Efficient SQL-Like Queries on .csv Files

1. Introduction and Background

Databases are a vital part of modern technology. Any company, group, or service interacting with data uses some sort of data store or database. As such, it is of much interest to interact with databases in an efficient manner. Data can be stored in a variety of formats, but commonly in .csv, which is a plaintext file consisting of individual records, separated by newlines, made of fields, which are separated by commas. Each field of a record corresponds to some column of a database, and the value in that record is the value in the row.

It is of great importance to data scientists to be able to query data for information of all kinds, such as counts, averages, and other various aggregations to gain info about the structure/characteristics of the data. Commonly, this involves loading data files (.csv, .json) into a database, with such databases as MySQL, SQLite, etc. Once loaded into a database, scientists can interact with the data using SQL, a query language.

There are a couple major drawbacks in using existing database systems for data science/analysis. For one, database representations of data-files are large and may take up many gigabytes of disk space. This is unwieldy for a single data scientist to create and keep locally. As well, these databases can take a very long time to create from a given data file, reaching potentially hours or days for large, many tens of gigabytes .csv files. On top of the time to create the database, data scientists are also held to maintain/update the database, and as well, interact with it via its (potentially complex) frontend in creating their queries and managing schemas. It is thus desirable to come up with a way for data scientists to interact with plaintext data files without the complications of loading data into a database.

This project is an implementation of a method to use familiar SQL query operations/syntax directly on .csv files, bypassing the cost of storing a unique database on disk (all operations are done in memory), the time cost incurred in building the database (all operations are done in-place on the file,) and the time interfacing with and maintaining a database.

The implementation is designed to be easy to use, simply calling ./select or ./group on a file, and the query in SQL-like syntax. The individual programs can be easily piped together for more complex queries.

Ideally, this method is faster, has a smaller memory footprint, and is easier to interact with than existing implementations for the same idea; mainly, csvkit and Pandas.

2. Related Work

Attempting to support SQL operations by simply interfacing .csv files with SQL-like language is not entirely new. There are a few key implementations facing this task, but all have shortcomings that I wish to address with my design.

One of these implementations is ‘csvsql’, (https://csvkit.readthedocs.io/en/latest/) a tool program part of a larger suite of tools called ‘csvkit’ aimed at making interacting with .csv files easier in a variety of ways. I initially discovered csvsql as a direct comparison, a competitor to benchmark my implementation by. As it turns out, however, csvsql is only a wrapper for the complex database suite SQLite, and the internal mechanism is not similar to my implementation; it simply translates your command-line query to SQL queries, and creates a SQLite database with your given file, only giving the illusion of a database-free system. Below, you can see that the time to run queries on a set of large files is unwieldy for any data scientist working on these types of files. Note especially that at 10GB and beyond, my PC (64GB RAM) did not have enough memory to create the database.

1GB	210s
5GB	1287s
10GB	Killed
15GB	Killed
20GB	Killed

Another proposed tool is Rainbow Query Language, another attempt at taking SQL-like language and working on .csv files. However, in my attempts to run queries with the tool, it turns out it does not support .csv files larger than 1GB.

Finally, the existing method that I chose to benchmark my approach against is Pandas. Pandas is a Python library for data scientists, and is far and away the most popular tool for data science today. Being so popular, I felt it was a natural benchmark, if my argument is to create the best tool for a data scientist to use on large .csv files, I wanted to compare against how it is currently done. As well, it is closer to the implementation style of my programs, in that it does not create an on-disk database representation. It does, however, create a representation in memory. The stipulation to using Pandas as a comparison, however, is that Pandas does not expose a SQL-like grammar for constructing and running queries. Instead, data scientists must learn Pandas syntax, Python syntax, and use a combination of Pandas methods and Python implementation techniques to achieve the same results (a guide on SQL-like programming in Pandas, and how I learned to run my tests, is found here: https://pandas.pydata.org/docs/getting_started/comparison/comparison_with_sql.html).

