Міністерство освіти і науки України

Національний технічний університет України

“​ Київський політехнічний інститут”

Кафедра АСОІУ

ЗВІТ

про виконання лабораторних робіт № 3-7

з дисципліни

“​ Сучасні технології програмування – 1 : Функціональне програмування”

Прийняв:ас. Виконав:

Очеретяний О.К. студент 2-го курсу

гр. ІП-82 ФІОТ

Троцюк Павло Сергійович

Київ 2020

Дані лабораторні я виконував на мові програмування Clojure. І так розпочнімо з 3 лабораторної роботи. Тут нам потрібно було зчитувати csv та tsv. Для зчитування .csv я використовував стандартну бібліотеку [clojure/data.csv](https://github.com/clojure/data.csv) і в ній використав звичайну функцію для зчитування данних:

(defn readCSV [name]  
 (with-open [reader (io/reader name)]  
 (doall  
 (csv/read-csv reader))))

Мал 1. Приклад функції для зчитування csv файлу

Для зчитування tsv я написав власний парсер. Просто розбиваємо на рядки і потім формуємо через таб слова. Функція наведена нижче:

(defn readTSV [name]  
 (map #(str/split % #"\t")  
 (str/split (slurp name) #"\r\n")))

Мал 2. Приклад функції для зчитування tsv файлу

В даному випадку получаємо наступний результат:

(["id" "name" "money"] ["1" "Paul" "200"] ["2" "Mark" "150"] ["3" "Ilia" "100"] ["4" "Sergiy" "175"])

Кожен рядок розбитий на так звані слова за всіма правилами csv. На мій погляд працювати з такими данними доволі таки важко, тому я вирішив використовувати колекцію map. Для цього я використав функцію [zip-map](https://clojuredocs.org/clojure.core/zipmap) яка отримує на вхід два списка(перший ключі, другий значення) і формує список мапів. Ось ця функція:

(defn data->maps [head & lines]  
 (map #(zipmap (map keyword head) %1) lines))

Мал 3. Приклад функції для вдосконалення файлу(переведення в мапи)

[Map](https://clojuredocs.org/clojure.core/map) формує новий список, ітеруючись по списку переданим в параметрах, а також приймає функцію, яка виконує певні дії, та повертає новий елемент списку.

І тепер маємо зовсім інакший результат:

({:id "1", :name "Paul", :money "200"}

{:id "2", :name "Mark", :money "150"}

{:id "3", :name "Ilia", :money "100"}

{:id "4", :name "Sergiy", :money "175"})

Коли ідея ясна, можна почати з самого початку цієї програми. Ми визиваємо нашу функцію getFile, яка в залежності від типу формату, визиває функцію формування нового файлу:

(defn getFile [name]  
 (if (checkFormat name) (makeTableCSV name) (makeTableTSV name)))

Мал 4. Приклад функції для отримання файлу

Ця функція за допомогою функції checkFormat вирішує, яку функцію викликати: csv чи tsv. Якщо ж файл з розширенням csv, то вона вертає true і викликається функція makeTableCSV. Якщо ж false – makeTableTSV. Функція checkFormat приймає ім’я та видає булеве значення:

(defn checkFormat [name]  
 (let [formatFile (str/split name #"\.")]  
 (if (= (str (nth formatFile 1)) "csv") true false)))

Мал 5. Приклад функції для визначення формату файлу

І також функції формування таблиць:

(defn makeTableTSV [name]  
 (apply data->maps (readTSV name)))  
  
(defn makeTableCSV [name]  
 (apply data->maps (readCSV name)))

Мал 6. Приклад функцій для побудови нових таблиць

Коли вже в нас все готове, залишилось тільки красиво це все вивести. Для цього я написав функцію printable:

(defn printTable [table]  
 (loop [l 0]  
 (when (< l (count table))  
 (loop [k 0]  
 (when (< k (count (first (nth table l))))  
 (print (format "%40s| " (name (nth (getKeysForPrint l table) k))))  
 (recur (+ k 1))))  
 (recur (+ l 1))))  
 (println (str "\n" (str/join "" (take (\* 42 (getCountOfAllKeys table)) (repeat "-")))))  
 (loop [i 0]  
 (when (< i (count (first table)))  
 (loop [j 0]  
 (when (< j (count table))  
 (loop [z 0]  
 (when (< z (count (nth (nth table j) i)))  
 (print (format "%40s| " (nth (getValsForPrint j i table) z)))  
 (recur (+ z 1))))  
 (recur (+ j 1))))  
 (println)  
 (recur (+ i 1)))))

Мал 7. Приклад функції для красивого виводу таблиці на екран

Зараз все розберемо, головне не переживайте. Тут ми використовуємо рекурсивну ітерацію [loop](https://clojuredocs.org/clojure.core/loop), яка дуже схожа на звичайний цикл, в нас є умова when, а також рекурсивне повернення нашого значення через recur. Спочатку ми виводим наші ключи всіх мапів(вони є назвами колонок). Ітеруємось в зовнішньому циклі по кількістю списків в нашому загальному списку, а потім ітеруємось по кількості елементів в першій мапі вибраної колонки. Виділяємо 40 символів на один рядок та відділяємо його такою комбінацією “| ”. Для отримання ключів використовуємо наступну функцію:

(defn getKeysForPrint [numTable table]  
 (keys (first (nth table numTable))))

Мал 8. Приклад функції для взяття ключів з певної колонки

Після того як вивелись всі ключі, намалюємо горизонтальну лінію. Знаючи, що одна колонка 42 символа, нам потрібно підрахувати кількість колонок та помножити на 42. Для підрахунку кількості колонок використовуємо функцію getCountOfAllKeys:

(defn getCountOfAllKeys [table & countOfKeys]  
 (cond  
 (empty? table)  
 (first countOfKeys)  
 (empty? countOfKeys)  
 (getCountOfAllKeys (next table) (count (keys (first (first table)))))  
 *:else* (getCountOfAllKeys (next table)  
 (+ (count (keys (first (first table)))) (first countOfKeys)))))

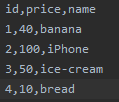
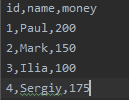
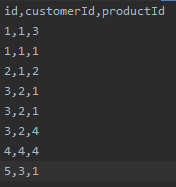
Мал 9. Приклад функції для отримання кількості колонк

