

DB2

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

Assignment 3 (12.05.2020)

Submission: Tuesday, 19.05.2020, 10:00 AM



Relevant videos: up to DB2 - Chapter 03 - Video #14.

<https://tinyurl.com/DB2-2020>

Please note important changes regarding the grading in DB2:

<https://forum-db.informatik.uni-tuebingen.de/t/neuer-modus-zur-notenfindung-in-der-vorlesung-db2/7758>

1. [15 Points] Decomposition and MonetDB

You are given the following table in `decompositon.sql`:

t				
id	a	b	c	d
1	6	'a'	3	'f'
2	7	'b'	4	'g'
3	8	'c'	3	'f'
4	9	'd'	4	'g'
5	0	'e'	3	'f'

Load the table into your MonetDB database by loading `decomposition.sql` using

```
mclient -l sql <dbname> <path/to/decomposition.sql>
```

- (a) Split table `t` into four tables where each table holds one of the columns of `t` and the `id`. Write down the result. The *decomposed* tables should look as follows:

ta	
id	a
...	...

tb	
id	b
...	...

tc	
id	c
...	...

td	
id	d
...	...

- (b) Find any **non-trivial functional dependencies** in table `t` and write them down.
- (c) Remove any unneeded rows in tables `ta`, ..., `td`. A row is unneeded if it can be removed and we are still able to restore table `t` using a SQL query. Provide the shortened tables `ta`, ..., `td`.
- (d) Formulate a MonetDB SQL query over any instances of shortened tables `ta`, ..., `td` whose result is equal to `t`.
- (e) Explain why MonetDB does **not** apply this particular space-saving scheme to decompose a wide table. Briefly compare the scheme to MonetDB's method of reassembling wide tables from BATs.

2. [15 Points] Page Layout in *PostgreSQL*

Load `documents.sql` to create a table `documents(title CHAR(4), doc TEXT)`. Rows in `documents` are – due to type `TEXT` of column `doc` – of variable length and hence can grow and exceed the free page space on `UPDATE`. How does *PostgreSQL* handle that?

Use the *PostgreSQL* extension `pageinspect`¹ to observe *PostgreSQL*'s behavior. **Hand in all queries you used** and describe your findings **briefly**. Proceed as follows:

- (a) Make use of function

```
heap_page_item_attrs(get_raw_page('documents', <page>), 'documents')
```

to inspect the organization of rows on all pages of table `documents`. Next to each row's header information (`lp: slotno`, `lp_off: row pointer`, `lp_len: row size`, `t_ctid: row version-chain pointer`²), the function extracts the raw data of all attributes in an array `t_attrs` of type `bytea[]`. Use function `convert(str BYTEA)→TEXT` provided in `documents.sql` to convert this attribute data to `TEXT`.

- (b) Find out the *RID* (`page, slotno`) and the *row size* of the row containing 'doc1' as well as the free space left on its page.
- (c) Perform an `UPDATE` on 'doc1' doubling the size of its `doc` column, thus exceeding the free page space. Does the *RID* still point to the same row? How are the pages and the physical location of the row data reorganized?
- (d) Perform an `UPDATE` on 'doc2' growing its *row size* to more than 8 kB. The new row cannot fit into any page – even an empty one. How does *PostgreSQL* cope with that?
- How does the *row size* of the new row compare to the size of the inserted `doc`-value?
 - Read chapter 66.2.1 “*Out-of-line, on-disk TOAST storage*”³ of the *PostgreSQL* documentation. Explain in your own words, how “*sliced bread*”⁴ relates to *PostgreSQL* in terms of our current problem.
 - Search the system catalog table `pg_class`⁵ to find the `relname` of the TOAST table associated with table `documents`.
 - Query table `pg_toast.<TOAST_relname>` to find out, how many *chunks* (rows of the toast table) have been created to store your new `doc`-value.

¹<https://www.postgresql.org/docs/current/pageinspect.html>

²See Slide 08 of Chapter 5 “Row Updates”

³<https://www.postgresql.org/docs/current/storage-toast.html#STORAGE-TOAST-ONDISK>

⁴<https://en.wikipedia.org/wiki/Toast>

⁵<https://www.postgresql.org/docs/current/catalog-pg-class.html>