Exercise Sheet 1, 2020

6.0 VU Advanced Database Systems

March 23, 2020

Aufgabe 1 (Disk Access) *[3 Punkte]*

(a) Seq Scan of a means that the whole file needs to be read. The file size is $n_a * R_a = 45000000B = 45000KB$. Since the DBMS block size is 50 KB, the blocking factor is $bfr_a = \text{DBMS}/R_a \approx 33$. Since the files are stored in an unspanned way, the amount of blocks to be read is thus $n_a/bfr_a \approx 910$. Using the same calculation for the Seq Scan of r, we have $n_a/\text{DBMS}/R_r = 160$ blocks to be read. For the exercise, I assume that disk blocks have to be read as a whole.

a) Disk A:

i. r: The DBMS blocks are assumed to be stored in a spanned way in $B = 160 * 50/4 = 2000$ blocks. The data to be read is 8000 KB. The access time is $t_s + t_r + t_{tr} + b * t_{t2t}$, where $t_s = 9$, $t_r = (0.5/7500) * 60 * 1000$, $t_{tr} = (8/210) * 1000$ and $t_{t2t} = 1$. b is the amount of tracks to be read $b = 8000/252$. Therefore, Seq Scan on r will be performed in approximately $9 + 4 + 38 + 32 * 1 = 83$ ms.

ii. a: The DBMS blocks are assumed to be stored in a spanned way in $910 * 50/4\text{KB} = 11375$ blocks. The data to be read is 45500 KB. The access time is $t_s + t_r + t_{tr} + b * t_{t2t} = 9 + ((0.5/7500) * 60 * 1000) + ((45.5/210) * 1000) + ((45500/252) * 1) \approx 410$ ms.

b) Disk B:

i. r: The DBMS blocks are assumed to be stored in a spanned way in $B = 160 * 50/1000 = 8$ blocks. The data to be read is 8 MB. Since there is no seek time and rotational delay involved, the access time is simply $t_{tr} = (1/520) * (8/1) \approx 15$ ms.

ii. a: The DBMS blocks are assumed to be stored in a spanned way in $910 * 50/1000\text{KB} \approx 46$ blocks. The data to be read is 46 MB. The access time is $t_{tr} = (1/520) * (46/1) \approx 88$ ms.

(b) The DBMS now uses a primary index. First, the index is loaded, then the Seq Scan on a is executed and finally the Index Scan on r. The index data to be loaded is

4000 KB. We can use the timings of the Seq Scan from above. The data loaded of r is $42 * 8 = 336$ B, which can be stored in one block. Therefore, 50 KB are loaded.

a) Disk A: The data to be loaded for the index is 4000 KB. The access time for the index is $t_s + t_r + t_{tr} + b * t_{t2t} = 9 + ((0.5/7500) * 60 * 1000) + ((4/210) * 1000) + ((4000/252) * 1) \approx 47$ ms. The Index Scan is a random disk access and the data fits in a single block and track. Its corresponding time is $t_s + t_r + t_{tr} + b * t_{t2t} = 9 + ((0.5/7500) * 60 * 1000) + ((.05/210) * 1000) + (0 * 1) \approx 13$.

b) Disk B: The data to be loaded for the index is 1 MB. The access time for the index is $t_{tr} = 1/370000 \approx 2$ ms. The Index Scan also loads 1 MB. The access time is the same.

Part	Disk A	Disk B
a	493	103
b	148	92

It can be clearly seen that Disk B is faster than Disk A. The only suboptimal part of Disk B is the high block size. Also, unsurprisingly, the execution plan using the Index Scan is faster than the naive execution plan. The difference for Disk B is rather small, however.