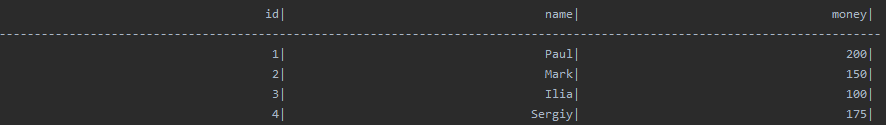
Рекурсивно ітеруємось по нашій таблиці, в нас тут є 2 умови, якщо пуста табличка то вернути вже підраховану кількість ключів та якщо це перший вхід в функцію то вернути наступні колонки та кількість ключів в першій колонці. І якщо ви ітеруємось десь всередині, то просто вертаємо наступні колонки та додаємо до значення підрахованих ключів, поточне значення кількості ключів. Вернімось до головної функції. Використовуючи конструкцію take number (repeat element) отримаємо список з 42 \* кількість ключів елементами “-”, потім просто за допомогою функції join об’єднуємо це все в одну строчку і виводимо її через переведення каретки. Ну і останній крок це виведення всіх значень. Знаючи наперед, що кількість елементів у всіх колонках однакова, ітеруємось у зовіншньому циклі по кількості елементів в першій колонці. Далі ітеруємось по кількостям колонок і вже потім ітеруємось по першим елементам колонок, потім по другим, тобто горизонтально. Для отримання значеннь поточної колонки з заданим індексом використовуємо наступну функцію:

(defn getValsForPrint [numTable numRow table]  
 (vals (nth (nth table numTable) numRow)))

Мал 10. Приклад функції для взяття значень з певної колонки

Вона майже така як і для ключів, тільки для ключів брали перший елемент, а тут nth. Впринципі з 3-ю лабораторною роботою завершили, подивимось на результат(створимo 3 csv файли – test1.csv, test2.csv, test3.csv):

    
 test1.csv test2.csv test3.csv



Мал 11. Приклад результату 3-ої лабораторної роботи(test1.csv)

Перейдімо до наступних лабораторних. Тут вже прийшлось серйозно подумати як все буде виглядати. Розглянемо мою реалізацію даної проблеми. Будемо йти від функції -main і до кінця. Точка входа в програму:

(defn -main [& args]   
 (do (createHelp) (view)))

Мал 12. Точка входу в програму

Тут ми просто викликаємо дві функції: перша для виводу справки(рекомендую для прочитання), друга основна для формування ваших запитів та виводу на екран.

(defn createHelp [] (println (slurp "Help.txt")) true)

Мал 13. Зчитування та вивід справки на екран

Функція slurp просто зчитує дані з файлу та поміщає їх у рядок. Переглянемо функцію view:

(defn view []  
 (println "Write your query below:")  
 (let [input (read-line)  
 commands ["SELECT" "FROM" "WHERE" "DISTINCT"  
 "ORDER" "BY" "AND" "OR" "NOT" "ASC" "DESC" "COUNT" "AVG" "MIN"  
 "INNER" "FULL" "OUTER" "LEFT" "JOIN" "ON" "GROUP"  
 "HAVING" "CASE" "WHEN" "THEN" "ELSE" "END" "AS"]]  
 (if (= "QUIT" (str/upper-case input)) true (do (printTable (getResult input commands)) (view)))))

Мал 14. CLI нашої програми

Ця функція запитує дані на ввід за допомогою команди read-line та дивиться якщо це вихід з програми то рекурсивно визиває цю функцію ще раз та виводить результат запиту. Тут містяться усі ключові слова, які потім використовуються в різних функціях. З функцією rintable ми вже знайомі(в красивому форматі виводить таблицю), осталось познайомитись із функцією getResult, яка за допомогою нашого запиту певним чином фільтрує наші таблиці.

(defn getResult [expr commands]  
 (let [words (deeperParseForWords (re-seq #”\”[^\”]+\”|[\S]+” expr) commands)]  
 (if (not= 0 (compare (str/upper-case (first words)) (first commands)))  
 [] (processColumns words commands))))

Мал 15. Основна функція для отримання таблиці

Ця функція парсить наш запит на окремі слова, по регулярному виразу, який розділяє слова по пробілам та бере в одне слово, яке міститься в лапках. Функція deeperParseForWords, кидає в одне слово всі символи, які містяться між ключовими словами SELECT та FORM.

(defn deeperParseForWords [words commands]  
 (let [indexOfFrom (.indexOf (map #(str/upper-case %) words) (nth commands 1))  
 indexOfDistinct (.indexOf (map #(str/upper-case %) words) (nth commands 3))]  
 (if (= -1 indexOfDistinct)  
 (concat (subvec (vec words) 0 1) (conj [] (str/join " " (subvec (vec words) 1 indexOfFrom)))  
 (subvec (vec words) indexOfFrom (count words)))  
 (concat (subvec (vec words) 0 2) (conj [] (str/join " " (subvec (vec words) 2 indexOfFrom)))  
 (subvec (vec words) indexOfFrom (count words))))))

Мал 16. Функція для об’єднання всіх колонок в 1 рядок

Далі дивимось чи наш запит починається з SELECT. Якщо так, то визиваємо функцію getCorrectTable, яка коректно визначає на якій позиції знаходяться наші колонки, щоб передати їх далі в обробку.

(defn getCorrectTable [words commands]  
 (if (not= 0 (compare (str/upper-case (nth words 1)) (nth commands 3)))  
 (if (not= 0 (compare (str/upper-case (nth words 2)) (nth commands 1)))  
 [] (getMainTable (map #(str/trim %) (str/split (nth words 1) #”,”)) words commands))  
 (if (not= 0 (compare (str/upper-case (nth words 3)) (nth commands 1)))  
 [] (getMainTable (map #(str/trim %) (str/split (nth words 2) #”,”)) words commands))))

Мал 17. Функція для коректного визначення індексу колонок

Тут ми просто дивимось чи присутнє ключове слово DISTINCT, якщо так то зміщуємо на 1 індекс колонок, ні то ставимо його як 1. Також додатково парсим колонки по комі, та видаляємо лишні пробіли. Переходимо до функції getMainTable:

