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)))))

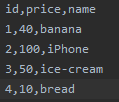
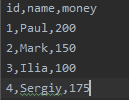
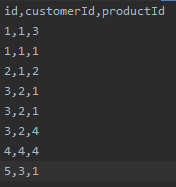
Мал 9. Приклад функції для отримання кількості колонк

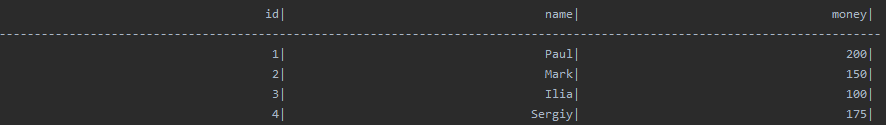
Рекурсивно ітеруємось по нашій таблиці, в нас тут є 2 умови, якщо пуста табличка то вернути вже підраховану кількість ключів та якщо це перший вхід в функцію то вернути наступні колонки та кількість ключів в першій колонці. І якщо ви ітеруємось десь всередині, то просто вертаємо наступні колонки та додаємо 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))

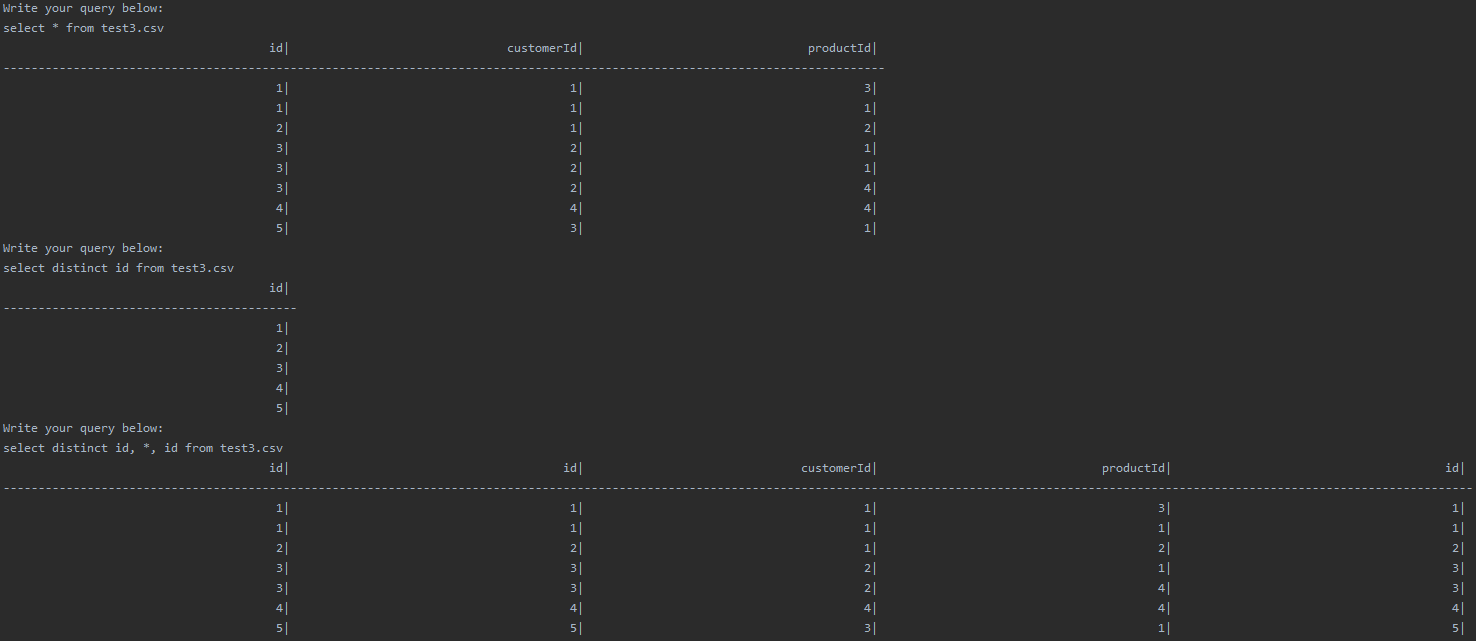
Мал 21. Приклад функції, яка бере з таблиці дані та робить окремі колонки

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))))))

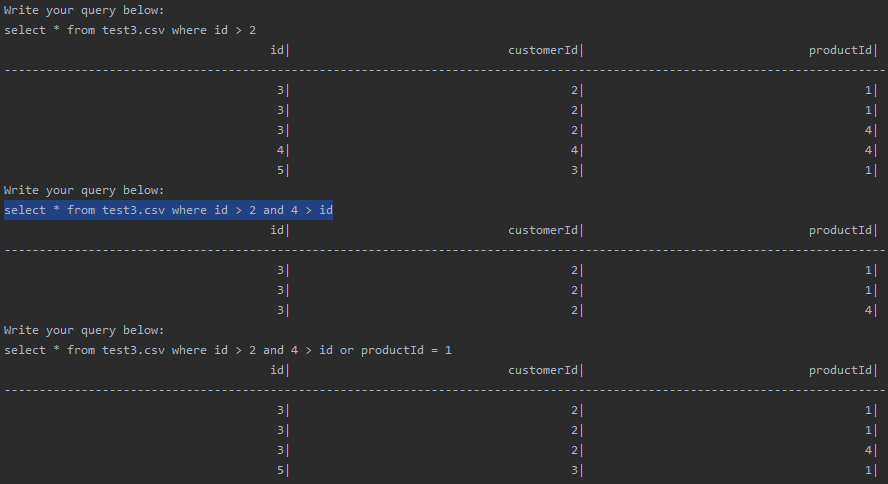
Мал 29. Функція, яка перевіряє умову з одним AND or OR та вертає булеве значення

