# CatDB

DSCI 551
Nealson Setiawan

## Introduction

CatDB is a database management system fully implemented in Python and with all the necessary functions such as CRUD operations and query operations such as Selection, Projection, Filtering, Grouping, Aggregation, and Sorting.

Each operations do not load the data entirely into memory but loaded via chunks, and performed mapReduce concepts in processing the data. In fact, most of the operations loads in the data like a data stream through use of Python generators. Every iteration of the generator reads the next X amount of bytes of the file, and sends it into the query processing or CRUD operations. By doing this, we would not have to store the chunks physically on disk, which helps avoid writing an entirely duplicate copy of the data. This significantly reduces I/O cost. Furthermore, every time a file is read, a chunk dictionary, which contains the start and end byte of each chunk, is written. The DBMS will read this to create chunks. This has the benefit of having each chunk as close to the required chunk size. This is demonstrated when I load a small file into the system and it splits it into chunks.

The database management system is called CatDB because the query language for the DBMS is written in the perspective of a cat demanding the database system things. An example of a query with projection, selection, and grouping is: "**My cat demands [all] in paw-session of [iris] with ones that [petalLength is bigger than 1.4] to be glued together by [species]**"

As shown, attributes to be used in each query operation is highlighted with brackets for clear syntax. This query language is accessed and used to interact with the database management system via a simple command line interface built using Python's in-built functions.

## Planned Implementation

(FROM PROJECT PROPOSAL, MODIFIED TO MATCH CURRENT IMPLEMENTATION)

Data Model:

- NDJSON, every line is a valid JSON object

Actual contents of data:

- The actual contents will be stored in a JSON format
- When files are larger than memory, data will be loaded into chunks by reading every "chunksize" bytes per iteration, and stored each in NDJSON format

Query Language:

<Detailed manual is listed far below>

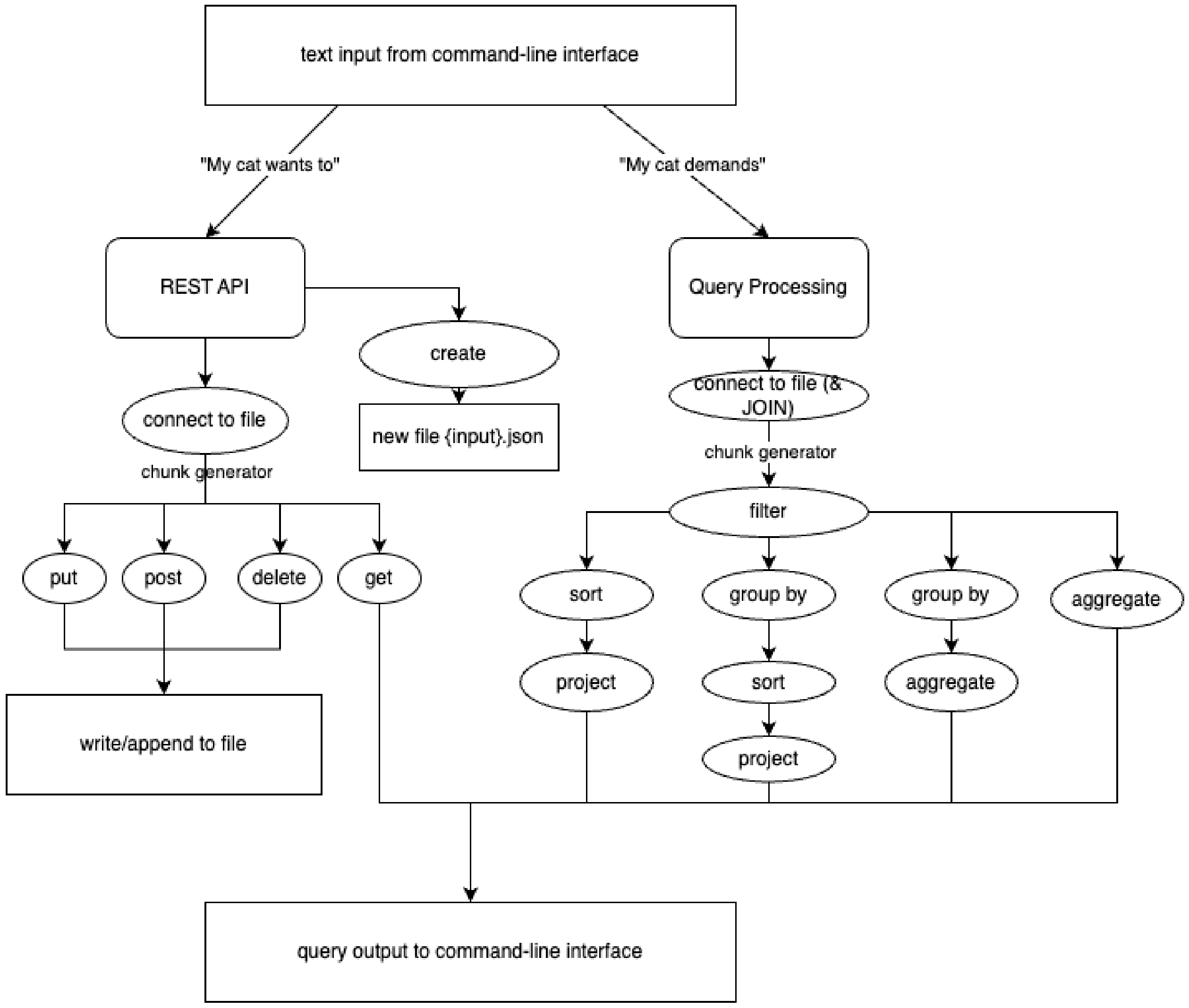
- The query language is to be written in the perspective of a cat wanting something from its owner
- So, sorting: arrange by, join: to be glued together by, etc.

CRUD:

- Implemented using REST API functions

## Architecture Design

### Flow Chart



Description:

The architecture of my database system starts with checking which header is in the text inputted in the command-line interface or CLI for short: "My cat wants to" or "My cat demands".

If the former is in the text input, the CRUD operations part of the query processing engine activates. If the create keyword is in text input, it outputs a new json file. Else, it starts a data stream by creating a generator that reads every "X" amount of bytes and write/append to a new file. The output of the GET function goes to the CLI.

On the other hand, if the latter is in the text input, the query processing side starts and it opens a data stream of the file and joins the two files if specified. It then sends this generator to the filter function. Then, it can go to one these 4 paths: "SP/GSP/GA/A". Whatever the output is, it outputs it to the CLI.

Every time the "connect to file" function is called, it first checks if a chunk dictionary is created or not, and creates one accordingly. Then it reads the chunk dictionary to find out the first chunk's start and end byte. Then it reads that specific start and end byte and outputs that section to the next operation. At the next iteration, it reads the chunk dictionary for the next chunk's start and end byte and return that section. This repeats until there are no more chunks in the file. This whole process is the "chunk generator" object that is returned every time a "connect to file" function is called. By doing it this way, we avoid loading everything into memory, but only loading section after section.

## Implementation

### Functionalities

Explanation:

Every time a text is inputted into the CLI, an instance of the class "query_pipeline" object is initialized. The query_pipeline class contains attributes such as a string query, as well as binary flags for each operation that is set to true when its respective key words is in the text input. This is to control the flow of the query execution. The query_pipeline instance then calls a function called "getFlags", which turns on the respective binary flags for each operation based on the text inputted. After the flags are on, query_pipeline instance then starts the query/CRUD operations.

Files directory:

**data (folder)**

- cdict (folder): contains chunk dictionary to each json file
- iris.json
- yelp_checkin.json
- yelp_tip.json

**scripts (folder)**

- crud_functions.py: implementations for crud
- crud_helper.py: helper functions for crud
- query_functions.py: implementations of query_pipeline class & query operations
- launch.py: to launch CLI

Implementation of query execution engine:

**CLASS query_pipeline()**

Attributes:

1. String **query**: text input from CLI
2. String **folder_path**: path of data folder relative to the scripts
3. String **tmp_path**: path of temporary folders for temporary storage of chunks or data relative to the scripts
4. Int **chunksize**: chunksize to be used in splitting the function
5. Boolean **{set of binary flags for operations indicating if respective keywords is in text input or not}:** binary flags to be passed to respective functions to control the flow of the query
6. Various data types **parameters of each operation:** the parameters of each function

Functions:

1. set_chunksize(self, chunksize): set the attribute chunksize to the input chunksize
2. getFlags(self): bunch of if statements like:
    a. if 'in-pawsession of' in self.query: self.s_on = True
    b. code to retrieve everything in the square brackets as parameters for the operation
3. evaluateQuery(self): through the binary flags
4. query operations:
    a. selection
    b. filter
    c. sort
    d. grouping

    e. aggregation

<u>Details on Memory Management implementation:</u>

When connecting to a file, a chunk dictionary JSON file is written to a disk in the "cdict" folder in the same folder the data is located in with name: "{file connected}.json_{chunksize}.json". An example name would be: "iris.json_1000.json" with iris.json the file name and 1000 the chunksize.

