. This sequential scan, and also all other nodes, has a certain `cost`, a number of `rows`, and a `width`. The `cost` is a measurement of execution time, but the unit is a bit odd. It's actually the time required to fetch one of a sequence of pages from disc. It's meaningful to compare costs, to see where more time is spent, but not to translate a cost to an execution time. The cost is specified as an interval, on line 13 it's `0.00..4.00`. The first value is the estimated time before output starts, and the last value is the time when the node has completed. The next thing that's specified is `rows`, which is the number of rows emitted by this node. It's not the number of rows handled, since the number of output rows might be limited, by for example a `WHERE` clause. The last thing that's specified is `width`, which is the number of bytes occupied by each output row.

Phew! That was just the first node, but now we at least have an idea of the structure of the query planners output. To continue, we move one level up in the plan tree, to the arrow one step to the left, above line 13 which we just analyzed. We find that arrow on line 12, which says it will create a hash table of the result of the sequential scan it performed on line 13. Note that first `cost` value includes the cost of child nodes, that's why it's 4 here, since the cost of completing the child node on line 13 was 4. We can also see that it's a very fast operation to store the 200 rows in the hash table. Actually, the time consumed is negligible, since the cost of completing this node is also 4.

Continuing up the plan tree we find the next node, indicated by the next arrow upwards and left, on line 8. There it's specified that there will be a hash full join, which will be completed after 13.25 time units. A hash join means that the database table which hasn't been handled yet will be read sequentially (as specified on line 10), and for each row the previously created hash table will be checked for a matching row to join. In this case, the table not accessed yet is `phone`, as specified in the `FULL JOIN` on line 6 in Listing 9, and on line 10 in Listing 10. As was concluded above, the data that has been stored in a hash table is the rows of the `laptop` table. The join condition is that `laptop.id` shall be equal to `phone.id`, as specified on line 6 in Listing 9, and on line 9 in Listing 10.

Before proceeding up the plan tree, note that the arrows on lines 10 and 12 have the same indentation. This means they are sibling nodes, on the same level in the plan tree, and that both must be executed before the parent node can execute.

Now we've again reached a level in the plan tree where there's a sibling node. This is revealed by the presence of another arrow, on line 15, with the same indentation as the one we're currently looking at, on line 8. Looking at the node on line 15, we see that it's preceded by another node, on line 16. That the node on line 16 precedes the node on line 15 can be seen from the fact that it's arrow is below and to the right of the arrow on line 15. The node on line 16 is another sequential scan, just like the one on line 10. The `materialize` node on line 15 means the the data read by the sequential scan node below, will be stored in a list in memory.

Now it's finally time for the root node on line 3, which handles the `LEFT JOIN` on line 7 of Listing 9. As specified on line 3 in Listing 10, it does a `nested loop` join. As the name implies, this is quite time consuming, which can also be seen by the fact that this nested loop join increases the cost from 6.50 to 492.81. This is by far the most costly operation in the entire plan tree. What actually happens is that each row in the joined phone/laptop set is compared to each row in the employee set. The reason that a nested loop join is required is that the join condition, `laptop.employee_id = employee.employee_id OR phone.employee_id = employee.employee_id`, because of the `OR` can't be decided by just looking for the existence of something.

At last, all that's left in the plan tree are the filters on lines 4 and 6. Line 4 is the join condition of `laptop` and `employee` mentioned above, and line 6 is the `WHERE` clause, `employee.department_id = 2 or laptop.department_id = 2`.

7 Be Aware of Correlated Subqueries

While subqueries are very useful, as was illustrated in Section 4, there are also situations where they can significantly slow down a query. This happens if we write a *correlated subquery*, which is a subquery depending on the outer query, by using a value returned by that outer query. In this case, it's not sufficient to execute the subquery just once, but it must instead be executed once for each row returned by the outer query. This leads to a nested loop, which may make the query unnecessarily slow.

