CS411 Project Track2 Report

Team 423
Team Members: Yichang Liang, Shijie Guo
Huamin Zhang, Sixuan Shen
12/28/2017

Introduction

People often get data files floating around in different file systems from our various activities, such as experiments, crawling the web, data sharing through emails or servers, etc. Such data is in the form of CSV files, Excel, tab-delimited text, or JSON format. People want to ask questions or compute over the data sometimes without putting the data into some database system such as MySQL or MongoDB. In this case, this project might be the thing you are looking for.
In this report, we will discuss the design and implementation details of our program, which provides an ad-hoc SQL-based data querying system in CSV format without importing it into a database system. Our program is in Python and it works perfectly with commodity personal computing environment, such as Windows, Mac OS, or Linux. Also, we will compare the performance of our system with q - Text as Data system across different queries and datasets and evaluate the results. Finally, we will discuss the pros and cons of the program along with some potential improvements.

How to use it

Our project supports ad-hoc SQL-based queries over data files in the CSV format. It supports one time selection from one or more tables. It will be command line based. Users need to put the corresponding csv files under the same folder with python files.

Environment Configuration:
IDE: PyCharm
Compiler: Anaconda with Python 2.7
Libraries: Pandas, numpy, sqlparse, egenix-mx-base

Usage:
Run commendLine.py. Users can build index or query through it.
Build Index: index (followed by csv filenames)
Eg. Index review.csv business.csv photos.csv
Query:
Enter SQL languages with the correct format.

SQL Format:
SELECT

(attribute names) eg. A.Year, A.Film, A.Name

FROM

(csv filenames) eg. movies.csv M1, movies.csv M2

WHERE

(conditions) eg. M.imdb_score = (3.1 + A.Winner)*2 AND (M.language like 'S%' OR A.Winner = 1)

ON

((join conditions) eg. M1.director_name = M2.director_name, M1.director_name = M3.director_name)

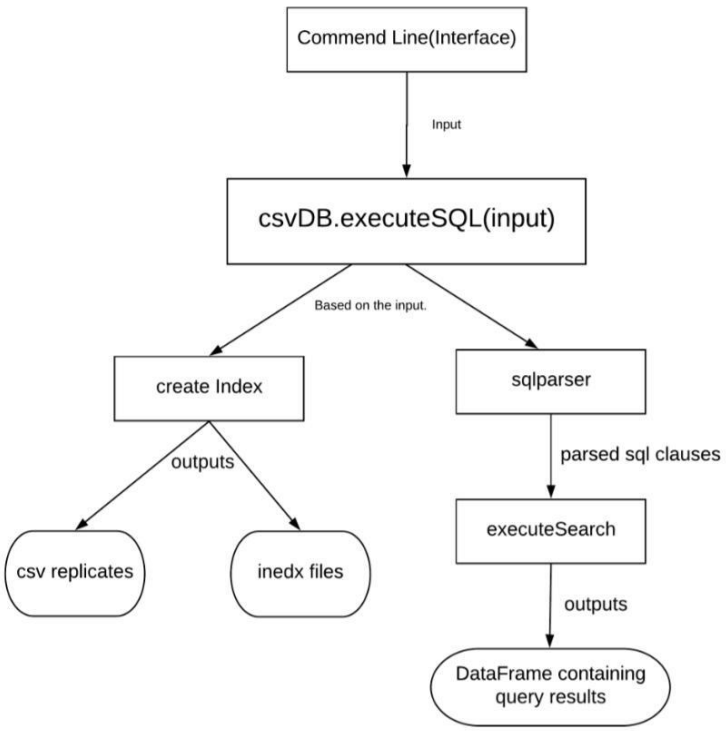
(DISTINCT)

(write “DISTINCT” at the very end to remove duplicates)

