**DB2**

Forum: https://forum-db.informatik.uni-tuebingen.de/c/ss20-db2

# Assignment 9 (30.06.2020)

Submission: Tuesday, 07.07.2020, 10:00 AM

Relevant videos: up to DB2 - Chapter 11 - Video #60.

https://tinyurl.com/DB2-2020

1. [15 Points] **Performance Impact of Indexes**

   Obviously, indexes have impact on the performance of data modifying operations. But, can an additional index also **decrease** the performance of a SELECT-query?

   Consider the following table:

```
CREATE TABLE skewed (
   sort        INTEGER NOT NULL,
   category    INTEGER NOT NULL,
   interesting BOOLEAN NOT NULL
);

INSERT INTO skewed
   SELECT i AS sort, i % 1000 AS category, i > 50000 AS interesting
   FROM   generate_series(1, 1000000) i;

CREATE INDEX skewed_category ON skewed (category);
ANALYZE skewed;
```

   We now want to query the first twenty interesting rows in category 42:

```
SELECT *
FROM   skewed
WHERE  interesting AND category = 42
ORDER  BY sort
LIMIT  20;
```

   (a) Print the query plan using EXPLAIN ANALYZE and describe in your own words, how query evaluation proceeds and how this is supported by index skewed_category.

   In general, it is a good idea to support top-$N$ queries (ORDER BY/LIMIT) by an index on the sort criteria. Let us test that!

   (b) Create an additional B$^+$Tree index skewed_sort on column sort of table skewed.

   (c) You will notice that index skewed_sort will decrease the performance of our query. **But why?** Again, print the query plan, explain the evaluation strategy in your own words and compare the estimated costs to the plan of 1a. Why is it – in general – a good idea to use the index on sort to support this query? What is the issue that causes the performance drain in the concrete example?

   (d) Can you think of another index which employs the idea of using the sort order, but does not suffer the problem of 1c? Find the index which fits the given query best. Measure and explain its benefit compared to 1a.

2. [7 Points] **UB-Trees: B$^+$Tree on Z-order values**

One possibility to index a table on multiple columns are *composite indexes*. However, these do not support both dimensions equally. Instead, the first dimension dominates the order of entries.

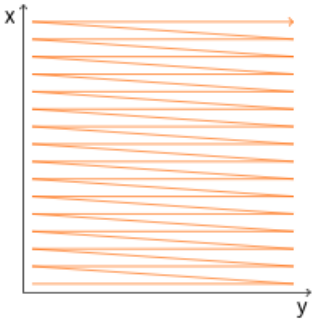
Consider a table `points(x INT, y INT)` representing points in a two-dimensional space:

```
-- A two dimensional space
CREATE TABLE points (x INT, y INT);

-- Populate space with points
INSERT INTO points
  (SELECT x, y
   FROM   generate_series(0,99) AS x,  generate_series(0,99) AS y);

-- Create primary key index
CREATE UNIQUE INDEX points_x_y ON points USING btree (x,y);
```

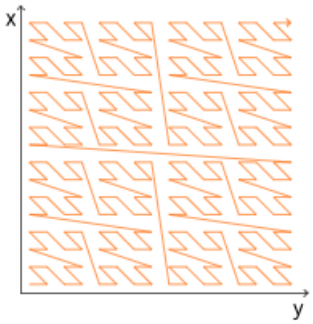
The *composite index* reflects spacial locality of points in a two-dimensional space only on the first dimension `x`. Locality regarding *both* dimensions can be achieved with an *expression index* that combines both dimensions in a single locality-preserving value, called *Z-order value*:

```
CREATE INDEX points_z_value_x_y ON points USING btree (z_order_value(x,y));
ANALYZE;
```

The Z-order value of a point (`x`,`y`) is calculated using the bit-interleaving function provided in `zorder.sql`.[1] This reflects locality for both input dimensions:




$\langle x,y \rangle$ **(concatenated)**




**Z-Order**

For both B$^+$Tree indexes, `points_x_y` and `points_z_value_x_y`:

- Choose an arbitrary leaf page. Uses function `bt_page_stats(indexname, pageno)` provided by extension `pageinspect`[2] to identify a leaf page.
- Collect all points $(x,y)$ referred by this leaf page. Use function `bt_page_items(indexname TEXT, pageno INT)` provided by `pageinspect` to access the *RID*s of all items on the B$^+$Tree page.
- Illustrate the $(x,y)$ location of these points in the two-dimensional space, plotting them on a grid of size $100 \times 100$ with *Gnuplot*.

  You can access a web-based version of Gnuplot at http://gnuplot.respawned.com.[3]
- Compare the two plots: describe and explain your findings **briefly**.

[1] See Wikipedia for more information on Z-order values: https://en.wikipedia.org/wiki/Z-order_curve
[2] https://www.postgresql.org/docs/current/pageinspect.html#id-1.11.7.31.6
[3] Examples for *"Data"* and *"Plot script"* can be found in files `data.txt` and `points.plot`.

3. [8 Points] **External Merge Sort**

Given a input table of unsorted values distributed over 16 pages. Each page covers up to two elements:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pages 1-8: | 38,58 | 23,31 | 36,59 | 21,35 | 19,53 | 13,25 | 56,22 | 54,21 |
| pages 9-16: | 34,58 | 19,14 | 48,46 | 47,32 | 60,40 | 58,60 | 55,24 | 50,54 |

Sort the table using **External Merge Sort**, with an available working memory of `B` = **3 pages**.
For each *Pass* of the algorithm, write down the *output runs* written to secondary memory.

**Note:** *Pass 0* does **not** use *Replacement Sort* and returns sorted runs of size `B` pages.