# (67506) Databases – Spring 2022 – Exercise (3)

**Muaz Abdeen 300575297**

**Mohammad Ghanayem 208653220**

## *Question (2): Indexing*

***A*** := authors (name, conference, year, institution, count, adjustedcount)

***C*** := conferences (conference, area, subarea)

***I*** := institutions (institution, region, country)

**1.**

(a)

```
SELECT DISTINCT year
FROM authors
WHERE institution='Hebrew University of Jerusalem';
```

(b)

```
                                    QUERY PLAN
-------------------------------------------------------------------------------
HashAggregate  (cost=3904.75..3905.28 rows=53 width=4)
               (actual time=19.875..19.905 rows=43 loops=1)
     Group Key: year
     ->  Seq Scan on authors  (cost=0.00..3902.30 rows=978 width=4)
                              (actual time=0.428..19.060 rows=929 loops=1)
    Filter: ((institution)::text = 'Hebrew University of Jerusalem'::text)
    Rows Removed by Filter: 163895
 Planning Time: 0.100 ms
 Execution Time: 19.964 ms
(7 rows)
```

Here the planner has decided to use a two-step plan: the child plan node scans the entire table sequentially, this is the meaning of `Seq Scan`, and then the upper plan node `HashAggregate` groups the records into a temporary hash table using the attribute `year` as a `Group Key`.

The grouping cost estimate `(cost=3904.75..3905.28 rows=53 width=4)` means that Postgres expects that the grouping will start at cost `3904.75` and finishes at `3905.28`, the difference is the estimated cost in an arbitrary unit of computation to perform this operation. `rows` is the estimated number of rows this `HashAggregate` will return, and `width` is the estimated size in bytes of the returned

rows. The actual estimate `(actual time=0.071..0.074 rows=4 loops=1)` tell us actual time involved in execution. Here we have a new value `loop` which says the entire table was scanned one time.

The `Seq Scan` estimated cost `(cost=0.00..3902.30 rows=978 width=4)` is `3902.30` units of computation, starting at `0.00`. where the actual cost was `(actual time=0.428..19.060 rows=929 loops=1)`. The `WHERE` clause has been applied as a "`Filter`" condition attached to the `Seq Scan` plan node. This means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition.

(c) `CREATE INDEX ON authors(institution);`

(d)

```
                                    QUERY PLAN
 ------------------------------------------------------------------------------------
HashAggregate  (cost=1636.41..1636.94 rows=53 width=4)
               (actual time=1.694..1.728 rows=43 loops=1)
      Group Key: year
      ->  Bitmap Heap Scan on authors  (cost=32.00..1633.96 rows=978 width=4)
                                (actual time=0.107..0.882 rows=929 loops=1)
            Recheck Cond: ((institution)::text = 'Hebrew University of
                           Jerusalem'::text)
            Heap Blocks: exact=53
            ->  Bitmap Index Scan on authors_institution_idx
                     (cost=0.00..31.76 rows=978 width=0)
                     (actual time=0.093..0.093 rows=929 loops=1)
                     Index Cond: ((institution)::text = 'Hebrew University of
                                  Jerusalem'::text)
Planning Time: 0.124 ms
Execution Time: 1.799 ms
```

Here the planner has decided to use a three-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the intermediate plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. The most upper plan is the grouping one as before.

Notice that the actual time using our defined index is `1.799 ms`, which is almost tenth of the cost of the query without this index `19.964 ms`.

2.a. (1) Without index we have to scan the entire table sequentially, that is to read all the blocks where the table is stored. Each block contains $\left\lfloor \frac{2000}{180} \right\rfloor = \mathbf{11}$ rows, then the `authors` table takes $\left\lceil \frac{12000}{11} \right\rceil = \mathbf{1091}$ blocks to store.

So, the query cost is: **1091** .

(2) Optimal branching factor is: $d = \left\lfloor \frac{b+s}{p+s} \right\rfloor = \left\lfloor \frac{2000+8}{8+8} \right\rfloor = \mathbf{125}$ .

(3) Step 1: Find first relevant leaf: $\left\lceil \log_{\lceil \frac{d}{2} \rceil} N \right\rceil = \left\lceil \log_{\lceil 63 \rceil} 12{,}000 \right\rceil = \mathbf{3}$ .

Step 2: Read all relevant leaves: since count values uniformly distributed over [1,20] range then there are: $\frac{12{,}000}{20} = \mathbf{600}$ rows corresponding to `count=2`, then there are $\left\lceil \frac{m}{\lceil \frac{d}{2} \rceil - 1} \right\rceil = \left\lceil \frac{600}{62} \right\rceil = \mathbf{10}$ leaves matching these rows.

Total cost $= \mathbf{13}$ .

2.b. (1) As the previous section, we have to traverse the entire tree, read all blocks, the query cost is: **1091** .

(2) The same as the previous section, $d =$ **125** .

(3) In addition to the step 1 and step 2 in the previous section, there is a step 3 here: to access the matching rows, we have to read all the **600** blocks containing the matching rows. The total $= \mathbf{613}$ .

2.c. (1) Scan the entire table: **1091** .

(2) Optimal branching factor is: $d = \left\lfloor \frac{b+s}{p+s} \right\rfloor = \left\lfloor \frac{2000+26}{8+26} \right\rfloor = \mathbf{59}$ .

(3) First, we traverse the tree down to the first matching leaf: $\lceil \log_{\lceil 30 \rceil} 12{,}000 \rceil = \mathbf{3}$ . Second, we traverse all matching leaves, notice the table have at most $\left\lceil \frac{12{,}000}{20 \times 80} \right\rceil = \mathbf{7}$ matching rows (maybe less, depends on how many `year=1999` values are there), which fit in $\left\lceil \frac{7}{29} \right\rceil = \mathbf{1}$ one leaf.

Total cost $= \mathbf{4}$ .

2.d. (1) Optimal branching factor is: $d = \left\lfloor \frac{b+s}{p+s} \right\rfloor = \left\lfloor \frac{2000+22}{8+22} \right\rfloor = \mathbf{67}$ .

(2) We have WHERE with a disjoint condition "or", so we have to scan the tree two times, one for each value, the calculations for both are the same.

Traversing the tree down costs: $\lceil \log_{\lceil 34 \rceil} 12{,}000 \rceil = \mathbf{3}$ , traversing the matching leaves costs: $\left\lceil \frac{150}{33} \right\rceil = \mathbf{5}$ , where $150 = \left\lceil \frac{12{,}000}{80} \right\rceil$ since the conference values uniformly distributed over 80 values. Now, since we are also indexing on name, there is no need to read the block containing the row.

The total cost for both conditions is $= 2 \times (3 + 5) = \mathbf{16}$ .

2.e. (1) Optimal branching factor is: $d = \left\lfloor \frac{b+s}{p+s} \right\rfloor = \left\lfloor \frac{2000+8}{8+8} \right\rfloor = \mathbf{125}$ .

(2) Tree traverse until leaf costs at most $\lceil \log_{\lceil 63 \rceil} 12{,}000 \rceil = \mathbf{3}$ .

There are 19 `count` values to scan, which spans $\frac{12{,}000}{20} \times 19 = 11{,}400$ rows, so the matching rows will be in at most $\left\lceil \frac{11400}{62} \right\rceil = \mathbf{184}$ leaves.

There are approximately 11,400 matching rows, each may be in a different block, but we will go over each block of the table at most once = **1091** .

Total $= 3 + 184 + 1091 = \mathbf{1278}$ .