Final Report

Team CSV++

Introduction

The ad-hoc data computing system is developed in Python with the goal of directly handling multiple .csv files that includes as large as millions of tuples without the step of loading data into an existing database such as MYSQL. With functions such as indexing (B+Tree indexing), this project provides both the flexibility and efficiency of handling .csv format data. In this report, we will describe the design of system architecture, the implementation details, the performance evaluation compared with q - Text as Data, and further discussion such as the advantages and disadvantages of the design and implementation, potential future improvements to the system, etc.

System architecture

The overall architecture of our system mainly includes the following parts:

- **User interface**, which accept queries and demonstrate the results to users. The user interface can mainly handling two functions: 1) indexing the .csv file according to the assigned attribute name and .csv film name, and 2) accepting queries in the sql SELECT-FROM-WHERE format. Code can be found in the Terminal.py.
- **Preprocessor**, which loads data and builds B+tree indexing files in the specified folder (default in the ./btree folder) according to the inputs (attribute, csv) in the user interface. In the B+tree indexing, key is the attribute value and the corresponding pointer stored is the byte-offset of the first byte in each tuple. Code can be found in the mybtree.py.
- **Query parser and query planner/optimizer**, which checks the format of queries input, parses them, and making the logical plan of executing them from the user inputs in the interface. In the query parser, the parsing is grouped into three part: the SELECT part, the FROM part, and the WHERE part. The FROM and the WHERE part will be parsed firstly, with all conditions from the WHERE part parsed into single table selection or two tables theta join (natural join as a special case of the theta join) algebras listed in the sequence of the optimized logical plan. The logical plan is optimized combining the rule-based and dynamic programming optimization, which executes the single table selection firstly (rule-based) followed by double table joins (with the join sequence optimised by dynamic programming). After parsing the FROM and WHERE parts, each condition will be passed into the physical execution layer. After the physical execution layer finishes the execution of selection and join algebras, results (lists of tuple pointers) will be returned to the query parser, which parses the WHERE part into projection algebra and prints the result out. Code can be found in the SQLparse.py and select_and_print.py.
- **Query processor/operator**, which contains several B+tree based physical execution plan to realize operators such as selection, comparison, arithmetic operations, etc. on the data. Query processor will accept inputs from conditions parsed in the query parser

and pass the result back into the query parser to run the projection algebras. Code can be found in the join.py.

- **Output printing**, which load data and print the result out according to the result of query processor and query parser. This part is integrated in the Query parser and query planner/optimizer and code can be found in select_and_print.py.

Implementation

Library

We utilized the following libraries in our system:

csv, pandas, BTrees.OOBTree, os, system, pickle/cPickle, sqlparse, numpy, time, etc.

Implementation details in different parts:

- User interface: find details in the User Interface Usage Document
- Preprocessor

  This part splits the CSV files such that the Btree files can fit into main memory (which is selected as 100MB). Users can specify maximum number of tuples in the splitted CSV files. After the file is splitted, user can specify the attributes that are necessary for the query processing. Then the Btree is built for each attribute selected using Python library *Btrees*. The selected attribute and the offset for each tuple are used as key and dictionary for the Btree, respectively. Then we use *cpickle* library (or *_pickle* in Python3) to serializing and de-serializing the Btree object.
- Query parser and query planner/optimizer

  This part mainly reads the raw SQL as a string into the code and by using sqlparse, each query condition within WHERE statement is extracted. Then these conditions are transferred to the query processor in a specific order such that the query time can be optimized based on the optimization rules we build. With the results generated from query processor, SELECT statement is parsed and the corresponding attributes are printed out in the terminal.
- Query processor/operator

  This part mainly focuses on the implementation of the join part which corresponds to the FROM and WHERE clauses. The basic idea is following the rule-based and cost-based optimization algorithms in the query planner/optimizer. Rule-based algorithm: the program will push down all the single file conditions down and implement them before any join function. All the necessary filter function will be applied to the preliminary result before Cartesian production. Cost-based algorithm: during the join condition between two files, whether to touch the file will based on the size of the offset lists and a estimation threshold has been tested and selected as 500 to get the optimal performance.

  Detailed procedure:

First, implement all the single file condition using single_join_filter_one function and get several offset lists for each condition. Then, use the and_condition_single, or_condition_single and except_condition_single function to combine all the offset lists for the same file and get one final offset list for each file.