The chunk dictionary looks like this:

```
{"0": [0, 998], "1": [1000, 2000], "2": [2000, 3000], "3": [3000, 4000], "4": [4000, 5000], "5": [5000, 5936], "6": [5936, 6976], "7": [6976, 7912], "8": [7912, 8952]}
```

To create this chunk dictionary, the function "get_chunk_d" is called, located in crud_helper.py. Below are details of this function:

**CREATE CHUNK DICTIONARY**

Function: get_chunk_d
Located in: crud_helper.py(get_chunk_d)
Input:

1. fd: opened file handle/descriptor
2. fname: file name
3. save_folder_path:
4. chunksize: chunksize to read the file with

Output: write to disk a chunk dictionary
PSEUDOCODE:

```
1. Lazily read each **chunksize** amount of bytes of the file and keep track of every '}\n' character in the file and put into a list called NL_list (newline list)
2. Integer divide each byte in NL_list by chunksize to assign each byte to a unique chunk ID and save it to i_list
3. Since the i_list is in the same oder as NL_list, assign each NL_list value to its respective ID in i_list and save it in a dictionary with the ID as key and bytes as value. Keep it as a dictionary cont_d.
4. Keep only the largest and smallest byte of every ID
5. Create a "cdict" folder under the same folder that the file is stored in
6. Write this dictionary into a "cdict" as a JSON file
```

Then after, we want to read the file based on the chunk dictionary:

**READ FILE BASED ON CHUNK DICTIONARY**

Function: chunk_json
Located in: crud_functions.py(chunk_json)
Input:

1. fd: opened file descriptor/handle
2. chunk_d: chunk dictionary in Python dictionary form

Output: a generator for each chunk from the file

PSEUDOCODE:

```
1. Create i_chunk to keep track of number of current chunk
2. Create a while True loop:
    a. If i_chunk is not in the chunk dictionary keys --> break the loop
```

b. Save the current start and end byte of current chunk as b_list
c. If current chunk is first chunk --> seek the file descriptor cursor to 0
    i. Else --> seek the file descriptor cursor to start byte in b_list
d. From the current cursor's position, read the next (difference between start and end byte) bytes
e. Split the read chunk string with }\n and store as f_list
f. Every splitted string is a valid JSON object, so for every item in f_list, JSON load and append it all into a list called json_list
g. yield the json_list
    i. Yielding stops the function temporarily and outputs the current json_list and cause the function to be an iterable object
    ii. When we call the next iteration of this function, we will read the next chunk of the file and repeat until the chunk is finished
    iii. THIS CREATES THE GENERATOR ASPECT
h. increases the i_chunk by 1 to repeat with next chunk

Details of CRUD implementations:

My CRUD operations are implemented using REST API as it is a good way of manipulating JSON files.

**CONNECT/FROM**
Located in: crud_functions.py(connect_DB)
Keyword: "in-pawsession of"
Input: f_path: file path
Output: chunk_gen: chunk generator of file in file_path
Helper functions:

1. chunk_json: read the file based on its chunk dictionary
2. get_chunk_d: write to cdict folder a chunk dictionary of the file with chunksize

PSEUDOCODE:

```
1. Open the file in read binary mode
2. Check if cdict JSON file exists
    a. If not, create the chunk dictionary file by calling get_chunk_d function
3. Read the chunk dictionary
4. Call chunk_json function to get a generator based on the chunk dictionary
5. Return the chunk generator from chunk_json
```

**PUT**
Located in: crud_functions.py(put)
Keyword: "add/change this thing"
Input:

1. data: data to be added (data in the form of key:val)
2. url: url is location in file (url in form of: "key:val/key:val/..."
3. chunk_gen: chunk_generator from connect_db function

PSEUDOCODE:

```
1. Get key-value pairs from the url and JSON load data into a dictionary
2. For every chunk, check if the url (location where the data is being put in) exists or not
    a. If exists, replace everything under url as data
    b. If not exist, append the data to the end of the JSON file
    c. Append the (both updated or not) chunk to a temporary JSON file
3. Remove the original JSON file, and its chunk dictionary
4. Rename the temporary JSON file to the original JSON file's name
5. Recreate the chunk dictionary
```

**POST**
Located in: crud_functions.py(post)
Keyword: "slip in this thing"
Input:

1. data: data to be added (data in the form of key:val)
2. url: url is location in file (url in form of: "key:val/key:val/..."
3. chunk_gen: chunk_generator from connect_db function

PSEUDOCODE:

1. Get key-value pairs from the url and JSON load data into a dictionary
2. For every chunk, check if the url (location where the data is being put in) exists or not
    a. If exists, insert data under the url with random key
    b. If not exist, append the data to the end of the JSON file with random key
    c. Append the (both updated or not) chunk to a temporary JSON file
3. Remove the original JSON file, and its chunk dictionary
4. Rename the temporary JSON file to the original JSON file's name
5. Recreate the chunk dictionary

**GET**
Located in: crud_functions.py(get)
Keyword: "steal this thing"
Input:
1. chunk_gen: chunk generator from connect_db function
2. url: url is location in file (url in form of: "key:val/key:val/..."

Output: List of content that matches URL

PSEUDOCODE:
1. Get key-value pairs from the url and JSON load data into dictionary
2. For every chunk, check if there is parent directory is matching the url
    a. If yes, return content
    b. If no, return empty list

**DELETE**
Located in: crud_functions.py(delete)
Keyword: "destroy this thing"
Input:
1. chunk_gen: chunk generator from connect_db function
2. url: url is location in file (url in form of: "key:val/key:val/..."
3. f_path: path of the file
4. chunksize: chunksize to be used

PSEUDOCODE:
1. Get key-value pairs from the url and JSON load data into dictionary
2. For every chunk, check if there is parent directory is matching the url
    a. If yes, remove entry from the chunk
    b. Append to temporary file
3. Remove chunk dictionary and original file
4. Rename the temporary file as original file
5. Recreate chunk dictionary

**CREATE**
Located in crud_functions.py(create)
Keyword: "make this thing"
Input:
1. f_name: file name of one to be created

PSEUDOCODE:

```
1. JSON dump empty dict {} into a JSON file
```

<u>Details of Query implementation:</u>

**JOIN (HELPER FUNCTION) <INNER JOIN ONLY>**

**CONCEPT: CHUNK-BASED NESTED LOOP JOIN**

Located in: nested_loop_join(query_functions)

Keyword: "to get along with"

Input:

1. out_path: the path to write the joined chunks onto
2. c_gen1: the first file's chunk generator
3. c_gen2: the second file's chunk generator
4. join_var_list: the list of variables to be joined (first entry is left var, second entry is right var)

PSEUDOCODE:

```
1. For every chunk in 1st c_gen
    a. For every chunk in 2nd c_gen
        i. For every dictionary in chunk1
            1. For every dictionary in chunk2
                a. If the value of d1 and d2 is equal, add both dicts together
                    i. rename the left dict's join_var as left.join_var, and same
                       for right
                    ii. write to the out_path
                b. If we get a KeyError (means that the join variable doesn't exist
                   on both dict), we pass
```

**SCAN_JOIN (Both FROM/SCAN and JOIN)**

Located in query_functions.py(SCAN_JOIN)

Keyword: "in paw-session of"

Input:

1. fname_list: list of file names to be scanned/joined
2. is_join: indicator whether to join or not
3. join_var: list of join_vars

Helper Function:

1. connect_db: to connect to the file and get chunk generator
2. nested_loop_join: nested loop join the two chunk generators

PSEUDOCODE:

```
1. If is_join flag is off, connect to the first entry of fname_list and return the chunk gen
2. if is_join flag is on, connect to both files and plug it to the nested_loop_join function
    a. then connect to the new temporary/joined file and return chunk generator for it
```