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

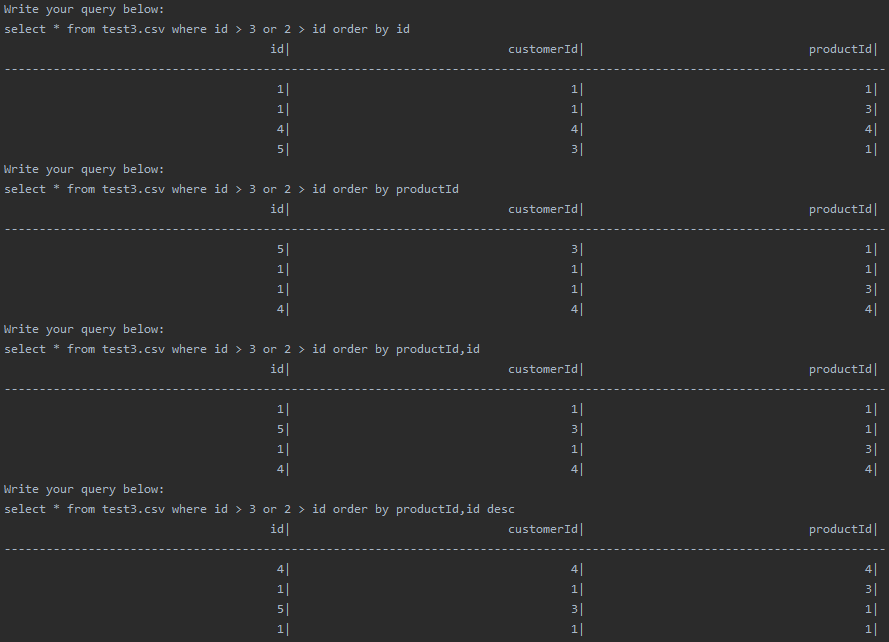
Давайте розберемо кожну умову. Перша умова – якщо розмір мапи = 1 та ці значення рівні то повертаємо 0. Друга – розмір мапи = 1, але різні значення, то повертаємо різницю. Третя – якщо розмір мапи більше 1 та поточні перші елементи рівні, викликаємо цю ж функцію, тільки передаємо третій параметр наступні 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))))  
 (if (string? (first column)) (read-string (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. Нам потрібно повернути значення в вигляді числа, а не тексту, тому також робимо перевірку і якщо це текст, то переводимо його в число (таке можливо коли поступає 1 рядок). 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))  
 updatedMinMap (if (string? (first (vals minValueMap)))  
 (assoc minValueMap (first (keys minValueMap))  
 (read-string (first (vals minValueMap)))) minValueMap)]  
 (list (set/rename-keys updatedMinMap {(first (keys updatedMinMap)) *:min*}))))

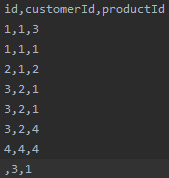
Мал 43. Функція для знаходження мінімального значення

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. Також для того щоб переконатись, що в нашій мапі міститься тип число, а не текст, ми робимо перевірку, якщо ж це текст, то апдейтим за допомогою [assoc](https://clojuredocs.org/clojure.core/assoc) нашу мапу. А в кінці просто робимо як і з іншими агрегатними функціями – вертаємо список з мапою всерединою з ключе min.

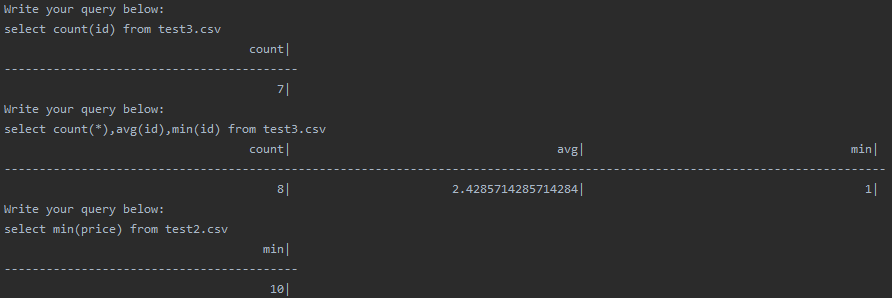
(defn minForReduceMap [& body]  
 (let [firstValueOfMap (get (first body) (first (keys (first body))))  
 secondValueOfMap (get (second body) (first (keys (second body))))]  
 (if (or (= "" (first (vals (second body))))  
 (and (map? (first body))  
 (<= (compareForStringNumber firstValueOfMap secondValueOfMap)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

І так перейдімо до ключового слова JOIN:

(defn checkForJoin [file words commands]  
 (if (= -1 (.indexOf (map #(str/upper-case %) words) (nth commands 18)))  
 file (joinTables file words commands)))

Мал 47. Функція для перевірки наявності ключового слова JOIN

Тут все просто, якщо наявне ключове слово то викликаємо функцію joinTables, якщо ні – вертаємо назад наш файл. Перейдемо до функції joinTables:

(defn joinTables [table words commands]  
 (let [firstJoinIndex (.indexOf (map #(str/upper-case %) words) (nth commands 18))]  
 (cond  
 (= -1 firstJoinIndex)  
 table  
 (and (= 0 (compare (str/upper-case (nth words (- firstJoinIndex 1))) (nth commands 14)))  
 (= 0 (compare (str/upper-case (nth words (+ firstJoinIndex 2))) (nth commands 19))))  
 (let [forJoiningTable (getFile (nth words (+ 1 firstJoinIndex)))  
 tableJoiningIndex (if (= (nth words (+ 1 firstJoinIndex))  
 (subs (nth words (+ 3 firstJoinIndex))  
 1 (str/last-index-of (nth words (+ 3 firstJoinIndex)) "]")))  
 (+ 3 firstJoinIndex) (+ 5 firstJoinIndex))  
 nameOfColumnInCondition (subs (nth words tableJoiningIndex)  
 (+ 2 (str/last-index-of (nth words tableJoiningIndex) "]"))  
 (count (nth words tableJoiningIndex)))  
 sameKeys (vec (set/intersection (set (keys (first table))) (set (keys (first forJoiningTable)))))  
 sameConditionKeys (not= -1 (.indexOf sameKeys (keyword nameOfColumnInCondition)))  
 correctKeysJoiningTable (renameMapKeys forJoiningTable sameKeys (nth words (+ 1 firstJoinIndex)))]  
 (joinTables (changeJoin  
 table correctKeysJoiningTable  
 (if sameConditionKeys  
 (subvec (vec (map #(if (or (= (nth words (+ 3 firstJoinIndex)) %)  
 (= (nth words (+ 5 firstJoinIndex)) %))  
 (if (= (getTableNameJoinClause %)  
 (nth words (+ 1 firstJoinIndex)))  
 (str % "(" (nth words (+ 1 firstJoinIndex)) ")") %) %) words))  
 (+ 3 firstJoinIndex)  
 (+ 6 firstJoinIndex))  
 (subvec (vec words) (+ 3 firstJoinIndex) (+ 6 firstJoinIndex)))  
 (nth words (+ 1 firstJoinIndex)) "innerJoin")  
 (subvec (vec words) (+ 6 firstJoinIndex) (count words)) commands))  
 (and (= 0 (compare (str/upper-case (nth words (- firstJoinIndex 2))) (nth commands 15)))  
 (= 0 (compare (str/upper-case (nth words (- firstJoinIndex 1))) (nth commands 16)))  
 (= 0 (compare (str/upper-case (nth words (+ firstJoinIndex 2))) (nth commands 19))))  
 (let [forJoiningTable (getFile (nth words (+ 1 firstJoinIndex)))  
 tableJoiningIndex (if (= (nth words (+ 1 firstJoinIndex))  
 (subs (nth words (+ 3 firstJoinIndex))  
 1 (str/last-index-of (nth words (+ 3 firstJoinIndex)) "]")))  
 (+ 3 firstJoinIndex) (+ 5 firstJoinIndex))  
 nameOfColumnInCondition (subs (nth words tableJoiningIndex)  
 (+ 2 (str/last-index-of (nth words tableJoiningIndex) "]"))  
 (count (nth words tableJoiningIndex)))  
 sameKeys (vec (set/intersection (set (keys (first table))) (set (keys (first forJoiningTable)))))  
 sameConditionKeys (not= -1 (.indexOf sameKeys (keyword nameOfColumnInCondition)))  
 correctKeysJoiningTable (renameMapKeys forJoiningTable sameKeys (nth words (+ 1 firstJoinIndex)))]  
 (joinTables (changeJoin  
 table correctKeysJoiningTable  
 (if sameConditionKeys  
 (subvec (vec (map #(if (or (= (nth words (+ 3 firstJoinIndex)) %)  
 (= (nth words (+ 5 firstJoinIndex)) %))  
 (if (= (getTableNameJoinClause %)  
 (nth words (+ 1 firstJoinIndex)))  
 (str % "(" (nth words (+ 1 firstJoinIndex)) ")") %) %) words))  
 (+ 3 firstJoinIndex)  
 (+ 6 firstJoinIndex))  
 (subvec (vec words) (+ 3 firstJoinIndex) (+ 6 firstJoinIndex)))  
 (nth words (+ 1 firstJoinIndex)) "fullOuterJoin")  
 (subvec (vec words) (+ 6 firstJoinIndex) (count words)) commands))  
 (and (= 0 (compare (str/upper-case (nth words (- firstJoinIndex 1))) (nth commands 17)))  
 (= 0 (compare (str/upper-case (nth words (+ firstJoinIndex 2))) (nth commands 19))))  
 (let [forJoiningTable (getFile (nth words (+ 1 firstJoinIndex)))  
 tableJoiningIndex (if (= (nth words (+ 1 firstJoinIndex))  
 (subs (nth words (+ 3 firstJoinIndex))  
 1 (str/last-index-of (nth words (+ 3 firstJoinIndex)) "]")))  
 (+ 3 firstJoinIndex) (+ 5 firstJoinIndex))  
 nameOfColumnInCondition (subs (nth words tableJoiningIndex)  
 (+ 2 (str/last-index-of (nth words tableJoiningIndex) "]"))  
 (count (nth words tableJoiningIndex)))  
 sameKeys (vec (set/intersection (set (keys (first table))) (set (keys (first forJoiningTable)))))  
 sameConditionKeys (not= -1 (.indexOf sameKeys (keyword nameOfColumnInCondition)))  
 correctKeysJoiningTable (renameMapKeys forJoiningTable sameKeys (nth words (+ 1 firstJoinIndex)))]  
 (joinTables (changeJoin  
 table correctKeysJoiningTable  
 (if sameConditionKeys  
 (subvec (vec (map #(if (or (= (nth words (+ 3 firstJoinIndex)) %)  
 (= (nth words (+ 5 firstJoinIndex)) %))  
 (if (= (getTableNameJoinClause %)  
 (nth words (+ 1 firstJoinIndex)))  
 (str % "(" (nth words (+ 1 firstJoinIndex)) ")") %) %) words))  
 (+ 3 firstJoinIndex)  
 (+ 6 firstJoinIndex))  
 (subvec (vec words) (+ 3 firstJoinIndex) (+ 6 firstJoinIndex)))  
 (nth words (+ 1 firstJoinIndex)) "leftJoin")  
 (subvec (vec words) (+ 6 firstJoinIndex) (count words)) commands)))))

