Exercise 1 (Summer 2020)

6.0 VU Advanced Database Systems

Information

General

Work through the exercises below and write a report on your answers. Submit the report as a single pdf file (max. 20MB) in TUWEL and register for an exercise interview. **You can only receive points for this exercise if you attend the interview.** We expect you to explain your work in your report, i.e., it is not enough to only write the final answers. Show how you arrived at your answers and, where applicable, discuss your results. **We will not accept any handwritten reports!**

Some parts of this exercise involve working on a PostgreSQL DBMS. For these exercises we will provide you with access to a PostgreSQL server (version 11.7). You can connect via SSH at `bordo.dbai.tuwien.ac.at` and access the server via `psql`. You will receive your account information via TUWEL.

Deadlines

at latest March 23rd	**23:59**	Upload your submission on TUWEL
at latest March 23rd	**23:55**	Register for an exercise interview in TUWEL

Exercise Interviews

In the solution discussion, the correctness of your solution as well as your understanding of the underlying concepts will be assessed.

The scoring of your submission is primarily based on your performance at the solution discussion. Therefore it is possible (in extreme cases) to get 0 points even though the submitted solution was technically correct.

Please be punctual for your solution discussion. Otherwise we cannot guarantee that your full solution can be graded in your assigned time slot. Remember to bring your student id to the solution discussion. It is not possible to score your solution without an id.

Question Sessions

About a week before the submission deadline, we offer question sessions to help you with any problems you have with the exercises. The goal of these sessions is to help you understand the material, not to check your solutions or solve the exercises for you. In this spirit we also ask that you engage with the exercise sheet before coming to the session. **Participation is completely optional.**
Exact times and locations will be announced on TUWEL.

TUWEL Forum

You can use the course forum on TUWEL for clarifying questions regarding the exercise sheets. Please do not post your solutions (even partial) on the forum.

Attribution

The dataset used in the final exercise is based on the stackexchange data dump available here `https://archive.org/details/stackexchange`.

Exercises

Exercise 1 (Disk Access in Databases) *[3 points]* Compare the performance of the following two disks with regard to the tasks below. Discuss the differences in performance according to the tasks. (In the following, we use KB for 10^3 bytes and MB for 10^6 bytes.)

Disk A (Magnetic)
- Block size: 4 KB
- Rotational speed: 7500 rpm
- Average seek time: 9ms
- Transfer rate: 210 MB/s
- Track-to-track seek time: 1ms
- Average track size 252 KB

Disk B (SSD)
- Block size: 1 MB
- Transfer rate: 520 MB/s

Finally, assume that your DBMS is configured to store data in an *unspanned* organization in 50 KB blocks. The blocks of each file are stored *contiguously*.

We consider the files `artist` and `record`, referred to as a and r, respectively. The number of records n and record size R for each file is given as follows:
- $n_a = 30\,000 \qquad R_a = 1\,500$ bytes
- $n_r = 1\,000\,000 \qquad R_r = 8$ bytes

The following two tasks concern two different Postgres query plans for the query

$$\texttt{artist} \bowtie_{artist.id=record.by} \texttt{record}$$

For Disk A and Disk B, calculate the I/O time (assuming that the files are not already in main memory) required in the execution of each of these plans.

You can assume that the intermediate results and hashes are stored in main memory and don't require any additional I/O.

(a) The DBMS has created the query plan shown in Figure 1 which uses only *sequential scans* (Called `Seq Scan` in Postgres query plans).

A sequential scan simply reads the whole file in a sequential manner, possibly applying filters, and collecting the information required by the next step.

```
Hash Join
  Hash Cond: (r.by = a.id)
  ->  Seq Scan on record r
  ->  Hash
        ->  Seq Scan on artist a
```

Figure 1: Query plan A

(b) An index for `records.by` is introduced. The query planner has now decided on an index scan instead of a sequential scan to read the records file (see Figure 2).

An `Index Scan` looks up a value in an index and – if present – reads the respective entry to which the index points. In this particular case, every `artist.id` is looked up in an index for `records.by`. Entries are then only read if the `artist.id` occurs in the index. **Important:** Note you can therefore make no assumptions on the order in which entries are read, i.e., you can not assume that the reads sequential.

```
Nested Loop
->  Seq Scan on artist a
->  Index Scan using record_by_idx on record r
Index Cond: (by = a.id)
```

Figure 2: Query plan B

Assume the index also needs to be read from disk and has a size of 4000 KB. There are 42 rows in the result of the query. How much time is now spent on disk access with Disk A and Disk B in this new plan? Discuss your results and how they compare to part (a) of this exercise.

