Design

The query system aims at handling csv file with more than 1 millions rows and hundreds of attributes (file size larger than 500 MB in general). The systems is solely based on disk. That is, the system will access both indexing files and csv files from disk, not from main memory.   In order to accelerate the performance, we will perform preprocessing before the query system is initiated. For each csv file, we will build save   hash tables for attributes that satisfy the indexing criteria. The indexing will only have 10% of the size of the csv files. For each query, the system only need to retrieve partial content requested in the query using indexing. The csv files are retrieved from disk every time when there is a query request. This approach has no requirement on user's operating system but takes longer to output query result however still depends on the CPU power of one's operating system.

After preprocessing stage is over, the query system will be ready for handling query request. The next step would be fully understand our query request. Given an input query sentence, we intend to parse the query sentence into three main sections: SELECT, FROM, and WHERE. Each section will further be parsed to obtain basic information such as csv files needed for this query, attributes the user intend to select, statements needed for evaluation, attributes needed to evaluate the statements in WHERE, etc.   We also assume that in WHERE statement, at most two individual attributes from distinct csv files will be given. That is, if the user intends to query on three csv files but gives a where statement involving three individual attributes A,B, C from each file, statement such as A*B > C will not appear. The WHERE part is also further parsed to form two lists, condition and connection. Condition contains each individual of the WHERE statement such as A >3 or A = 2. Connection contains the logical operators that are used for connecting condition back to the WHERE statement. The correct order of conditions and connections will be also identified by detecting parentheses. We also can use parsing to identify the join-on attribute if there is a join request in the query. For these cases, we will deal the join on statements (statement involving two distinct attributes from two different csv files but with equal sign).   We also observe that query requests can be divided into three different cases. Each case can be handled using different evaluation functions accordingly.

Case 1:   SELECT xxx FROM xxx, xxx
If no WHERE statement is given, simply return needed attributes. No practical meaning for considering return two huge table from join. So we will only implement it for one single table.
Case 2:   SELECT xxx FROM xxx WHERE xxx, xxx, xxx
Query request only involves one table and join in not needed. Simply evaluate WHERE
Case 3: SELECT xxx FROM xxx, xxx, xxx WHERE xxx, xxx, xxx
Most complicated case. Need to join multiple tables and also to evaluate complicated WHERE statements.

We intend to first evaluate WHERE conditions to decrease the needed rows of each csv file as much as possible before we perform JOIN and SELECT. Based on our assumption that no more than 2 individual attributes from different csv files will be appeared in the WHERE statements, we can divide conditions into three different categories. 1. Single attributes evaluation, 2. Single table evaluation, 3. Multiple table evaluation. Single attributes evaluation suggests that the a WHERE statement only involves one attribute such as A >2. Row numbers returned will related to only one table. Single table evaluation allows the statement to contain 2 attributes. However, both attributes much come from the same table and wow numbers returned will again only related to one table. Multiple table evaluation allows the statement to contain 2 attributes but two attributes must come from different tables. Row numbers returned will   Then we see that the query Case 2 can be   handled using both single attribute evaluation and single table evaluation. Query Case 3 may need all three evaluation categories. As mentioned earlier, if there is a join request , we need handle with care for equality statements. In practice, we know that evaluating the join on statement will return huge number of row positions (worst case could be all row indices for self join), so instead of perform evaluation in the evaluation function, we design the program to detect such equality statements and skip to the next WHERE condition.

Now suppose we have successfully obtained row numbers for each condition in the WHERE statement. We need to aggregate our results according to the logical operators.   Two different combination will be considered. 1.Combine within the same table. 2. Combine from multiple tables. Again, we can see results from Case 2 query only need to combine within the same table. Case 3 query need more careful handle.

Then based on all combined position, we simply perform join on join-on attributes identified in parsing stage and select requested attributes with specific requirement such as DISTINCT.

With the design, we can also handle self-join conveniently without specifying in the program as self-join can always be counted as Case 2 or Case 3. Also, there is no need to particularly deal with null value. The null value is treated as a particular key during indexing and can be easily retrieved during evaluation process.

Implementation

We mainly use the following Python libraries. Dictionaries and list comprehension are also used for hash table purpose and acceleration.