**FILTER**

Located in: query_functions.py(FILTER)

Keyword: "with ones that"

Input:

1. f_on: filter flag
2. c_gen: chunk generator
3. var_dict: dict of variables to be filtered and its values (ex: sepalLength: 15)
4. comp_dict: dict of variables and its compare operator (ex: 'sepalLength': '>')

PSEUDOCODE:
```
1. If f_on flag is off, for every chunk in c_gen, yield chunk
2. If f_on flag is on, for every chunk in c_gen,
    a. Create a container list res_chunk (result chunk)
    b. For every dict in chunk:
        i. If the dict passes ALL the var_dict and comp_dict filter requirements, append
           dict to res_chunk
    c. If the res_chunk is not empty, yield res_chunk
        i. On the iteration, the filter operation will repeat for the next chunk
```

**SORT**
**CONCEPT: K-WAY External Sort (where K is number of chunks)**
Located in: query_functions.py(SORT)
Keyword: "arrange by"
Input:
1. sort_on: sorting flag
2. c_gen: chunk_generator
3. sort_key: key to be sorted by
4. ascending: 'A' for ascending and 'D' for descending

PSEUDOCODE:
```
1. If sort_on is off, for every chunk in c_gen, yield chunk
2. If sort_on is on, check ascending string
    a. If ascending 'A':
        i. # SORT PHASE
        ii. Create a folder 'tmp/sort' if not exist
        iii. For every chunk, sort the chunk ascendingly based on the sort key and write the
             chunk to the tmp/sort folder named sc_{i}.json, where i is the number of chunk
        iv. # MERGE PHASE
        v. Open a file tmp2.json (2nd temporary file) in append mode
        vi. Create an output buffer priority queue (a list for now)
        vii. Create a list of all opened file handles for all the sort chunks
        viii. Get the first line of every sorted chunk and push it into the output buffer (as a
              priority queue data structure implemented as heapq in Python's built-in library)
        ix. While output buffer is not empty
            1. Get the last in the output buffer and write it to tmp2.json
            2. Then get the next line of the sorted chunk that had its line pushed out
            3. If the next line is not empty (which means that if it's not the last of the
               chunk), read the line into a dict and push it into the priority queue
        x. REPEAT UNTIL EACH SORTED CHUNK IS EXHAUSTED
    b. If ascending 'D':
        i. Repeat everything above except that sort in a descending manner, and push the
           largest element into the output buffer priority queue
    c.  Remove the 'tmp/sort' folder
    d.  Connect to tmp2.json
    e.  For every chunk in the chunk generator, yield the chunk
```

**PROJECT**
Located in: query_functions.py(PROJECT)
Keyword: "My cat demands"
Input:

1. p_list: list of parameters to project
2. c_gen: chunk generator

PSEUDOCODE:

```
    1.  If 'all' is in p_list:
        a.  for chunk in c_gen, yield chunk
    2.  Else:
        a.  For every chunk in c_gen,
            i.    Create a container chunk
            ii.   For every dict in chunk,
                  1.  append only the key-value pairs of the keys in p_list
                  2.  yield the chunk
```

**AGGREGATE**
**CONCEPT: MAPREDUCE**
Located in: query_functions.py(AGG)
Keyword: "check each"
Input:

1. c_gen: chunk generator
2. agg_dict: dictionary of var and aggregate values (ex: {'sepalLength': 'biggest number'})

**NOTE: I made it so that sum(), max(), min(), mean() skips non-numeric values (or missing), but count() does not! So dividing sum() / count() might lead different results to mean() if dataset has missing values.**

PSEUDOCODE:

```
    1.  Create container dict result_d and reduce_d
    2.  # MAP PHASE
    3.  For every chunk in c_gen:
        a.  For every key, value in agg_dict:
            i.    If value == 'smallest number':
                  1.  For every dict in chunk, get the minimum value dict[key]
                  2.  append to result_d with 'min(key)' as key
            ii.   If value == 'biggest number':
                  1.  For every dict in chunk, get the maximum value of dict[key]
                  2.  append to result_d with 'max(key)' as key
            iii.  If value == 'count':
                  1.  For every dict in chunk, count = count + 1
                  2.  append to result_d with 'count(*)' as key
            iv.   If value == 'together number':
                  1.  For every dict in chunk, get total sum of dict[key]
```

```
                2.  append to result_d with 'sum(key)' as key
            v.      If value == 'middle number':
                1.  For every dict in chunk, get the [sum and count]
                2.  append to result_d with 'mean(key)' as key
4.  # REDUCE PHASE
5.  For every key, v in result_d:
    a.  If 'max' in k:
        i.      Append to reduce_d the maximum of the list of values in v
    b.  If 'min' in k:
        i.      Append to reduce_d the minimum of the list of values in v
    c.  If 'count' in k:
        i.      Append to reduce_d the sum of the list of values in v
    d.  If 'sum' in k:
        i.      Append to reduce_d the sum of the list of values in v
    e.  If 'mean' in k:
        i.      Append to reduce_d the (sum of the first element of the list of values in v) /
                (sum of the second element of the list of values in v)
    f.  yield reduce_d
    g.  THIS REPEATS FOR EVERY ITEM IN RESULT D
```

**GROUP BY**
**CONCEPT: 2-PASS GROUP BY**
Location in: query_functions.py(GROUPING)
Input:

1. c_gen: chunk generator
2. group_var: grouping variable

PSEUDOCODE:

```
1.  Create folder tmp/gBy if not exists to store group by chunks
2.  # 1-PASS
3.  For every chunk and its dict, store every possible values of dict[group_var] as keys with empty
    lists as g_val
4.  # 2-PASS
5.  For every chunk in c_gen:
    a.  For every dict in chunk:
        i.      Append the dict where the value is the key
    b.  For every key, value in g_val:
        i.      Write the values to a file with the key as name so to create a separate JSON file
                for each group
        ii.     REPEAT UNTIL EACH JSON FILE IS POPULATED
6.  For every key in g_dict:
    a.  connect to its JSON file
    b.  yield key, group_chunk_gen
```

**EVALUATE QUERY FUNCTION (QUERY FLOW EXECUTION)**
Located in: query_functions(evaluateQuery)

PSEUDOCODE:

1. If tmp folder doesn't exist, make one
2. If CRUD flag is False:
   - a. SCAN_JOIN to get a chunk generator
   - b. FILTER the chunk generator above to get a filtered chunk generator
   - c. 4 cases now:
     - i. If group flag and agg flag is both False
       1. Plug in filtered_cgen to SORT()
       2. Get sort_cgen to PROJECT()
       3. For every chunk in projected_cgen, for every d in chunk, yield d
     - ii. If group flag is False and agg flag is True
       1. Plug in filtered_cgen to AGG()
       2. For every chunk in agg_cgen, yield chunk
     - iii. If group flag is True and agg flag is False
       1. Create folder tmp/group_final if doesn't exist yet
       2. Plug in filtered_cgen to GROUPING()
       3. For every key, c_gen pair in grouped_cgen:
          - a. Remove tmp2.json if exist
          - b. Plug c_gen to SORT()
          - c. Plug in sort_cgen to PROJECT()
          - d. For every chunk in PROJECT_cgen, write the chunks to a file named g_{}.json
            - **i. THIS IS TO AVOID LOADING ALL THE CHUNKS INTO MEMORY AT ONCE**
          - e. Connect to this file and get a grouped_cgen
          - f. For every chunk in grouped_cgen,
            - i. Insert {'group': key} at first position of chunk
            - ii. For every dict in chunk, yield d
     - iv. If both group and agg flags are True
       1. Plug in Filtered_cgen to GROUPING()
       2. For every key, c_gen pair in grouped_cgen:
          - a. Plug c_gen to AGG()
          - b. For every chunk in AGG_cgen: append to a dict
            - **i. THIS IS FINE BECAUSE THE AMOUNT OF AGG() WON'T EXCEED MEMORY LIMITS**
            - ii. yield this dict
3. If CRUD flag is True:
   - a. Check if create is on:
     - i. Plug file name to create()
     - ii. For every item in list[self.f]:
       1. yield item
   - b. Else connect to the file:
     - i. If put flag is on:
       1. Plug this c_gen to PUT() with url
       2. yield [self.data, self.url, self.f]
     - ii. If post flag is on:
       1. Plug this c_gen to POST() with url
       2. yield [self.data, self.url, self.f]
     - iii. If delete flag is on:

                1. Plug this c_gen to DELETE()
                2. yield [self.url, self.f]
            iv. If get flag is on:
                1. Plus this c_gen to GET() with url
                2. For every chunk in c_gen:
                    a. yield chunk

**ASSUMPTIONS**

1. **The chunksize CANNOT be less than the amount of bytes in one line or it will fail**

LIMITATIONS

1. Only can group with one variable
2. Only can sort using one variable
3. Only can do Nested loop **INNER** join, this can be pretty slow.
4. If aggregate can only project aggregated variables not any other variables
5. Not much error handling is implemented. **If error occurs, the whole thing crashes**.
6. **Multiple filtering** on the **SAME VARIABLE** does not work for now, I ran out of time!
7. **Multiple filtering** with **SORTING** sometimes work and sometimes doesn't
8. Graphics is LIMITED.