Aufgabe 2 (Selectivity) *[3 Punkte]*

(a) The DBMS uses equi-depth histograms with ten buckets and a max value of 100. The min value is smaller than 5. Then we have for each bucket $B = 5000/10 = 500$ rows.

a) Due to the equi-depth histogram and since `score` $\in [0, 100]$, we can deduce that 50 lies in the $(40, 59]$, i.e. the 5th interval. Since we can assume uniformity inside the buckets, we calculate that $4 * 500 + 500 * (50 - 40)/(59 - 40) \approx 2263$ scores lie below 50. Thus, $sel_{\texttt{score}<50} = 2263/5000 \approx .453$.

b) In the same way, we get, $sel_{\texttt{score}>87} = (1 * 500 + 500 * (91 - 87)/(91 - 85))/5000 = .167$.

(b) Again, we assume uniformity inside the buckets.

a) The selectivity for equality is $sel_{\texttt{score}=0} = sel_{\texttt{score}=k} = 1/78 \approx .012$.

b) The selectivity for inequality is $sel_{\texttt{score!}=62} = 1 - sel_{\texttt{score}=62} = 1 - 1/78 \approx .987$

(c) a) Since we can assume that `id` attribute is the primary key and thus has 15 unique values, $sel_{\bowtie_{\texttt{exams.courseid=course.id}}} = 1/15 \approx .067$.

b) In SQL, all tuples would be returned for this operation. But the relational algebra operation operates with sets, therefore theoretically the selectivity would be $sel_{\bowtie_{\texttt{courseid}}} = sel_{\bowtie_{\texttt{course.id}}} 1/15 = 0.67$.

(d) I will assume that the DBMS further uses equi-depth histograms. We try to execute operations with low selectivity first, as less tuples will be affected subsequently. We prioritize selections over joins. The `course` table has the fewest rows. The `reservation` table has the most rows, but appears in both joins anyways.

a) $\sigma_{\texttt{building}=\text{Freihaus}}\texttt{room}$, since $sel_{\texttt{building}=\text{Freihaus}} = 1/5 = .2$

b) $\sigma_{\texttt{capacity}<300}\texttt{room}$, since $sel_{\texttt{capacity}<300} = 1 - sel_{\texttt{capacity}>300} = 1 - (1005/4) * (480 - 300)/(480 - 120)/1005 \approx .875$, as `room` stores `capacity` in 4 buckets approximately 251 rows each.

c) $\sigma_{\texttt{coursename}=\text{ADBS}\vee\texttt{ects}>6}\texttt{course}$, since $sel_{\texttt{coursename}=\text{ADBS}\vee\texttt{ects}>6} = \min(sel_{\texttt{coursename}=\text{ADBS}} + sel_{\texttt{ects}>6}, 1) = \min(1/520+(162-6)/(162-5), 1) \approx .995$. I choose $\min(x+y, 1)$ here as it can not be ascertained how the attributes `coursename` and `ects` interact. The `coursename` attribute is not unique. See the note about the `ects` distribution below.

d) `room` $\bowtie_{\texttt{name}=\texttt{room}}$ `reservation`, since $sel_{\bowtie_{\texttt{room.name=reservation.room}}} = 1/1000 = 0.001$. The `name` attribute is not unique.

e) `reservation` $\bowtie_{\texttt{coursename}=\texttt{name}}$ `course`, since $sel_{\bowtie_{\texttt{course.name=reservation.coursename}}} = 1/520 \approx 0.002$

(e) Since there is no information about indexes, hash tables or ordering and since none of the join attributes are unique, I would choose a Hash Join for both joins.

Further information that might improve selectivity estimates is the `course.ects` distribution. One might strongly suspect that `course.ects` is not uniformly distributed, so more information on that attribute (histogram information) would help. Also, for the `OR`, it would be interesting to have interaction statistics for the `coursename` and `ects` attributes in order to improve the estimate.

Aufgabe 3 (The Query Planner and You) *[4 Punkte]*

(a) PostgreSQL chooses Sort-Merge Join by default in this case.

(b) What was illustrative was that it took postgres multiple minutes to execute the `EXPLAIN (ANALYZE)` command for the nested loop join. I had to stop the process after half an hour and used the normal `EXPLAIN` command. I reproduced this problem also locally. What can be assumed, though, is that this method will be the slowest. It is on an order of magnitudes costlier than the other join strategies.