(defn getMainTable [columns words commands]  
 (let [fileWithAllColumns (getFile (nth words (+ 1 (.indexOf (map #(str/upper-case %) words)  
 (nth commands 1)))))  
 filteredFile (orderBy (whereClause (checkForJoin fileWithAllColumns words commands)  
 words commands) words commands)]  
 (if (checkForGroupBy words commands)  
 (groupBy filteredFile columns words commands)  
 (if (= -1 (.indexOf (map #(str/upper-case %) words) (nth commands 3)))  
 (getAllColumns filteredFile columns commands)  
 (if (= -1 (.indexOf columns "\*"))  
 (getAllColumns (first (myDistinct (mergeColumns  
 (getAllColumns filteredFile columns commands)))) columns commands)  
 (getAllColumns (first (myDistinct filteredFile)) columns commands))))))

Мал 18. Основна функція, яка формує таблицю

Тут з самого початку ми використовуємо функцію з 3-ої лабораторної для зчитування нашого файлу і відразу фільтруємо його по 3-х параметрах: дивимось чи немає наступних ключових слів WHERE, ORDER BY, JOIN. Якщо ж такі є, то виконуємо відповідні дії. Коли таблиця сформована, дивимось чи наявне слово GROUP, якщо так, то викликаємо функцію, яка виконує відповідні дії, якщо ж відсутнє, то перевіряємо на наявність ключового слова DISTINCT. Якщо ж таке слово відсутнє, то викликаємо функцію getAllColumns і передаємо наш відфільтрований файл, колонки та команди. Якщо ж наявне слово DISTINCT, то дивимось чи є назва колонки \*. Якщо так, то просто викликаємо функцію [distinct](https://clojuredocs.org/clojure.core/distinct), яка видаляє всі однакові елементи. Якщо ж ні, то ми з’єднуємо всі колонки в одну таблицю, але є один нюанс(merge {:a 1} {:a 2} => {:a 2}), тобто нам потрібно змерджити колонки, примінити distinct і знову витягнути всі колонки. Так це затратно, но приходиться інколи чимсь жертвувати. Розглянемо функцію, яка мерджить наші колонки:

(defn mergeColumns [table]  
 (for [i (range (count (first table)))]  
 (mergeColStep i 0 table)))

Мал 19. Функція яка склеює всі колонки

Тут ми можемо бачити команду for, яка генерує новий список довжиною, яку ми вкажемо. В цьому випадку ми вказуємо, що нам потрібно ітеруватись від 0 до загальної довжини першої колонки. Далі в тілі функції ми визиваємо іншу функцію, яка бере всі поточні мапи в даному рядку, та мерджить їх:

(defn mergeColStep [indexRow indexColumn table & map]  
 (if (= 1 (- (count table) indexColumn))  
 (merge (first map) (nth (nth table indexColumn) indexRow))  
 (mergeColStep indexRow (+ 1 indexColumn) table (merge (first map) (nth (nth table indexColumn) indexRow)))))

Мал 20. Функція яка склеює в одну мапу всі інші

Тут ми рекурсивно проходимось по всім колонкам та об’єднуємо їх в одну велику мапу, допоки значення індекса не достигне максимуму(щоб не вийти за межі таблиці).

Перейдімо до розгляду функції, яка дає нам всі колонки:

(defn getAllColumns [file columns commands]  
 (map #(let [splitCol (re-seq #"\"[^\"]+\"|[\S]+" %)  
 indexOfNameCase (.indexOf (keys (first file)) (keyword (last splitCol)))]  
 (cond  
 (= 0 (compare % "\*"))  
 file  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (= 2 (- (.indexOf % ")") (.indexOf % "(")))  
 (= 0 (compare "\*" (subs % (+ 1 (.indexOf % "(")) (- (count %) 1))))  
 (str/starts-with? (str/upper-case %) (nth commands 11)))  
 (if (not= -1 (.indexOf (keys (first file)) *:count*))  
 (getColumn "count" file) (countFunc file))  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (str/starts-with? (str/upper-case %) (nth commands 11)))  
 (if (not= -1 (.indexOf (keys (first file)) *:count*))  
 (getColumn "count" file)  
 (countFunc (getColumn (subs % (+ 1 (.indexOf % "(")) (- (count %) 1)) file)))  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (str/starts-with? (str/upper-case %) (nth commands 12)))  
 (if (not= -1 (.indexOf (keys (first file)) *:avg*))  
 (getColumn "avg" file)  
 (avgFunc (getColumn (subs % (+ 1 (.indexOf % "(")) (- (count %) 1)) file)))  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (str/starts-with? (str/upper-case %) (nth commands 13)))  
 (if (not= -1 (.indexOf (keys (first file)) *:min*))  
 (getColumn "min" file)  
 (minFunc (getColumn (subs % (+ 1 (.indexOf % "(")) (- (count %) 1)) file)))  
 (and (> (count splitCol) 1) (= (str/upper-case (first splitCol)) (nth commands 22)))  
 (if (or (not= -1 indexOfNameCase) (not= -1 (.indexOf (keys (first file)) *:case*)))  
 (getColumn (if (= -1 indexOfNameCase) "case" (last splitCol)) file)  
 (if (= (str/upper-case  
 (nth splitCol (- (count splitCol) 2))) (nth commands 27))  
 (checkCaseCondition file % (nth splitCol (- (count splitCol) 1)) commands)  
 (checkCaseCondition file % "case" commands)))  
 *:else* (getColumn % file)))  
 columns))