A correlated subquery can appear in either the `WHERE` clause or the `SELECT` clause. Let's start with an example where it's placed in the `WHERE` clause, Listing 11. Here, the subquery on lines 2-3 selects employees who has the employee of the outer query, `e.name`, as manager. The name of an employee is included in the result if the subquery finds any such employee, which means only names of managers are listed. This subquery is correlated since `e.employee_id`, on line 3, is different for each row in the outer query, forcing the subquery to be evaluated once for each row in the outer query.

```
1 SELECT name FROM employee e
2 WHERE EXISTS (SELECT * FROM employee d
3               WHERE d.managers_employee_id = e.employee_id);
```

Listing 11: A correlated subquery in the `WHERE` clause. This query lists the names of all employees who are managers.

Correlated subqueries can often be avoided by using `JOIN` instead of a subquery, which is in fact the case with the query in Listing 11. That query can be rewritten as in Listing 12. Running `EXPLAIN` on the queries in Listings 11 and 12 shows that the query without a correlated subquery is between two and three times faster, even though the query planner managed to avoid using a nested loop for the subquery. Note, however, that the result of `EXPLAIN` depends on the architecture of the computer where it's executed.

```
SELECT DISTINCT mgr.name FROM employee mgr
INNER JOIN employee emp
    ON emp.managers_employee_id = mgr.employee_id;
```

Listing 12: This query gives the same result as the query in Listing 11, without using a correlated subquery.

Next, it's time for an example of a correlated subquery in the `SELECT` clause, see Listing 13. Here, the correlated subquery counts how many employees have the same `department_id` as the employee in the outer query. Since that number depends on which department the employee in the outer query is working in, the subquery must be evaluated for each row in the outer query.

```
SELECT emp.employee_id, (
    SELECT COUNT(dep.department_id) AS emps_in_dep
    FROM employee dep
    WHERE dep.department_id = emp.department_id
    GROUP BY department_id)
FROM employee emp;
```

Listing 13: A correlated subquery in the `SELECT` clause. This query lists all employees, and for each employee shows how many are working in the same department as that employee

Running EXPLAIN on the query in Listing 13 gives the result of Listing 14. That result shows that the subquery will be executed in a nested loop, which is revealed by the text SubPlan on line 4. It means that the subplan will be executed for all rows returned by the node outside the subplan, that is the sequential scan on line 3.

```
1                              QUERY PLAN
2 -------------------------------------------------------------
3  Seq Scan on employee e  (cost=0.00..264.94 rows=106 width=12)
4    SubPlan 1
5      ->  GroupAggregate  (cost=0.00..2.48 rows=5 width=12)
6            Group Key: d.department_id
7            ->  Seq Scan on employee d  (cost=0.00..2.33
8                                         rows=21 width=4)
9                  Filter: (department_id = e.department_id)
```

Listing 14: The result of running EXPLAIN on the query in Listing 13. The output has been reformatted to fit page width.

Also the query in Listing 13 can be changed to use a JOIN instead of a correlated subquery, as is done in Listing 15. This time, when there was a nested loop, EXPLAIN tells that the JOIN version is about 24 times faster than using a correlated subquery.

```sql
SELECT emp.employee_id, dep.emps_in_dep FROM employee emp
INNER JOIN (
    SELECT count(department_id) AS emps_in_dep, department_id
    FROM employee GROUP BY department_id) dep
ON dep.department_id = emp.department_id;
```

Listing 15: This query gives the same result as the query in Listing 13, without using a correlated subquery.

Before leaving this section it must be emphasized that subqueries really are useful, and shall most certainly be used. The purpose here is only to show how to write efficient queries. In fact, even correlated subqueries might very well be useful. They can't always be replaced by joins or other constructs, sometimes we have to use them. What matters most is that we're aware of the problems they might cause.

