CX4123 Big Data Report

Meenachi Sundaram Siddharth: U1923790H
Arora Srishti: U1922129L
Acharya Atul: U1923502C

April 2023

Name	**Detailed Individual Contribution**	**Percentage (100% in total)**
Acharya Atul	Filter, Binary Search, Aggregation Functions, Group By, Projection	33.33%
Arora Srishti	Filter, ZoneMap Filtering, Query Preprocesing	33.33%
Meenachi Sundaram Siddharth	Block Abstraction, File I/O Handling, Query Processing	33.33%

Contents

1 Data Storage **3**
1.1 Design of Column Store . . . 3
1.2 File I/O operations . . . 4
1.3 Exception Handling . . . 4

2 Data Processing **5**
2.1 Column Scanning for Query Processing . . . 5
2.2 Minimum and Maximum Aggregation . . . 6
2.3 Enhancements . . . 6

3 Experiment Results **7**
3.1 Program Output and Evaluation . . . 7
3.2 Analysis . . . 7

1 Data Storage

1.1 Design of Column Store

To implement the storage of the data in a column-store format, we choose to separate the data in the various different columns into different files, each representing one column. These files are assumed to be on a simulated disk and can be read from and written only at a block-level granularity. More details on this will follow in the section on reading from and writing to files. We attempt to make as many optimizations as possible during storage to improve the efficiency of the system. Following are the detailed descriptions of how the various columns are stored on (simulated) disk:

1. **Station**: We observe that the two possible values for the station are either "Changi" or "Paya Lebar", corresponding to the 2 sites of observation. A naïve method of storing this column would be to use a fixed-length string of **10 characters (10 bytes)** since the maximum length of this field would be 10 characters for "Paya Lebar". However, this results in a waste of space. In order to optimize the processing, as well as to compress and save on storage, we choose to encode the values as follows:

   (a) Changi: 0

   (b) Paya Lebar: 1

   In this manner, just 1 **bit** is required to store each entry of this column, resulting in a **40:1 compression ratio**.

2. **Timestamp**: We choose to encode the provided timestamps into the Unix timestamp format, which occupies **8 bytes (64 bits)**. For the given data, there are possibilities to further compress the entries in this column; however, in the interest of **efficient query processing**, as well as to account for **easy scalability**, we have chosen the Unix timestamp format. In particular, this allows for easy conversion to a local timestamp data type, which allows for much more efficient query processing. Also, any encoding based on the limited time range will not allow for the integration of future data.

   Exploration: One encoding we explored for storing timestamps was to split the timestamp columns into four separate columns for: year, month, day, and time. We encoded these 4 columns as follows:

   (a) Year: Since it was limited between 2002 and 2021 (20 unique values), we used 8-bit integers (1 byte each) to encode the year.

   (b) Month: We stored months unmodified in 8-bit integers (1 byte each).

   (c) Day: We stored days unmodified in 8-bit integers (1 byte each).

   (d) Time: We stored time as 16-bit integers (2 bytes each) where it represents the number of minutes elapsed since midnight. We choose this because some data points are not exactly recorded at half an hour intervals.

   The reason we did not proceed with this system design choice was that although this occupies less space:

   (a) The solution will not easily extend when the data goes big and several more years of data are added.

(b) If the second-level granularity is required in the future, the time field cannot support it.

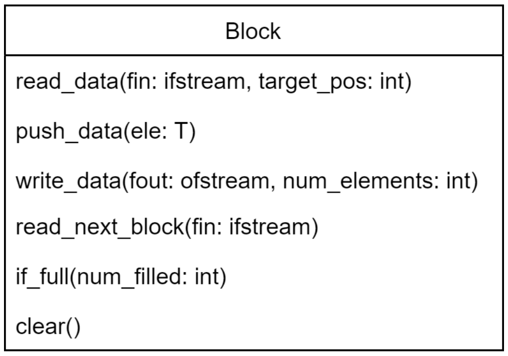
(c) For query processing, the "GROUP BY" operation on the column store could be much more elegantly implemented when simultaneous access to both the year and month is possible. If not, extra disk I/Os would be incurred.