Nevertheless, throughout this document, Pandas is my benchmark. Pandas does, importantly, have an upfront cost to build an in-memory representation for the course of the session, so it does not completely ignore the cost of a database; it just uses a light implementation of one. The upfront time costs to build a database are listed below for Pandas:

1GB	20.1s
5GB	90.3s
10GB	175.3s
15GB	Killed
20GB	Killed

On my 32GB machine, it will not build files larger than ~10GB, Killing a process when loading a 15GB file. The workaround for the comparison, should a user want to process larger files, is to iteratively process chunks of a .csv file, using a for loop. Pandas can build individual databases over these chunks, leaving it up to the programmer to implement their own aggregation methods for the chunks. This, again, is not convenient for a data scientist, as mentioned earlier.

3. Proposed Technique

There are two implementations for the methodology: a SELECT program, to perform SELECT operations on .csv files, and GROUP, a program to perform GROUP BY operations on files. Both are implemented in C.

3.1. SELECT Implementation

3.1.1. Overview

The SELECT implementation, at its core, follows this process:

- Process provided SQL query into Abstract Syntax Tree using *bison*
- Spawn N threads to process (size)/N chunks of the .csv file in parallel
- For each line, walk the query AST, evaluating boolean comparisons
- If the resulting query is a boolean TRUE, use the line (e.g. print)

The query is passed to the program in text. A custom parser, written using *flex* and *bison*, converts the passed query into an Abstract Syntax Tree (AST), with a defined grammar representing a subset of the SQL language, supporting SELECT queries of the form “SELECT * FROM table WHERE condition(s)”.

Once this internal query AST is made, the given file is split into N equal chunks (over lines, so not perfectly equal,) for parallel processing. Each thread will read a series of lines. The core functionality of the program is to walk the AST, referencing fields in each row, and evaluating the boolean expressions made by the query. Once done, all the nodes of the AST for that line will have been filled in, and if the

outermost boolean value is TRUE, the line is deemed "valid" by the query, a.k.a., it is to be returned by SELECT. Here, it can be processed, such as printing to stdout.

The optimal value for N, the number of threads to spawn, was determined experimentally, and is shown below.

3.1.2. Code Listings/ Specifics

main calls the flex/bison functions to generate the parser for the specified query, then calls the wrapper function *main_driver*.

main_driver does the following:

1. Opens the .csv file
2. Calculates file metadata information, such as chunk size given N threads
3. Creates the "type_table" used to determine the variable type of all columns in the header
4. Spawns N threads, and calls the function *thread_function* with [a,b] byte offset regions in file specified to process on

thread_function is responsible for calling all relevant processing instructions. Each thread runs *thread_function* to process the passed [a, b] file offset region in the file.

1. Gets a new file descriptor for the .csv file
2. Seeks to byte offset *a*
3. For each line in its region:
    a. Builds a "row lookup table", caching the values located at each column of the line into an array for fast access
    b. Evaluates the AST by calling *AST*, passing the thread's unique row lookup table
    c. Processes line if it is indeed valid to the query

get_nl returns the byte address (off_t) of the next newline given a current file pointer (FILE*).

entry_comparison evaluates boolean expressions comparing two given entries from the AST.

determine_type determines if a given string literal is a int, float or string and returns the corresponding type enum.

build_row_lookup_table iterates through a passed .csv line and populates an array of Row_Entry*, where each Row_Entry is given its int, float or string literal value, alongside with a type determined by passing its field to *determine_type*. The point of a row lookup table, as said above, is to cache all fields of a line and their types.

AST is the heart of the implementation, and recursively walks the binary tree of Node(s) generated by the flex/bison functions. Don't worry about those, just know that the result of running flex/bison is a Node* to the start of an AST. *AST* does different things depending on if the node is one of three types, representing boolean expressions, actual literals/numbers, or logical AND, OR. At the end, the Node(s) store "retvals", the boolean truthiness of their expressions.

3.2. GROUP BY Implementation

3.2.1. Overview

The GROUP BY implementation follows this process:

- Spawn (size)/N thread to process the file in chunks, in parallel
- For each line, first compute the unique aggregation key, then insert the line into a hash table based on this key
- Perform necessary aggregation(s), some of which are done on-the-fly, like SUM or COUNT

One key performance hotspot is the extraction of unique keys from the lines, based on user-specified aggregation columns. GROUP BY queries work by grouping multiple column attributes together. Thus, the implementation uses the concatenation of an individual row's aggregation-column values as a key, and inserts the row itself into the hash table with this key.