- Pandas: for general csv read, merge, and access
- Numpy: for general computation and evaluation
- re: for string parse and replacement
- csv: for preprocessing reading csv lines

1. Preprocessing

To ease evaluation, we only use three types: pure string, float, and int to store keys for each attributes.
For a csv file, we first filter out the attributes that worth indexing. The criterias are that 1. the number of unique keys of one attribute is at most 30% of the entire size of the csv file and 2. if it is a string-based attributes (not float or int), the maximum of character length is less than 100. For the attributes that satisfy the criterias, we will build hash table and save the indexing on disk as json file. The buckets of hash tables contain the line number of corresponding key for an attribute.

2. Parse

Many helpers functions are built. Main function will output all useful information.

- split_input: Split query according to SELECT, FROM, WHERE and output three individual parts. Check for specific requirement in SELECT such as DISTINCT.
- split_conditions: Split WHERE statement into lists of conditions and connections. Output correct order using function parentheses.
- Parentheses: Sort the split_conditions by counting the depth of the parentheses and marking the position of conditions. The precedence rules: (1) conditions in the same depth of parentheses are executed from left to right, (2) conditions in deeper parentheses are executed precedently. Note that there is no precedence between operators without parentheses, all conditions are executed from left to right.
- find_attributes: Split SELECT statement (mainly based on comma) and identify wanted attributes. Also record original attributes' name from tables in the same order in SELECT to handle rename problem in the query. Perform same operation to WHERE statement but split on mathematical operators or LIKE operator. Find attributes needed for evaluation and record the original attributes' name from tables in the same order for renaming purpose. Aggregate results from SELECT and WHERE to output overall needed attributes for tables. Also record the table where each attribute comes from in each WHERE condition.
- deserialization: Main function for parsing input. Call all helper functions to output all useful information and identify if the query needs to perform join. If yes, recheck conditions in WHERE statement to find join-on attributes.
- find_index_order: Load attributes' indexing from disk for attributes that need evaluation and find corresponding correct order of indexing for the csv files.

3. Selection:

Two main tasks are evaluation of WHERE conditions and combine returned positions for each table.

3.1 Evaluation

Helper functions are built for special evaluation such as LIKE.   Build three evaluation function for three evaluation categories discussion in Section 1. Main function is built to handle three query cases. In order to evaluate string-type equations and inequalities, we will use eval command in Python. Use pre-indexing as input, the evaluation functions operates on unique keys and return corresponding row indices in the dictionary.

- isMatch: Using dynamic programming for regular expression matching to implement the LIKE operator. We use dp(i,j) to handle calls match(text[i:,pattern[j:]]), saving us expensive string-building operations and allowing us to cache the intermediate results.
- detect_type: function built for handling special cases when string type evaluation needed to made for integer type data. Output updated WHERE statement to accommodate data type.
- single_attribute_eval: Evaluate a condition with only one attribute. Use isMatch for conditions involving LIKE and numpy library for all other types of equations and inequalities. Output row indices of the related table.
- Singe_table_eval: Evaluate a condition with two attributes from the same table. Use isMatch for conditions involving LIKE and numpy library for all other types of equations and inequalities. Output the row indices of the related table.
- Multi_table_eval: Evaluate a condition with two attributes from distinct tables. First find the table with smaller number of unique keys. If equal or not equal is in the condition, simply take intersection or union to obtain desired keys and then retrieve row indices from indexing. If other   inequalities are present, then loop through the smaller table to obtain needed keys. Output row numbers for both tables.
- Eval: Main function for evaluation. Take all conditions as input. For each condition, find out which category it belongs to and call corresponding helper function. Output dictionaries of row indices and use condition indices as keys.

3.2 Combination

Two helper functions are built for two combine categories. Main function calls helper functions for 3 query cases. Output needed row indices for each table in a dictionary using table index as keys.

- self_combine: Perform combination for positions returned for one table query. Simply perform intersection of positions for logical operator AND and union for logical operator OR.
- combine_multi: Perform combination for positions returned for Case 2 query and Case 3 query.   Initialize final output positions with the first non-empty conditional based positions from Eval. Initialize check list recording tables have appeared in the condition. Initialize boolean appear_together indicating if any condition has involved two tables. Loop through logical connections in WHERE statements. If appear_togher is False, update table positions accordingly with positions from next condition. Perform intersection for AND and union for OR. If appear_togther is True, update table positions for all tables in check list accordingly.

- Combine_Condition: For query Case 1, simply return all row indices. If join is False, call self_combine to return wanted row indices. If join, call combine_multi to return dictionary of wanted positions for each table.

4. Join

Use pandas library function to perform join on join-on attribute. The order of join will be based on the input order of FROM statement. Change order wisely to accelerate the query.