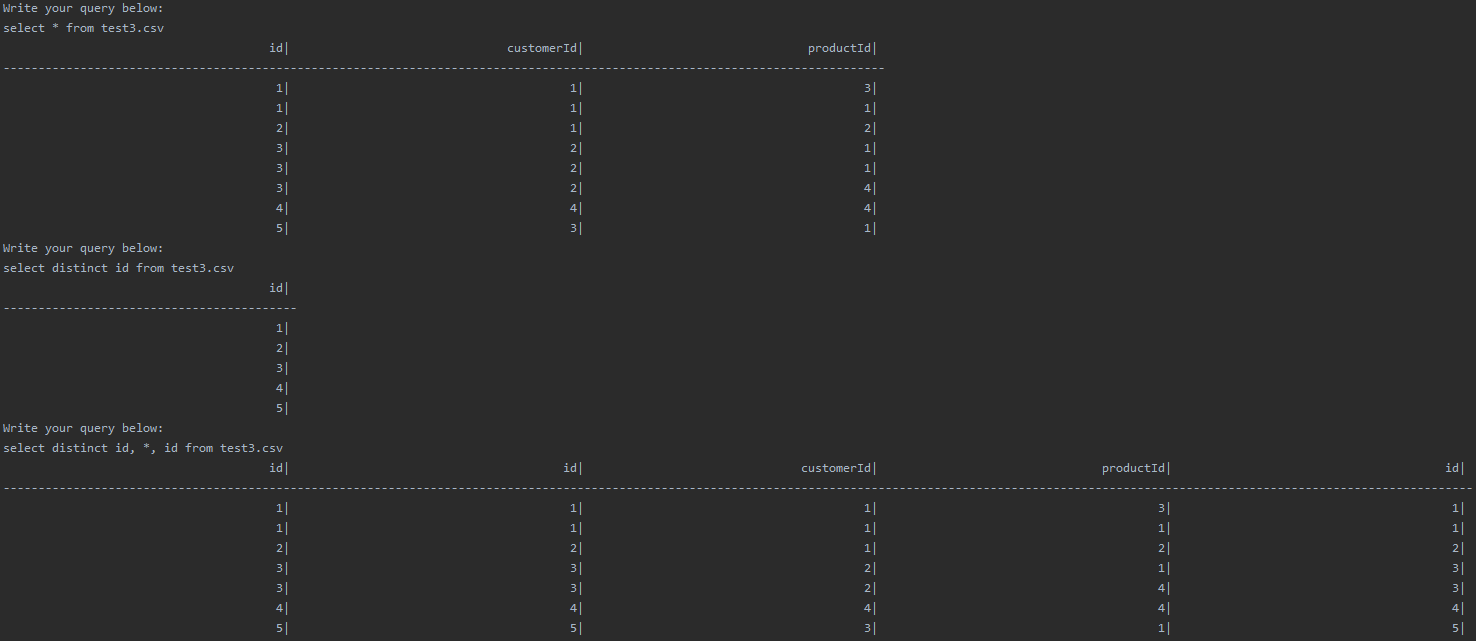
Мал 21. Приклад функції, яка бере з таблиці дані та робить окремі колонки

Розглянемо повністю всі умови(тут є 4,5,7 лаби). З самого початку ми розділяємо назву нашої колонки по регулярному виразу який ви можете бачити вище. Також відразу дивимось чи є останнє слово після розділу в назві колонок таблиці. Далі ми дивимось, якщо колонка з назвою \*, то просто видаємо готовий файл. Якщо назва починається з слова count, та слово заключене в круглі дужки є \* то викликаємо агрегатну функцію countFunc для цілого файлу. Якщо ж це якесь інше слово в дужках, то виділяємо це слово і передаємо його разом із файлом у нашу функцію countFunc. Таким самим чином все працює і для avgFunc та minFunc. Також у цих функціях ми перевіряємо наявність їх вже у таблиці. Якщо ж такі присутні, то ми просто їх вертаємо. Так і робимо з наступною умовою, яка передбачена для CASE ключового слова. Якщо ж такого немає, то викликаємо спеціальну функцію, яка формує нову колонку. Ну і якщо це просто назва колонки, то передаємо в функцію getColumn назву колонки та файл. Розглянемо функцію getColumn.

(defn getColumn [columnName file]  
 (map #(select-keys % [(keyword columnName)]) file))

Мал 22. Функція, яка дістає по заданому ключу всі значення

Тут ми просто пробігаємся за допомогою map та дістаємо усі значення по ключу і вертаєм колонку. Давайте протестуємо, те що поки є:



Мал 23. Результати

Тепер розглянемо WHERE.

(defn whereClause [table words commands]  
 (if (not= -1 (.indexOf (map #(str/upper-case %) words) (nth commands 2)))  
 (filter #(checkAllConditions % (parseConditions words commands) commands) table) table))

Мал 24. Головна функція, яка керує фільтрацією по умовах

Ми дивимось чи присутнє ключове слово WHERE в нашому запиті, якщо ні, то вертаємо таблицю, якщо ж так то фільруємо нашу таблицю за допомогою функції checkAllConditions, яка вертає булеві значення для кожного рядка, тобто чи пройшов рядок всі критерії. Також ми парсимо наші умови за допомогою функції parseConditions:

(defn parseConditions [words commands]  
 (let [indexOfWhere (.indexOf (map #(str/upper-case %) words) (nth commands 2))  
 lastIndexOfAnd (if (not= -1 (.indexOf (map #(str/upper-case %) (reverse words)) (nth commands 6)))  
 (- (count words) (.indexOf (map #(str/upper-case %) (reverse words)) (nth commands 6)) 1) -1)  
 lastIndexOfOr (if (not= -1 (.indexOf (map #(str/upper-case %) (reverse words)) (nth commands 7)))  
 (- (count words) (.indexOf (map #(str/upper-case %) (reverse words)) (nth commands 7)) 1) -1)  
 lastIndexOfNot (if (not= -1 (.indexOf (map #(str/upper-case %) (reverse words)) (nth commands 8)))  
 (- (count words) (.indexOf (map #(str/upper-case %) (reverse words)) (nth commands 8)) 1) -1)]  
 (if (= -1 (max lastIndexOfAnd lastIndexOfOr lastIndexOfNot))  
 (subvec (vec words) (+ 1 indexOfWhere) (+ 4 indexOfWhere))  
 (subvec (vec words) (+ 1 indexOfWhere) (+ 4 (max lastIndexOfAnd lastIndexOfOr lastIndexOfNot))))))

Мал 25. Функція, яка парсить наші умови для WHERE

Спочатку ми визначаємо індекс нашого where. Далі визначаємо останні входження ключових слів AND OR NOT. Якщо ж такі слова не присутні то просто беремо 3 перші слова після нашого WHERE, якщо ж присутні, то беремо максимальне з них і додаємо 4, тобто ще 3 слова після нього буду зчитуватись. Тепер перейдімо до розбору функції checkAllConditions.