The query using the hash join is slightly faster than with the sort-merge join, but much more costly. In particular, it can be observed that the sort-merge spends less time on SEQ SCAN but more time on SORT. In total, as the cost is nearly a quarter of the hash join, it is clear why postgres chooses it as default join in this casev.

It is surely possible to improve performance by tweaking the numbers in the `CREATE` statement. For the merge join, the SEQ SCAN and SORT operations on the `R` table are way more costly than the ones on the other tables. This is because the `R` table contains 2.5 times more rows than the rest. Changing the second parameter of `generate_series(.)` in this `CREATE` statement would improve performance.

The difference is not as pronounced for the hash join, but less rows would also improve performance, particularly of the top-most `SORT`.

(c) An index on the attributes of `R` and `S` would certainly help to speed up the query. The tables `T` and `U` are joined first, and only on the attributes `a` and `d`. As the values are not unique, a clustering index will be used. In principle, a multi-key index and bitmap index would be fitting. In the end, I decided for a multi-key btree index, as it is supposed to perform a bit better than a bitmap index with logical AND.

I created following indexes:

```
create index rabc on r using btree(a, b, c);
create index sbcd on s using btree(b, c, d);
create index uad on u using btree(a, d);
create index tad on t using btree(a, d);
```

These indexes reduce the cost of the sort-merge join query from approximately 100,000 to 60,000. The indexes for table R and S yield the biggest reductions, as the SEQ SCAN and SORT operations can be reduced to an INDEX ONLY SCAN. The SEQ SCAN and SORT operations are replaced with (less costly) INDEX SCAN and Materialize steps.

Table	Table Size	Index Size
r	21000	15000
s	8656	6148
t	872	456
u	872	456

(d) I implemented the semi join version of the query in SQL:

```
select distinct a, b, c, d from u
natural join t
where (a, b, c) in (select distinct a, b, c from r)
and (b, c, d) in (select distinct b, c, d from s);
```

With this query, the cost is reduced from 100,000 to 22,500. The query plan is changed towards using a hash join for the semi joins and a merge join for the natural join, as could be expected. The execution time is reduced to approximately 3.5 seconds, which is just slightly faster than the hash join execution time above.

Interestingly, indexes are not used in this query plan.

(e) The semi joins are not actually realized as semi joins, but as full hash joins. I wasted way too much time trying to get Hash Semi Join to show up in the query plan. The hint was only a little helpful for this endeavour, as it still implied an $n!$ search space of all enabled parameters.

Turns out, the solution was to rewrite the query entirely using EXISTS:

```
select count(distinct(u.a,u.b,t.c,u.d)) from u
natural join t
where exists (select * from r where r.a = u.a and r.b = u.b and r.c = t.c)
```

`and exists (select * from s where s.b = u.b and s.c = t.c and s.d = u.d);`

The original query plan with a slighlty higher cost of 22900 can be seen in figure 1.

```
Aggregate  (cost=22889.18..22889.19 rows=1 width=8)
  ->  Merge Join  (cost=22189.64..22500.89 rows=155315 width=16)
Merge Cond: ((u.a = t.a) AND (u.d = t.d) AND (r.c = t.c))
    ->  Sort  (cost=20451.87..20501.87 rows=20000 width=28)
Sort Key: u.a, u.d, r.c
      ->  Hash Join  (cost=17910.00..19023.10 rows=20000 width=28)
Hash Cond: ((u.a = r.a) AND (u.b = r.b) AND (s.c = r.c))
        ->  Hash Join  (cost=5082.00..5932.03 rows=20000 width=24)
Hash Cond: ((u.d = s.d) AND (u.b = s.b))
          ->  Seq Scan on u  (cost=0.00..309.00 rows=20000 width=12)
          ->  Hash  (cost=4782.00..4782.00 rows=20000 width=12)
            ->  HashAggregate  (cost=4582.00..4782.00 rows=20000 width=12)
Group Key: s.b, s.c, s.d
              ->  Seq Scan on s  (cost=0.00..3082.00 rows=200000 width=12)
        ->  Hash  (cost=11953.00..11953.00 rows=50000 width=12)
          ->  HashAggregate  (cost=11453.00..11953.00 rows=50000 width=12)
Group Key: r.a, r.b, r.c
            ->  Seq Scan on r  (cost=0.00..7703.00 rows=500000 width=12)
    ->  Sort  (cost=1737.77..1787.77 rows=20000 width=12)
Sort Key: t.a, t.d, t.c
      ->  Seq Scan on t  (cost=0.00..309.00 rows=20000 width=12)
```