The main function will handle three query cases. Return final table with renaming. Using row indices returned for each table, only read in needed rows from csv files on disk and perform join.

- WHERE: For Case 1 query, simply return table. If join is False, return table with needed row indices. If join is needed, retrieve table one by one and join on join-on attribute.

5. Projection

Main function are designed to deal with specific request in SELECT statement such as SELECT * and SELECT DISTINCT. Use numpy library for choose unique tuples.

- SELECT: If * is present in the SELECT statement, simply return the table from WHERE function. If join is True, rename each attributes to the names called in SELECT statement and carefully deal with name of merge on attributes.

Evaluation and results

In this section, we will discuss the performance of our system by running several queries over it and compare the result with q-command result of the same query to check correctness, and then compare the running time of our system with the q-command execution time to check performance.
We will use 9 queries in total to evaluate our system, the first 6 queries were from the final demo, and we add three more queries to invoke more cases that are not covered in the final demo.

1. Test queries on final demo

(1). Queries in one table

a)
```
SELECT R.review_id, R.funny, R.useful
```

```
FROM review-1m.csv R
WHERE R.funny >= 20 AND R.useful > 30
```

Output result of the fastest version:

```
read query time: 0.23822903633117676
select row time: 0.0003380775451660156
combine pos time: 0.01694202423095703
perform join time: 1.7797338962554932
Total time with final selection: 2.0371079444885254
[520 rows x 3 columns]
Total row number is:  520
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	36.48s	2.03s

b)

```
SELECT B.name, B.city, B.state
FROM business.csv B
WHERE B.city = 'Champaign' AND B.state = 'IL'
```

Output result of the fastest version:

```
read query time: 0.03623199462890625
select row time: 0.0005288124084472656
combine pos time: 0.000370025634765625
perform join time: 0.2952260971069336
Total time with final selection: 0.33687877655029297
[1084 rows x 3 columns]
Total row number is:  1084
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	15.11s	0.33s

(2). Queries in two tables

a)

```
SELECT B.name, B.postal_code, R.stars, R.useful
FROM business.csv B, review-1m.csv R
```

```
WHERE B.business_id = R.business_id AND B.name = 'Sushi
Ichiban' AND B.postal_code = '61820'
```

Output result of the fastest version:

```
read query time: 0.17359709739685059
select row time: 0.05512595176696777
combine pos time: 0.00016713142395019531
perform join time: 2.75698184967041
Total time with final selection: 2.987962007522583
[47 rows x 4 columns]
Total row number is:  47
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	43.67s	2.98

b)

```
SELECT R1.user_id, R2.user_id, R1.stars, R2.stars
FROM review-1m.csv R1, review-1m.csv R2
WHERE R1.business_id = R2.business_id AND R1.stars = 5 AND
R2.stars=1 AND R1.useful>50 AND R2.useful>50
```

Output result of the fastest version:

```
read query time: 0.4866318702697754
select row time: 0.0032570362091064453
combine pos time: 0.05077409744262695
perform join time: 3.179759979248047
Total time with final selection: 3.722900867462158
[2 rows x 4 columns]
Total row number is:  2
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	32.55s	3.72

(3). Queries on three tables

a)

```
SELECT B.name, B.city, B.state, R.stars, P.label
```

```
FROM business.csv B, review-1m.csv R, photos.csv P
WHERE B.business_id = R.business_id AND B.business_id =
P.business_id AND B.city = 'Champaign' AND B.state = 'IL' AND
R.stars = 5 AND P.label = 'inside'
```

Output result of the fastest version:

```
read query time: 0.1824491024017334
select row time: 0.004486083984375
combine pos time: 0.00041294097900390625
perform join time: 2.4877820014953613
Total time with final selection: 2.677725076675415
[618 rows x 5 columns]
Total row number is:  618
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	44.32s	2.67s

b)

```
SELECT B.name, R1.user_id, R2.user_id
FROM business.csv B, review-1m.csv R1, review-1m.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
```

Output result of the fastest version:

```
read query time: 0.43301892280578613
select row time: 0.004939079284667969
combine pos time: 0.05269098281860 3516
perform join time: 3.7835159301757812
Total time with final selection: 4.2760169506073
[2 rows x 3 columns]
Total row number is:  2
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	42.12s	4.27s

2. More queries

In this section we tried three more queries to check whether our system can handle some cases which were not tested during the final demo.

