Heinrich Heine
University Düsseldorf

Institut für Informatik
Datenbanken und Informationssysteme
Prof. Dr. Stefan Conrad
Philipp Grawe
Thomas Germer

Winter semester 20/21
Assignment 6 (Project 2)

Relational Databases and Data Analysis

Rules

- Do not import any libraries besides the standard Python libraries and the libraries which are preinstalled in the Docker image.
- Some exercises have additional restrictions – read the exercise text carefully!
- To be admitted to the exam, you have to achieve a combined 80 % in both projects.
- Some exercises come with test scripts with which you can check your solution. Those tests only serve as a guideline. Passing them does not guarantee full points.
- Each test **must** finish in less than a minute and **must** require less than 4 GB of RAM.
- **Any** amount of copying from anywhere or providing solutions to other students will result in 0 points. You must work on the projects **yourself**!

Project description

In this project, we will take a closer look at the internals of a database system.

Industrial databases such as PostgreSQL are quite complex and the code is therefore hard to follow. Instead, we will look at a very simple database engine implemented in Python which you can find in the file `database.py`.

The database is implemented as a Python `dict`. The keys are the names of the relation and the values are Python `lists`. Those lists contain `dicts`, which represent the tuples in a relation. The keys of those `dicts` are the attribute names. This is slightly easier to work with but not very efficient – a real database would store the attribute names only once per relation. But as this database is education-oriented, we can live with this loss in efficiency.

In the database, you can find the following relations. See `database.py` for an example.

- `time(time_id, year, month, day, timestamp)`
- `location(location_id, state, district, city, latitude, longitude)`
- `product(product_id, name, category, subcategory, price)`
- `sale(sale_id, time.time_id, location.location_id, product.product_id, quantity)`
- `campaign(campaign_id, timestamp_start, timestamp_end)`

Exercise 1 *Query Optimization* 30 Points

(a)

The first step is to implement a reasonably efficient natural join in Python.

A naive implementation might loop over two relations and compare all values where the attribute names match, which would result in a $O(n^2)$ algorithm when working on relations with n tuples.

Instead, you should implement a natural join with $O(n \log n + k)$ or amortized $O(n + k)$ computational complexity where k is the size of the resulting relation. Your implementation **must** pass the test with large relations in less than 1 second.

(b)

The execution time of a query can vary substantially depending on the order in which joins are executed. Your task is to implement the query below such that it runs in less than a second.

```
SELECT
    product.price,
    sale.quantity
FROM
    sale,
    time,
    product,
    location
WHERE
    sale.time_id = time.time_id AND
    sale.product_id = product.product_id AND
    sale.location_id = location.location_id AND
    time.year = 2021 AND
    location.state = 'some_state' AND
    product.category = 'some_category'
```

However, there are some restrictions. You **must** solve this exercise only with the following Python functions. For-loops and other looping methods are not allowed for this exercise (b).

- `inner_join(relation, relation)`
- `natural_join(relation, relation)`
- `where_equal(relation, attribute_name, attribute_value)`
- `rename_attribute(relation, old_attribute, new_attribute)`
- `select_attributes(relation, attributes)`

See `test_database.py` for examples of those functions and see the function `example_query_allowed_but_slow` in `test_ex1_b.py` for acceptable function use.

(c)

Draw a diagramm of your query from the previous exercise as in chapter 4 on page 12 and compute the sizes of the intermediate relations for each operation in the tree.

Make sure that your drawing is legible and unambiguous. If it is not clear which number belongs where, we will guess incorrectly on purpose.

Exercise 2 *B+ Tree* 30 Points

The B+ tree is an associative data structure which is central to almost all database engines. Each tree node (except for the root node) stores between m and $2m$ keys inclusively. This guarantees a lookup, insertion and deletion time of $O(\log_m n)$ where n is the number of values stored in the tree.

The leafs of the B+ tree store the data values and are linked, which allows for efficient range queries.

In `ex2_bp_tree.py`, you will find the function `make_bp_tree`. This function creates a B+ tree from key-value pairs and returns the root node of the tree.

(a)

The B+ tree implementation contains a small mistake. It still "works", but it is technically not a B+ tree. Implement tests to find this error.

- You do not need to use a test framework. A single file (`test_ex2_a.py`) which will run your tests and print a precise error message is sufficient.
- You do not need to fix the error.

(b)

Your task is to implement the function `find_inclusive(self, key1, key2)` of the `BPTreeNode` class in `ex2_bp_tree.py`, which should return all values corresponding to keys between `key1` and `key2` (inclusive).

For example, the following query will print `["value2", "value3", "value3again"]`:

```
key_value_pairs = [
    (1, "value1"), (2, "value2"), (3, "value3"),
    (3, "value3again"), (4, "value4")]
]
root = make_bp_tree(key_value_pairs)
print(list(root.find_inclusive(2, 3)))
```

The algorithmic complexity of your implementation must be $O(\log_m n + k)$ where n is the number of values stored in the tree and k is the number of returned values. The test for the large range query should finish in less than 1 second.

(c)

The following query computes the revenue of all sales during an advertisement campaign.