(defn checkAllConditions [row conditions commands]  
 (let [lastIndexOfAnd (getLastIndex (map #(str/upper-case %) conditions) (nth commands 6))  
 lastIndexOfOr (getLastIndex (map #(str/upper-case %) conditions) (nth commands 7))  
 numberOfAnd (numberOfRepeated (map #(str/upper-case %) conditions) (nth commands 6))  
 numberOfOr (numberOfRepeated (map #(str/upper-case %) conditions) (nth commands 7))]  
 (cond  
 (= 3 (count conditions)) (checkTruth row conditions)  
 (= 4 (count conditions))  
 (if (not= -1 (.indexOf (map #(str/upper-case %) conditions) (nth commands 8)))  
 (not (checkTruth row (subvec (vec conditions) 1))) false)  
 (= 1 (+ numberOfAnd numberOfOr))  
 (if (= 1 numberOfAnd)  
 (andOrNotCondition row conditions commands (nth commands 6))  
 (andOrNotCondition row conditions commands (nth commands 7)))  
 (> lastIndexOfAnd lastIndexOfOr)  
 (checkAllConditions  
 row  
 (conj (subvec (vec conditions)  
 0 (+ 1 (indexBeforeLastBoolOp conditions lastIndexOfAnd commands)))  
 (andOrNotCondition  
 row  
 (subvec (vec conditions)  
 (+ 1 (indexBeforeLastBoolOp conditions lastIndexOfAnd commands)) (count conditions))  
 commands (nth commands 6)))  
 commands)  
 *:else* (checkAllConditions  
 row  
 (conj (subvec (vec conditions)  
 0 (+ 1 (indexBeforeLastBoolOp conditions lastIndexOfOr commands)))  
 (andOrNotCondition  
 row  
 (subvec (vec conditions)  
 (+ 1 (indexBeforeLastBoolOp conditions lastIndexOfOr commands)) (count conditions))  
 commands (nth commands 7)))  
 commands))))

Мал 26. Головна функція для побудови списка з булевими значеннями, які відповідають тому, чи підходить даний рядок під наші умови чи ні.

Тут знову ж таки ми визначаємо останні індекси слів AND, OR, а також скільки разів вони повторюються. Для цього використовуємо власноруч написані функції getLastIndex та numberOfRepeated. Швиденько пройдемось по них:

(defn getLastIndex [coll word]  
 (if (not= -1 (.indexOf (reverse coll) word))  
 (- (count coll) (.indexOf (reverse coll) word) 1) -1))

Мал 27. Функція для знаходження індексу останнього входження елементу

Тут ми просто беремо перевертаємо наш список та використовуємо звичайну функцію .indexOf, яка знаходить перше входження елемента. Якщо ж цей індекс = -1, то вертаємо -1. Якщо ні, то беремо віднімаємо від загальної кількості елементів наш індекс та одиницю.

(defn numberOfRepeated [coll x & counter]  
 (cond  
 (empty? Counter)  
 (if (= 0 (compare (first coll) x))  
 (numberOfRepeated (next coll) x 1) (numberOfRepeated (next coll) x 0))  
 (empty? Coll)  
 (first counter)  
 (= 0 (compare (first coll) x))  
 (numberOfRepeated (next coll) x (inc (first counter)))  
 *:else* (numberOfRepeated (next coll) x (first counter))))

Мал 28. Функція для знаходження числа повторів елемента в списку

Тут ми просто рекурсивно проходимось по кожному елементу та дивимось чи елемент співпадає з нашим, якщо так, то ще раз рекурсивно визиваємо цю функцію, збільшуючи при цьому наш лічильник.

Перейдімо до розбору функції checkAllConditions. Спочатку ми дивимось чи довжина умови = 3. Якщо так, то це є звичаний запит типу id > 3 і приміняємо до нього функцію checkTruth, яка вертає нам булеве значення, розберемо її пізніше. Далі дивимось, якщо довжина = 4, то припускаємо що це запит типу not id = 3. Перевіряємо чи дійсно перше слово NOT, якщо так то робимо теж саме тільки з запереченням що і при довжині 3. Якщо ні, вертаємо false. Далі дивимось чи сума повторів ключових слів = 1. Якщо так, то визначаємо, яке ж все таки це ключове слово і передаємо в функцію andOrNotCondition, розглянемо її пізніше. Тепер розказую головну ідею: ми по частинам дивимось умови, вертаємо true false і склеюємо з нашими умовами. Для прикладу: “cond1 and cond2 and cond3 or cond4” => “cond1 and cond2 and true” => “cond1 and false” => “false”. Ми беремо частину без останньої умови та приклеюємо туди результат функції andOrNotCondition. Тепер перейдімо до розгляду цієї функції:

(defn andOrNotCondition [row condition commands logicOp]  
 (let [indexOfNot (.indexOf (map #(str/upper-case %) condition) (nth commands 8))]  
 (if (= (str/upper-case logicOp) (nth commands 6))  
 (cond  
 (= 9 (count condition))  
 (and (not (checkTruth row (subvec (vec condition) 1 4)))  
 (not (checkTruth row (subvec (vec condition) 6 9))))  
 (and (= 8 (count condition)))  
 (if (= 0 indexOfNot)  
 (and (not (checkTruth row (subvec (vec condition) 1 4)))  
 (checkTruth row (subvec (vec condition) 5 8)))  
 (and (checkTruth row (subvec (vec condition) 0 3))  
 (not (checkTruth row (subvec (vec condition) 5 8)))))  
 (= 7 (count condition))  
 (and (checkTruth row (subvec (vec condition) 0 3))  
 (checkTruth row (subvec (vec condition) 4 7)))  
 (= 6 (count condition))  
 (and (not (checkTruth row (subvec (vec condition) 1 4))) (nth condition 5))  
 (= 5 (count condition))  
 (and (checkTruth row (subvec (vec condition) 0 3)) (nth condition 4)))  
 (cond  
 (= 9 (count condition))  
 (or (not (checkTruth row (subvec (vec condition) 1 4)))  
 (not (checkTruth row (subvec (vec condition) 6 9))))  
 (and (= 8 (count condition)))  
 (if (= 0 indexOfNot)  
 (or (not (checkTruth row (subvec (vec condition) 1 4)))  
 (checkTruth row (subvec (vec condition) 5 8)))  
 (or (checkTruth row (subvec (vec condition) 0 3))  
 (not (checkTruth row (subvec (vec condition) 5 8)))))  
 (= 7 (count condition))  
 (or (checkTruth row (subvec (vec condition) 0 3))  
 (checkTruth row (subvec (vec condition) 4 7)))  
 (= 6 (count condition))  
 (or (not (checkTruth row (subvec (vec condition) 1 4))) (nth condition 5))  
 (= 5 (count condition))  
 (or (checkTruth row (subvec (vec condition) 0 3)) (nth condition 4))))))