The hash table implementation uses linear probing to resolve collisions, and is initialized to a large size at the beginning to avoid resizing (currently.) The values stored at the buckets are dynamic arrays storing individual lines that hash to that group key.

Aggregations are done either during processing, such as keeping up a COUNT or a total SUM, while some are done at the end, such as taking an AVERAGE.

The optimal value for N, the number of threads to spawn, was determined experimentally, and is shown below.

3.2.2. Code Listings/ Specifics

main does the following:

1. Opens the file, collects size metadata for chunk sizes/threads
2. Reads the contents of the entire file into an array
3. Spawns N threads to do processing on [a, b] newline-aligned byte offset segments
4. Hash-inserts all lines into the hash table

createHashTable creates a hash table object (HashTable*).
hashFunction is a simple hashing function

hashInsert inserts a new line into a hash table. Given the grouping key created by *getUniqueKey*, it either adds it to a new bucket using linear probing or inserts it into a previous one.

getCols acquires the literal values of each specified aggregation column for the GROUP BY query, to be used by *getUniqueKey*.

thread_function1 is what the threads run. It parallelizes the intensive process of extracting columns with *getCols* and creating a "unique key" for each line. It stores these values in the fileArray created in *main*.

4. Evaluation

4.1. Experimental Testbed

Processor: Intel(R) Xeon(R) CPU E5-2680, 28 cores
Memory: 256GB
Storage: Intel® Optane™ SSD 900P

4.2. Workloads

4.2.1. Generation of Workloads using TCP-H

TCP-H is an industry-standard database benchmarking tool, which provides the tool "dbgen" for generating benchmarking tables for database operations. For testing my implementations, I utilized the dbgen tool to generate the "lineitem" table, and perform all queries on this table. The Appendix lists all necessary information to create this table using the dbgen tool.

4.2.2. SELECT Workloads

Below are all the test queries I used to test the SELECT implementation. These are all ready to run with the select program. The testing purpose of each group of queries is listed below, for different result characteristics.

```
// Trying to strain the AST
((l_shipmode == TRUCK) OR ((l_shipmode == MAIL) AND (l_quantity > 33))
((l_shipmode == TRUCK) OR ((l_shipmode == MAIL) AND (l_quantity > 33)) OR ((l_shipmode != TRUCK) AND ((l_shipmode !=
MAIL) AND (l_quantity < 33))
(((l_shipmode == TRUCK) OR ((l_shipmode == MAIL) AND (l_quantity > 33)) OR ((l_shipmode != TRUCK) AND ((l_shipmode !=
MAIL) AND (l_quantity < 33))) AND (l_partkey >= 60000 AND l_extendedprice >= 45000)

// Return many rows
l_orderkey > 1 // the entire table
l_orderkey >= 500000 OR (l_shipmode == TRUCK OR l_shipmode == MAIL OR l_shipmode == RAIL OR l_shipmode == FOB OR
l_shipmode == REG AIR) // also the entire table due to the second condition

// Return few rows
l_orderkey >= 50000 AND l_orderkey <= 50500
```

4.2.3. GROUP BY Workloads

Below are all the test queries I used to test the GROUP BY implementation.

```
// Essentially sort the entire table; no aggregation
GROUP BY l_orderkey

// Aggregation
GROUP BY l_shipmode, AVG(l_extendedprice) // average price of each shipping mode (AIL, RAIL, etc)
GROUP BY l_shipdate, SUM(l_quantity) // total number of units shipped for each day

GROUP BY l_shipmode, l_shipdate, l_extendedprice, l_discount, AVG(l_quantity) (4 columns)
```