3. **Temperature**: We choose to store the temperatures directly as **floating-point values**, which each occupy **4 bytes (32 bits)**. This is due to the output requirement imposed, where it would not suffice for us to round to the closest integer, and we are required to display the precise value.

4. **Humidity**: We choose to store the humidity in an identical manner to the temperature, i.e., as **floating-point values**, due to the same reasoning.

1.2 File I/O operations

In order to simulate a realistic big data system, file I/O operations are implemented using a block abstraction, whereby files can only be read from or written to via a block. On the right is a simple class description of the Block class that we designed. This design is made **generic** to blocks holding any data type through the use **templates** in C++. In particular, for a given block size and a data type, this class will automatically determine the number of elements of that particular type that can be read and stored in one block.



```
Block
read_data(fin: ifstream, target_pos: int)
push_data(ele: T)
write_data(fout: ofstream, num_elements: int)
read_next_block(fin: ifstream)
if_full(num_filled: int)
clear()
```

This design works directly for all columns that are required to be stored in our context except for the "Station" column due to our design choice to encode them using just 1 bit, which is smaller than 1 byte. Since memory (and disk) is only byte-addressable, 8 entries of the "Station" must be grouped and stored together in 1 byte. This is handled by using **template specialization** of the relevant Block methods for the case when the data type is **bool**, as is the case for the "Station" column. In this manner, we truly obtain the 40:1 compression ratio we set out to achieve by encoding each entry of this column as just 1 bit.

At every stage of processing a query, values from the columns are read from the corresponding files with only a block-level granularity. Also, any qualified positions which are obtained are stored in a **block-sized memory buffer** which when filled, is outputted to a file. During the next stage of processing, the qualified positions resulting from the previous stage are read from the respective file. As a result, our design is scalable to the scenario when the data becomes so large that even the qualified positions at a step of the query processing cannot be stored in the main memory.

1.3 Exception Handling

1. **Missing values**: We handle missing values by changing their value to one that can never logically occur for the columns in question. Only the "Temperature" and "Humidity" columns can have missing values which are marked by the string "M". During preprocessing when the input CSV file is being converted to the separate column files, if "M" is encountered for either of these columns, it is converted to the string "-1000", which is later accordingly converted to the float -1000, and stored. We make this choice by taking into account the semantic meaning

of the columns in question; the temperature can never become −1000°C in Singapore, and the percentage of humidity can never become negative.

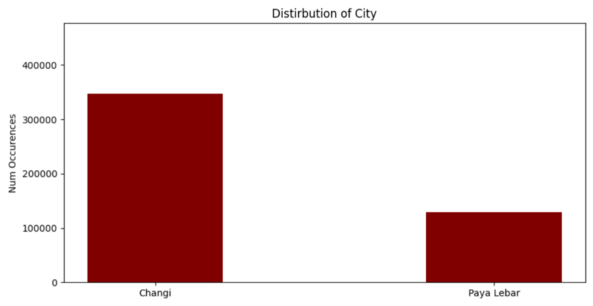
2. **Missing month**: We handle missing months by not processing them in any special manner. This is because we delegate of the handling of this corner case to the query processing; we design our query processing to be agnostic to the possibility that no value may be present for an entire month.

2 Data Processing

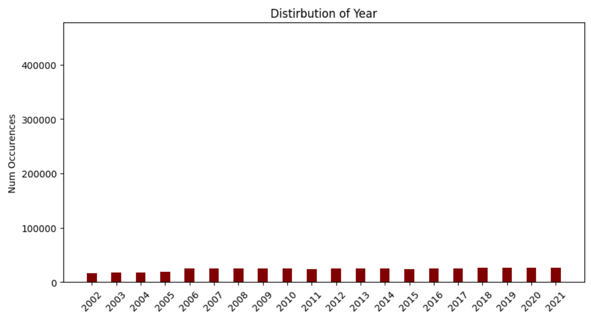
We first discuss the task at hand, as it is relevant to the manner in which we design our data processing. The task effectively requires us to group by months within 2 particular years given in the query, then find qualified results where the maximum and minimum values are achieved in the "Temperature" and "Humidity" columns. Based on this query, we made a generic class-based design for the various components of the query processor.