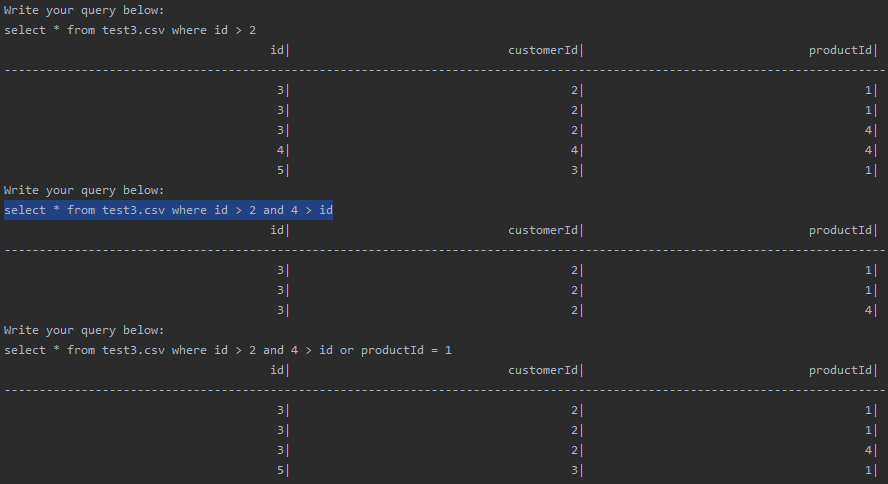
Мал 29. Функція, яка перевіряє умову з одним AND or OR та вертає булеве значення

З самого початку цієї функції ми дивимось чи ключове слово AND чи OR, далі просто перераховуємо всі випадки, які можуть бути і відповідно для кожної простої умови викликаємо функцію checkTruth, яка теж вертає true або false. Ну візьмемо для прикладу випадок коли розмір умови = 5. Тут ясно що перші 3 елементи це умова, другий елемент це ключове слово і останній це true або false. І так для всіх довжин умови. Перейдімо до функції checkTruth.

(defn checkTruth [row condition]  
 (let [indexOfValue (if (= -1 (.indexOf (keys row) (keyword (nth condition 0)))) 0 2)  
 indexOfColumn (if (= 0 indexOfValue) 2 0)]  
 (cond  
 (and (= 0 (compare (nth condition 1) "="))  
 (every? #(Character/isDigit %) (nth condition indexOfValue)))  
 (if (= (read-string (nth condition indexOfValue))  
 (read-string (get row (keyword (nth condition indexOfColumn))))) true false)  
 (and (= 0 (compare (nth condition 1) ">"))  
 (every? #(Character/isDigit %) (nth condition indexOfValue)))  
 (if (= 0 indexOfValue)  
 (if (> 0 (compareForStringNumber (read-string (nth condition indexOfValue))  
 (read-string (get row (keyword (nth condition indexOfColumn))))) 0)  
 true false)  
 (if (> 0 (compareForStringNumber (read-string (get row (keyword (nth condition indexOfColumn))))  
 (read-string (nth condition indexOfValue))) 0) true false))  
 (and (= 0 (compare (nth condition 1) "=")))  
 (if (= 0 (compare (get row (keyword (nth condition indexOfColumn)))  
 (subs (nth condition indexOfValue) (+ (str/index-of (nth condition indexOfValue) "\"") 1)  
 (str/last-index-of (nth condition indexOfValue) "\"")))) true false)  
 (and (= 0 (compare (nth condition 1) ">")))  
 (if (= 0 indexOfValue)  
 (if (> (compare (subs (nth condition indexOfValue) (+ (str/index-of (nth condition indexOfValue) "\"") 1)  
 (str/last-index-of (nth condition indexOfValue) "\""))  
 (get row (keyword (nth condition indexOfColumn)))) 0) true false)  
 (if (> (compare (get row (keyword (nth condition indexOfColumn)))  
 (subs (nth condition indexOfValue) (+ (str/index-of (nth condition indexOfValue) "\"") 1)  
 (str/last-index-of (nth condition indexOfValue) "\""))) 0) true false)))))

Мал 30. Функція, яка перевіряє найпростіші умови

Тут головним критерієм є те, чи другий елемент умови “=” чи “>”. Далі відразу в умові перевіряється чи є значення текстом чи цифрою. І в залежності від того виконуємо порівняння. Якщо ж це операція “>”, то має значення порядок. Тут ми теж дивимось на якому місці стоїть значення і назва колонки. Наприклад якщо це цифра і це знак “>”, ми потрапляємо у другу умову і дивимось на якому місці стоїть значення. Отже з WHERE розібрались. Давайте протестуймо.



Мал 31. Результати

Перейдімо до розгляду ключового слова ORDER BY:

(defn orderBy [table words commands]  
 (let [indexOfOrder (.indexOf (map #(str/upper-case %) words) (nth commands 4))]  
 (if (= -1 indexOfOrder)  
 table (mergeSort table (resolve (symbol (checkOrder words commands)))  
 (getColumnsOrderBy words (inc indexOfOrder))))))

Мал 32. Основна функція order by

Як завжди спочатку ми знаходимо індекс нашого ключового слова. Якщо він = -1, то просто вертаємо нашу таблицю, якщо ні то сортуємо дані за допомогою функції mergeSort, яка приймає таблицю, послідовність сортування та колонки по яких сортувати. Розглянемо маленькі функції checkOrder та getColumnsOrderBy.

(defn getColumnsOrderBy [words indexOfBy]  
 (map #(keyword %) (str/split (nth words (inc indexOfBy)) #",")))

Мал 33. Функція для розділення колонок те переведення їх в keywords

Тут все доволі просто, ми ділимо рядок по комі та робимо з кожного елемента keyword.

(defn checkOrder [words commands]  
 (cond  
 (not= -1 (.indexOf (map #(str/upper-case %) words) (nth commands 10))) "pos?"  
 *:else* "neg?"))

Мал 34. Функція, яка визначає порядок сортування

Тут ми просто дивимось чи є в нашому запиті слово DESC, якщо так то вертаємо рядок “pos?”, ні – “neg?”.