4.3. Comparison of Multithreading Performance

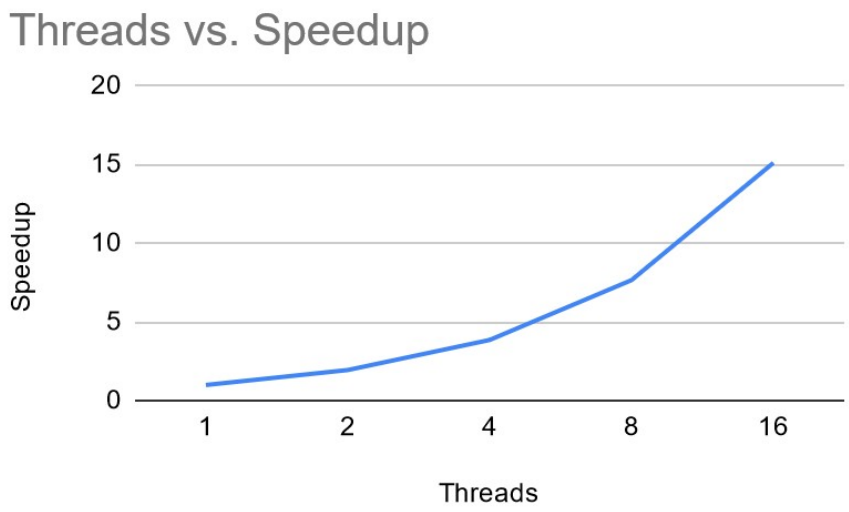
Of interest to me while developing the implementations was the performance of the multithreading solutions. Using multithreading helped me squeeze out maximum performance, and it was interesting to

see how each utilized the cores on the machine. I found that SELECT scales linearly with the number of cores used, while GROUP BY's implementation has a hard limit (8 cores on my 28-core machine.)

4.3.1. SELECT

As discussed above, SELECT spawns N threads to operate on disjoint chunks of the .csv file, each doing IO and processing. Thus, the processing is embarrassingly parallel, and a linear performance increase w.r.t. number of threads is expected. Indeed, this is achieved. For the benchmark, I performed reads only (no output writes) on a 10GB file, and ran the experiment with an increasing number of threads from 1 to 16 in powers of two. Note that I stop at 16 because my testbed has 28 cores. Here are the results, tabulated and shown in a graph:

Scaling # Threads, 10GB File, No Writes		
Threads	Execution Time (s)	**Speedup**
1	110.3	1
2	56.5	1.95
4	28.6	3.86
8	14.4	7.66
16	7.3	15.11



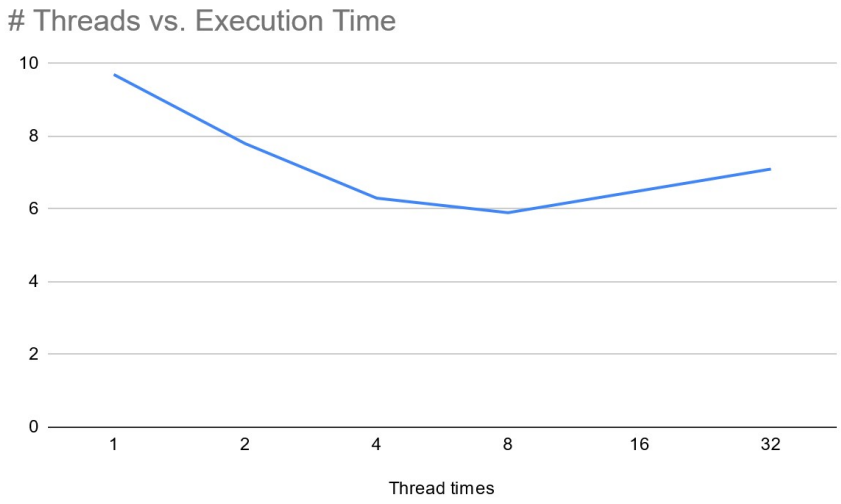
As you can see, the trend is roughly linear, where the biggest rate of change is 8-16. This is good performance, and really shows the power of multithreading; for every core used, the file is read that many times faster than 1 core. The ability for the SELECT implementation to read a 10GB file and do all of the complicated processing in less than 5 seconds is seriously impressive.

As a final note, there is no locking in this implementation. Each thread gets its own disjoint byte range to operate on the file, and have unique data structures to mutate, notably the row_lookup_table and AST, which each thread writes to a local copy of. Avoiding locking helps the performance time quite significantly.