DATASETS

**Iris Dataset [15,349 bytes]**

Link: Iris Dataset (JSON Version) (kaggle.com)

Modification before loading:

1. Removed brackets at start and end and converted to NDJSON format
    a. This cause every line to be a valid JSON object

**Note: I have set it that if you load the Iris dataset, the chunksize will be 10,000 bytes and will result in around 16 chunks. This is to showcase how my approach to memory management is able to be used to be used in devices with low memory capacity.**

**SteamDB Dataset [158MB]**

Link: Video Games on Steam [in JSON] (kaggle.com)

Modification before loading:

1. Removed brackets at start and end and converted to NDJSON format
    a. This cause every line to be a valid JSON object

**Note: The dataset contains a lot of missing values. This is to show that my dataset can handle missing values relatively alright.**

**Yelp Business and Checkin and Tip Dataset[287MB & 180.6MB]**

Link: Yelp Dataset (kaggle.com)

Modification before loading:

1. To demonstrate joining, it would take too long to join 180.6MB and 287MB together, so I instead took the first 10,000 entries of yelp_tip and yelp_checkin and yelp_business and named it **yelp_tip_sample.json and yelp_checkin_sample.json and yelp_business_sample.json**, which resulted in 19.8MB and 22.7MB and 8MB

## TECH STACK

Backend: Libraries(JSON, OS, Shutil, re, numpy, itertools, sys, random, string, heapq)
Frontend: Libraries(Colorama, Python built-in while function)

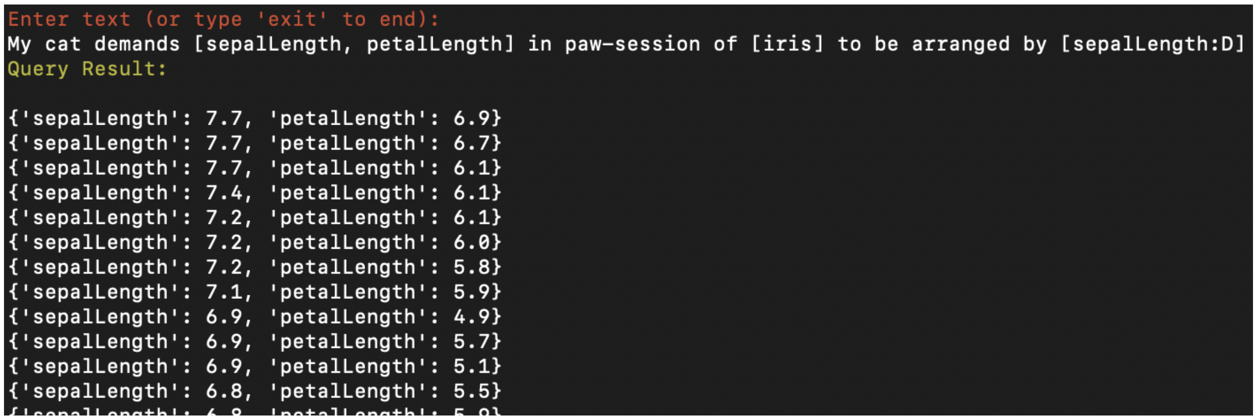
## SAMPLE QUERIES (IMPLEMENTATION SCREENSHOTS):

**NOTE: Default chunksize for IRIS dataset is 10,000 bytes, and the other two 4MB**

IRIS

**PROJECTION + SORT**

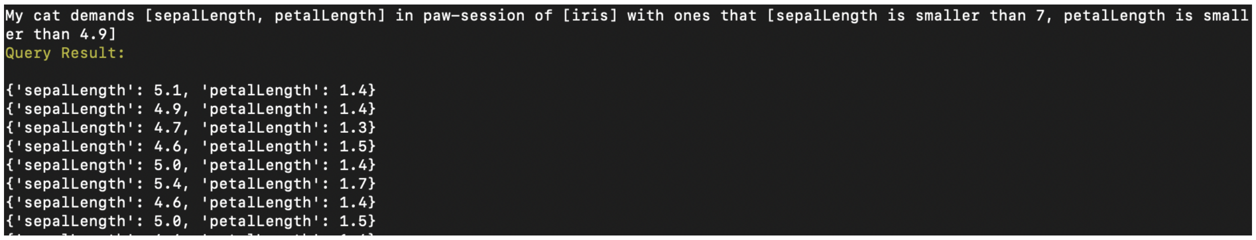
My cat demands [sepalLength, petalLength] in paw-session of [iris] to be arranged by [sepalLength:D]



```
Enter text (or type 'exit' to end):
My cat demands [sepalLength, petalLength] in paw-session of [iris] to be arranged by [sepalLength:D]
Query Result:

{'sepalLength': 7.7, 'petalLength': 6.9}
{'sepalLength': 7.7, 'petalLength': 6.7}
{'sepalLength': 7.7, 'petalLength': 6.1}
{'sepalLength': 7.4, 'petalLength': 6.1}
{'sepalLength': 7.2, 'petalLength': 6.1}
{'sepalLength': 7.2, 'petalLength': 6.0}
{'sepalLength': 7.2, 'petalLength': 5.8}
{'sepalLength': 7.1, 'petalLength': 5.9}
{'sepalLength': 6.9, 'petalLength': 4.9}
{'sepalLength': 6.9, 'petalLength': 5.7}
{'sepalLength': 6.9, 'petalLength': 5.1}
{'sepalLength': 6.8, 'petalLength': 5.5}
```

**PROJECTION + MULTIPLE FILTER**

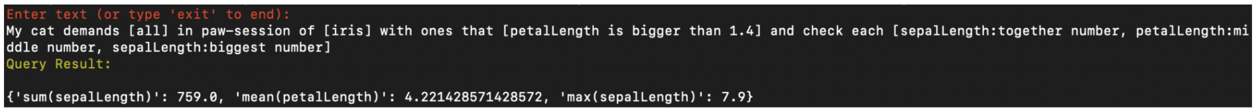
My cat demands [sepalLength, petalLength] in paw-session of [iris] with ones that [sepalLength is smaller than 7, petalLength is smaller than 4.9]



```
My cat demands [sepalLength, petalLength] in paw-session of [iris] with ones that [sepalLength is smaller than 7, petalLength is small
er than 4.9]
Query Result:

{'sepalLength': 5.1, 'petalLength': 1.4}
{'sepalLength': 4.9, 'petalLength': 1.4}
{'sepalLength': 4.7, 'petalLength': 1.3}
{'sepalLength': 4.6, 'petalLength': 1.5}
{'sepalLength': 5.0, 'petalLength': 1.4}
{'sepalLength': 5.4, 'petalLength': 1.7}
{'sepalLength': 4.6, 'petalLength': 1.4}
{'sepalLength': 5.0, 'petalLength': 1.5}
```

**FILTER + AGG**

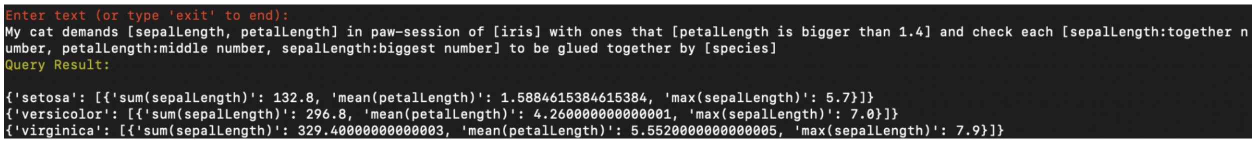
My cat demands [all] in paw-session of [iris] with ones that [petalLength is bigger than 1.4] and check each [sepalLength:together number, petalLength:middle number, sepalLength:biggest number]



```
Enter text (or type 'exit' to end):
My cat demands [all] in paw-session of [iris] with ones that [petalLength is bigger than 1.4] and check each [sepalLength:together number, petalLength:mi
ddle number, sepalLength:biggest number]
Query Result:

{'sum(sepalLength)': 759.0, 'mean(petalLength)': 4.221428571428572, 'max(sepalLength)': 7.9}
```