Мал 48. Основна функція з`єднання таблиць

З самого початку ми знаходимо перше входження ключового слова. Головна ідея це з’єднання таблиць зліва направо. Дивимось який в нас тип з`єднання та рекурсивно викликаємо нашу функцію, але передаємо туди вже з`єднанні дві таблиці. І також обрізаємо наші умови з’єднання, допоки не залишиться ключових слів. Перша умова – це умова виходу з функції, якщо не залишилось ключових слів. І остальні 3 це просто перевірка який тип з`єднання потрібно використати: inner join, full outer join, left join. Почнімо з нашого inner join’а. В усіх трьох типах в нас будуть такі самі змінні. Перша це ми наперед підвантажуємо наш другий файл, щоб з’єднати його з першим. Друга це ми знаходимо на якому місці знаходиться умова з назвою другої таблиці. Третє це назва колонки другої таблиці по якій буде відбуватись з’єднання. Потім ми знаходимо однакові за назвою колонки, щоб перейменувати їх в другій таблиці, щоб якось їх відрізняти + при мерджі, в нас значення першої заміняться на значення другої. Далі дивимось чи в умові в нас співпадають колонки. Якщо так то потрібно буде замінити саму колонку в нашій умові. І останнє це ми перейменовуємо наші колонки в другій таблиці. Для цього я написав функцію renameMapKeys, в яку ми передаємо нашу другу таблицю, однакові ключі та назву таблиці. Розглянемо цю функцію:

(defn renameMapKeys [table sameKeys tableName]  
 (map #(set/rename-keys % (zipmap sameKeys (map (fn [currKey]  
 (keyword (subs (str currKey "(" tableName ")") 1))) sameKeys))) table))

Мал 49. Функція для перейменування ключів

В цій реалізації ми використовуємо функцію [rename-keys](https://clojuredocs.org/clojure.set/rename-keys), яка приймає на вхід мапу та мапу з ключами. Приклади можете подивитись за наведеним посиланням вище. Тут ми просто заміняємо назву колонки на назву колонки + (назва таблиці). Тут все досить просто.

Повернімось до головної функції. Маючи перейменовані ключі та чи потрібно змінювати умову, можна рекурсивно викликати нашу функцію, передаючи основні параметри в функцію changeJoin, Там ми ще додатково парсим та вирішуємо, яку функцію викликати для джоіна. Взагалі код для цих 3 віток майже одинаковий за виключенням того, що ми передаємо різний текст в typeOfJoin параметра функції changeJoin. Туди ми передаємо дві таблиці, умову(за необхідності змінюємо її, а саме якщо колонки для з`єднання однакові, то додаємо до колонки другої таблиці (назва другої колонки). Перейдімо до changeJoin:

(defn changeJoin [currentTable forJoiningTable condition forJoiningTableName typeJoin]  
 (let [textInParenthesesFirst (if (and (not= -1 (.indexOf (first condition) "("))  
 (not= -1 (.indexOf (first condition) ")"))  
 (> (.indexOf (first condition) ")")  
 (.indexOf (first condition) "(")))  
 (subs (first condition)  
 (inc (str/last-index-of  
 (subs (first condition)  
 0 (.indexOf (first condition) "(")) "."))  
 (inc (.indexOf (first condition) ")"))) [])  
 textInParenthesesLast (if (and (not= -1 (.indexOf (last condition) "("))  
 (not= -1 (.indexOf (last condition) ")"))  
 (> (.indexOf (last condition) ")")  
 (.indexOf (last condition) "(")))  
 (subs (last condition)  
 (inc (str/last-index-of  
 (subs (last condition)  
 0 (.indexOf (last condition) "(")) "."))  
 (inc (.indexOf (last condition) ")"))) [])]  
 (cond  
 (= "innerJoin" typeJoin)  
 (if (= (getTableNameJoinClause (first condition)) forJoiningTableName)  
 (innerJoin '() currentTable forJoiningTable  
 (if (empty? textInParenthesesLast)  
 (last (str/split (last condition) #"\.")) textInParenthesesLast)  
 (if (empty? textInParenthesesFirst)  
 (last (str/split (first condition) #"\.")) textInParenthesesFirst))  
 (innerJoin '() currentTable forJoiningTable  
 (if (empty? textInParenthesesFirst)  
 (last (str/split (first condition) #"\.")) textInParenthesesFirst)  
 (if (empty? textInParenthesesLast)  
 (last (str/split (last condition) #"\.")) textInParenthesesLast)))  
 (= "fullOuterJoin" typeJoin)  
 (if (= (getTableNameJoinClause (first condition)) forJoiningTableName)  
 (fullOuterJoin '() currentTable forJoiningTable  
 (if (empty? textInParenthesesLast)  
 (last (str/split (last condition) #"\.")) textInParenthesesLast)  
 (if (empty? textInParenthesesFirst)  
 (last (str/split (first condition) #"\.")) textInParenthesesFirst))  
 (fullOuterJoin '() currentTable forJoiningTable  
 (if (empty? textInParenthesesFirst)  
 (last (str/split (first condition) #"\.")) textInParenthesesFirst)  
 (if (empty? textInParenthesesLast)  
 (last (str/split (last condition) #"\.")) textInParenthesesLast)))  
 (= "leftJoin" typeJoin)  
 (if (= (getTableNameJoinClause (first condition)) forJoiningTableName)  
 (leftJoin '() currentTable forJoiningTable  
 (if (empty? textInParenthesesLast)  
 (last (str/split (last condition) #"\.")) textInParenthesesLast)  
 (if (empty? textInParenthesesFirst)  
 (last (str/split (first condition) #"\.")) textInParenthesesFirst))  
 (leftJoin '() currentTable forJoiningTable  
 (if (empty? textInParenthesesFirst)  
 (last (str/split (first condition) #"\.")) textInParenthesesFirst)  
 (if (empty? textInParenthesesLast)  
 (last (str/split (last condition) #"\.")) textInParenthesesLast))))))