Figure 1: EXISTS Semi-Join using Hash Join

Now, when the parameters `hashagg` and `sort` are disabled, the planner produces a semi join, as can be seen in figure 2. I won't claim here that I found this combination analytically, that is for sure. The only thing I can deduce, is that Sort and HashAggregate do exist figure 1 and are gone in figure 2. The cost increased dramatically to approximately 4,000,000.

(f) Interestingly, when joining `R` and `S` first, the cost decreases to approximately 1,500,000. This is a nice decrease, but still way costlier than the non-hash semi join variant. In order to be sure, I also tried first joining `R` and `T`, which results in a cost of 2,400,000. In total, I claim that the largest table should be among the two tables joined naturally.

Finally, to ascertain the performance with the indexes defined above: The Hash Semi Join cost can be reduced to approximately 270,000. This is still 10x larger than the version without semi join, though.

Aufgabe 4 (Query Optimization) *[5 Punkte]*

```
Aggregate  (cost=3942806.59..3942806.60 rows=1 width=8)
 ->  Hash Semi Join  (cost=27063.00..3942418.30 rows=155315 width=16)
Hash Cond: ((u.d = s.d) AND (r.b = s.b) AND (r.c = s.c))
  ->  Hash Semi Join  (cost=19504.00..2647691.54 rows=155315 width=28)
Hash Cond: ((u.a = r.a) AND (u.b = r.b) AND (t.c = r.c))
   ->  Hash Join  (cost=609.00..24171.15 rows=155315 width=24)
Hash Cond: ((u.a = t.a) AND (u.d = t.d))
    ->  Seq Scan on u  (cost=0.00..309.00 rows=20000 width=12)
    ->  Hash  (cost=309.00..309.00 rows=20000 width=12)
     ->  Seq Scan on t  (cost=0.00..309.00 rows=20000 width=12)
   ->  Hash  (cost=7703.00..7703.00 rows=500000 width=12)
    ->  Seq Scan on r  (cost=0.00..7703.00 rows=500000 width=12)
  ->  Hash  (cost=3082.00..3082.00 rows=200000 width=12)
   ->  Seq Scan on s  (cost=0.00..3082.00 rows=200000 width=12)
```

Figure 2: EXISTS Semi-Join using Hash Semi Join

(a) The query uses an SQL RegEx pattern. I build a partial index for that. This index simply with the given query, which I assume to be constant.

```
create index name_mnoe on users (displayname)
where substring(displayname from '%#"[mn][eo]{2,}#"%' for '#') IS NOT NULL;
```

The index takes up 1152 KB of storage, which is just 1/6 of the table size. Using this index, the query plan now uses a Bitmap Index Scan to filter the rows, as can be seen in figure 3.

```
Aggregate  (cost=304.90..304.91 rows=1 width=8)
 ->  Bitmap Heap Scan on users  (cost=5.12..304.63 rows=107 width=0)
  Recheck Cond: ("substring"((displayname)::text, '%#"[mn][eo]{2,}#"%'::text, '#'::text)
 ->  Bitmap Index Scan on username_mneo  (cost=0.00..5.09 rows=107 width=0)
  Index Cond: ("substring"((displayname)::text, '%#"[mn][eo]{2,}#"%'::text, '#'::text) I
```

Figure 3: Query Plan for (a)

This leads to execution time dropping to approximately 0.5 ms.