**FILTER + AGG + GROUP**

My cat demands [sepalLength, petalLength] in paw-session of [iris] with ones that [petalLength is bigger than 1.4] and check each [sepalLength:together number, petalLength:middle number, sepalLength:biggest number] to be glued together by [species]

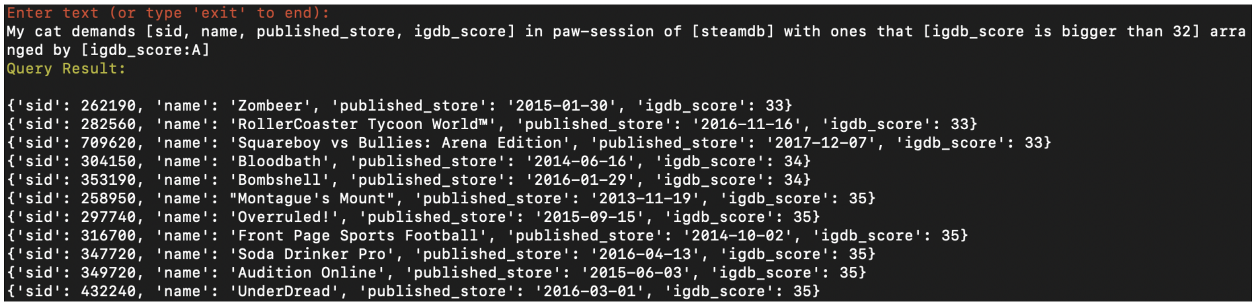


```
Enter text (or type 'exit' to end):
My cat demands [sepalLength, petalLength] in paw-session of [iris] with ones that [petalLength is bigger than 1.4] and check each [sepalLength:together n
umber, petalLength:middle number, sepalLength:biggest number] to be glued together by [species]
Query Result:

{'setosa': [{'sum(sepalLength)': 132.8, 'mean(petalLength)': 1.5884615384615384, 'max(sepalLength)': 5.7}]}
{'versicolor': [{'sum(sepalLength)': 296.8, 'mean(petalLength)': 4.260000000000001, 'max(sepalLength)': 7.0}]}
{'virginica': [{'sum(sepalLength)': 329.40000000000003, 'mean(petalLength)': 5.5520000000000005, 'max(sepalLength)': 7.9}]}
```

## STEAMDB

### PROJECTION + FILTER + SORT(ASCENDING)

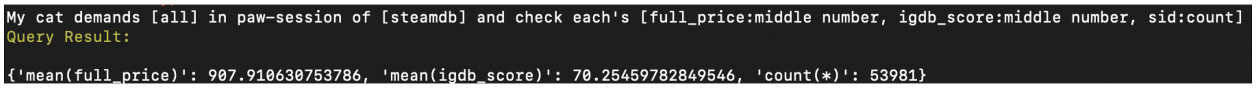
My cat demands [sid, name, published_store, igdb_score] in paw-session of [steamdb] with ones that [igdb_score is bigger than 32] arranged by [igdb_score:A]



```
Enter text (or type 'exit' to end):
My cat demands [sid, name, published_store, igdb_score] in paw-session of [steamdb] with ones that [igdb_score is bigger than 32] arra
nged by [igdb_score:A]
Query Result:

{'sid': 262190, 'name': 'Zombeer', 'published_store': '2015-01-30', 'igdb_score': 33}
{'sid': 282560, 'name': 'RollerCoaster Tycoon World™', 'published_store': '2016-11-16', 'igdb_score': 33}
{'sid': 709620, 'name': 'Squareboy vs Bullies: Arena Edition', 'published_store': '2017-12-07', 'igdb_score': 33}
{'sid': 304150, 'name': 'Bloodbath', 'published_store': '2014-06-16', 'igdb_score': 34}
{'sid': 353190, 'name': 'Bombshell', 'published_store': '2016-01-29', 'igdb_score': 34}
{'sid': 258950, 'name': "Montague's Mount", 'published_store': '2013-11-19', 'igdb_score': 35}
{'sid': 297740, 'name': 'Overruled!', 'published_store': '2015-09-15', 'igdb_score': 35}
{'sid': 316700, 'name': 'Front Page Sports Football', 'published_store': '2014-10-02', 'igdb_score': 35}
{'sid': 347720, 'name': 'Soda Drinker Pro', 'published_store': '2016-04-13', 'igdb_score': 35}
{'sid': 349720, 'name': 'Audition Online', 'published_store': '2015-06-03', 'igdb_score': 35}
{'sid': 432240, 'name': 'UnderDread', 'published_store': '2016-03-01', 'igdb_score': 35}
```

### AGG

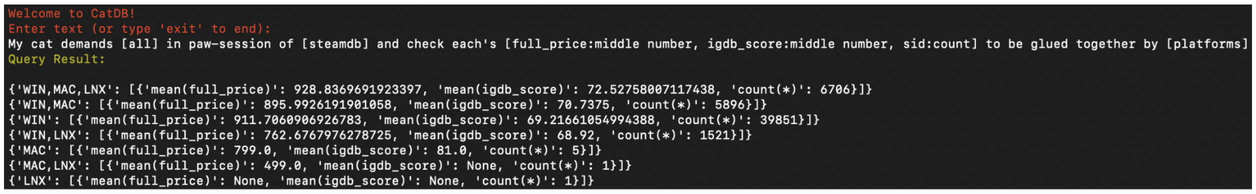
My cat demands [all] in paw-session of [steamdb] and check each's [full_price:middle number, igdb_score:middle number, sid:count]



```
My cat demands [all] in paw-session of [steamdb] and check each's [full_price:middle number, igdb_score:middle number, sid:count]
Query Result:

{'mean(full_price)': 907.910630753786, 'mean(igdb_score)': 70.25459782849546, 'count(*)': 53981}
```

### AGG + GROUP

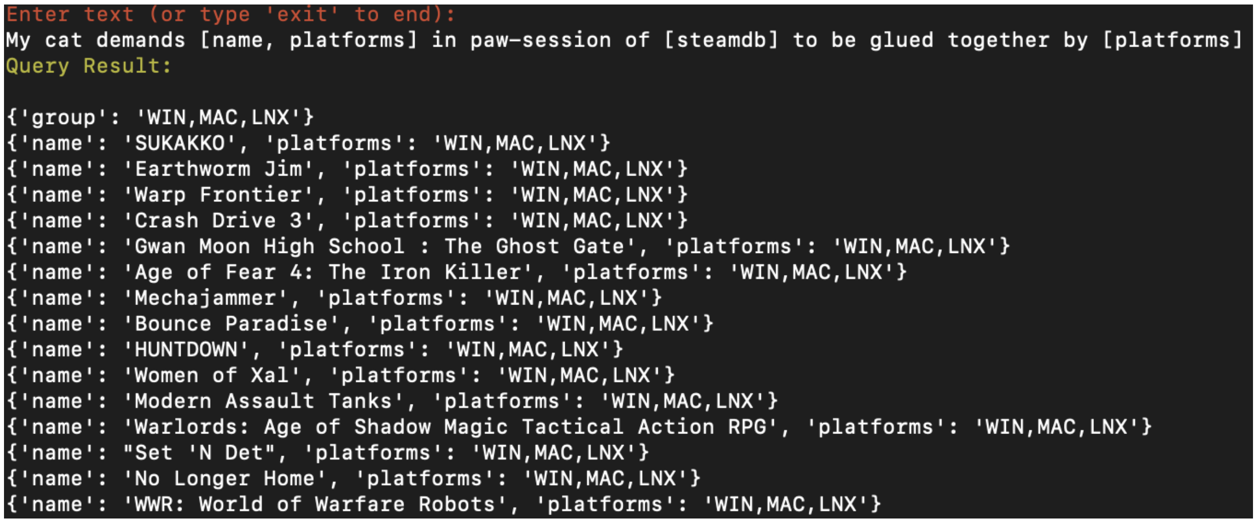
My cat demands [all] in paw-session of [steamdb] and check each's [full_price:middle number, igdb_score:middle number, sid:count] to be glued together by [platforms]