Second, based on the size of the offsetlist, select how to implement the join condition between two files. If both offset lists are larger than size of 500, the program will rebuild two small btrees for the two files based on the common attribute and offset lists using get_small_btree and compare them using double_join_filter, double_join_filter_plus, double_join_filter_multi. If only one of the offset lists is larger than 500, the program will rebuild one btree for the larger offset list and compare them using btree_A_a_file, btree_A_a_file_plus, btree_A_a_file_multi, A_a_btree_file, A_a_btree_file_plus and A_a_btree_file_multi. If both offset lists are smaller than size of 500, the program will directly touch both of the two files using A_a_B_b_file, A_a_B_b_file_plus and A_a_B_b_file_multi. If there is no single file condition results, just directly use double_join_filter, double_join_filter_plus, double_join_filter_multi based on the original btrees.

Third, use the all the necessary filter functions, AB_AC, A_AB_and, A_AB_or, A_AB_B_and, A_AB_B_or to filter all the results before the next Cartesian production step which is really time-consuming.

Fourth, use cross_prod and AB_AB function if necessary to get the final results.

Last, use permute_list to transform the join result to print subroutine and print the final result.

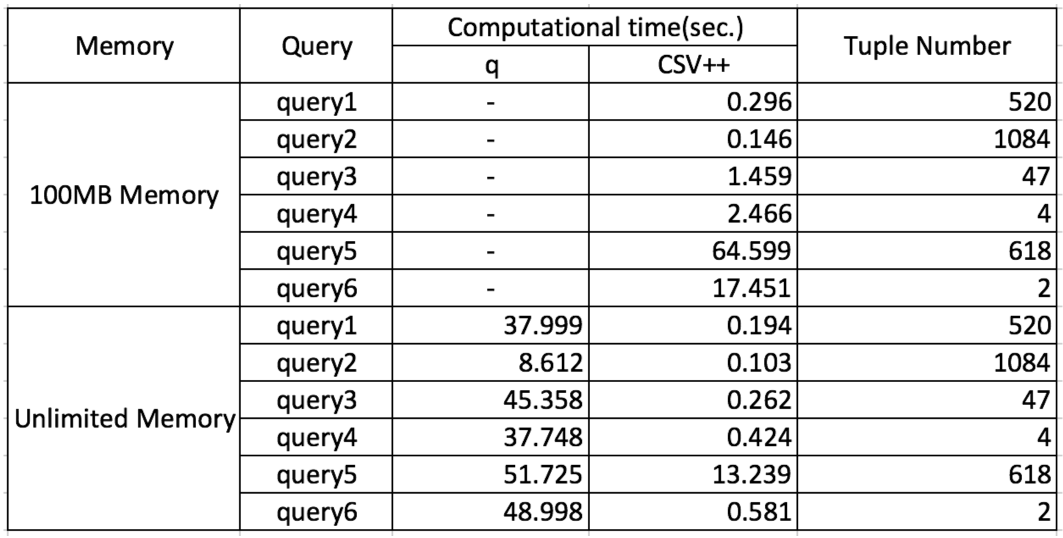
All the documentation can be found in join.py script.

- Output printing

After the execution of the selection and join algebra in the query planner/optimizer and query processor/operator (join.py). We got lists of byte offset corresponding to the different .csv file. Then we parse the SELECT part in the query and conform the attribute and .csv of each printed colume, With the byte offset as the pointer, we scan the result in correspondign .csv file colume by colume, combine them into the final result, and print them out.

Performance evaluation

We compare our system with the state of art system, q - Text as Data. As shown in the table below, our database outperforms q in terms of the cost for all queries with and without the memory limitation. Such advantages result from the better design of the system, for example, the usage of Btree, rule-based and cost-based optimization. However, it should be noted that our system might require extra time and memory to preprocess the data and store the Btree file. But the preprocess only need to be done once, along with importing the data and then users will spend less time for future use.