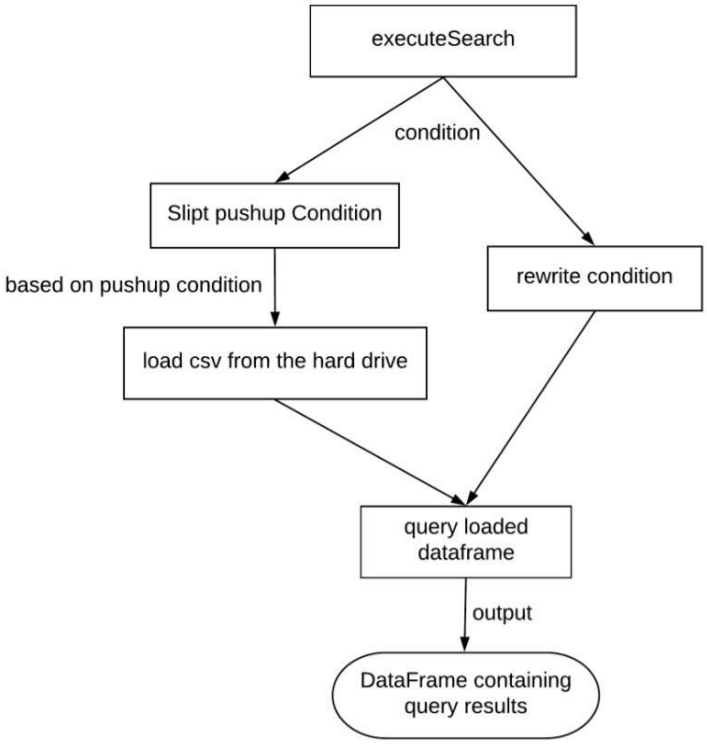
Example:
SELECT B.name, R1.user_id, R2.user_id FROM business.csv B, r.csv R1, r.csv R2 WHERE R1.stars = 5 AND R2.stars = 1 AND B.city = 'Champaign' ON (B.business_id = R1.business_id, B.business_id = R2.business_id) DISTINCT

Design

Below is a flow chart indicating how the program works. *CsvDB.executeSQL()* accepts commends and forward the commend to different modules. When an indexing request is accepted, the program will call *csvDB.indexData()*. Upon calling *csvDB.indexData()*, two outputs are generated, replicates of csv files and index files. Replicates contain the same data as the original csv file, but sorted on different columns. When a query is accepted, the program will use parse the query into different parts with sqlparser. Then pass the parsed SQL to *csvDB.executeSearch()* and it will return the query results.