(resolve (symbol str)) – переведення рядка в функцію. Перейдемо до самого сортування:

(defn mergeSort  
 ([data] (mergeSort data neg?))  
 ([data ord] (mergeSort data ord []))  
 ([data ord columns]  
 (if (< (count data) 2)  
 data  
 (mergeStep  
 (mergeSort (first (split-at (/ (count data) 2) data)) ord columns)  
 (mergeSort (second (split-at (/ (count data) 2) data)) ord columns)  
 ord columns))))

Мал 35. Функція, яка сортує таблицю в якій елементами є hash-map

Ця функція є перегруженою. Якщо ми просто передаємо таблицю, то за умовчанням відбудеться сортування від меншого до більшого. Також ми вказуємо за якими колонками потрібно сортувати. Думаю про сам алгоритм ви можете самі почитати, а короткими словами, ми просто ділимо наш список допоки не буде по 1 елементу і далі за допомогою функції mergeSort склеюємо це все. Подивимось на цю функцію:

(defn mergeStep [l r ord columns]  
 (cond (empty? l) r  
 (empty? r) l  
 (and (empty? columns) (ord (mapCompare (first l) (first r))))  
 (cons (first l) (mergeStep (next l) r ord columns))  
 (and (empty? columns) (not (ord (mapCompare (first l) (first r)))))  
 (cons (first r) (mergeStep l (next r) ord columns))  
 (ord (mapCompare (first l) (first r) columns))  
 (cons (first l) (mergeStep (next l) r ord columns))  
 *:else* (cons (first r) (mergeStep l (next r) ord columns))))

Мал 36. Функція, яка виконує з’єднання двох списків за заданами критеріями

Зрівнюємо ми наші елементи за допомогою функції mapCompare, яка вертає -1, якщо перша мапа менше другої, 0 якщо однакові і 1 якщо більше. В неї ми можемо передавати колонки по яким зрівнювати, по замовчуванню воно сортує по всім колонкам, допоки не знайде різницю в значеннях. Розглянемо цю функцію:

(defn mapCompare [a b & keysOfMap]  
 (cond  
 (and (empty? keysOfMap)  
 (= 0 (compareForStringNumber (get a (first (keys a))) (get b (first (keys b)))))  
 (= 1 (count (keys a))))  
 0   
 (and (empty? keysOfMap)  
 (not= 0 (compareForStringNumber (get a (first (keys a))) (get b (first (keys b)))))  
 (= 1 (count (keys a))))  
 (compareForStringNumber (get a (first (first keysOfMap))) (get b (first (first keysOfMap))))   
 (and (empty? keysOfMap)  
 (= 0 (compareForStringNumber (get a (first (keys a))) (get b (first (keys b))))))  
 (mapCompare a b (next (keys a)))   
 (and (= 1 (count (first keysOfMap)))  
 (= 0 (compareForStringNumber (get a (first (first keysOfMap))) (get b (first (first keysOfMap))))))  
 0  
 (and (> (count (first keysOfMap)) 1)  
 (= 0 (compareForStringNumber (get a (first (first keysOfMap))) (get b (first (first keysOfMap))))))  
 (mapCompare a b (next (first keysOfMap)))  
 (empty? keysOfMap)  
 (compareForStringNumber (get a (first (keys a))) (get b (first (keys b))))  
 *:else* (compareForStringNumber (get a (first (first keysOfMap))) (get b (first (first keysOfMap))))))

Мал 37. Функція, яка порівнює мапи

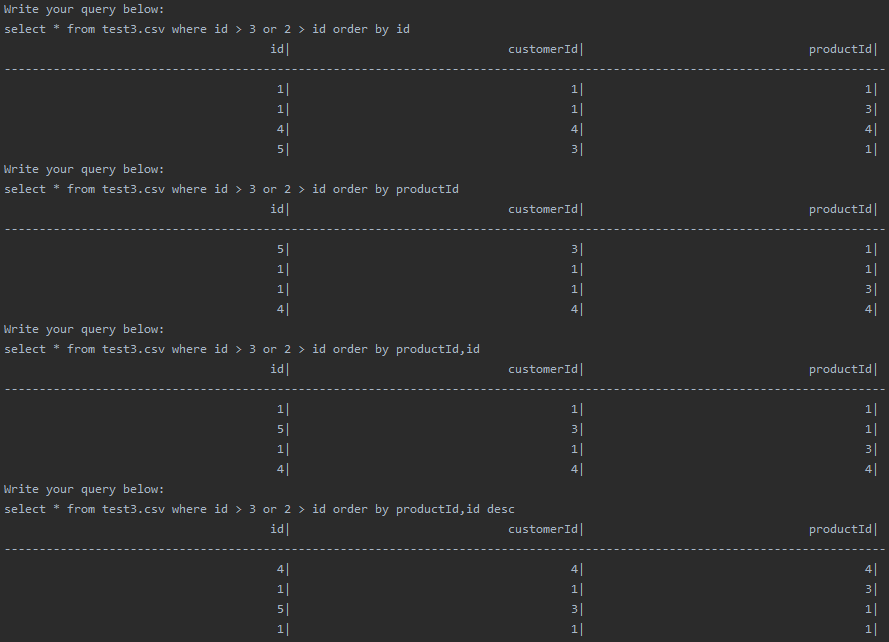
Давайте розберемо кожну умову. Перша умова – якщо розмір мапи = 1 та ці значення рівні то повертаємо 0. Друга – розмір мапи = 1, але різні значення, то повертаємо різницю. Третя – якщо розмір мапи більше 1 та поточні перші елементи рівні, викликаємо цю ж функцію, тільки передаємо третій параметр наступні ключі. Четверта – якщо значення останнього ключа рівні вертаємо 0. П’ята – теж саме, що і 4 тільки вертаємо різницю, бо вони різні. Шоста - якщо поточні значення поточних ключів різні, то вертаємо різницю. Ну і якщо нічого з цього не підходить то просто вертаємо різницю перших ключів. Тут ми використовуємо функцію compareForStringNumber.

(defn compareForStringNumber [a b]  
 (let [alterA (str a)  
 alterB (str b)]  
 (cond  
 (every? #(Character/isDigit %) alterA)  
 (compare (read-string alterA) (read-string alterB))  
 *:else* (compare alterA alterB))))

Мал 38. Функція, цифри в тексті та звичайний текст і звичайні цифри

Тут ми просто перевіряємо чи є кожен символ елемента цифрою, якщо так то переводимо це все в цифри та зрівнюємо.