I also tried changing the pattern matching while using a different index, but the execution time did not notably improve.

(b) The query uses a correlated subquery, which greatly increases the complexity. I eliminate this subquery by simply joining both tables.

```
select b.name, b.class from badges b
join users u on u.id=b.userid
where u.reputation < b.class;
```

The query plan can be seen in figure 4.

```
Hash Join  (cost=1388.54..2554.59 rows=18219 width=15)
Hash Cond: (b.userid = u.id)
Join Filter: (u.reputation < b.class)
->  Seq Scan on badges b  (cost=0.00..1022.56 rows=54656 width=19)
->  Hash  (cost=1077.13..1077.13 rows=24913 width=8)
->  Seq Scan on users u  (cost=0.00..1077.13 rows=24913 width=8)
```

Figure 4: Query Plan for (b)

Query time is thus reduced to approximately 56 ms.

I do not see the possibility of any (reasonable) index improving this.

(c) The query could be restructured mainly by introducing group by and aggregation while still using an uncorrelated subquery.

```
select b.userid from badges b
where b.name ilike 'v%'
group by b.userid
having count(*) = (
select count(distinct(b2.name)) from badges b2
where b2.name ilike 'v%'
);
```

This reduces the query to approximately 140 ms. I then attempted to create an index on the `name` attribute, as the pattern matching should be sped up by that.

```
create index b_name on badges using btree (name text_pattern_ops);
```

I first was perplexed that the planner only used an Index Scan when using LIKE instead of ILIKE. I figured that the ILIKE function might be immutable or otherwise unsuited for indices. Therefore, I restructured the query into an equivalent form using LIKE.

```
select b.userid from badges b
where lower(b.name) like 'v%'
group by b.userid
having count(*) = (
select count(distinct(b2.name)) from badges b2
where lower(b2.name) like 'v%'
);
```

The fitting index for this query is thus:

```
create index b_name on badges using btree (lower(name) text_pattern_ops);
```

The query time could now be further reduced to approximately 10 ms.

```
GroupAggregate  (cost=205.73..206.33 rows=1 width=4)
  Group Key: b.userid
  Filter: (count(*) = $0)
  InitPlan 1 (returns $0)
    ->  Aggregate  (cost=102.53..102.54 rows=1 width=8)
          ->  Bitmap Heap Scan on badges b2  (cost=4.60..102.45 rows=30 width=11)
          Filter: (lower(name) ~~ 'v%'::text)
            ->  Bitmap Index Scan on b_name  (cost=0.00..4.59 rows=30 width=0)
                Index Cond: ((lower(name) ~>=~ 'v'::text) AND (lower(name) ~<~ 'w'::text))
  ->  Sort  (cost=103.19..103.26 rows=30 width=4)
  Sort Key: b.userid
          ->  Bitmap Heap Scan on badges b  (cost=4.60..102.45 rows=30 width=4)
        Filter: (lower(name) ~~ 'v%'::text)
      ->  Bitmap Index Scan on b_name  (cost=0.00..4.59 rows=30 width=0)
      Index Cond: ((lower(name) ~>=~ 'v'::text) AND (lower(name) ~<~ 'w'::text))
```

Figure 5: Query Plan for (b)

(d) The query seems to retrieve the posts of users which have an autobiographer or teacher badge and which on average upvoted more than the post was upvoted after the post was submitted. This is a query that I can easily depict out there in the wild.

First, I attempted to remove the condition `h.userdisplayname = u.displayname` can be removed, as the preceding condition is already carried out on foreign/primary keys. But somehow this resulted in longer execution times, so I kept the condition.

One of the slowest nodes is the Seq Scan of `B`. I try to speed this up by using the `B.name` index defined above. Also, the SIMILAR TO function can simply be replaced with an equality operator.

Next, I created an index on the `creationdate`, which led to a tiny decrease.

An index on `posthistory.userdisplayname` then cut cost by almost a half.

Just to be sure, I also tried adding an index on `users.displayname`, which didn't change anything, so I removed it.