Exercise 2 (Selectivity) *[3 points]*

(a) Your DBMS saves equi-depth histograms to support selectivity calculation for query planning. In particular, for the column `score` of a table `exams` with 5000 rows, the following 9 values divide the column values into 10 groups of equal size: $5, 15, 20, 40, 59, 64, 76, 85, 91$. Furthermore, the DBMS has stored the maximum value in the column: 100.

Estimate the selectivity for the following two predicates as accurately as possible. Assume that values are evenly distributed inside the buckets.

i) `score < 50`

ii) `score > 87`

(b) Histograms provide little useful information for estimating the selectivity of equalities. Saving the number of different values of an attribute allows for reasonable approximation in those cases. For the column `score` you know that there are 78 different values; estimate the selectivity of the following two constraints.

i) `score = 0`

ii) `score != 64`

(c) Our `exams` relation also has an attribute `courseid`, which is a *foreign key* to `id` attribute of the `course` relation. The `course` relation has 15 rows. Estimate the selectivity of the following two operations:

i) `exams` $\bowtie_{courseid=id}$ `course`

ii) $\pi_{courseid}$(`exams`)

(d) Given the following query and statistics information for the rows and their attributes. Choose a good ordering of joins and selection operations based on the available information. *Don't forget to give reasons for your solution in your report.*

```sql
SELECT * FROM room ro
  JOIN reservation re ON re.room = ro.name
  JOIN course c ON c.coursename = re.coursename
  WHERE (c.coursename = 'ADBS' OR c.ects > 6)
    AND ro.capacity < 300
    AND ro.building != 'Freihaus';
```

Attribute	Histogram	Maximum	Distinct Values
course.coursename	–	–	520
course.ects	$\{2, 3, 5\}$	162	20
course.courseid	–	–	780
room.capacity	$\{25, 120\}$	480	120
room.building	–	–	5
room.name	–	–	1000

Table	Rows
course	780
room	1005
reservation	34 000

(e) Which join strategies would you choose in task (d)? What further information could help to inform the choice of join strategies or improve your selectivity estimations?

Note for exercises 3 and 4: For more consistent results make sure to manually trigger statistics collection in Postgres by using `ANALYZE`[1] after you create tables and indexes.

Exercise 3 (The Query Planner and You) *[4 points]*

In this exercise we will investigate the behavior of the PostgreSQL query planner of a complex *join* as our guiding example. We have four relations R, S, T, U as described below.

```
R(a,b,c)
S(b,c,d)
T(a,c,d)
U(a,b,d)
```

We want to compute the unique results of the query $R \bowtie S \bowtie T \bowtie U$. First, create relations with random test data from the `planner_gen.sql` file that is avaiable in TUWEL.
We will first consider the direct translation of our query to SQL:

```
SELECT distinct(a,b,c,d)
FROM r
NATURAL JOIN s
NATURAL JOIN t
NATURAL JOIN u;
```

(a) Which join strategy is chosen by default?

(b) There are three major types of join strategies that PostgreSQL chooses from. Nested loop joins, sort-merge joins and hash joins. For each of these strategies, configure the query planner in such a way that you get a plan using only one join implementation. Compare the performance of these plans among each other and to the plan of task (a). You can disable join strategies using the commands `set enable_{hashjoin|mergejoin|nestloop}=0;`.

Try to explain the differences in performance. Can you change the creation process (the numbers in the `CREATE` statements) in such a way that the relative performance changes significantly? If so, argue why.

(c) Add indexes to the DB that you believe may be useful for the query. Investigate the effect of your indexes on the query plan and the performance for this query. How did the

[1] `https://www.postgresql.org/docs/11/sql-analyze.html`

plan and the performance change, and how big are your indexes relative to the relation data?

Hint: You can use the following SELECT to check the relation data and index size of relation ***RELNAME****, respectively:*'

```
SELECT pg_size_pretty(pg_relation_size('RELNAME')) as data_size,
       pg_size_pretty(pg_indexes_size('RELNAME')) as index_size;
```

Theory tells us that a good (albeit not optimal) equivalent way to compute our join is to compute the unique tuples of $(U \bowtie T) \ltimes R \ltimes S$ instead[2]. **Try the following steps without indexes first. At the end, see how indexes change the performance in Task (e).**

(d) Reformulate the query to directly expresses the computation as $(U \bowtie T) \ltimes R \ltimes S$. Don't forget to use the *distinct* keyword to only obtain the unique tuples. Remember to reenable all features of the planner, e.g., using the statement `RESET ALL;`. Does the query plan match your expectations? Discuss the performance compared to task (a).

