Exercise 1 (Summer 2019)

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 9.6). You can connect via SSH at `bordo.dbai.tuwien.ac.at` and access the server via `psql`. You will receive your account information via email.

Deadlines

at latest April 8th	**12:00**	Upload your submission on TUWEL
at latest April 9th	**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. **Every group member has to be able to explain all parts of your submission.**

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) *[1 points]*

Consider a magnetic disk with the following characteristics:

- Block size: 8 kB
- Rotational speed: 10000 rpm
- Average seek time: 4ms
- Transfer rate: 130 MB/s
- Track-to-track seek time: 0.2ms
- Average track size 512kB

(a) Calculate the *average rotational delay* and the *transfer time* for a single block. What is the average time to locate and transfer a block?

(b) What is the average time it would take to transfer 20 random blocks. Compare it with the time it would take to transfer 20 consecutive blocks.

Now consider a file `students` with 40000 fixed-length records. Each record has the following fields `name` (120 bytes), `birthdate` (8 bytes), `mnr` (12 bytes), `program` (4 bytes). An additional byte is used as a deletion marker. The file is stored on the disk described above.

(c) Calculate the record size in bytes.

(d) Calculate the *blocking factor* and the number of blocks used by the file, assuming an *unspanned* organization. Furthermore, calculate the wasted space in each disk block because of the unspanned organization.

(e) Calculate the average time needed to search for an arbitrary record, using linear search, if the file blocks are stored on consecutive disk blocks. Compare it with the case where the file blocks are not stored contiguously.

(f) Assume that the records are ordered via some key field. Calculate the average number of block access and the average time needed to search for an arbitrary unique record in the file using binary search. You can assume any

Exercise 2 (I/O in Query Plans) *[2 points]*

Consider the files `works_on`, `project` and `employee`, referred to as w, p, e respectively. The number of records n and record size R (in bytes) for each file is given as follows: $n_w = 55000, n_p = 2000, n_e = 125000, R_w = 14, R_p = 120, R_e = 65$.

(a) Assume your DBMS is backed by a SSD with 340 MB/s transfer rate that writes/reads in blocks of 64kB. Your DBMS organizes its data in blocks of 16kB with *unspanned organization*. For each file, its blocks are *contiguous*. The DBMS has created the query plan shown in Figure 1 which relies on *sequential scans* (Called `Seq Scan` in Postgres query plans).

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

How much time is spent on disk I/O in the execution of this query in the worst case? (You can assume that the intermediate results and hashes are stored in main memory and don't require any additional I/O.)

```
Hash Join
   Hash Cond: (w.ssn = e.essn)
   ->  Hash Join
       Hash Cond: (w.pno = p.pnumber)
       ->  Seq Scan on works_on w
       ->  Hash
           ->  Seq Scan on project p
               Filter: ((pname)::text = 'Aquarius'::text)
   ->  Hash
       ->  Seq Scan on employee e
           Filter: ((bdate)::text > '1957-12-31'::text)
```

Figure 1: Query plan A

(b) A hash index for `project` is introduced. The query planner has now decided on a bitmap index scan instead of a sequential scan to read project (see Figure 2).

A `Bitmap Index Scan` returns a bitmap of potential tuple locations by scanning the index; it does not access the table itself. The bitmap is used by an ancestor `Bitmap Heap Scan` node that reads and checks the returned potential locations.

```
Hash Join
  Hash Cond: (w.ssn = e.essn)
  ->  Hash Join
      Hash Cond: (w.pno = p.pnumber)
      ->  Seq Scan on works_on w
      ->  Hash
          ->  Bitmap Heap Scan on project p
              Recheck Cond: ((pname)::text = 'Aquarius'::text)
              ->  Bitmap Index Scan on project_pname_idx
                  Index Cond: ((pname)::text = 'Aquarius'::text)
  ->  Hash
      ->  Seq Scan on employee e
          Filter: ((bdate)::text > '1957-12-31'::text)
```

Figure 2: Query plan B

Assume that there are 3 projects with name `Aquarius` and that the bitmap index returns exactly the locations of those 3 tuples. Assume that the index has to be read from disk where it uses 18kB of contiguous space. (You can again assume that the intermediate results and hashes are stored in main memory and don't require any additional I/O.)

How much time is now spent on disk access in this new query? Discuss your results and how they compare to part (a) of this exercise. Can you think of cases where bitmap scans are more beneficial?

Exercise 3 (Selectivity) *[3 points]*

(a) Your DBMS saves equi-depth histograms to support selectivity calculation for query planning. In particular, for the column `votes` of a table `movie_reviews` with 4000 rows, the following 7 values divide the column values into 8 groups of equal size: $4, 12, 16, 23, 36, 62, 133$. Furthermore, the DBMS has stored the maximum value in the column: 175.

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

i) `votes < 20`

ii) `votes > 140`

(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 `votes` you know that there are 180 different values; estimate the selectivity of the following two constraints.

i) `votes = 5`

ii) `votes != 9`

Consider how a DBMS can keep track of this number. What trade-offs are introduced?