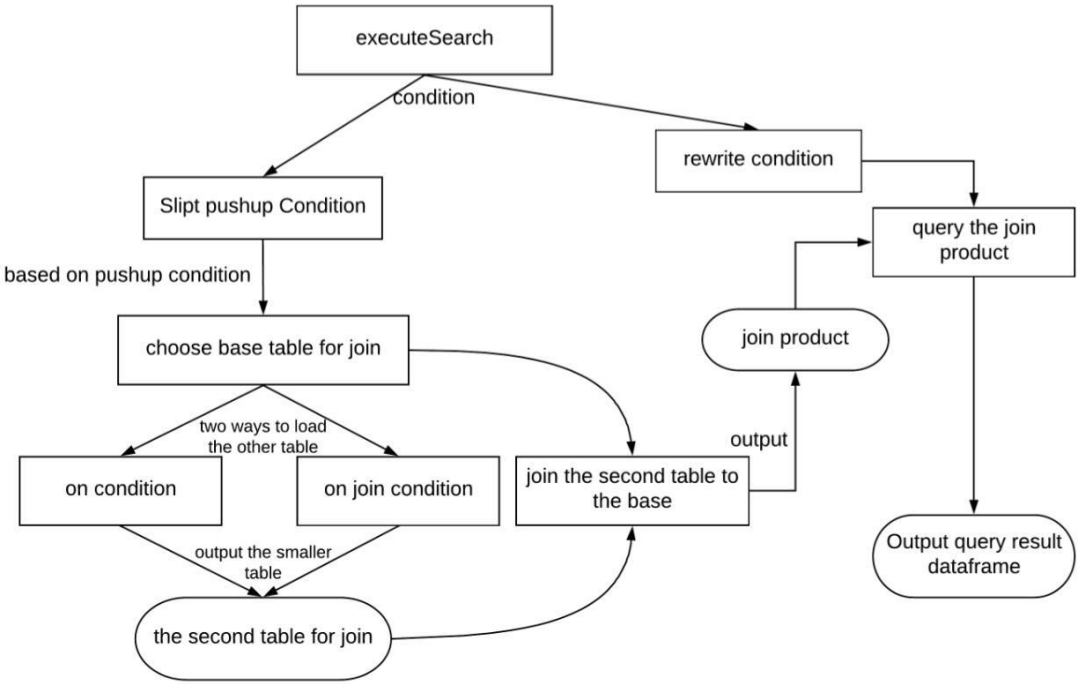


The flow chart below shows how *csvDB.executeSearch()* processes a query including up to one csv file. First, the program split the condition with " AND " and decide which of them are "push up conditions". "Push up conditions" are conditions that can be processed before join stage, or in this case, before loading the csv file from the memory. In our program, "Push up conditions" are conditions that compare one column to some constant values. For example, "M.age >= 15" or "M.name = 'John'" are push up conditions while "M.age >= M.gpa" is not. The advantage of doing so is obvious. It reduces the size of the table in the main memory, which makes it faster to query it or to join it with another table. All the "Push up conditions" are stored in a list and passed to *indexV1.loadTable()*. This method will load a csv file from the hard drive based on the input push up conditions. The output would be a pandas dataframe in the main memory. We will further discuss how this method works in the next section. On the other hand, the condition is rewritten into a form that can be accepted by pandas dataframes. Finally, simply call "*dataframe[eval(rewritten condition)]*" and pandas will do the final query process.



However, things get different if the query includes multiple tables. The flow chart below shows how the program processes an SQL including two tables. In this case, in order to further boost the speed and save the memory, we don't load two tables into the main memory simultaneously. Instead, we have a method to estimate the size of loaded table based on push up conditions without actually load it into the main memory. Then we compare the sizes of two tables and choose the one with fewer records to load. That table is called the "base table" for join. After that, we want to load another table. Since we already have the base table from the previous step, now we have two ways to load the other table. The first method is to load it based on push up conditions like we did before. The second method is to load it based on the

base table and join conditions. The program will estimate the result of both and choose the best solution. In other words, load the table with fewer records. And now we have two so called “optimized” tables and may proceed to join them. We use a method from pandas to do the join, which joins two tables on their common column. Finally, we rewrite condition, query the join product and return the result.



Things get exponentially complicated if we have more than two tables. However, the basic idea is the same. One last thing we would like to mention is how we handle “OR” operators in the query. Basically, “push up condition” no longer makes any sense if there exists any “OR” s in the query and we have to load the full table. We don’t want to do that so we came up with a solution to eliminate “OR” s in the query. The solution is simple, for every “OR” in the query we convert the query into three queries. For example, “A OR B” is converted into “A AND B”, “NOT A AND B” and “A AND NOT B”. We process them separately and merge the results. According to our test result, doing so efficiently reduces the runtime by 50%. However, the performance greatly depends on the query conditions so the actual performance may vary.

Implementation

In this part, we will focus on the pre-process and query optimization part, and the query clause parse with sqlparse package and the final query process with pandas package has been discussed in the previous Design part.

Index creation

Overall it will create the primary index on each attribute, which means it will make same number of replications for that table. The table will be stored in replication folder, and the corresponding index is stored under the index folder.

There will be two types of index; however they both use attribute's value as key and the value's record's start position in the file as value. The difference is one is b+ tree which can use cursor to get the neighbor's key and values. And another one is a kind of on disk dictionary, which will only hold the key value pair. B+ tree will only store integers and key will be at most 30 digits long, the Dictionary can hold integer or string and can be max 500 digits long.