Мал 50. Функція для парсингу та визначення типу з`єднання

З самого початку ми пробуємо виділити нашу колонку якщо вона такого типу назва\_колонки(назва\_таблиці). Якщо не виходить, то просто будем парсити по “.”. Далі нам потрібно поставити на першу позицію назву колонки першої таблиці. Для того щоб дістати назву колонки з умови використовуємо функцію getTableNameJoinClause:

(defn getTableNameJoinClause [str]  
 (if (and (= 0 (.indexOf str "[")) (not= -1 (.indexOf str "]")))  
 (subs str 1 (.indexOf str "]"))  
 (nth (str/split str #"\.") 0)))

Мал 51. Функція для парсингу назви таблиці

Тут все просто. Дивимось чи слово починається з квадратної дужки. Якщо там – беремо текст з неї, ні – беремо текст до першої точки.

І так в функції changeJoin ми просто викликаємо потрібний нам join та передаємо туди пустий список(який потім стане результатом), дві таблиці(спочатку перша потім друг), два ключових слова по яким потрібно з’єднувати. І тепер перейдімо до реалізації саме цих типів з’єднань.

(defn innerJoin [resultTable currentTable forJoiningTable  
 currentTableKeyword forJoiningTableKeyword & indexOfIter]  
 (let [acceptableElements (if (empty? indexOfIter)  
 (getAcceptableElements currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword 0)  
 (getAcceptableElements currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword (first indexOfIter)))]  
 (cond  
 (empty? indexOfIter)  
 (innerJoin  
 (map #(merge (nth currentTable 0) %) acceptableElements)  
 currentTable forJoiningTable currentTableKeyword forJoiningTableKeyword 1)  
 (= (first indexOfIter) (- (count currentTable) 1))  
 (concat  
 resultTable  
 (map #(merge (nth currentTable (first indexOfIter)) %) acceptableElements))  
 *:else* (innerJoin  
 (concat  
 resultTable  
 (map #(merge (nth currentTable (first indexOfIter)) %) acceptableElements))  
 currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword (+ 1 (first indexOfIter))))))

Мал 52. Inner join

Розказую головну ідею всіх join’ів. З самого початку, коли ми маємо дві таблиці та дві колонки по яким з’єднувати, ми добавляємо допоміжний аргумент, за допомогою якого ми будемо ітеруватись по нашій першій таблиці. Для кожного рядка першої таблиці будемо знаходити елементи з другої таблиці, які підходять по нашій умові та будемо мерджити наш рядок зі всіма тими які підходять. І так для кожного рядка поки значення нашого лічильника не достигне останнього індекса першої таблиці. Розберімо функцію, яка знаходить підходящі рядки з другої таблиці:

(defn getAcceptableElements [currentTable forJoiningTable  
 currentTableKeyword forJoiningTableKeyword indexOfIter]  
 (filter #(= (get (nth currentTable indexOfIter) (keyword currentTableKeyword))  
 (get % (keyword forJoiningTableKeyword))) forJoiningTable))

Мал 53. Функція, яка шукає потрібні елементи для з’єднання

Сюди ми передаємо дві таблиці, дві колонки та поточний індекс рядка. Потім просто фільтруємо другу таблиці по нашій умови. Беремо значення з поточного рядка першої таблиці та зрівнюємо його з іншими(за заданими ключами).

І так коли ми отримали поточні підходящі елементи, переходимо до наших умов. Перша – коли в нас ще немає лічильника ми беремо 0 по індексу елемент та мерджимо і рекурсивно визиваємо нашу функцію. Друга умова – це в нас береться останній рядок першої таблиці і ми повинні вернути результат. Тут ми просто використовуємо функцію [concat](https://clojuredocs.org/clojure.core/concat), яка з’єднує два списки в один(concat [1 2] [3 4] => [1 2 3 4]) і вертаємо нашу таблицю. І остання умова, це коли ми між 0 та останнім індексом. Просто рекурсивно викликаємо нашу функцію та з’єднуємо поточний результат з тим, що вже є. Такий перший етап використовується і в двох інших join’ах тільки добавляються деякі деталі. Розглянемо саме ці деталі:

(defn leftJoin [resultTable currentTable forJoiningTable  
 currentTableKeyword forJoiningTableKeyword & indexOfIter]  
 (let [acceptableElements (if (empty? indexOfIter)  
 (getAcceptableElements currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword 0)  
 (getAcceptableElements currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword (first indexOfIter)))  
 emptyMapForJoiningKeys (pasteEmptyAndMakeMap (first forJoiningTable))]  
 (cond  
 (empty? indexOfIter)  
 (leftJoin  
 (if (empty? acceptableElements)  
 (list (merge (nth currentTable 0) emptyMapForJoiningKeys))  
 (map #(merge (nth currentTable 0) %) acceptableElements))  
 currentTable forJoiningTable currentTableKeyword forJoiningTableKeyword 1)  
 (= (first indexOfIter) (- (count currentTable) 1))  
 (concat  
 resultTable  
 (if (empty? acceptableElements)  
 (list (merge (nth currentTable (first indexOfIter)) emptyMapForJoiningKeys))  
 (map #(merge (nth currentTable (first indexOfIter)) %) acceptableElements)))  
 *:else* (leftJoin  
 (concat  
 resultTable  
 (if (empty? acceptableElements)  
 (list (merge (nth currentTable (first indexOfIter)) emptyMapForJoiningKeys))  
 (map #(merge (nth currentTable (first indexOfIter)) %) acceptableElements)))  
 currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword (+ 1 (first indexOfIter))))))