Отож, протестуємо наше сортування. Будемо комбінувати з WHERE.



Мал 39. Результати

Для завершення 5-ої лабораторної залишилось тільки розібрати агрегатні функції. Почнімо)

І почнемо ми з всім відомої функції count. Розглянемо її реалізацію:

(defn countFunc [table]  
 (list (hash-map *:count* (count (filter #(not (every? (fn [value] (= "" value)) (vals %))) table)))))

Мал 40. Агрегатна функція для підрахунку кількості елементів, які не є пустими

Тут все досить просто. Ми створюємо нову колонку в якій міститься тільки 1 мапа ключем якої є count, а значенням виступає наступне: ми дивимось, якщо усі значення рядка є пустими то ми відсіюємо цей рядок з таблиці за допомогою функції [filter](https://clojuredocs.org/clojure.core/filter).

Далі в нас функція, яка рахує середнє значення:

(defn avgFunc [column]  
 (let [sumOfMaps (if (= 1 (count column))  
 (if (number? (first (vals (first column))))  
 (first column) (hash-map (first (keys (first column)))  
 (read-string (first (vals (first column))))))  
 (reduce sumForReduceMap column))  
 numberOfElementsMap (countFunc column)  
 numberOfElements (get (first numberOfElementsMap) (first (keys (first numberOfElementsMap))))  
 avgValueMap (update sumOfMaps (first (keys sumOfMaps)) / (double numberOfElements))]  
 (list (set/rename-keys avgValueMap {(first (keys avgValueMap)) *:avg*}))))

Мал 41. Функція для підрахунку середнього арифметичного

З самого початку ми підраховуємо суму всіх значень. Тут ми дивимось, якщо у нас тільки 1 рядок, то сумою і буде значення того рядка. Якщо ж ні, то робимо [reduce](https://clojuredocs.org/clojure.core/reduce) в який передаємо нашу функцію sumForReduceMap.

(defn sumForReduceMap [& body]  
 (let [firstValueOfMap (get (first body) (first (keys (first body))))  
 secondValueOfMap (get (second body) (first (keys (second body))))]  
 (cond  
 (= "" (first (vals (second body))))  
 (first body)  
 (and (number? firstValueOfMap) (number? secondValueOfMap))  
 (merge-with + (first body) (second body))  
 (and (string? firstValueOfMap) (string? secondValueOfMap))  
 (update (first body) (first (keys (first body)))  
 #(+ (read-string %) (read-string secondValueOfMap)))  
 *:else* (update (first body) (first (keys (first body)))  
 #(+ % (read-string secondValueOfMap))))))

Мал 42. Функція для reduce, щоб правильно додавати значення мап

Тут нам в параметрах спочатку приходять 2 перші агрументи. Потім першим аргументом є загальне значення, яке ми вертаємо з цієї функції, а другим буде наступний елемент списку. Фіксуємо ці елементи і починаємо дивитись на умови. Якщо наступний елемент порожній вертаємо перший. Якщо обидва параметри є числами, то мерджимо ці дві мапи з додаванням їхніх значень. Якщо обидва рядки, то оновлюємо перший елемент на значення їх суми. Якщо перший цифра, а другий рядок, то теж оновлюємо перший елемент. Просто тут переводимо в число тільки другий елемент.

Повернімося до головної функції. Також за допомогою numberOfElementsMap та numberOfElements ми взнаємо кількість ненульових рядків. Потім ми просто апдейтим нашу суму, ділячи її на значення numberOfElements. В кінці перейменовуємо ім’я колонки на avg та закидуємо його у список.

І остання агрегатна функція – min.

(defn minFunc [column]  
 (let [minValueMap (if (= 1 (count column)) (first column) (reduce minForReduceMap column))]  
 (list (set/rename-keys minValueMap {(first (keys minValueMap)) *:min*}))))

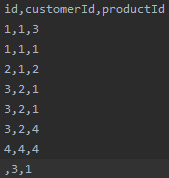
Мал 43. Функція для знаходження мінімального значення

Тут ми теж спочатку дивимось чи в нас один елемент, якщо так то вертаємо його, якщо ні, робимо reduce і передаємо туди нашу функцію minForReduceMap. Ця функція дуже схожа до попередньої(там де ми робили редюс авг), тільки тут ми просто дивимось чи ці елементи є мапами, і якщо перший менший від другого то вертаємо перший і навпаки, також якщо наступний елемент є порожнім ми просто вертаємо перший. А в кінці просто робимо як і з іншими агрегатними функціями – вертаємо список з мапою всерединою з ключе min.

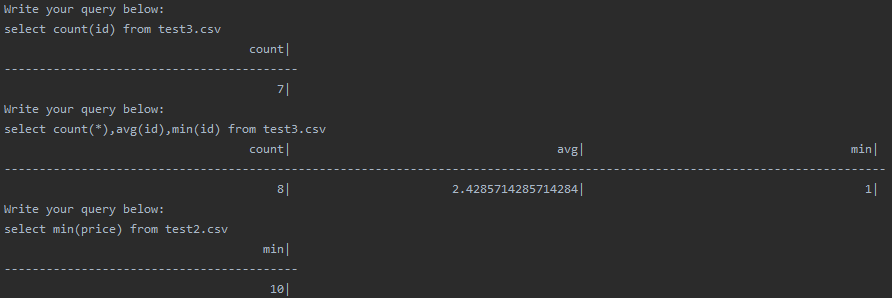
(defn minForReduceMap [& body]  
 (let [firstValueOfMap (get (first body) (first (keys (first body))))  
 secondValueOfMap (get (second body) (first (keys (second body))))]  
 (if (or (= "" (first (vals (second body))))  
 (and (map? (first body))  
 (<= (compareForStringNumber firstValueOfMap secondValueOfMap) 0)))  
 (first body) (second body))))

Мал 44. Функція для reduce, щоб правильно порівнювати мапи

І так з 5-ою лабораторною ми завершили, тому протестуймо наші агрегатні функції(трішки змінимо наш файл test3.csv зробивши останній рядок з полем id порожнім).



Мал 45. Модифікований файл test3.csv



Мал 46. Результати