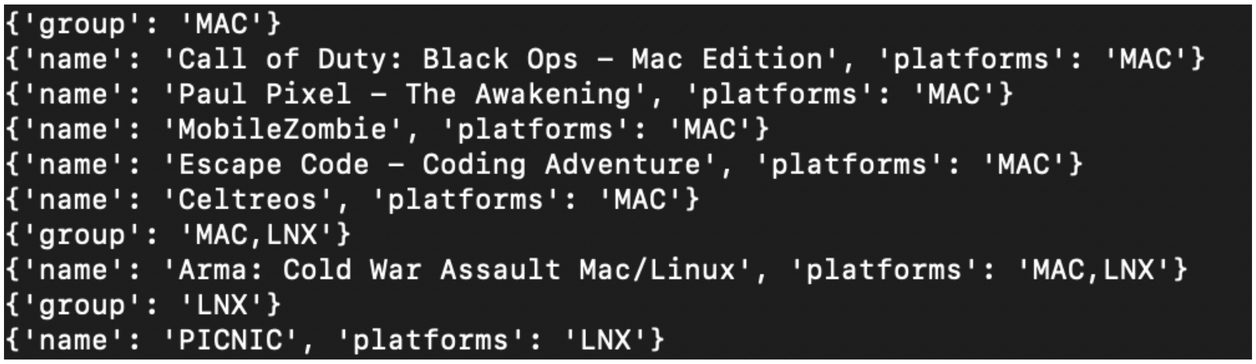
```
Welcome to CatDB!
Enter text (or type 'exit' to end):
My cat demands [all] in paw-session of [steamdb] and check each's [full_price:middle number, igdb_score:middle number, sid:count] to be glued together by [platforms]
Query Result:

{'WIN,MAC,LNX': [{'mean(full_price)': 928.8369691923397, 'mean(igdb_score)': 72.52758007117438, 'count(*)': 6706}]}
{'WIN,MAC': [{'mean(full_price)': 895.9926191901058, 'mean(igdb_score)': 70.7375, 'count(*)': 5896}]}
{'WIN': [{'mean(full_price)': 911.7060906926783, 'mean(igdb_score)': 69.21661054994388, 'count(*)': 39851}]}
{'WIN,LNX': [{'mean(full_price)': 762.6767976278725, 'mean(igdb_score)': 68.92, 'count(*)': 1521}]}
{'MAC': [{'mean(full_price)': 799.0, 'mean(igdb_score)': 81.0, 'count(*)': 5}]}
{'MAC,LNX': [{'mean(full_price)': 499.0, 'mean(igdb_score)': None, 'count(*)': 1}]}
{'LNX': [{'mean(full_price)': None, 'mean(igdb_score)': None, 'count(*)': 1}]}
```

### GROUP

**My cat demands [name, platforms] in paw-session of [steamdb] to be glued together by [platforms]**



```
Enter text (or type 'exit' to end):
My cat demands [name, platforms] in paw-session of [steamdb] to be glued together by [platforms]
Query Result:

{'group': 'WIN,MAC,LNX'}
{'name': 'SUKAKKO', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Earthworm Jim', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Warp Frontier', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Crash Drive 3', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Gwan Moon High School : The Ghost Gate', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Age of Fear 4: The Iron Killer', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Mechajammer', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Bounce Paradise', 'platforms': 'WIN,MAC,LNX'}
{'name': 'HUNTDOWN', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Women of Xal', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Modern Assault Tanks', 'platforms': 'WIN,MAC,LNX'}
{'name': 'Warlords: Age of Shadow Magic Tactical Action RPG', 'platforms': 'WIN,MAC,LNX'}
{'name': "Set 'N Det", 'platforms': 'WIN,MAC,LNX'}
{'name': 'No Longer Home', 'platforms': 'WIN,MAC,LNX'}
{'name': 'WWR: World of Warfare Robots', 'platforms': 'WIN,MAC,LNX'}
```

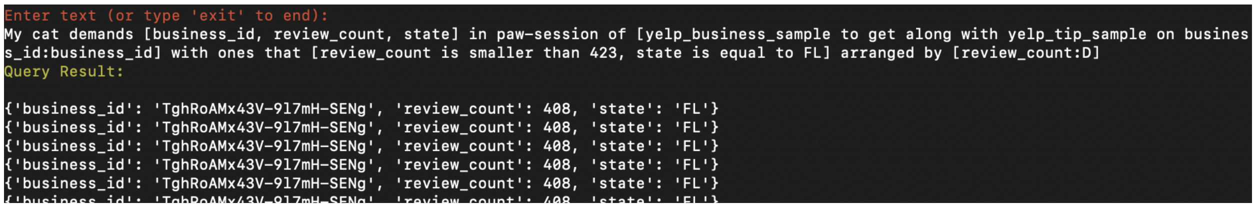


```
{'group': 'MAC'}
{'name': 'Call of Duty: Black Ops - Mac Edition', 'platforms': 'MAC'}
{'name': 'Paul Pixel - The Awakening', 'platforms': 'MAC'}
{'name': 'MobileZombie', 'platforms': 'MAC'}
{'name': 'Escape Code - Coding Adventure', 'platforms': 'MAC'}
{'name': 'Celtreos', 'platforms': 'MAC'}
{'group': 'MAC,LNX'}
{'name': 'Arma: Cold War Assault Mac/Linux', 'platforms': 'MAC,LNX'}
{'group': 'LNX'}
{'name': 'PICNIC', 'platforms': 'LNX'}
```

YELP

**JOIN + MULTIPLE FILTER + PROJECT + SORT(DESCENDING)**

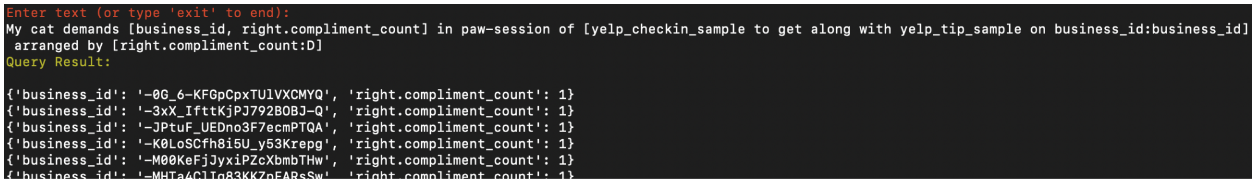
My cat demands [business_id, review_count, state] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on business_id:business_id] with ones that [review_count is smaller than 423, state is equal to FL] arranged by [review_count:D]



```
Enter text (or type 'exit' to end):
My cat demands [business_id, review_count, state] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on busines
s_id:business_id] with ones that [review_count is smaller than 423, state is equal to FL] arranged by [review_count:D]
Query Result:

{'business_id': 'TghRoAMx43V-9l7mH-SENg', 'review_count': 408, 'state': 'FL'}
{'business_id': 'TghRoAMx43V-9l7mH-SENg', 'review_count': 408, 'state': 'FL'}
{'business_id': 'TghRoAMx43V-9l7mH-SENg', 'review_count': 408, 'state': 'FL'}
{'business_id': 'TghRoAMx43V-9l7mH-SENg', 'review_count': 408, 'state': 'FL'}
{'business_id': 'TghRoAMx43V-9l7mH-SENg', 'review_count': 408, 'state': 'FL'}
{'business_id': 'TghRoAMx43V-9l7mH-SENg', 'review_count': 408, 'state': 'FL'}
```