Мал 54. Left Join

Тут ми бачимо добавилась така штука як emptyMapForJoiningKeys. Ця конструкція слугує для добавлення напівпустого рядка, якщо відсутні рядки в acceptableElements. В іннер джоіні ми просто конкатували з пустим списком і в нас залишалось все незмінне. Тут ми при пустому acceptableElements добавляємо наступний рядок: дані рядка з першої таблиці + всі колонки з другої але з пустими значеннями. Функція, яка генерує пусті мапи:

(defn pasteEmptyAndMakeMap [map]  
 (zipmap (keys map) (take (count map) (repeat ""))))

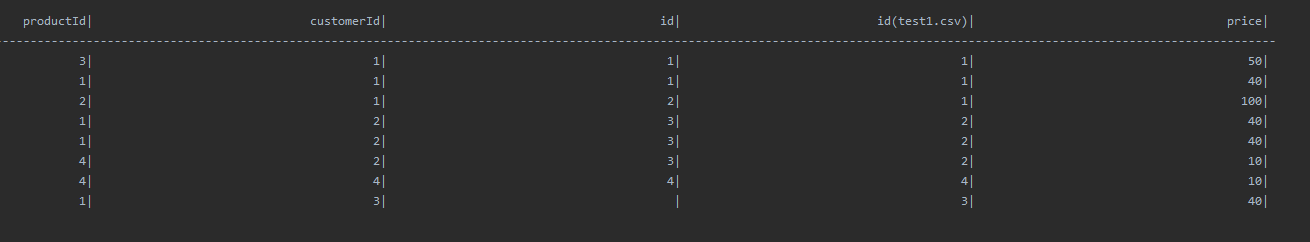
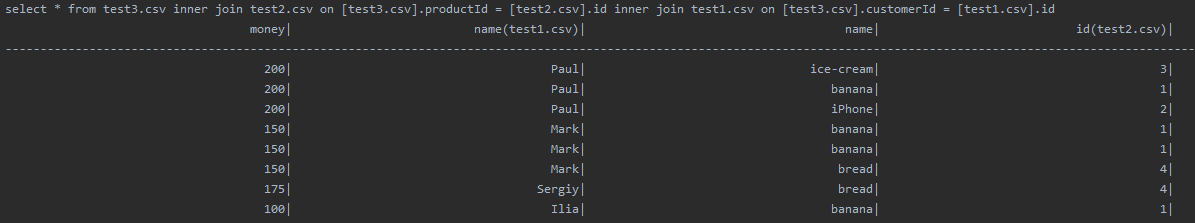
Мал 55. Отримання пустої мапи

Ця функція просто приймає мапу і генерує нову з такими самими ключами тільки пустими значеннями. В left join все абсолютно те саме, що і в іннер, тільки при пустих acceptableElements добавляється напів порожній рядок. Перейдімо до останнього типу:

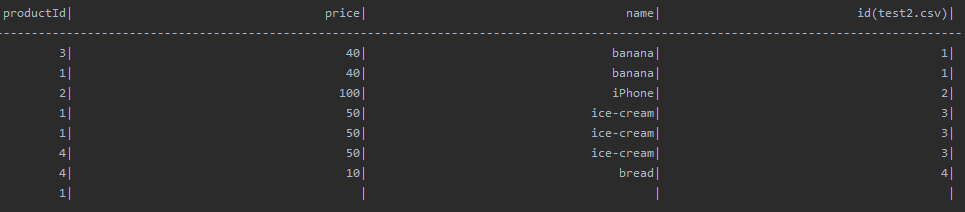
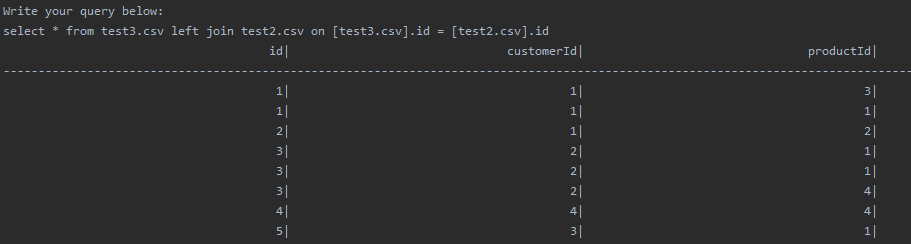
(defn fullOuterJoin [resultTable currentTable forJoiningTable  
 currentTableKeyword forJoiningTableKeyword & indexOfIterListOfUsed]  
 (let [acceptableElements (if (empty? indexOfIterListOfUsed)  
 (getAcceptableElements currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword 0)  
 (getAcceptableElements currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword (first indexOfIterListOfUsed)))  
 emptyMapForJoiningKeys (pasteEmptyAndMakeMap (first forJoiningTable))  
 emptyMapCurrentKeys (pasteEmptyAndMakeMap (first currentTable))]  
 (cond  
 (empty? indexOfIterListOfUsed)  
 (fullOuterJoin  
 (if (empty? acceptableElements)  
 (list (merge (nth currentTable 0) emptyMapForJoiningKeys))  
 (map #(merge (nth currentTable 0) %) acceptableElements))  
 currentTable forJoiningTable currentTableKeyword forJoiningTableKeyword 1 acceptableElements)  
 (= (first indexOfIterListOfUsed) (- (count currentTable) 1))  
 (concat  
 resultTable  
 (if (empty? acceptableElements)  
 (list (merge (nth currentTable (first indexOfIterListOfUsed)) emptyMapForJoiningKeys))  
 (map #(merge (nth currentTable (first indexOfIterListOfUsed)) %) acceptableElements))  
 (map #(merge emptyMapCurrentKeys %)  
 (into '() (set/difference (set forJoiningTable) (set (second indexOfIterListOfUsed))))))  
 *:else* (fullOuterJoin  
 (concat  
 resultTable  
 (if (empty? acceptableElements)  
 (list (merge (nth currentTable (first indexOfIterListOfUsed)) emptyMapForJoiningKeys))  
 (map #(merge (nth currentTable (first indexOfIterListOfUsed)) %) acceptableElements)))  
 currentTable forJoiningTable currentTableKeyword  
 forJoiningTableKeyword (inc (first indexOfIterListOfUsed))  
 (concat (second indexOfIterListOfUsed) acceptableElements)))))