Memory	Query	Computational time(sec.)		Tuple Number
		q	CSV++	
100MB Memory	query1	-	0.296	520
	query2	-	0.146	1084
	query3	-	1.459	47
	query4	-	2.466	4
	query5	-	64.599	618
	query6	-	17.451	2
Unlimited Memory	query1	37.999	0.194	520
	query2	8.612	0.103	1084
	query3	45.358	0.262	47
	query4	37.748	0.424	4
	query5	51.725	13.239	618
	query6	48.998	0.581	2

Further discussion

Advantages:

- Exceptional performance (average 100-fold efficiency improvement without memory limit) handling millions rows of .csv file compared with existing q-Text As Data and ~10-fold efficiency improvement compared with MySQL database system.
- As an Ad-hoc database system, we don't need to load .csv data into the system, which is more convenient for user to handle data.

Disadvantages:

- Currently we can only realize regular SELECT-FROM-WHERE clauses without GROUP BY or HAVING clause
- In the current version we do not support LIKE condition
- In the current version we do not support aggregate functions such as sum(), max(), min()
- Compared with commercially available database systems, we don't have a GUI which can be more user-friendly for regular users without programming experience.

Potential future improvements

- Add the support of LIKE condition, GROUP BY/HAVING clause, and aggregate functions.
- Design a more user-friendly interface.

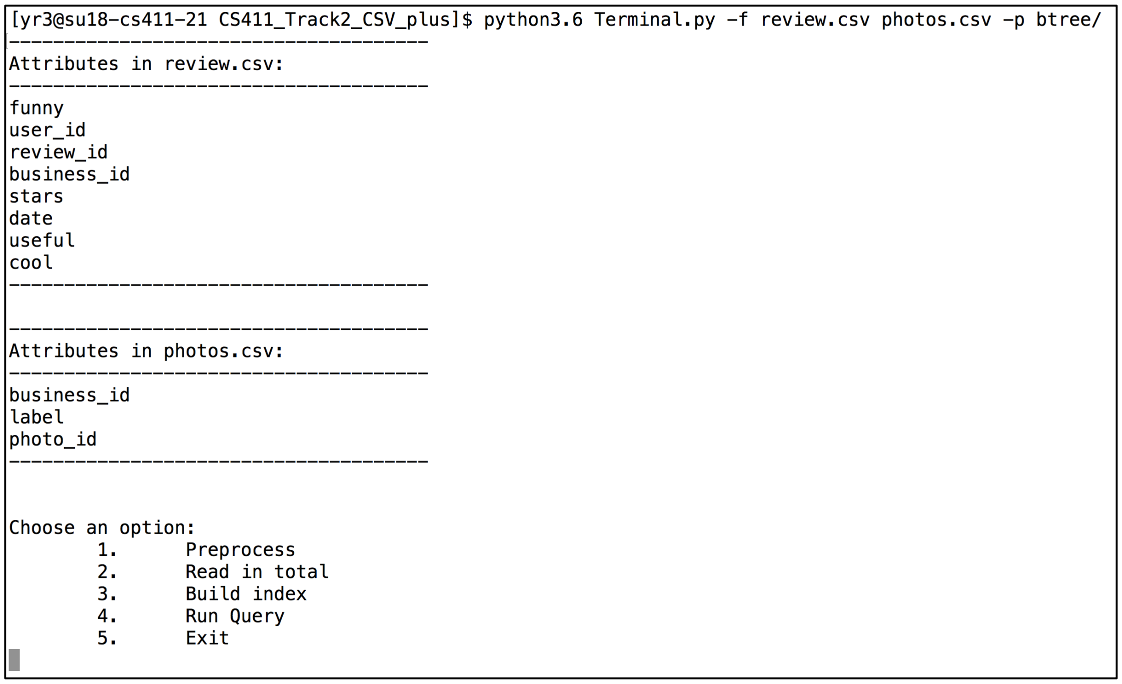
User interface usage documentation

Users can follow following steps to use this system:

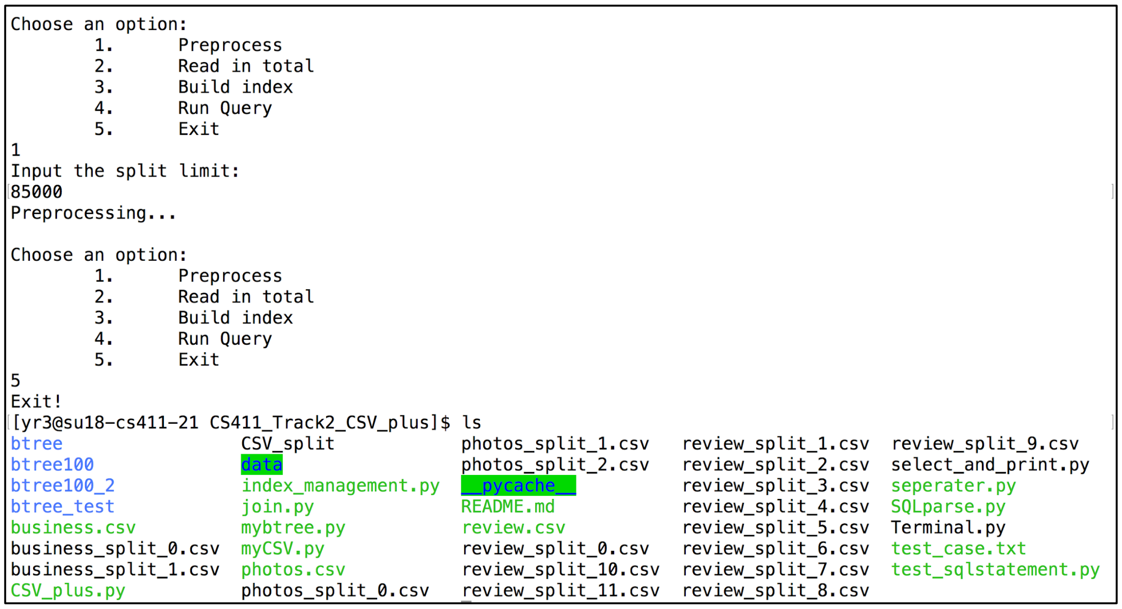
- Open the directory in your terminal and make sure the csv files stored in the directory of python codes.
- Run the following line to enter the system

  python Terminal.py -f *List of CSV File Name* -p *Path of btree file*
- As shown in the following graph, all the attributes in the csv files are printed. There are five options we can choose:
- If we are running code under memory limit, input '1' first and split csv files, we can specify the maximum number of rows in each divided csv file. A dictionary file is created to record the number of files that csv files are divided into.
- If we are running code without memory limit, input '2' first to update csv split dictionary so that the number of each csv file is 1.
- If we want to build index for some attributes, we can input '3' and write the following statement (The third part is to determine whether the attribute is a number). The Btree files will be saved in the path specified previously.

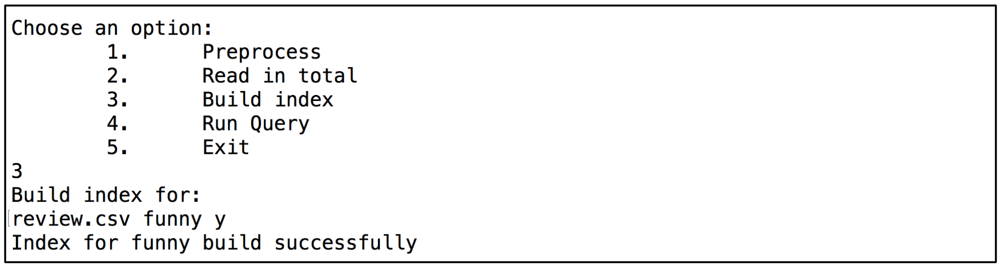
  CSV File Name *Attribute Name* *y/n*
- If we want to conduct query on the csv files, input '4' and write the SQL command. The query results and consuming time will be printed out.
- If we want to exit, write '5'.



