Databasesystems 2
Forum: https://forum-db.informatik.uni-tuebingen.de/c/ss18-db2

Assignment 2 (01.05.2018)

Submission: Tuesday, 08.05.2018, 10:00 AM

1. [10 Points] **Free Page Space in *PostgreSQL***

   In *PostgreSQL*, deletion of rows leaves holes of free space scattered all over the pages of a table. To reuse this space, holes can either be filled on insertion of new rows, or the data of all remaining rows can be rearranged in order release some empty pages.

   Both strategies can be observed in *PostgreSQL*:

   - Load `supplier.sql` from folder `/assignments02` to create a table with the following schema:

     ```
     Supplier(s_suppkey    INT,
              s_name       CHAR(25),
              s_address    VARCHAR(40),
              s_nationkey  INT,
              s_phone      CHAR(15),
              s_acctbal    DECIMAL(15,2),
              s_comment    VARCHAR(101))
     ```

   - Use the *page* and *slot* information of `supplier`'s system column `ctid`[1] (Current Tuple Identifier) to answer the following questions. Describe your observations **briefly** and include your queries.

     (a) How many pages does the table's heap file occupy?

     (b) Delete all rows with an *odd* `s_suppkey`. How are the remaining rows distributed afterwards?

     (c) Insert a new row of arbitrary value. Guess where it was placed? Check whether *PostgreSQL* meets your expectations.

     (d) Now, execute the command `VACUUM`[2]. Did anything happen? Insert another row and check.

     (e) What are the effects of `VACUUM FULL` instead? Explain why `VACUUM` is executed by the system periodically while the execution of `VACUUM FULL` is avoided. Under which circumstances can `VACUUM FULL` still be a good choice?

2. [10 Points] **Page Fragmentation in *PostgreSQL***

   PostgreSQL's storage layout is based on fixed-sized pages. In general, this leads to some *fragmentation*: Due to their variable length, records often don't cover the whole available page space. Instead, a small portion of each page may be wasted because it fits none of the existing records.

   To analyze this effect with respect to the *average record length* of a table, we will make use of *PostgreSQL*'s `pageinspect`[3] extension. The command `CREATE EXTENSION pageinspect;` makes some additional functions available that allow us to inspect pages on a low level:

   - `get_raw_page(`*`relname`*`,`*`page`*`)`: Return a *raw copy* of a page. (Page number *page* starts from 0.)
   - `page_header(`*`rawpage`*`)`: Inspect the page header. Reveals information about the `upper` and `lower` byte-offset of the page's free-space area.
   - `heap_page_items(`*`rawpage`*`)`: Inspect all record pointers of the page. Reveals information about each record's length (`lp_len` in byte).

[1] https://www.postgresql.org/docs/current/static/ddl-system-columns.html
[2] http://www.postgresql.org/docs/current/static/sql-vacuum.html
[3] https://www.postgresql.org/docs/current/static/pageinspect.html

Load `fragmentation.sql` from folder `/assignments02` to create two tables (1) `lineitem` and (2) `customer`. Use the functions above to write two *SQL* queries **for each table** which compute

(a) the *average record length* (in byte) **and**

(b) the ratio $\frac{\text{total free page space}}{\text{overall size of all pages}^{4}}$. Exclude the last page from your computation.

3. [10 Points] **From MonetDB to C**
In the lecture, we previously discussed a C program `mmap.c` in which we printed the entire column `a` contained in the table `unary`. This task will require you to write a C function similar to `scan_tail(...)` in `mmap.c`.

You are now given two files `binary.sql` and `print.c`. When `binary.sql` is run in MonetDB, it creates a table `binary(a INT, b DOUBLE)`. Then column `a` is populated with numbers ranging from 1 to 100, while column `b` is populated with the same number divided by two.

(a) Run `binary.sql` in MonetDB to create and populate the table `binary`.

(b) Locate the two `tail`-files associated with columns `a` and `b` of table `binary`. Hand in the `tail`-files with your solution.

(c) Now, using these two `tail`-files, complete the function stub given in `print.c`:

```
void scan_tails(int32_t *tail_a, double *tail_b, off_t record_count)
```

The function `scan_tails` is intended to print each record of the table `binary`. It also has to show the correct `oid`. When running `print.c`, the output should look as follows:

```
[ 0@0, 1, 0.500000 ]
[ 1@0, 2, 1.000000 ]
[ 2@0, 3, 1.500000 ]
...
[ 97@0, 98, 49.000000 ]
[ 98@0, 99, 49.500000 ]
[ 99@0, 100, 50.000000 ]
```

(d) Examine the following two lines in the function body of `main()` in `print.c`:

```
a_count = tail_a_size / sizeof(int32_t);
b_count = tail_b_size / sizeof(double);
```

How do the values of `a_count` and `b_count` relate to each other? Explain briefly.