4.3.2. GROUP BY

As discussed above, the implementation for GROUP BY's multithreading is slightly different from that of SELECT's. The processing done in parallel is not entirely disjoint; memory allocations need to happen in parallel on the data structures, and the hash table structure is shared. As a result, I didn't expect it to be an embarrassingly parallel task and to scale linearly. The results show that the performance increase with multithreading is not nearly as significant as with SELECT's implementation. The following table and graph shows a test done with an increasing number of threads up to 16, on a 2GB file, using an arbitrary query.

# Threads	Execution Time	Speedup
1	9.7	1
2	7.8	1.24
4	6.3	1.54
8	5.9	1.64
16	6.5	1.49



As you can see, the increase in performance is not even 2x, and does not increase linearly, instead bottoming out at 8 threads. Whether this is due to my implementation, or the nature of the task, is a question I was unable to answer, after a lot of investigation and trying different memory allocation/other strategies to reduce this disparity.

4.4 Comparison against previous techniques

To benchmark the implementations, a sequence of data files was generated using the TCP-H benchmark table generator. Specifically, the **lineitem** table is used. Data files are generated such as 1GB, 5GB, 10GB, 15GB, and 20GB.

The main point of comparison against the C implementation is csvkit, a wrapper for SQLite, and Pandas, the most common data science library for querying .csv files directly.

4.4.1. C Implementation Compared

To benchmark the C implementation, I created a set of test queries and ran them both on the C implementation(s) as well as an equivalent implementation in Pandas. The test queries are listed below with a comment indicating their testing purpose.

4.1.2. SELECT

I compare these queries to Pandas implementations. The first results show a query set that returns a smaller portion of the table. The second results show the effect when the entire table is returned.

Pandas

1GB	1.12s
5GB	4.72s
10GB	9.6s
15GB	21.4s
20GB	46.6s

C

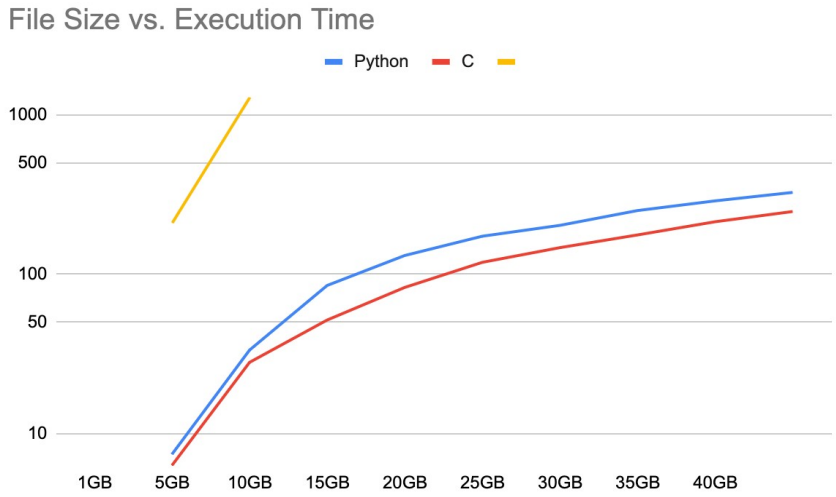
1GB	0.62s
5GB	2.6s
10GB	4.6s
15GB	8.5s
20GB	11.8s

Pandas

1GB	2.6s
5GB	11.9s
10GB	23.2s
15GB	42.5
20GB	58.2s

C

1GB	0.57s
5GB	2.2s
10GB	4.3s
15GB	6.7s
20GB	10.2s



The results of the above tests show that the C implementation is on average about 4x faster than the Pandas implementation. These are really good results, especially when considering that this is ignoring the aforementioned upfront cost from Pandas to read the .csv file first. Even in the 5GB case, this adds a minimum of 90.3s to each operation. Also note that the more of the table is returned, the longer Pandas takes, whilst this has no effect on the C, as it will process the entire table regardless. This is because each line must be looked at; the result of the AST must be evaluated.