(1) Parentheses and OR operator

```
SELECT B.name, B.city, B.state FROM business.csv B WHERE
B.stars > 3 AND (B.city = "Champaign" OR B.city = "Urbana")
```

This query tests the ability of our system to handle parentheses and OR operator.

Output result of the fastest version:

```
read query time: 0.04079914093017578
select row time: 0.0013320446014404297
combine pos time: 0.012190818786621094
perform join time: 0.2994260787963867
Total time with final selection: 0.35550403594970703
[868 rows x 3 columns]
Total row number is:  868
```

The result is identical to what the q-command and R SQL command return.

Query methods	q-command	Disk-based
Running time	20.00s	0.35s

(2) LIKE operator

```
SELECT B.name, B.postal_code, R.stars, R.useful FROM
business.csv B, review-1m.csv R WHERE B.business_id =
R.business_id AND B.name LIKE "%Chinese%" AND B.postal_code <>
"61820" AND R.stars>4 AND R.useful>5
```

This query tests the ability of our system to handle LIKE operator.

Output result of the fastest version:

```
read query time: 0.3607368469238281
select row time: 5.842043161392212
combine pos time: 0.07024884223937988
perform join time: 2.104969024658203
Total time with final selection: 8.380295038223267
[14 rows x 4 columns]
Total row number is:  14
```

Query methods	q-command	Disk-based
Running time	46.60s	8.38s

As we can observe in the form above, the LIKE operator is relatively slow in our implementation compared to the other operators. This is because our isMatch function which implements the LIKE functionality is a pattern matching function, and the additional running time it contributes to the whole query is O(NLP), where N is the number of tuples that are passed through other conditions in the query, L is the average length of the text and P is the length of the pattern.

(3) Queries on four tables

```
SELECT DISTINCT B.name FROM business.csv B, review-1m.csv R1,
review-1m.csv R2, review-1m.csv R3 WHERE B.business_id =
R1.business_id AND R1.business_id = R2.business_id AND
R2.business_id = R3.business_id AND R1.stars = 5 AND R2.stars
= 1 AND R3.stars=5 AND R1.useful>50 AND R2.useful>50
```

This query tests the ability of our system to handle 4-table-join and DISTINCT operator

Output result:

```
read query time: 0.5171840190887451
select row time: 0.007827043533325195
combine pos time: 0.05449795722961426
perform join time: 5.800078868865967
Total time with final selection: 6.381330966949463
[1 rows x 1 columns]
Total row number is:  1
```

Query methods	q-command	Disk-based
Running time	44.28s	6.38s

Discussion

1.Advantages

Achieved higher speed with the query accuracy remained. After processing, the disk-memory based query system is fast, about 10 times faster that the q query, due to the hashtable

technique and less redundant disk reading operations since originally every query has to access the full csv file on the disk whose speed is limited by the data transfer bandwidth.
In our implementation, before we do operations on the dataset, we keep track of the attributes used in the query clause and only these attributes are read during the processing. This allows us narrow down the size of the dataset as much as possible before we actually read and manipulate the dataset. We found that this step helps us reduce the running time significantly. For instance, if there are 30 columns in the original CSV file but only 3 of them are used in the query string, we project those three columns so that the size of the dataset is reduced to 1/10 of its original size.

2.Disadvantages

Need time to do preprocessing, which is about one minute, before entering all queries.
The main memory requirement for the main-memory based query system is more stringent, which needs M is larger than the sum of sizes of all tables.
Our system is not very efficient when handling range-based conditions since we only used hash tables for indexing, in order to make it faster when dealing with range-based conditions, we can use B+ tree for indexing. We also see that most of the time the program spent on is accessing csv file from disk. We have tested out the accessing time If the csv file is largely based CPU occupancy at the certain time. If some other programs are open, then the access time will increase dramatically.

3.Things we have learned and future works

Through this project we have learned and applied techniques which contains parsing, indexing, query optimization using pandas dataframe and Numpy. And manipulating strings using re. Through this project we have a better understanding on how indexing and query optimization make queries more efficient by seeing how the response time reduced.
Furthermore, we realized the trade off between performance and memory usage is important, we shouldn’t feel free to put everything into main memory anymore when working with real-world data and applications, though we had already used to put everything into memory when programming with smaller inputs.
As mentioned above, we implemented indexing using hash tables in our system, which is not efficient enough when dealing with range based queries. We are looking forward to improve our performance on range scan using B+ tree. We may also look at other faster csv reading techniques to accelerate the reading time.