Hint: Semi joins correspond to the IN or EXISTS keywords in SQL.

(e) Investigate the query plan the query from task (d). How are the *semi joins* realized in the query plan?

Find a way to make the query planner decide on a plan that explicitly uses a *Hash Semi Join* for the query from Task (d). Explain how and discuss the performance of this version of the query.

Hint: Observe what primitives are used by the query planner to realize the semi joins. Disable those primitives (again via set enable_*) until the planner is forced to use an actual semi join.

(f) Using your query and settings from task (e), try changing which two relations are joined, e.g., try $(R \bowtie S) \ltimes T \ltimes U$. Is there any difference in performance?

Exercise 4 (Optimizing Queries) *[5 points]*

Investigate the following queries on the database provided to you as `optdb.sql`[3] at the path `/home/student/optdb.sql` on the `bordo` server. Use what you have learned in the course to improve the performance of the following queries as much as possible.

You can take whatever action you like as long as you don't change the results of the query. Make sure your optimized query is as general as the original one. (E.g., don't hardcode values from the database since they might change.) Be sure to include a discussion of your thought process, approaches you've tried that didn't work well and relevant query plans in your report. **Explain why you believe that no further reasonable optimzations exist!**

Hint: It is often hard to see when the query planner makes bad estimations in the default output format. You can use the Postgres EXPLAIN Visualizer[4] *to get a better overview of what steps are slow, costly or badly estimated. You are welcome to use screenshots of the visualized query plans in your report where apt.*

For the sake of comparability **please work on** `bordo.dbai.tuwien.ca.at` and optimize for performance there.

[2] This is only the case because of the concrete structure of the query, i.e., the way in which the relations are connected via their attributes. This is not true in general

[3] You can import it by executing it in psql: `psql -a -f path_to_optdb.sql`. If you import from psql with the `\i` command you will have to exit and restart `psql` before you see the tables properly.

[4] `http://tatiyants.com/pev/#/plans/new`

(a) We start off by a simple but critical query that has been causing lag in your application. Your colleagues tell you that fixing the problem requires the query to take at most 2ms.

```
SELECT count(*) FROM users
  WHERE substring(displayname from '%#"[mn][eo]{2,}#"%' for '#') IS NOT NULL;
```

(b) The following query takes many minutes (or hours) to execute, making it basically useless. Read its query plan using `explain` to figure out why. Find a way around the issue and aim to have the query run in significantly under 100ms.

```
SELECT b.name, b.class FROM badges b
WHERE b.userid IN (SELECT u.id from users u WHERE u.reputation < b.class);
```

(c) The following statement asks for all the userids of users which have received all badges starting with the letter v. It is a common bad practice to state these types of queries via counting constructs as in the query below. This is usually a very ineffective way of expressing this type of query. Find a different way to achieve the same result and, if possible, also find further ways to improve performance.

```
SELECT distinct b.userid FROM badges b
WHERE (SELECT count(distinct b2.name) FROM badges b2
       WHERE b2.name ILIKE 'v%' AND b2.userid=b.userid)
       =
      (SELECT count(distinct b2.name) FROM badges b2 WHERE b2.name ILIKE 'v%');
```

For reference: Our optimized version of the query takes around 50ms to execute. Don't worry if you don't reach this time, the reference time is only intended to help you to evaluate your solution and to motivate you.

(d) Even though nobody understands what it does, the following query is crucial for the software your team is working on. For complex queries it is often best to approach the optimization process step by step. Use `EXPLAIN ANALYZE` to identify slow nodes in the query plan and find ways to speed up execution of the query. Make sure to document how your decisions were motivated in your report.

```
SELECT distinct p.* FROM users u, posthistory h, posts p, badges b
WHERE u.id = h.userid
AND h.userid = u.id AND h.userdisplayname = u.displayname
AND p.id = h.postid
AND b.userid = u.id
AND (b.name SIMILAR TO 'Autobiographer' OR b.name SIMILAR TO 'Teacher')
AND (SELECT AVG(u2.upvotes)
     FROM users u2
     WHERE u2.creationdate > p.creationdate)
  >=
 (SELECT COUNT(*) FROM votes v
  WHERE v.votetypeid = 3 AND v.postid = h.postid);
```

For reference: Our optimized version of the query takes around 75ms. Don't worry if you don't reach this time, the reference time is only intended to help you to evaluate your solution and to motivate you.