**JOIN + PROJECTION + SORT(DESCENDING)**

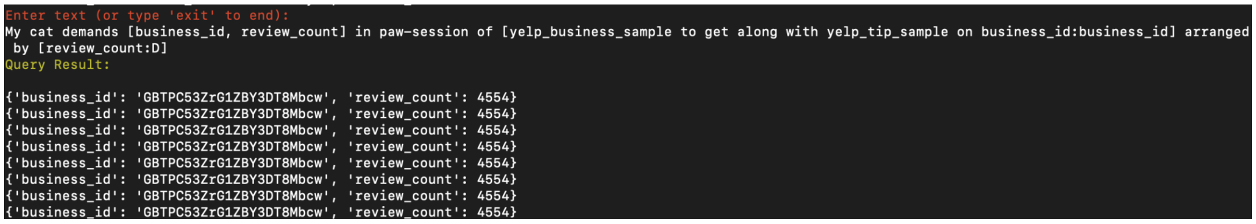
My cat demands [business_id, right.compliment_count] in paw-session of [yelp_checkin_sample to get along with yelp_tip_sample on business_id:business_id] arranged by [right.compliment_count:D]



```
Enter text (or type 'exit' to end):
My cat demands [business_id, right.compliment_count] in paw-session of [yelp_checkin_sample to get along with yelp_tip_sample on business_id:business_id]
 arranged by [right.compliment_count:D]
Query Result:

{'business_id': '-0G_6-KFGpCpxTUlVXCMYQ', 'right.compliment_count': 1}
{'business_id': '-3xX_IfttKjPJ792BOBJ-Q', 'right.compliment_count': 1}
{'business_id': '-JPtuF_UEDno3F7ecmPTQA', 'right.compliment_count': 1}
{'business_id': '-K0LoSCfh8i5U_y53Krepg', 'right.compliment_count': 1}
{'business_id': '-M00KeFjJyxiPZcXbmbTHw', 'right.compliment_count': 1}
{'business_id': '-MHTa4ClIq83KKZpFARsSw', 'right.compliment_count': 1}
```

**JOIN + PROJECTION + SORT(DESCENDING)**

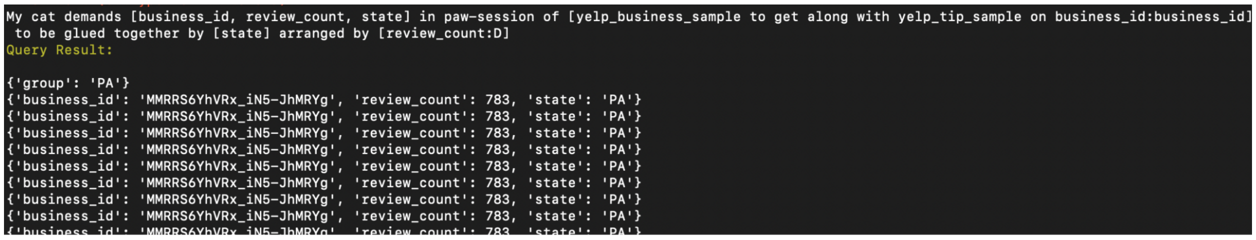
My cat demands [business_id, review_count] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on business_id:business_id] arranged by [review_count:D]



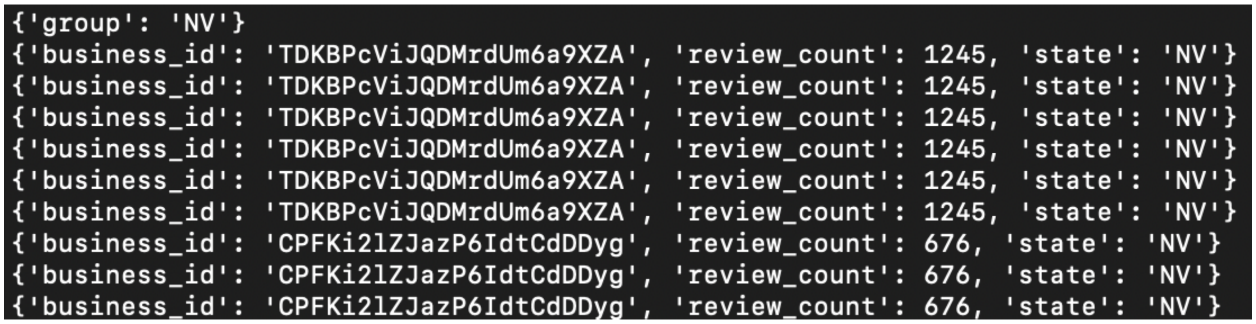
```
Enter text (or type 'exit' to end):
My cat demands [business_id, review_count] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on business_id:business_id] arranged
 by [review_count:D]
Query Result:

{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
{'business_id': 'GBTPC53ZrG1ZBY3DT8Mbcw', 'review_count': 4554}
```

**JOIN + GROUP + PROJECTION + SORT(DESCENDING)**

My cat demands [business_id, review_count, state] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on business_id:business_id] to be glued together by [state] arranged by [review_count:D]



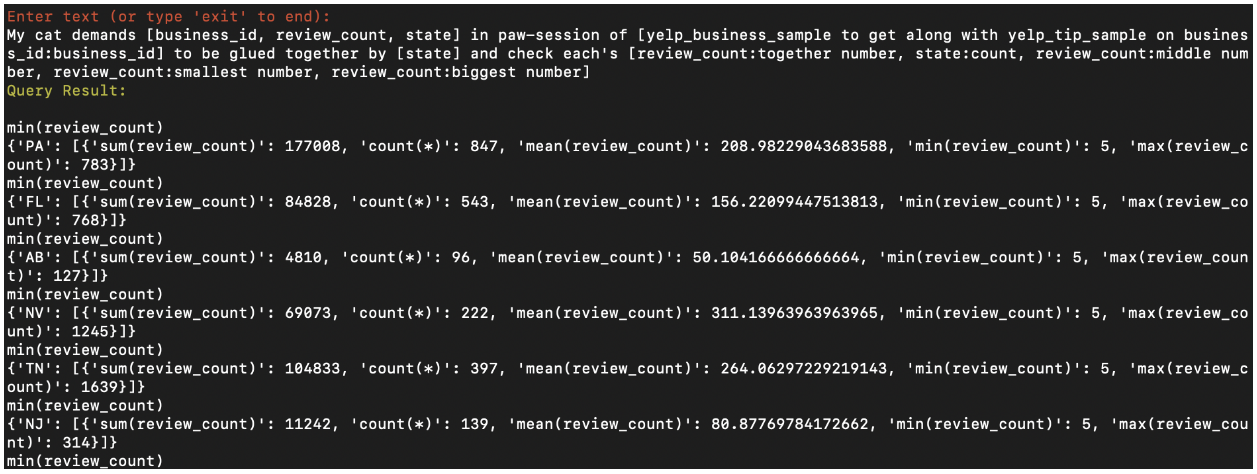
```
My cat demands [business_id, review_count, state] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on business_id:business_id]
 to be glued together by [state] arranged by [review_count:D]
Query Result:

{'group': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
{'business_id': 'MMRRS6YhVRx_iN5-JhMRYg', 'review_count': 783, 'state': 'PA'}
```



```
{'group': 'NV'}
{'business_id': 'TDKBPcViJQDMrdUm6a9XZA', 'review_count': 1245, 'state': 'NV'}
{'business_id': 'TDKBPcViJQDMrdUm6a9XZA', 'review_count': 1245, 'state': 'NV'}
{'business_id': 'TDKBPcViJQDMrdUm6a9XZA', 'review_count': 1245, 'state': 'NV'}
{'business_id': 'TDKBPcViJQDMrdUm6a9XZA', 'review_count': 1245, 'state': 'NV'}
{'business_id': 'TDKBPcViJQDMrdUm6a9XZA', 'review_count': 1245, 'state': 'NV'}
{'business_id': 'TDKBPcViJQDMrdUm6a9XZA', 'review_count': 1245, 'state': 'NV'}
{'business_id': 'CPFKi2lZJazP6IdtCdDDyg', 'review_count': 676, 'state': 'NV'}
{'business_id': 'CPFKi2lZJazP6IdtCdDDyg', 'review_count': 676, 'state': 'NV'}
{'business_id': 'CPFKi2lZJazP6IdtCdDDyg', 'review_count': 676, 'state': 'NV'}
```

**JOIN + GROUP + AGGREGATION + SORT(DESCENDING)**

My cat demands [business_id, review_count, state] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on business_id:business_id] to be glued together by [state] and check each's [review_count:together number, state:count, review_count:middle number, review_count:smallest number, review_count:biggest number]