2.1 Column Scanning for Query Processing

We choose to scan (filter) the columns in the order: (i) Timestamp, (ii) Station. We choose to do this because the query restricts the range of possible outputs to those that occur in just 2 years. Given that the dataset has 20 years of data, we can roughly estimate that the selectivity of such a year-based filter would be roughly $\frac{2}{20} = 0.1$. On the other hand, the query restricts the range of possible outputs to those taken from 1 station out of a total of 2 stations. Hence, we can estimate that the selectivity of such a station-based filter would be roughly $\frac{1}{2} = 0.5$. Hence, it is much more optimal to perform the year-based query first, as the number of qualified tuples obtained would be a lot less. The output of each intermediate step is qualified positions, each of which is represented by a single integer. At each filter, the qualified positions are outputted to a temporary file.



(a) City Distribution



(b) Year Distribution

Figure 1: Distirbution of Columns

The scan is achieved here by using the "AtomicPredicate" class which serves as a generic method to implement any comparison operation between a given constant value in the query, and the values in the column, as they are scanned.

After these 2 filters, qualified positions are then sent to the "GroupBy" operator which groups by month, and sends the position, year and month into a custom data structure that we call "GroupByYearMonthPosition". Recall from the design of our block that it is generic to any data type, even custom types and so these qualified positions, which are no longer simple integers, can also be handled using the block abstraction, and accordingly the qualified "GroupByYearMonthPositions" are outputted to a temporary file.

Then, the qualified "GroupByYearMonthPositions" are read from the temporary file, and the "Temperature" and "Humidity" columns are read. When these columns are processed, effectively a **shared scan** is used, since all the aggregates, - maximum temperature, minimum temperature, maximum humidity, and minimum humidity - are all simultaneously processed. Then, the final aggregated values for each of these is stored in memory, which is justified due to the negligible size of the results.

Penultimately, the qualified "GroupByYearMonthPositions" are once again read from the temporary file, and using an "AtomicPredicate" for each of the aggregated values, the qualified positions where each of maximum/minimum values are outputted to 4 separate files, one for each aggregation. Once again, we use a **shared scan** here to get obtain rows matching any 4 of the aggregate values at once, rather than repeating the scan for each aggregate.

Finally, a "Projection" operation is performed where the qualified positions for each of the aggregations are read and only the required columns are retained (projection, in database terms).

2.2 Minimum and Maximum Aggregation

We implement a generic virtual "AggregationFunction" class which allows aggregation on any type of value and allows for easy extension to any type of aggregation operation. This is especially relevant for the context of big data, where we may be required to process other interesting aggregation queries such as "sum" and "average" for which we can easily extend and adapt the existing setup. The "MaxAggregation" and "MinAggregation" classes process the aggregation by iteratively considering one value after another, and replacing the result of the aggregate as required.

This design is very conducive with frameworks like **MapReduce** since this can equivalently represent a "Reduce" function which receives values of a particular subset to be aggregated. In fact, the method "save_aggregation" processes the aggregates of each month in a sequential manner, which can very easily be turned into a "Map" function which streams all data associated with a certain month to the "Aggregator" (in MapReduce terminology), which will then pass it on to the "Reduce" job. Hence, our design is very compatible for settings where the data may scale up fast.

2.3 Enhancements

We explore four major enhancements in processing the column store:

1. **Binary Search**: We observed that the "Timestamp" column was indeed sorted. We exploited this property of the column to perform the initial filter (on the year) as a binary search filter. This significantly reduces the number of page I/Os required during the initial scan.