8 A Comparison of Join, Subqueries and Union

There are many ways to combine SELECT statements into a single result set, this section is a comparison of JOIN, subqueries and UNION. First, let's see when a subquery is actually used to combine query results. As discussed above, in Sections 4 and 7, subqueries can be used for different purposes, and be placed in both a SELECT, FROM and WHERE clause. When placed in a WHERE clause, it restricts the output of a query by selecting a subset of all rows, it doesn't merge the result of different queries. It's only when placed in a SELECT or FROM clause that a subquery is used to merge query results.

Now it's clear which subqueries to include in the comparison, namely those in SELECT and FROM clauses. Let's first compare those subqueries with JOIN clauses, since those two have more in common with each other than with a UNION clause. Both a JOIN and a subquery serves to include more columns in the result set, and the rest of the SELECT statement can operate on those columns, for example with GROUP BY clauses, aggregate functions and WHERE clauses. In fact, as was seen in Section 7, they can, at least in some cases, both be used for the exact same purpose. This means that to same extent the choice is just a matter of taste, and which option is best known. One way they do differ is that a subquery is a complete SELECT, which means it can do more filtering and processing than a JOIN, before merging its result with the rest of the SELECT statement. On the other hand, a JOIN is the one that most often executes faster, but that varies between database management systems and hardware. Also, query planners are becoming better all the time, making execution time less dependent on how a query is written. Remember that execution time must be evaluated, not assumed, for example by using EXPLAIN. Finally, it's important that queries are as easy to understand as possible, even though that doesn't always point in the same direction regarding the choice between JOIN and subquery.

Now that we have some idea about the similarities and differences of JOIN and subquery, let's move to UNION, which is completely different. As opposed to the other two options, a UNION is performed after all operations on columns are completed in a SELECT. This means it merges rows, not columns, and no operations including columns can be performed after a UNION. That leaves very little, a UNION more or less merges the rows of two complete select statements. The result of this, apart from the obvious difference of adding rows instead of columns, is that a UNION might result in longer and more complex Sql. The reason is that both sets being merged must be complete SELECT statements, which leads to the same code being duplicated in both the merged SELECT statements, if used unwisely. This problem might, at least to some extent, be remedied by replacing SELECT statements with views.

Regarding execution time of JOIN and UNION, the short answer is that a UNION is faster since there's really no kind of merging done. The rows returned by the second query are just appended to the rows of the first query. In reality, this difference is far from always enough to motivate using a UNION, since the query planner is very good at optimizing, and we can also ourselves think about how we write JOIN clauses. Also, as always, execution time must be measured before any optimization is done. Never optimize for time unless it's proven that a problem exists, and that the optimization solves it. As an example of reducing execution time by using UNION, let's look at the query in Listing 4. As was seen in Listing 10, that statement resulted in a nested loop join, and was estimated to take 492.81 time units. It can, however, be rewritten to use a UNION of a query for phones and a query for laptops, instead of the FULL JOIN of phone and laptop on line 4 of Listing 4. This will place phone and laptop ids in the same column, since a UNION adds rows, not columns. However, that's no disaster, we can just manually add a column telling if a device is a phone or laptop. This rewritten query, using UNION, is found in Listing 16. Running EXPLAIN tells that the estimated execution time is now reduced to only 18.74 time units, since no nested loop is needed.

```
SELECT phone.id AS device_id, 'phone' AS device_type,
       CAST(emp.department_id AS TEXT)
FROM phone
LEFT JOIN employee emp ON phone.employee_id = emp.employee_id
WHERE emp.department_id = 2

UNION

SELECT laptop.id AS device_id, 'laptop' AS device_type,
       CONCAT(laptop.department_id, emp.department_id)
           AS department_id
FROM laptop
LEFT JOIN employee emp
    ON laptop.employee_id = emp.employee_id
WHERE emp.department_id = 2 OR laptop.department_id = 2;
```

Listing 16: This query outputs the same data as the query in Listing 4, but executes much faster

This query is notably longer than the one in Listing 4, which is often the downside of using `UNION`. It can, however, be made easier to read if parts of it, for example the two `SELECT` statements, are placed each in its own view, with an explaining name.