Project Description
COMP 6908
Fall 2021

Posted: Oct. 16
Due: November 23, 11:59pm.

1. goal

The goal of this project is to Implement a hypothetic database system that consists of the simplified versions for three relational algebra operations, *select*, *project* and *join*, a limited form for B+_tree index, and a few other utility functions. The language is Python. The following is a list of functions you are required to create and a description for their functionalities.

- *select(rel, att, op, val)*: Select tuples from relation *rel* which meet a select condition. The select condition is formed by *att*, *op*, and *val*, where *att* is an attribute in *rel*, *op* is one of the five strings, '<', '<=', '=', '>', '>=', corresponding respectively to the comparison operators, <, ≤, =, >, ≥, and *val* is a value. Returns the name of the resulting relation. The schema for the resulting relation is identical to that for *rel*.
- *project(rel, attList)*: project relation *rel* on attributes in *attList*, which is a list of strings, corresponding to a list of attributes in relation *rel*. Return the name of the resulting relation. The schema for the resulting relation is the set of attributes in *attList*.
- *join(rel1, att1, rel2, att2)*: join two relations *rel1* and *rel2* based on join condition *rel.att1* = *rel2.att2*. Returns the name of the resulting relation, with schema being the union of the schemas for rel1 and rel2, minus either *att1* or *att2*.
- *build(rel, att, od)*: build a B+ tree with an order of *od* on search key *att* of relation *rel*. Returns a reference to the root page of the constructed B+_tree.
- *removeTree(rel, att)*: remove the B+-tree on *rel.att* from the system. Its entry in the directory is deleted and all the pages occupied are returned to the page pool. If the B+_tree does not exist, do nothing.
- *removeTable(rel):* Remove the relation *rel* from the system. Its entry in the catalog is deleted and all the pages occupied are returned to the page pool. If the relation does not exist, do nothing.
- *displayTree(fname)*: display the structure of the B+_tree with root file *fname*. The parameter *fname* is a plain file name under *index* folder. Return the plain file name for the file under *treePic* folder where the tree is displayed. Note that you are not required to plot a B+_tree like the ones plotted in the lecture notes. The looking can be similar to a nested directory hierarchy. Refer to the sample in Section 4.
- *displayTable(rel, fname)*: display the relation instance for *rel* in a file with name *fname*.

2. Simulation setting

In real databases, tuples in relations are implemented as data records, which are stored in pages. Relational algebraic operations are implemented as low level functions operating on pages, like reading and writing pages, access fields in records stored in pages, etc. Similarly, nodes in B+_trees are pages, and accessing nodes implies reading pages. Python is a high level language, and therefore cannot operate on pages. We must do simulations. The simulation includes simulating pages with files, and simulating accesses on pages with operations on files. The following describes these in detail.

- Pages: You simply use text files to simulate pages (files with extension .txt). Records stored in a page are simulated by Python lists saved in a file. For purpose of this project, assume the capacity of each page is two, implying each file can host at most two Python lists.
- Reading and writing pages: In Python, you can use input operations like read(*), readline(*), etc, and output operations like print(*), write(*), etc., to read and write contents in files. You might use these operations to simulate page accesses. The problem is these operations work only for strings. When you read a structure from a file, it will be flattened into a string first and then returned to you. This means you will have to parse it in order to recover its structure. To avoid the hassle, Python provides a mechanism, json, that allows you to read and write structures directly without parsing. It is json you will use for the simulation of page access. (For detail of how to use json, refer to the Python manual.)
- Linkage between pages: In reality, a relation file may contain many pages. These pages are linked together with page pointers. How do you link the .txt files together? A simple strategy is to use a separate file, named *pageLink.txt*, say, to store a sequence of file names for the same relation. This sequence gives the order for the files.
- Schema table: In real database systems, the information about all the relations are kept in a single relation table, commonly named as *schema* (or *catalog*). Each tuple in this relation is of the form:

  (<relation name>, <attribute name>, <type>, <column position>)

  For example, in the database for the sailors club, the following tuple in relation *schema* tells *sid* is an attribute in relation *Sailors*:

  (‘Sailors’, ‘sid’, ‘string’, 0)

  The value for <column position> is 0 since *sid* is the first column (with index 0) in *Sailors* relation. You can use a file, *schema.txt* to simulate *schema* relation.