4.1.2. GROUP BY

Before comparing it to Pandas, I wanted to test the impact of increasing the number of columns on the GROUP BY queries for the C implementation. The C implementation showed an increased execution time as the number of columns increased:

One column

1GB	4.5s
5GB	19.8s
10GB	41.2s
15GB	73.8s
20GB	93.6s

Four columns

1GB	4.9s
5GB	21.8s
10GB	45.2s
15GB	79.8s
20GB	103.6s

Compared to the Pandas implementation, we can see that it, too, increases the execution time when the number of columns increases in the aggregation; note that Pandas also performs worse than the C implementation. However, it is less of a blowout than SELECT:

Python – 1 column

1GB	4.4s
5GB	22.1s
10GB	45.7s
15GB	79.3
20GB	103.2

C – 1 column

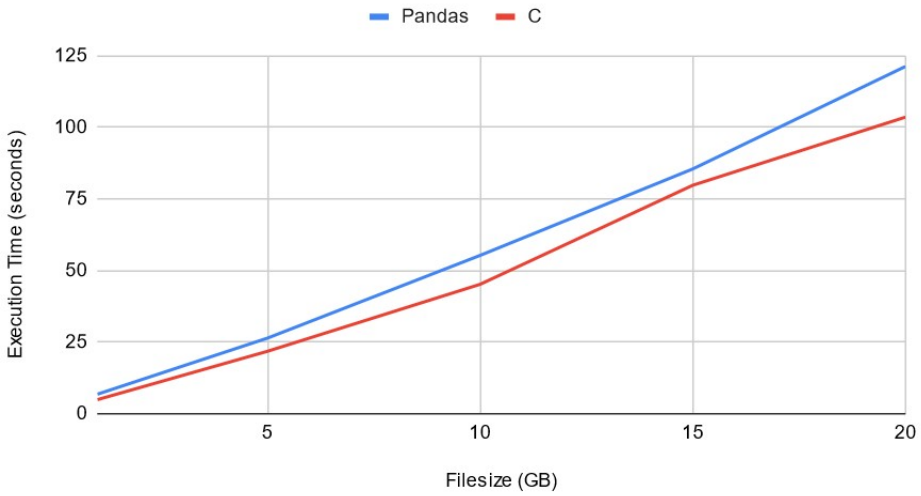
1GB	4.5s
5GB	19.8s
10GB	41.2s
15GB	73.8s
20GB	93.6s

Python – 4 columns

1GB	6.8s
5GB	26.4s
10GB	55.3s
15GB	85.5s
20GB	121.3s

C – 4 columns

1GB	4.9s
5GB	21.8s
10GB	45.2s
15GB	79.8s
20GB	103.6s



Next, I created a set of queries to stress test the hash table, i.e., to inflate the number of unique groups to be hashed. I wanted to test the effect this would have on the C; the Pandas implementation does not use a hash table.

The C numbers:

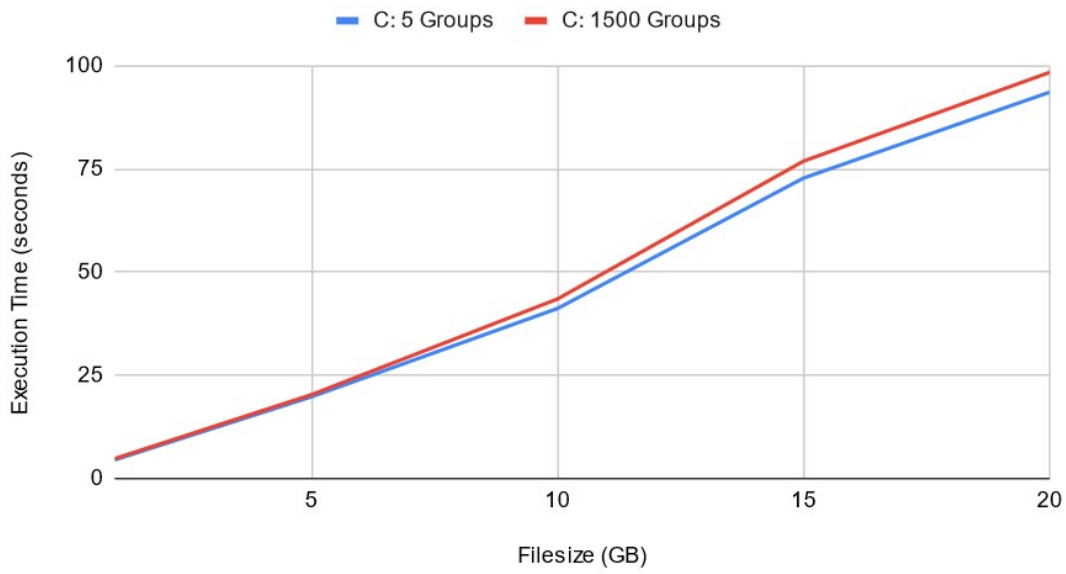
5 groups

1GB	4.5s
5GB	19.8s
10GB	41.2s
15GB	72.8s
20GB	93.6s

1500 groups

1GB	4.8s
5GB	20.3s
10GB	43.5s
15GB	76.9s
20GB	98.4s

You can notice that it does impact the C performance to a noticeable degree.



4.1.3. SELECT and GROUP BY Combined

Finally, to conclude testing, I made a full test suite combining a SELECT and GROUP BY query into one, to simulate a full data science query. This was done by piping the output of first running the “select” program into the “group” program. The results:

(l_shipmode == TRUCK) AND (l_quantity > 5) GROUP BY l_partkey, SUM(l_extendedprice) // For all units shipped via truck with at least 5 parts, sum up the total price for each distinct part

	Python
1GB	7.14s
5GB	34.2s
10GB	81.7s
15GB	127.3s
20GB	169.3s

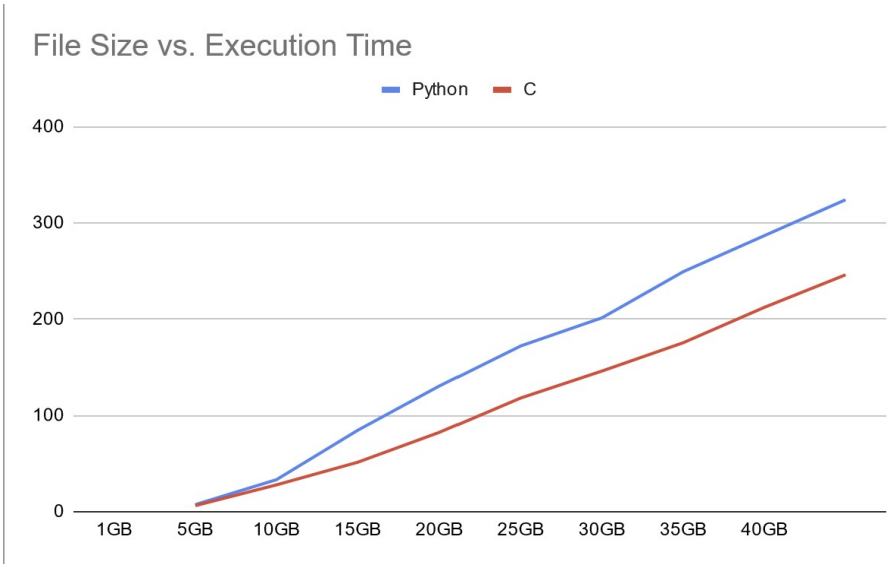
	C
1GB	6.28s
5GB	27.6s
10GB	52.3s
15GB	81.5s
20GB	108.1s

(((l_shipinstruct == TAKE BACK RETURN) OR (l_shipinstruct == DELIVER IN PERSON)) AND l_extendedprice > 50000) GROUP BY l_shipinstruct, AVG(l_discount) // If the delivery method was return or deliver, and the price was over 50k, calculate the average discount for each delivery method

	Python
1GB	8.42s
5GB	38.5s
10GB	92.5s
15GB	152.9s
20GB	198.8s

	C
1GB	6.7s
5GB	31.6s
10GB	63.3s
15GB	96.4s
20GB	133.3s

The results indicate that the C implementation performs better than the Pandas method, and as the size of the file grows, the disparity too grows in the C's favor. In the 20GB case, the most extreme difference is shown, where the Pandas can take up to a minute longer per query. Note, again, that this is ignoring the many minutes Pandas needs to load the file.