The project will prefer to create the index as it can support bigger than or smaller than, if it cannot create, it will try to create the dictionary. If both are not able to create, then there will not be index for this attribute.

Within creating index procedure, it will also replicate tables on indexed attributes, it will not only store the table's data, but also the information of how many records is before this row and how many is equal to it. When we use index find the row, we can also get the information of it. Like in sorted stars replication:

funny	user_id	review_id	text	business_ic	stars	date	useful	cool	sizInfo
0	MmUdixE_	JS-7tDWdl	After eatin	dKdApYVF	1	2/4/2015	1	0	0000000;0000533
0	L7EYMtFIE	ZnMx-Y2K	ugggggh i	RDw-K8fjs	1	#######	0	0	0000000;0000000
0	2nUUubRa	AXGBe6y-	Was recomme nded to	uloYxyRAN	1	#######	0	0	0000000;0000000
2	hijMIaASTI	rVcZO63_	Very disappoin	LtNhEt9TY	1	#######	6	2	0000000;0000000
0	yAyiQXKhv	AWWgUpI	I wish I cou	dKdApYVF	1	3/9/2015	1	0	0000000;0000000
0	bfU0y3zbV	CdwGZtrB	It's a pity. I	uloYxyRAN	1	#######	0	0	0000000;0000000
0	zD_erWb1.	tblyalt1gT(	Grew up in	dKdApYVF	1	#######	0	0	0000000;0000000
1	Nc6Ng7W	2v0IZTgXtj	Try to cancel a members hip here and you	sBzefmmiX	1	2/7/2014	0	1	0000000;0000000
0	NSJ-5-ZW	HCrGeIgG6	sad to say	dKdApYVF	1	#######	0	0	0000000;0000000
1	dSTXuF_x_	75AOtPgt(	After seein	CUivTcULs	1	3/9/2017	2	0	0000000;0000000
			Unser erster und letzter Besuch:						

The size information for the first one of stars == 0 is 0;533, it means there is 0 smaller than it, 533 equal to it. The record is essential for the query optimization.

Loading size information

The input will pass in the table's name, the column Name, and the left boundary, the right boundary, also two Integers implying left or right open or closed boundary.

The method is loadTableCheck(tableName,condition):

Condition is in format of ['colName',openOrClose,'leftBound',openOrClose,'rightBound']; openOrClose is int and 0 means no condition provided, 1 means open, 2 means closed, 3 means equal. For example 'reviews.csv' ['stars', 1 , '3', 2 , '4'] means 3<review.stars<=4. Then it will search on index. For left, as it is open, it will find key '3''s right neighbor's value. And for right, it will just find '4''s value, but if 4 is not there, will find '4' left neighbor's value. Then we will get the 'pointers' of a closed range, it means the two pointers will be the ones pointing to the values which need to be selected by this condition. Then by the pointer we can access the smaller and equal value for Left and Right, also I stored totalRecordNumber information globally in the index, by these information we can get how many record will be returned.

If the index is B tree, it can support < >, but if it is dictionary, it will only support equal, then

if there is no proper index on these attribute, it will just return totalRecordNumber and firstRecord's pointer, meaning we need to load the whole table as we don't know information.

Loading table

For loading table there will be two ways to load table. However, the basic core is same it will load data from the proper start of the file.

1. Load table on push up conditions

First, we can only have information from the query. It will use loadTableOPIN (tableName,conditions) return (tableName,colName ,start,NumberOfRecord) to get certain table's best condition and the pointer, and the numberOfRecord, then we do not need to load the information again if in the future we decide to load this table. By start and numberOfRecord, we can directly load the table. But now we only use the numberOfRecord.

Actually this method will loop on each table's attribute if it is in the condition, in the loop it will use loadTableCheck() to load information, then compare then find the best and return. If we have table A B C it will get every table's best conditions and then choose the one with lowest returned recordNumber.