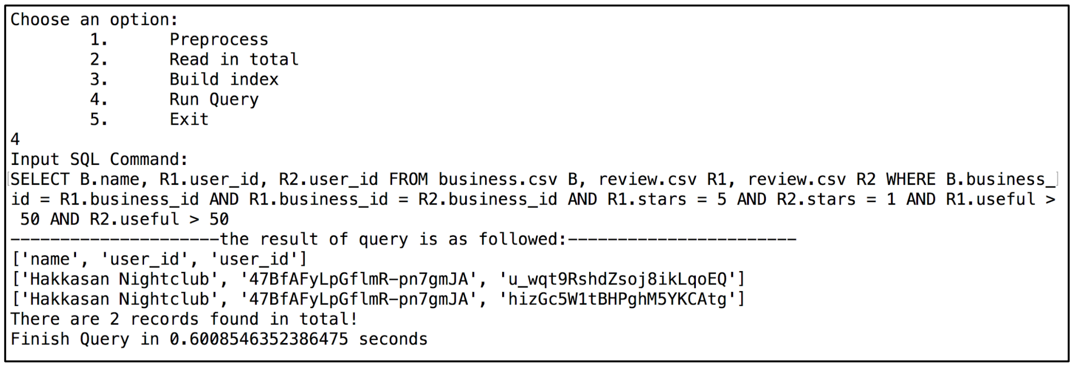
```
[yr3@su18-cs411-21 CS411_Track2_CSV_plus]$ python3.6 Terminal.py -f review.csv photos.csv -p btree/
--------------------------------------
Attributes in review.csv:
--------------------------------------
funny
user_id
review_id
business_id
stars
date
useful
cool
--------------------------------------

--------------------------------------
Attributes in photos.csv:
--------------------------------------
business_id
label
photo_id
--------------------------------------


Choose an option:
        1.      Preprocess
        2.      Read in total
        3.      Build index
        4.      Run Query
        5.      Exit
```



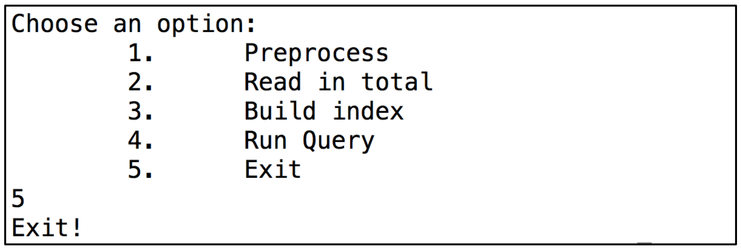
```
Choose an option:
        1.      Preprocess
        2.      Read in total
        3.      Build index
        4.      Run Query
        5.      Exit
1
Input the split limit:
85000
Preprocessing...

Choose an option:
        1.      Preprocess
        2.      Read in total
        3.      Build index
        4.      Run Query
        5.      Exit
5
Exit!
[yr3@su18-cs411-21 CS411_Track2_CSV_plus]$ ls
btree                 CSV_split            photos_split_1.csv   review_split_1.csv  review_split_9.csv
btree100              data                 photos_split_2.csv   review_split_2.csv  select_and_print.py
btree100_2            index_management.py  __pycache__          review_split_3.csv  seperater.py
btree_test            join.py              README.md            review_split_4.csv  SQLparse.py
business.csv          mybtree.py           review.csv           review_split_5.csv  Terminal.py
business_split_0.csv  myCSV.py             review_split_0.csv   review_split_6.csv  test_case.txt
business_split_1.csv  photos.csv           review_split_10.csv  review_split_7.csv  test_sqlstatement.py
CSV_plus.py           photos_split_0.csv   review_split_11.csv  review_split_8.csv
```



```
Choose an option:
        1.      Preprocess
        2.      Read in total
        3.      Build index
        4.      Run Query
        5.      Exit
3
Build index for:
review.csv funny y
Index for funny build successfully
```



```
Choose an option:
        1.      Preprocess
        2.      Read in total
        3.      Build index
        4.      Run Query
        5.      Exit
4
Input SQL Command:
SELECT B.name, R1.user_id, R2.user_id FROM business.csv B, review.csv R1, review.csv R2 WHERE B.business_
id = R1.business_id AND R1.business_id = R2.business_id AND R1.stars = 5 AND R2.stars = 1 AND R1.useful >
 50 AND R2.useful > 50
----------------------the result of query is as followed:-----------------------
['name', 'user_id', 'user_id']
['Hakkasan Nightclub', '47BfAFyLpGflmR-pn7gmJA', 'u_wqt9RshdZsoj8ikLqoEQ']
['Hakkasan Nightclub', '47BfAFyLpGflmR-pn7gmJA', 'hizGc5W1tBHPghM5YKCAtg']
There are 2 records found in total!
Finish Query in 0.6008546352386475 seconds
```



```
Choose an option:
        1.      Preprocess
        2.      Read in total
        3.      Build index
        4.      Run Query
        5.      Exit
5
Exit!
```