2. **Zone Maps**: During the preprocessing stage, we create zone maps for the timestamps to obtain the maximum and minimum values of each page for the "Timestamp" column. We store the zone map as a separate file. Later, when processing a query, the zone map file is read, and once again we make use of "AtomicPredicate" to effectively deconstruct the boolean expression $(year == year1)$ || $(year == year2)$ to the boolean expression $(year \geq year_1$ && $year \leq year_1)$ || $(year \geq year_2$ && $year \leq year_2)$. This is particularly useful since this allows for easy demarcation of the starting and ending points of the rows where the year

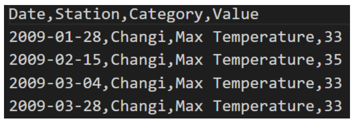
of the “Timestamp” column equals a certain year. Once again, this significantly reduces the number of page I/Os required during the initial scan.

3. **GroupByYearMonthPosition**: We make special note of the special data structure we created to handle the “GroupBy” operation. Observe that this data structure essentially serves the purpose of storing a qualified position, but also contains the additional metadata of the associated year and month to allow for easier processing of the aggregation. Here, we essentially make a **space-for-time** tradeoff. Observe that if we only stored the qualified positions, there would be no way of deducing the year and month associated with a given position, hence necessitating a read of the “Timestamp” once again, after it has already been filtered. This incurs much heavier disk I/Os and hence is not scalable. Instead, our solution ensures that such re-reads would never be incurred, thus resulting in significant savings.

4. **Shared Scan**: As highlighted in the explanation on column scanning, we utilize shared scans while computing the maximum and minimum for each of “Temperature” and “Humidity”.

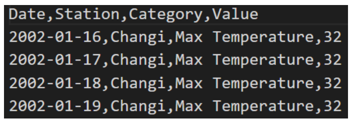
3 Experiment Results

3.1 Program Output and Evaluation



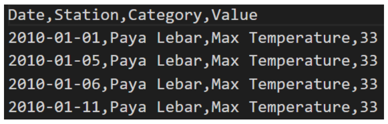
```
Date,Station,Category,Value
2009-01-28,Changi,Max Temperature,33
2009-02-15,Changi,Max Temperature,35
2009-03-04,Changi,Max Temperature,33
2009-03-28,Changi,Max Temperature,33
```

(a) U1922129L



```
Date,Station,Category,Value
2002-01-16,Changi,Max Temperature,32
2002-01-17,Changi,Max Temperature,32
2002-01-18,Changi,Max Temperature,32
2002-01-19,Changi,Max Temperature,32
```

(b) U1923502C



```
Date,Station,Category,Value
2010-01-01,Paya Lebar,Max Temperature,33
2010-01-05,Paya Lebar,Max Temperature,33
2010-01-06,Paya Lebar,Max Temperature,33
2010-01-11,Paya Lebar,Max Temperature,33
```

(c) U1923790H

Figure 2: Selected output for three matriculation IDs

The first four rows of the output files corresponding to all three matriculation IDs are displayed in Figure 2, confirming the successful execution of the program. To evaluate our output, let the input be U1922129L. The last digit = 9, which means we scan the years 2009 and 2019. The second last digit is even and thus we scan data corresponding to Changi station. Similarly, for U1923502C we scan Changi for the years 2002 and 2012, and for U1923790H we scan Paya Lebar for the years 2010 and 2020. The final output was successfully verified using both PostgreSQL and Numpy + Pandas.

3.2 Analysis

Table 1: Num IOs for filtering year

Algorithm	**Block Size (bytes)**					
	50	100	500	1000	5000	10000
Filter	19077	9539	1908	954	191	96
Binary Search	2093	1057	227	119	33	19
Zone Map	2067	1034	208	105	22	11

We observe that for a fixed input, the trend of the number of data IOs follows normal filter > binary search > zone map. This trend remains consistent as we vary the block size.