Мал 56. Full outer join

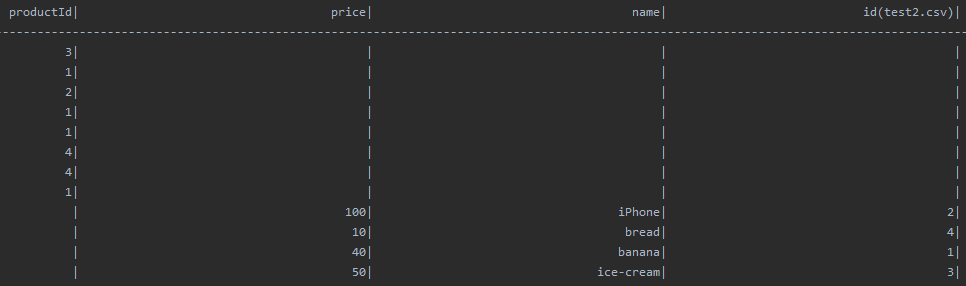
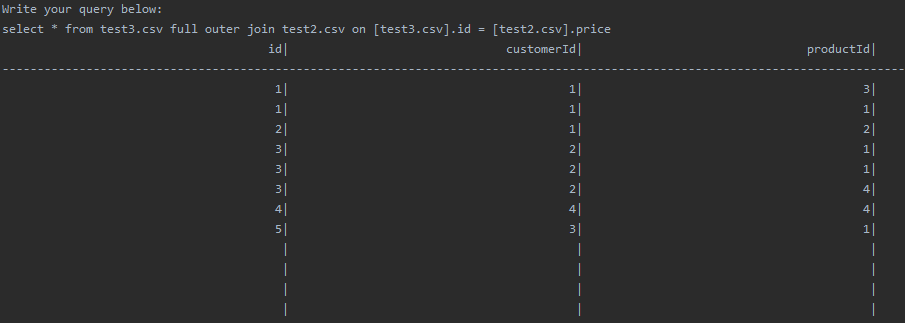
Тут вже поцікавіше. В нашому indexOfIterListOfUsed, зберігаються два типа даних. Перший це як і в попередніх версіях – лічильник. А вже другий це список всіх використаних елементів з другої таблиці, щоб в самому кінці ми з невикористаних рядків зробили напів порожні як в лефт джоіні тільки навпаки і вставили їх. Тобто тут відбувається все теж саме, що і в left join тільки тут ми ще додатково зберігаємо елементи які були використані із другої таблиці. І в умові коли вже в нас останній рядок ми також до результату додаємо список із напів порожніх елементів тільки тут присутні значення невикористаних елементів з другої таблиці. Начебто все по 6-й лабораторній роботі. Отож, перейдімо до останньої лабораторної. Але перед цим протестуємо наші join’и.



Мал 57. Два inner join’а



Мал 58. Left join



Мал 59. Full outer join

Тепер спокійно можна переходити до останньої лабораторної.

Не будемо тянути і перейдімо до моєї реалізації. Ще в 4 лабораторній роботі ми розбирали функцію getMainTable, де була перевірка на те, чи є в умові ключове слово GROUP:

(defn checkForGroupBy [words commands]  
 (if (= 1 (- (.indexOf (map #(str/upper-case %) words) (nth commands 5))  
 (.indexOf (map #(str/upper-case %) words) (nth commands 20))))  
 true false))

Мал 60. Функція для перевірки наявності GROUP

Якщо ж це слово наявне, викликається функція groupBy:

(defn groupBy [file columns words commands]  
 (let [distinctTable (getTableGroupBy  
 file (str/split (nth words (+ 2 (.indexOf  
 (map #(str/upper-case %) words)  
 (nth commands 20)))) #","))  
 havingCondition (conditionHaving (.indexOf (map #(str/upper-case %) words) (nth commands 21)) words)  
 resultTable (having distinctTable file havingCondition commands)]  
 (map #(let [splitCol (re-seq #"\"[^\"]+\"|[\S]+" %)  
 indexOfNameCase (.indexOf (keys (first file)) (keyword (last splitCol)))]  
 (cond  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (str/starts-with? (str/upper-case %) (nth commands 11)))  
 (groupByForAggregate file resultTable % "countFunc")  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (str/starts-with? (str/upper-case %) (nth commands 12)))  
 (groupByForAggregate file resultTable % "avgFunc")  
 (and (not= -1 (.indexOf % "(")) (not= -1 (.indexOf % ")"))  
 (str/starts-with? (str/upper-case %) (nth commands 13)))  
 (groupByForAggregate file resultTable % "minFunc")  
 (and (> (count splitCol) 1) (= (str/upper-case (first splitCol)) (nth commands 22)))  
 (if (or (not= -1 indexOfNameCase) (not= -1 (.indexOf (keys (first resultTable)) "case")))  
 (getColumn (if (= -1 indexOfNameCase) "case" (last splitCol)) resultTable)  
 (if (= (str/upper-case  
 (nth splitCol (- (count splitCol) 2))) (nth commands 27))  
 (checkCaseCondition resultTable % (nth splitCol (- (count splitCol) 1)) commands)  
 (checkCaseCondition resultTable % "case" commands)))  
 *:else* (getColumn % resultTable)))  
 columns)))