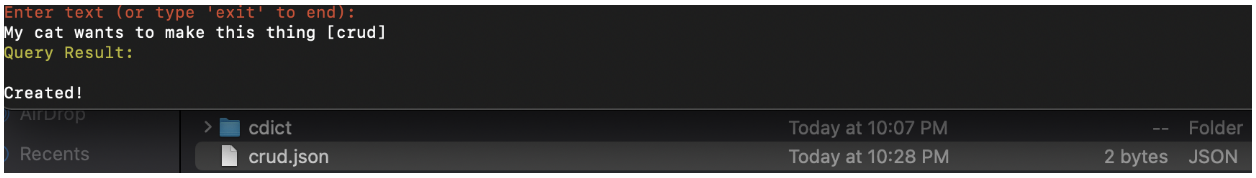
```
Enter text (or type 'exit' to end):
My cat demands [business_id, review_count, state] in paw-session of [yelp_business_sample to get along with yelp_tip_sample on busines
s_id:business_id] to be glued together by [state] and check each's [review_count:together number, state:count, review_count:middle num
ber, review_count:smallest number, review_count:biggest number]
Query Result:

min(review_count)
{'PA': [{'sum(review_count)': 177008, 'count(*)': 847, 'mean(review_count)': 208.98229043683588, 'min(review_count)': 5, 'max(review_c
ount)': 783}]}
min(review_count)
{'FL': [{'sum(review_count)': 84828, 'count(*)': 543, 'mean(review_count)': 156.22099447513813, 'min(review_count)': 5, 'max(review_co
unt)': 768}]}
min(review_count)
{'AB': [{'sum(review_count)': 4810, 'count(*)': 96, 'mean(review_count)': 50.104166666666664, 'min(review_count)': 5, 'max(review_coun
t)': 127}]}
min(review_count)
{'NV': [{'sum(review_count)': 69073, 'count(*)': 222, 'mean(review_count)': 311.13963963963965, 'min(review_count)': 5, 'max(review_co
unt)': 1245}]}
min(review_count)
{'TN': [{'sum(review_count)': 104833, 'count(*)': 397, 'mean(review_count)': 264.06297229219143, 'min(review_count)': 5, 'max(review_c
ount)': 1639}]}
min(review_count)
{'NJ': [{'sum(review_count)': 11242, 'count(*)': 139, 'mean(review_count)': 80.87769784172662, 'min(review_count)': 5, 'max(review_cou
nt)': 314}]}
min(review_count)
```

**NOTE: If you divide sum() and count(), it might lead to different results than mean() because sum() skips missing values while count() does not.**

## SAMPLE CRUD

**CREATE**

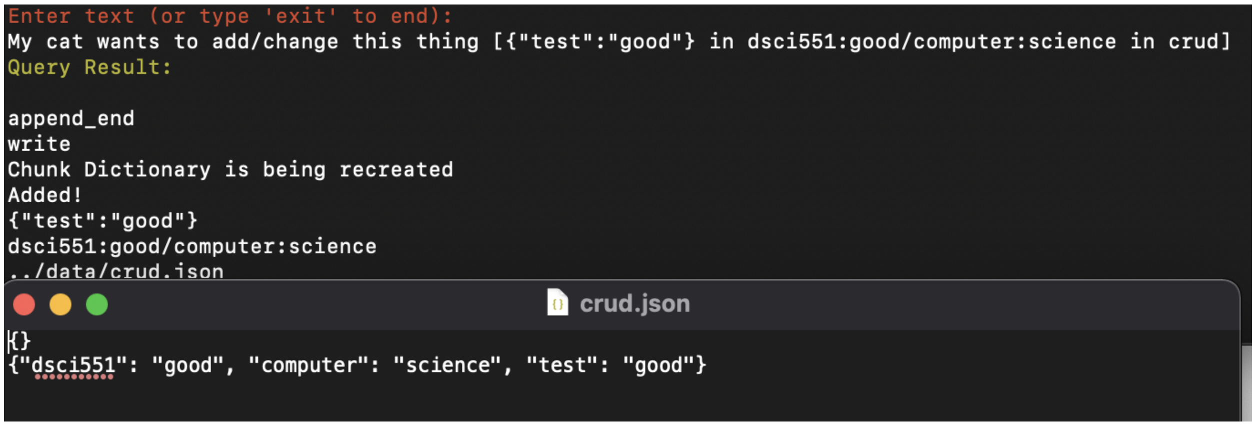
My cat wants to make this thing [crud]



```
Enter text (or type 'exit' to end):
My cat wants to make this thing [crud]
Query Result:

Created!
```

**PUT**

My cat wants to add/change this thing [{"test":"good"} in dsci551:good/computer:science in crud]



```
Enter text (or type 'exit' to end):
My cat wants to add/change this thing [{"test":"good"} in dsci551:good/computer:science in crud]
Query Result:

append_end
write
Chunk Dictionary is being recreated
Added!
{"test":"good"}
dsci551:good/computer:science
../data/crud.json
```

```
{}
{"dsci551": "good", "computer": "science", "test": "good"}
```

**QUERY MANUAL**

Projection: My cat demands

Filter: with ones that[var1 is equal to 10, var2 is bigger than 20]

- = : is equal to
- > : is bigger than
- < : is smaller than
- >= : is bigger or equal to
- <= : is smaller or equal to
- **NOTE: MULTIPLE FILTERING IS VERY UNSTABLE AS OF CURRENTLY**
- **NOTE: MULTIPLE FILTERING ON SAME VARIABLE DOESN'T WORK AS OF CURRENTLY**

Scan: in paw-session of [JSON_file name without the .json extension]

- Join: to get along with
- for example: in paw-session of [JSON1 to get along with JSON2 on join_var1:join_var2]

Group: to be glued together by [group_var]

- **NOTE: ONLY SINGLE VARIABLE GROUPING AS OF CURRENTLY**

Aggregate: check each [var1:together number, var2:count, var3:smallest number]

- max: biggest number
- min: smallest number
- count: count
- mean: middle number
- sum: together number

Order: arranged by [var:D or var:A]

- :D for descending, :A for ascending
- **NOTE: ONLY SINGLE VARIABLE SORTING AS OF CURRENTLY**

CRUD: My cat wants to
**ASSUME URL is in the form of "key:val/key:val/..."**
**ASSUME DATA is a valid form of JSON in the form of {"key": "val"}**
GET: steal this thing [URL in JSON_fileName]
PUT: add/change this thing [DATA in URL in JSON_fileName]
POST: slip in this thing [DATA in URL in JSON_fileName]
DELETE: destroy this thing [URL in JSON_fileName]
Create: make this thing [JSON_fileName]

## Learning Outcomes

From this project, I really get to understand first-hand algorithms that use chunks instead of loading into memory. This really solidified my understanding of external sorting algorithms, 2-pass algorithms (grouping), mapReduce (aggregation), and query execution engines. This is very useful for understanding how database management systems like Hadoop, MySQL, MongoDB works under the hood.

Furthermore, I really solidified my Python programming skills and especially with the use of Python generators. It allowed me to understand and become extremely familiar with the limitations and usefulness of Python generators.

## Challenges

The first big challenge was chunking JSON files. I knew I couldn't just randomly chunk every 10,000 bytes. This will lose the essential structure of the JSON file. Plus, I couldn't keep my chunks to around 10,000 bytes if I read every 100 lines since every line will contain different amount of bytes. I finally made the idea of a chunk dictionary after realizing that every valid JSON object ends with a newline delimiter in the NDJSON format which my database system supports. This quickly solved my problem.

Second, sorting is one of the most difficult algorithms to wrap my head around. It's only after really rereading the lecture and information on the algorithm that I could write the algorithm. Still, after thinking about a diagram, I realized I could use a priority queue to quickly implement an output buffer.

Finally, the hardest part in my opinion was to find out how to do query execution. The two parts that got to me was how to use generators and plug them into each other, and how to control the flow of execution. For the former, I realized that by iterating through the chunks in each of the functions, I could simply treat the functions as a gate that only allows some chunks or parts of the chunk to flow through. Also, to avoid loading everything to memory, if I require it, I will write everything into the disk. For the latter, I tried many things including using the iterator model the lecture taught us. But then I realized that the query generally follows 4 paths and that depends entirely on grouping and aggregation. I then realized binary flags would quickly help with this issue.

## Conclusion

The project was both a really difficult task and a really fun one. I really enjoyed all the brainstorming and ideas I could try to implement, only to see if it would work or not. Honestly, it gave me confidence that I could build anything I want to as this might have been the most complex thing I have built.

Furthermore, the project really solidified my understanding of sorting and chunk-based algorithms. Avoiding loading everything into memory was really difficult, but really rewarding to learn. It really makes me realize how different an algorithm can be if you don't load everything into memory.

In conclusion, the project was a great way to practically apply and solidify my understanding of these algorithms I learned in class. It is definitely a project I'm very proud of.

## Future Scope

I really did not have the time to explore implementing a good UI. I think the main thing to explore in the future is interactive GUI or displays of the datasets, changing in real time.