The new query can be seen below:

```
SELECT distinct p.* FROM users u, posthistory h, posts p, badges b
WHERE u.id = h.userid
AND h.userdisplayname = u.displayname
AND p.id = h.postid
AND b.userid = u.id
AND (lower(b.name) = 'autobiographer' OR lower(b.name) = '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);
```

Overall this reduces execution time to 650 ms.

```
SELECT distinct p.* FROM users u, posthistory h, posts p, badges b, votes v
WHERE u.id = h.userid
AND h.userdisplayname = u.displayname
AND p.id = h.postid
AND b.userid = u.id
AND (lower(b.name) = 'autobiographer' OR lower(b.name) = 'Teacher')
AND v.postid = p.id
group by p.id
having COUNT(*) <= (SELECT AVG(u2.upvotes)
FROM users u2
WHERE u2.creationdate > p.creationdate);
```

I then replaced the subselection with a join and a group by and ended up with a reduced execution time of approximately 250 ms. This seems to also be the result of the elimination of the nested loop. The final query plan can be seen in figure 6.

```
 Unique  (cost=11134.54..11134.70 rows=3 width=1057)
->  Sort  (cost=11134.54..11134.55 rows=3 width=1057)
Sort Key: p.id, p.posttypeid, p.acceptedanswerid, p.parentid, p.creationdate, p.score, p.v
wcount, p.body, p.owneruserid, p.ownerdisplayname, p.lasteditoruserid, p.lasteditordisplay
teditdate, p.lastactivitydate, p.title, p.tags, p.answercount, p.commentcount, p.favoritec
seddate, p.communityowneddate
->  GroupAggregate  (cost=6747.74..11134.52 rows=3 width=1057)
Group Key: p.id
Filter: ((count(*))::numeric <= (SubPlan 1))
->  Sort  (cost=6747.74..6747.76 rows=8 width=1057)
Sort Key: p.id
->  Gather  (cost=3764.15..6747.62 rows=8 width=1057)
Workers Planned: 1
->  Hash Join  (cost=2764.15..5746.82 rows=5 width=1057)
Hash Cond: (v.postid = p.id)
->  Parallel Seq Scan on votes v  (cost=0.00..2529.72 rows=120772 width=4)
->  Hash  (cost=2764.14..2764.14 rows=1 width=1061)
->  Nested Loop  (cost=1401.91..2764.14 rows=1 width=1061)
->  Nested Loop  (cost=1401.62..2763.64 rows=1 width=4)
->  Hash Join  (cost=1401.33..1910.97 rows=525 width=18)
Hash Cond: (b.userid = u.id)
->  Bitmap Heap Scan on badges b  (cost=12.79..521.05 rows=525 width=4)
Recheck Cond: ((lower(name) = 'autobiographer'::text) OR (lower(name) = 'Teacher'::text))
->  BitmapOr  (cost=12.79..12.79 rows=526 width=0)
->  Bitmap Index Scan on b_name  (cost=0.00..6.26 rows=263 width=0)
Index Cond: (lower(name) = 'autobiographer'::text)
->  Bitmap Index Scan on b_name  (cost=0.00..6.26 rows=263 width=0)
Index Cond: (lower(name) = 'Teacher'::text)
->  Hash  (cost=1077.13..1077.13 rows=24913 width=14)
->  Seq Scan on users u  (cost=0.00..1077.13 rows=24913 width=14)
->  Index Scan using posthistory_userdisplayname_idx on posthistory h  (cost=0.29..1.61 ro
Filter: (u.id = userid)
->  Index Scan using posts_pkey on posts p  (cost=0.29..0.50 rows=1 width=1057)
Index Cond: (id = h.postid)
SubPlan 1
->  Aggregate  (cost=548.31..548.32 rows=1 width=32)
->  Index Scan using users_creationdate_idx on users u2  (cost=0.29..527.55 rows=8304 widt
Index Cond: (creationdate > p.creationdate)
```

Figure 6: Query Plan for (b)