- B+_tree directory: If there exists a B+_tree on a search key for a relation, the information about this B+_tree must be saved in a relation, named as *directory*. Each tuple in this relation is of the following form:

  (<relation name>, <search key>, <root>)

  where <search key> is an attribute in the relation with the name <relation name>, and <root> is the address of the root page for the B+_tree. You can use a file, *directory.txt*, to simulate *directory* relation, and use the file name for the root file for the address of the root page.
- Page pool: This is a collection of available pages to the database systems. When a new relation or a new B+_tree is created, the system takes pages from the page pool, and when they are removed, the occupied pages are returned to the page pool. You can simulate the page pool with a file, say *pagePool.txt*. It contains a list of page names, like pg0.txt, pg1.txt, etc. These pages will be removed from or re-entered to the list when they are taken or released by the database system.

3. Data set

I will supply the data set for a sample database. This database contains three relations, with the following schemas.

Suppliers (<u>sid</u>: string, sname: string, address: string)
Products (<u>pid</u>: string, pname: string, color: string)
Supply (<u>sid</u>: string, <u>pid</u>: string, cost: float)

The instance for each relation schema is stored over a sequence of *.txt* files under the sub-folder with that relation name, and the schematic information for all of them is stored in file *schemas.txt*, all under the folder *data.* (Refer to the following section for detail.)

4. File organization

The files you need to deal with belong to several different categories, B+_tree files, RA files, data files, schema file, etc. How to organize these files properly is an issue. The simplest way is to place all the files in the same folder. This will make your project unreadable. It also makes debugging hard. A better way is to organize the files based on their categories, as described in the following.

```
<your user name>
    program
        relAlg.py
        buildTree.py
        remove.py
        display.py
        queries.py
    data
        Suppliers
            pageLink.txt
            page1.txt
            …
            pagen.txt
        Products
            …..
        Supply
            ……
        schemas.txt
        pagePool.txt
        other files
    index
        pagePool.txt
        directory.txt
        page1.txt
        …..
        pagen.txt
        other files
    treePic
        <relation name>_<search key>.txt
        …..
    queryOutput
        queryResult.txt
```

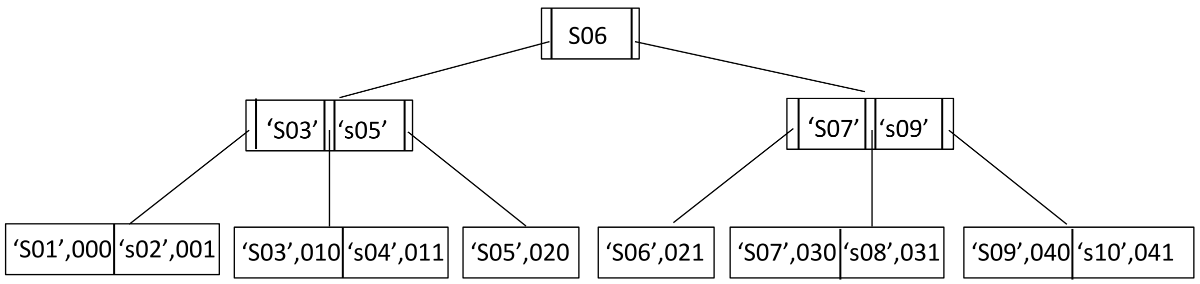
In the above structure, <your user name> is the folder named after your user name. It contains five sub-folders, named as *program*, *data*, *index*, *treePic* and *queryOutput*. The files in red will be provided by me. All the other files are to be your work, either developed by you, or as outputs generated by your programs. The following is a detailed description for each folder.

The *program* folder contains five Python files. These files include the eight functions described in Section 1: *relAlg.py* includes the three relational algebra functions, *buildTree.py* includes *build(*)*, *remove.py* includes *removeTree(*)* and *removeTable(*)* and *display.py includes displayTree(*)* and *displayTable(*).* The fifth file, *query.py*, contains the implementation for the queries using relation algebra functions. These queries are specified in Section 5.