(c) To approximate the selectivity of equality constraints even better, your DBMS additionally saves some further information on the most common values and their frequency. The list of values with frequencies for `votes`: $(5; 11\%), (9; 8\%), (15; 8\%), (26; 3\%)$. Furthermore, the histogram of task (a) is available for selectivity estimations. For this task, assume that the most frequent values listed above $(5, 9, 15, 26)$ were excluded for the calculation of the histogram.

Propose a reasonable selectivity of the following combined predicates. Argue your proposed methods for combining the selectivity over AND/OR.

i) `votes != 9 AND votes < 30`

ii) `votes = 15 OR votes > 50`

(d) Given the following query and statistics information (as specified in the previous tasks) 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.courseid = re.course
  WHERE (c.coursename = 'ADBS' OR c.ects > 6)
    AND ro.capacity < 300
    AND ro.building != 'Freihaus';
```

Attribute	Histogram	Maximum	Distinct Values
course.coursename	–	–	490
course.ects	$\{2, 3, 5, 8\}$	162	16
course.courseid	–	–	540
room.capacity	$\{18, 64, 95, 120\}$	480	120
room.building	–	–	11
room.name	–	–	1700

Table	Rows
course	540
room	1700
reservation	12800

(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: This is very similar to how Postgres manages its statistics. Check out the `n_distinct`, `histogram_bounds`, and `most_common_values` values in the `pg_stats` table; see `https://www.postgresql.org/docs/9.6/view-pg-stats.html`.

Note for Exercises 4 and 5: For more consistent results make sure to manually trigger statistics collection in Postgres by using `ANALYZE`[1] after your created your tables.

Exercise 4 (The Query Planner and You) *[4 points]* In this exercise we will investigate the behavior of the PostgreSQL query planner using a so-called *triangle join* as our guiding example. We have three binary relations R, S, T that each share one attribute name with one another and want to compute $R \bowtie S \bowtie T$. First, create the relations with random test data:

```
create table r as select (random()*100)::int as a, (random()*20)::int as b
       from generate_series(1, 10000);
create table s as select (random()*100)::int as b, (random()*20)::int as c
       from generate_series(1, 10000);
create table t as select (random()*10)::int as a, (random()*20)::int as c
       from generate_series(1, 10000);
```

We will first consider the direct translation of our query to SQL:

```
SELECT a,b,c FROM r NATURAL JOIN s NATURAL JOIN t;
```

(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? Argue why.

(c) What effect do indices have on the query plan and the performance for this query?

Theory tells us that a good (albeit not optimal) way to compute a triangle join is to compute it as $(R \bowtie T) \ltimes S$. This statement is equivalent to our original query[2].

(d) Reformulate the query in a way that more directly expresses the computation as $(R \bowtie T) \ltimes S$. The formulation used in tasks (a) and (b) returns some duplicate answers. For your new query it suffices to return the same *set of answers*, i.e., how often a answer is repeated does not matter. (Remember to reenable all features of the planner, e.g., using

[1] `https://www.postgresql.org/docs/9.6/sql-analyze.html`

[2] This is only the case because of the concrete structure of the query, i.e., the way in which R, S and T are connected via their attribute names. This is not true in general

the statement `RESET ALL;`) Does the query plan match your expectations? Discuss the performance compared to task (a).

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

(e) Can you find a way to make the query planner decide on a plan that consists of a hash join followed by a hash semi join for (d)? Explain how and discuss the performance of this version of the query.

(f) Continuing your results from task (e), try changing the relation order, i.e., try $(R \bowtie S) \ltimes T$ and $(S \bowtie T) \ltimes R$. Are there any differences?

Exercise 5 (Optimizing Queries) *[5 points] You have started a new job. Your team has heard of your experience in advanced database systems and has asked you if you could look over some of their old SQL queries. They suspect that changes over time have resulted in some legacy queries with suboptimal performance.*
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. Take care not to change the results of the query and make sure your optimized query is as general as the original one. (E.g., don't hardcode values from the database, you never know when they might change.) Be sure to include a discussion of your thought process, approaches you 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.

(a)
```
SELECT distinct(displayname) FROM users u
  WHERE id IN (SELECT owneruserid FROM posts p WHERE p.viewcount > u.views);
```

(b)
```
SELECT score FROM comments WHERE lower(substring(text for 3)) = 'yes';
```

(c) The following statement queries the ids of all those posts where all its votes have `votetypeid = 2`. It is a rather common bad practice to solve such problems via counting as in the query below. This is 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 postid FROM votes v
WHERE (SELECT COUNT(*) FROM votes v2
                      WHERE v2.postid = v.postid AND v2.votetypeid = 2)
        = (SELECT COUNT(*) FROM votes v2 WHERE v2.postid = v.postid);
```

(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

[3]You can import it by executing it in psql: `psql -a -f path_to_optdb.sql`
[4]`http://tatiyants.com/pev/#/plans/new`

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 p.*, c.*, u.* FROM posts p, comments c, users u, badges b
  WHERE c.postid=p.id
  AND u.id=p.owneruserid
  AND u.upvotes+3 >= (SELECT AVG(upvotes)
                      FROM users
                      WHERE u.creationdate > c.creationdate)
  AND EXISTS (SELECT 1 FROM postlinks l WHERE l.relatedpostid > p.id)
  AND u.id = b.userid
  AND (b.name SIMILAR TO 'Autobiographer' OR
       b.name SIMILAR TO 'Supporter');
```