Then after comparison, we will load the table by the method loadTableWithOpimazedInfo (tableName,colName,start,RecordNumbers) return the dataframe, by tableName and colName it can find corresponding table and index, then by start and RecordNumbers, it can load data from that pointer. Then it will return the dataFrame.

2. Load table on join conditions

If we already have a table A in memory and we need to join B and C, there will be two options: 1. Load data as previous one 2. Loop on record of B, load one by one. Still at first we need to load the information of the options, but the first way's information is already in memory, as at first step we need to load each table's information to choose the best one. We only care about the join one, it will load information of A and B, if we do the join. We will use loadTableOPTINJoin(tableName,colName,equalTos). Here equalTos will be the values of the on-join attribute of A's record, first it will use unique to make sure equalTos doesn't have duplicated values, then it will randomly pick 5 samples from it, then use loadTableCheck() to get RecordNumber for each value in the sample, then estimate the total cost, also as here it need to randomly frequently access disk so a penalty will be multiplied to it, now I use 20 to make it fair when comparing to the first way's loading method which is near to sequential reading. Then we will compare 4 values in this case: Way1 for B and C, Way2 for B and C then choose the best one to load table.

Way1's load table method is described previously. For way2 it will use: loadTableBySeperateJoin(tableName,colName,equalTos), the input is the same input for the loadINformation method. Unlike way1 we don't have start position for each record, then we need to load the position first, but only position, it won't use loadTableCheck which will load position and the csv file. It only uses a partial functionality of loadTableCheck. Then use similar way to load from that position. Then we will get a dataframe for each element in

equals, and then we will append them together and return.

API:

A. createIndex(filename)

Create index for a csv file. First, check if there already exists one, otherwise proceed.

B. loadTable(tableName, conditions)

Returns pandas dataframe. This method is for one table only. It first finds the best condition, and then use it to load table.

C. loadTableOPTINJoin(tableName, colName, equalTos)

Returns (tableName, colName, NumberOfRecord, equalTos). Load the way2 information of the join. EqualsTo is the list of attribute's values of table already in memory. Then it returns (tableName, colName, NumberOfRecord, equalTos), the tableName and colName is only for convenience for next step.

D. loadTableOPIN(tableName,conditions)

Returns (tableName, colName , start, NumberOfRecord). Load way1 information of just load table on condition in query.

E. loadTableWithOpimazedInfo(tableName, colName, start, mins)

Returns pandas dataframe. Load table by the way1 information from previous step.

F. loadTableBySeperateJoin(tableName, colName, equalTos)

Returns pandas dataframe load table by the way2 information from previous step.

Evaluation:

We have compared our program (csvDB system) with q-text system in terms of query results, runtime, and main memory usage in the same environment. Below are the experiment results:

Datasets:
Photos(business_id, caption, label, photo_id)
Review-1m(funny, user_id, review_id , text,business_id, stars, date, useful, cool)
Business(address,attributes_AcceptsInsurance,attributes_AgesAllowed,attributes_Al
cohol,attributes_Ambience…)

1. **SELECT R.review_id, R.funny, R.useful FROM review-1m.csv R WHERE R.funny >= 20 AND R.useful > 30**

	query results (rows)	Runtime (s)	main memory usage(mb)
csvDB	520	0.0237	92.78
q-text	521	98.12	1050

2. **SELECT B.name, B.postal_code, R.review_id, R.stars, R.useful FROM business.csv B JOIN review-1m.csv R ON (B.business_id = R.business_id) WHERE B.city = 'Champaign' AND B.state = 'IL'**

	query results (rows)	Runtime (s)	main memory usage(mb)
csvDB	6593	1.626	98.36
q-text	6593	130.3	1155

3. **SELECT B.name, B.postal_code, R.stars, R.useful FROM business.csv B JOIN review-1m.csv R ON (B.business_id = R.business_id) WHERE B.name = 'Sushi Ichiban' AND B.postal_code = '61820'**

	query results (rows)	Runtime (s)	main memory usage(mb)
csvDB	47	0.0320	93.12
q-text	47	142.12	1155