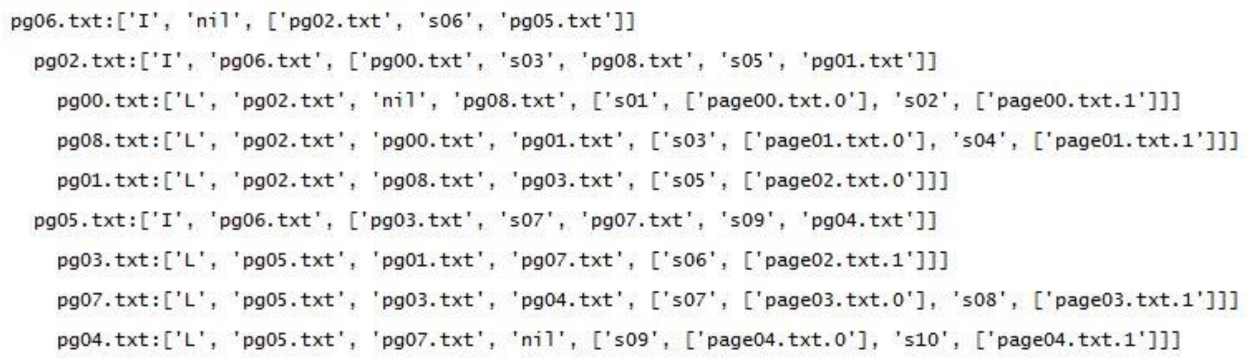
Under the *data* folder is a list of subfolders, one for each relation. Under each of these subfolders are the files of the tuples for that relation. This structure links each relation name to its instance in a natural way. For example, the instance for relation 'Suppliers' is the sequence of files with the order specified in *pageLink.txt* under the folder named *Suppliers*. The file *schemas.txt* contains schematic information about all the relations.

Under the *index* folder are the files for one or more B+_trees. The file *directory.txt* is where you save the information about all the B+_trees that you have built.

Under the *treePic* sub-folder is a list of files in which the B+_trees you create are displayed. The file name <relation>_<search key>.txt refers to the file where the B+_tree on <search key> for <relation> is displayed. The display must show the hierarchical structure of the B+_tree and the structure in each node. For example, the following is a B+_tree built on *Suppliers.sid* with an order of one.



where 'sxx' is a search key value and each three-digit number is a record id. The following is a possible display of this B+_tree:



```
pg06.txt:['I', 'nil', ['pg02.txt', 's06', 'pg05.txt']]
  pg02.txt:['I', 'pg06.txt', ['pg00.txt', 's03', 'pg08.txt', 's05', 'pg01.txt']]
    pg00.txt:['L', 'pg02.txt', 'nil', 'pg08.txt', ['s01', ['page00.txt.0'], 's02', ['page00.txt.1']]]
    pg08.txt:['L', 'pg02.txt', 'pg00.txt', 'pg01.txt', ['s03', ['page01.txt.0'], 's04', ['page01.txt.1']]]
    pg01.txt:['L', 'pg02.txt', 'pg08.txt', 'pg03.txt', ['s05', ['page02.txt.0']]]
  pg05.txt:['I', 'pg06.txt', ['pg03.txt', 's07', 'pg07.txt', 's09', 'pg04.txt']]
    pg03.txt:['L', 'pg05.txt', 'pg01.txt', 'pg07.txt', ['s06', ['page02.txt.1']]]
    pg07.txt:['L', 'pg05.txt', 'pg03.txt', 'pg04.txt', ['s07', ['page03.txt.0'], 's08', ['page03.txt.1']]]
    pg04.txt:['L', 'pg05.txt', 'pg07.txt', 'nil', ['s09', ['page04.txt.0'], 's10', ['page04.txt.1']]]
```

One can see how the previous B+_tree is simulated. For example, each .txt file simulates a page. Follows each colon is a Python list stored in the page preceding the colon. This list contains as its values some strings, as well as a nested list. The first value indicates whether a node is an internal or a leaf node, and the second value is a parental pointer. For leaf nodes, the lists also contain pointers to their left and right cousins. The nested lists simulate nodes in the B+_tree, where the child pointers are represented by page names.

The *queryOutput* folder contains a single file in which you print the resulting tables from the queries.

5. Your Task and Requirements

Your task for the project is to implement eight functions described in Section 1 under the assumption that only limited memory spaces are available, and run these functions to answer some basic queries. For marking purpose, your project should also meet some requirements so that it will be somewhat 'regularized'. These requirements are given in the following list.