5. Conclusion and Future Work

5.1. Conclusion

Over the course of this document, I have outlined the problem statement, evaluated what methods exist to solve the problem, shown the implementation architecture of my approach, and demonstrated the results compared to the existing approach Pandas.

The result is that my implementations, both SELECT and GROUP BY, outperform the contemporary implementation Pandas, on a per-query basis, where all queries take less time. In the case of SELECT, the speedup is almost 4x in the largest file cases, and even 2x at the least. This indicates my tool provides a monumental benefit for data scientists doing SELECT queries on large .csv files. In addition, this ignores the extremely large cost of reading the file itself by Pandas, the cumbersome task of writing the Python code to perform the operations, and aggregating results. Over all, my SELECT implementation is much faster, and far simpler to use, than Pandas. GROUP BY achieves similar results, performing around 1.5x-2x better than Pandas per query.

The result is that my implementations provide a much more performant and easier to use set of tools for SQL-like queries than the existing methods such as Pandas and SQLite.

5.2. Future Work

5.2.1. Improving Implementations/Bug Fixes

The implementations, in their current state, would benefit from a refactor. A lot of work can be done on both implementations to present the code cleaner, to make variable/function names more recognizable in scope, etc.

A key performance issue in the implementations lies in the GROUP BY implementation; the question of why the parallel performance does not scale linearly with time (or really at all) needs to be addressed, as it currently bottoms out at 8 threads with only a 1.7x improvement, as seen above. In addition, supporting a dynamically resizing hash table is essential when the number of groups is truly large and hard to estimate.

5.2.2. Extended Functionality

A potential area for future work on the project is to extend the functionality of the suite of tools by providing more SQL-like operations. One such operator is a “summary” operator, which provides a comprehensive description of the data in the .csv file, presenting aggregation information like mean, median, mode, min, and max for each column (if the column’s data is numeric) or a simple histogram (for string-based data). Another option for the toolkit could be a “sql” program, which takes as input an actual SQL query (only supporting SELECT, GROUP BY, and JOIN), and converts it to the command-line pipeline of operators in this toolkit, to help data scientists convert real SQL queries to this implementation’s pipeline.

Another useful feature of the tools would be more robustness in handling .csv files which are ill-formed, able to handle invalid rows (skip them,) or incorrect delimiters, etc. It would also be helpful to perhaps write a tool to *fix* the issues in the .csv file, to prepare it for correct processing by the tools.

5.2.3. Extended Range

To develop this toolkit further with the goal of processing extremely large files, on the order of hundreds of GB or even terabytes in size, one would need to move to a distributed computing environment to distribute the work over pieces of the file over a network. Existing frameworks like Hadoop easily manage the networked file storage, which does all the work under the hood, and allows the commands to be implemented and used. The implications of distributing the work do not impact SELECT, due to its embarrassingly parallel nature, but careful consideration would have to be made with GROUP BY due to the aggregation step and the requirement of combining results.

Another potential direction is to implement a way to view file system directories as tables, using our tools to view directories as "tab separated files" in order to obtain metadata about directories, allowing easier use and understanding of a large collection of files.

Appendix

Build Instructions

To build SELECT:

1. Build the flex: `flex –header-file=lex.yy.h ./select.l`
2. Build the bison: `bison -d ./select.y`
3. Re-build the flex: `flex –header-file=lex.yy.h ./select.l`
4. Compile the program, linking the pthread library as well as the lex functions: `gcc -pthread select.c select.tab.c lex.yy.c -lfl -o select`

To build GROUP BY:

1. Compile the program, link the pthread library: `gcc -lpthread group.c -o group`

I usually do `-Ofast` for both, but it is supposedly unstable. I still do it and have never had problems.

Testing Instructions

TCP-H/dbgen

In testing this, I have used the TCP-H benchmark generation tool for generating my testing tables. If you go to the TCP-H website and download the benchmark, you will download a directory containing the sub-directory "dbgen."

Go to the `dbgen` directory, and build it with `make`.

`dbgen` generates tables. I use the “lineitem” table. To generate this with a size of 1GB: `./dbgen -T L -S 1`. This calls `dbgen`, sets the table (`T`) to lineitem (`L`), and sets the “scale factor” (`-S`) to 1GB.