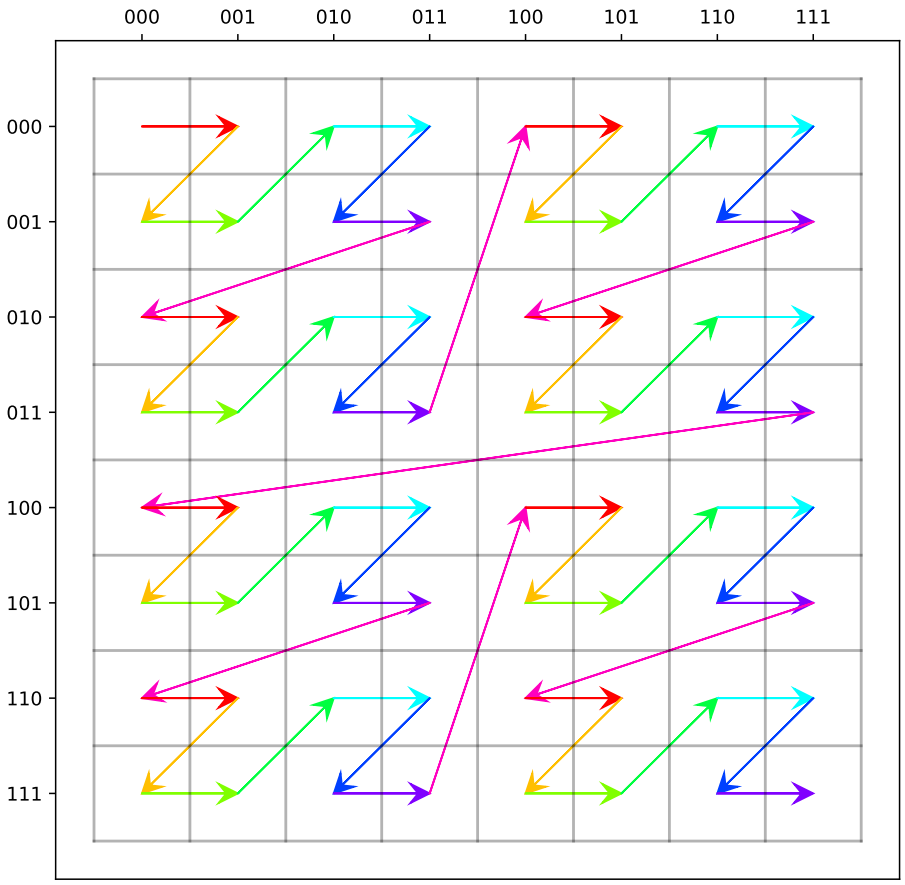
```
SELECT
    SUM(product.price * sale.quantity)
FROM
    sale, time, product
WHERE
    sale.time_id = time.time_id AND
    sale.product_id = product.product_id AND
    EXISTS (
        SELECT * FROM campaign WHERE
            campaign.timestamp_start <= time.timestamp AND
            time.timestamp <= campaign.timestamp_end
    )
```

Implement this query in Python using the B+ tree. No SQL allowed! Your query should take less than 1 second to execute.

Exercise 3 *UB-Tree* 10 Points

(a)

Reproduce the plot from the lecture for Z-order curves.



The setup for the plot is already provided for you. All you need to do is call the function `draw_arrow(x1, y1, x2, y2)` with the correct start- and endpoint coordinates.

(b)

In the file `ex3_b.txt`, explain how you calculated the start- and endpoints of the arrows in (a).

Exercise 4 *Bitmap Index* 30 Points

Bitmap indexes are a great choice for Boolean data or categorical data with small cardinality, like weekdays or months. The individual queries in this exercise should all execute in well under a second.

(a)

Implement a bitmap index for all twelve months to store whether a sale happened in that specific month.

(b)

Implement a function to count all sales that happened between two months inclusively. For this exercise, we will ignore that there are multiple years. An application where that might make sense could be the analysis of seasonal effects. For example, a merchant might be interested in the increase of sales numbers during December due to Christmas.

(c)

To get a more fine-grained data structure, implement a $\langle 31, 12 \rangle$-multi-component bitmap index for each sale to store whether a certain sale happened in a specific month and day. You already implemented half of it (a), so just do the same for each day of a month.

(d)

Implement a function to count the sales between two dates given as month and day values.

There is no need to worry about the range of the months y or days z on page 46 in chapter 5 which use 0-based indexing instead of 1-based indexing, as the value x, which represents the day in a year (if each month had 31 days) is never calculated.

(e)

Combining many bitmap-indexes can be costly. A solution for this problem are multi-component range-coded bitmap indexes. Implement such an index.

(f)

Implement the same query as in (d) using the multi-component range-coded bitmap index.

The algorithmic complexity of this query must be constant with respect to the number of days and months.

Submission

- Zip all files for this project and *then* upload them in the ILIAS.
- Do **not** use any buttons in the ILIAS to zip or extract files since they modify file extensions.
- All tests must run and pass in the environment of the given Dockerfile without further configuration or installations. We will not guess how your scripts were supposed to be executed.
- Jupyter Notebooks, Python code in PDFs, photos/screenshots of code and similar will not be accepted.
- All text and code files must be UTF-8-encoded. Encodings such as ISO-8859-1, Windows-1251, Windows-1252, UTF-16, UTF-32 or other encodings will not be graded.