1. Use the directory structure described in section 4 for the project. As a result, whenever you read or write a file in your code, use relative path name, like '../data/Suppliers/<file name>. Do not use absolute path name like 'C:/…'.
2. The eight functions described in Section 1 must be included in the four Python files, *relAlg.py, buildTree.py, remove.py, display.py* under *program* folder, as described in Section 4. In addition, these files should not contain any code other than function definitions.
3. Where you need to save structural data types (e.g., lists, tuples, etc.) into .txt files, use **json**.
4. *select(rel, att, op, val)*: if there exists a B+_tree on *rel.att*, use it for the search, otherwise use sequential search. In both cases, print the total number of pages read from B+_tree, if any, and from data files to the IDLE shell. (Note: do not include the cost for reading other files, such as *schemas.txt, pageLink.txt, directory.txt*, etc. Also, do not include the cost for writing files.) The output should be similar to the following format:

   When B+_tree is used:

   'With B+_tree, the cost of searching <att> <op> <val> on <rel> is <value> pages'

   When B+_tree is not used:

   'Without B+_tree, the cost of searching <att> <op> <val> on <rel> is <value> pages'

   Since, presumably, the memory space is limited, you should place at most one page from each relation in the memory at any given time.

5. *join(rel1, att1, rel2, att2)*: if there exists a B+_tree on either *rel1.att1* or *rel2.att2*, use it, otherwise, use nested-loops-page-at-a-time to implement the join (i.e., you are allowed to read only one page from either *rel1* or *rel2* at a time).
6. Run *build(*)* on the provided data set, and create two B+_trees with an order of 2, one on *Suppliers.sid*, and the other on *Supply.pid*.
7. Run *displayTree(*)* to display the structures of the two B+_trees you create under item 6 above. They should be displayed in files *Suppliers_sid.txt* and *Supply_pid.txt*, respectively, under folder *treePic*.
8. Include into file *query.py* under *program* folder the specifications for the following queries using the three relational algebra functions you create. Then run this file on the provided

database, and use function *displayTable(*)* to send the resulting relations to file *queryResult.txt.*

a. Find the name for the supplier ‘s23’ when a B+_tree exists on *Suppliers.sid.*
b. Remove the B+_tree from *Suppliers.sid*, and repeat Question a.
c. Find the address of the suppliers who supplied ‘p15’.
d. What is the cost of ‘p20’ supplied by ‘Kiddie’?
e. For each supplier who supplied products with a cost of 47 or higher, list his/her name, product name and the cost.

Unzip the package I supplied to you. It is the directory structure at the most basic level. Except for the data files under data folder, it does not contain anything else. Then work out the implementation to substantiate it.

6. Points to note

Since the requirements in Section 5 require you not to include any code other than definitions of functions into the four Python files mentioned there, you can call the functions defined in those files only from other Python files, or at the IDLE shell, which requires you to import them first. This is not a problem in the former case, since the Python file (*i.e., query.py*) from which you call a function and the Python file (e.g., *relAlg.py*) in which the function is defined are under the same folder for this project. To run a function at the IDLE shell, however, you must import the Python file in which the function is defined at the IDLE shell. There is a little hassle here. Before you type ‘import <file name without .py>’, you should insert the path for the folder under which the file is defined in sys.path manually. To avoid the hassle, you can open the file, and simply click on run-run module. Nothing visible will happen, except that the folder path has been inserted into sys.path, and also been set as the current working folder.

The files listed in the directory structure in Section 4 are by no means the only ones you should create. You can create other files if you like. In fact, it’s likely that you need to create some temporary files during the process of implementation.

Building a B+_tree is a non-trivial task. You need to insert search key values repeatedly. As discussed in the class, at some point you will need to handle overflow. It is almost certain that you will have to remove bugs repeatedly. This is also likely the case when you implement other functions. For debugging purpose, you can use a debugging tool, PDB, which is powerful and easy to use. (For details of how to use it, refer to the Python help manual.) It is always a good idea to start with small B+_trees, and gradually increase their sizes. This means that you need to maintain a page pool. When you need a page, you take it from the page pool. When you remove a B+_tree, its pages should be returned to the page pool. Similar situation happens for your implementation of the three relation algebra functions.

7. Submission

- Folder <your username> as specified in Section 4
- Folder <report> that contains the following files:
  - A *pdf* file that includes (1) the cover page which includes the names and student ids of the two group members, and the course number and title; (2) a description of the directory structure, and information about each file. (Do not list all the functions defined in each .py file.), and (3) screenshots for the output that shows the IO costs from those queries that use *select(*)* in the list of 8.a – 8.e in Section 5.
  - *txt* files where the B+_trees are displayed.
  - *queryResult.txt*. (Please write the query before the result it generates for each query.)
- zip both folders into a single file and drop to the dropbox.