4. **SELECT DISTINCT B.name FROM business.csv B JOIN review-1m.csv R JOIN photos.csv P ON (B.business_id = R.business_id AND B.business_id = P.business_id) WHERE B.city = 'Champaign' AND B.state = 'IL' AND R.stars = 5 AND P.label = 'inside'**

	query results (rows)	Runtime (s)	main memory usage(mb)
csvDB	13	2.019	92.52
q-text	13	138.37	1186

5. **SELECT R1.user_id, R2.user_id, R1.stars, R2.stars FROM review-1m.csv R1 JOIN review-1m.csv R2 ON (R1.business_id = R2.business_id) WHERE R1.stars = 5 AND R2.stars = 1 AND R1.useful > 50 AND R2.useful > 50**

	query results (rows)	Runtime (s)	main memory usage(mb)
csvDB	2	0.0270	93.14
q-text	2	103.05	1081

6. **SELECT B.name, R1.user_id, R2.user_id FROM business.csv B JOIN review-1m.csv R1 JOIN review-1m.csv R2 ON (B.business_id = R1.business_id AND R1.business_id = R2.business_id) WHERE R1.stars = 5 AND R2.stars = 1 AND R1.useful > 50 AND R2.useful > 50**

	query results (rows)	Runtime (s)	main memory usage(mb)
csvDB	2	0.0534	93.15
q-text	2	131.03	1154

First, we can see that JOIN cost a lot in q-text system which means it need more time if we join more large tables. However, since we use the condition in WHERE clause and estimate the joined table size in advance, the size of tables that need to be joined are much smaller. Thus, the cost of JOIN will be reduced a lot in our csvDB system.
Second, our system need smaller memory size because we will only load these rows that satisfied WHERE condition.
According to the performance and result of two systems, we can say that our system has strikingly improvement of processing queries on runtime and main memory usage with high accuracy (~100%).

Discussion

Advantages：

1. Our program achieves great performance in terms of runtime. It handles most experiment queries within under 0.1s due to the convoluted query optimization that was discussed in

Implementation part.
2. The performance in terms of main memory usage is also significant compare to q-text.

Disadvantages：
1. There is a tradeoff between disk space and speed. Instead of using main memory, this program takes much of the space of the hard drive to hold indexing data and csv files. Also, you can reduce the use of disk space, and it will slow the speed of query.
2. The process of building the index (pre-process part), which boosts the query speed, will take much time.
3. The program can only deal with "SELECT-FROM-WHERE" query, which means we can't deal with more complex format such as "GROUP BY" or "HAVING" clause. The system is strict with this syntax, and it will not return the expect result if the input is different with our format. Furthermore, we also need a ON clause such as A.attr = B.attr or the query will be slow since it will do a Cartesian product.

P.S. During the demo, the program returns some duplicated records for query 3a. That bug has been fixed

Potential Improvements and Future works:
1. Reduce the usage of the disk space as well as the cost of time on building index. This can be done through writing more complex index files and compressing the size of duplicates. And hopefully the indexing speed can also be improved with this approach.
2. Add support to "GROUP BY" and "HAVING". Details remain to be discussed.

Conclusion

In the track2 project, we built a small database query system like a real database and we fully applied what we have learnt in class. First, we define the basic framework and use pandas in python to achieve the basic functions which a database is supposed to have. For improving the performance, we build different indexes on the disk —— on-disk B+ tree and on-disk dictionary. We also estimate the time of loading each file before executing the join operation to achieve the best way of joining tables. The rule-based query optimization, the selection is pushed up before the join, is also applied in our program.

Thanks to this project, we get a thorough understanding of the database system from query parse to query optimization and then process the query on the pre-process data. This project is challenging and inspiring. Finally, our deepest gratitude goes to our professor Kevin C.C. Chang and TAs for their work and thoughtful suggestions that helped us to finish such an interesting project.

Github: https://github.com/moqi112358/CS411-Fall2017-Project-track2-csvDB