Мал 61. Основна функція Group By

Данна функція може нагадувати нам getAllColumns, адже дійсно ці функції схожі, тільки тут я ряд відмінностей. З самого початку ми робимо нашу табличку без повторень і з тими рядками, які були вказані в group by. Тобто якщо ми в group by вказали лише id, то таблиця буде складатися лише з id. Розглянемо функцію getTableGroupBy:

(defn getTableGroupBy [file columns]  
 (let [elementsForDelete (vec (set/difference (set (map #(subs (str %) 1) (keys (first file)))) (set columns)))]  
 (distinct (map #(deleteElementsMap % elementsForDelete) file))))

Мал 62. Функція для взяття унікальних комбінацій елементів вказаних в group by

Тут ми просто беремо засовуємо в [set](https://clojuredocs.org/clojure.core/set) наші ключі та беремо логічну різницю між всім ключами та тими які передані в group by. Потім пробігаємся по всім рядкам таблиці та видаляємо ті ключі, які ми знайшли вище. І в самому кінці робимо операцію distinct, про яку я вже розказував в 4 лабораторній. Подивимось як саме ми видаляємо наші ключі:

(defn deleteElementsMap [map keysForDelete]  
 (if (empty? keysForDelete)   
 map  
 (deleteElementsMap (dissoc map (keyword (first keysForDelete))) (next keysForDelete))))

Мал 63. Функція для видалення заданих ключів з мапи

Це рекурсивна функція, яка проходиться по кожному елементу для видалення та за допомогою операції [dissoc](https://clojuredocs.org/clojure.core/dissoc) видаляє заданий ключ. Працює допоки в списку є ключі для видалення. Отож повернімось до головної функції. Далі ми виділяємо(якщо є) умови фільтрації по нашим агрегатним функціям за допомогою ключового слова HAVING. Ну до цього дійдемо пізніше. І результуюча таблиця, це вже відфільтрована завдяки having. Далі все ж за тим сценарієм як в getAllCoumns. Тільки тут викликається спеціальна функція groupByForAggregate. Розгляньмо її:

(defn groupByForAggregate [file resultTable column aggFunc]  
 (cond  
 (= "countFunc" aggFunc)  
 (map #(first (countFunc  
 (getColumn (subs column (+ (.indexOf column "(") 1) (- (count column) 1))  
 (filter (fn [currentRowFile] (containsForMap currentRowFile %)) file))))   
 resultTable)  
 (= "avgFunc" aggFunc)  
 (map #(first (avgFunc  
 (getColumn (subs column (+ (.indexOf column "(") 1) (- (count column) 1))  
 (filter (fn [currentRowFile] (containsForMap currentRowFile %)) file))))  
 resultTable)  
 (= "minFunc" aggFunc)  
 (map #(first (minFunc  
 (getColumn (subs column (+ (.indexOf column "(") 1) (- (count column) 1))  
 (filter (fn [currentRowFile] (containsForMap currentRowFile %)) file))))   
 resultTable)))

Мал 64. Функція для формування нової колонки з даними агрегатних функцій

Тут ми дивимось яку функцію потрібно викликати і в залежності від того мапимся по нашій колонці та викликаємо ту агрегатну функцію, яку задали. В цю агрегатну функцію ми передаємо відфільтровану колонку по наступному критерію: чи є наш рядок з результуючої таблиці підмножиною рядка з нормальної цілої таблиці. Якщо так то кидаєм його в нашу колонку. Для того щоб визначити чи є одна мапа підмножиною іншою, ми використовуємо функцію containsForMap:

(defn containsForMap [m1 m2]  
 (and (every? (set (keys m1)) (keys m2))  
 (every? #(= (m1 %) (m2 %)) (keys m2))))

Мал 65. Функція, яка перевіряє чи є одна мапа підмножиною іншої

Тут ми просто дивимось за допомогою функції [every?](https://clojuredocs.org/clojure.core/every_q) чи містить в собі одна мапа всі ключі іншої і чи значення цих ключів є рівними. Якщо так то true, ні – false.

Ось таким чином і відбувається наш group by. Тепер подивимось як відбувається фільтрація за допомогою нашого having, але спочатку подивимось на те, як ми дістаємо наші умови:

(defn conditionHaving [havingIndex words]  
 (if (= -1 havingIndex) [] (if (> (+ havingIndex 4) (count words))  
 [] (subvec (vec words) (+ 1 havingIndex) (+ 4 havingIndex)))))

Мал 66. Функція для виділення умови для having

Дана функція просто бере перші 3 слова після ключового слова having, та якщо ці 3 слова виходять за межі розміру всіх слів, вертаємо пустий вектор.

Ну а тепер розглянемо having:

(defn having [distinctTable mainTable condition commands]  
 (let [boolTable (getBoolTableOfResults distinctTable mainTable condition commands)]  
 (if (empty? condition)  
 distinctTable  
 (filter #(nth boolTable (.indexOf distinctTable %)) distinctTable))))

Мал 67. Основна функція having

Головна ідея: в нас є наша унікальна таблиця, без повторень при group by. Для того щоб відфільтрувати її нам потрібна булева таблиця, яка буде містити в собі true false’и в відповідності до рядка. Для прикладу: [{id: 1} {id: 2} {id: 3}] [true false false] => [{:id 1}]. Дивимось, якщо умови відстуні то просто вертаємо таблицю. Якщо ж вони є то фільтруємо нашу таблицю по нашим умовам(в нашому випадку вона 1). І так подивимось як ж ми отримуємо нашу булеву таблицю:

(defn getBoolTableOfResults [distinctTable mainTable condition commands]  
 (map #(checkConditionForCurrRow  
 (filter (fn [currentRow] (containsForMap currentRow %)) mainTable) condition commands)  
 distinctTable))

Мал 68. Функція, яка формує відповідну булеву таблицю

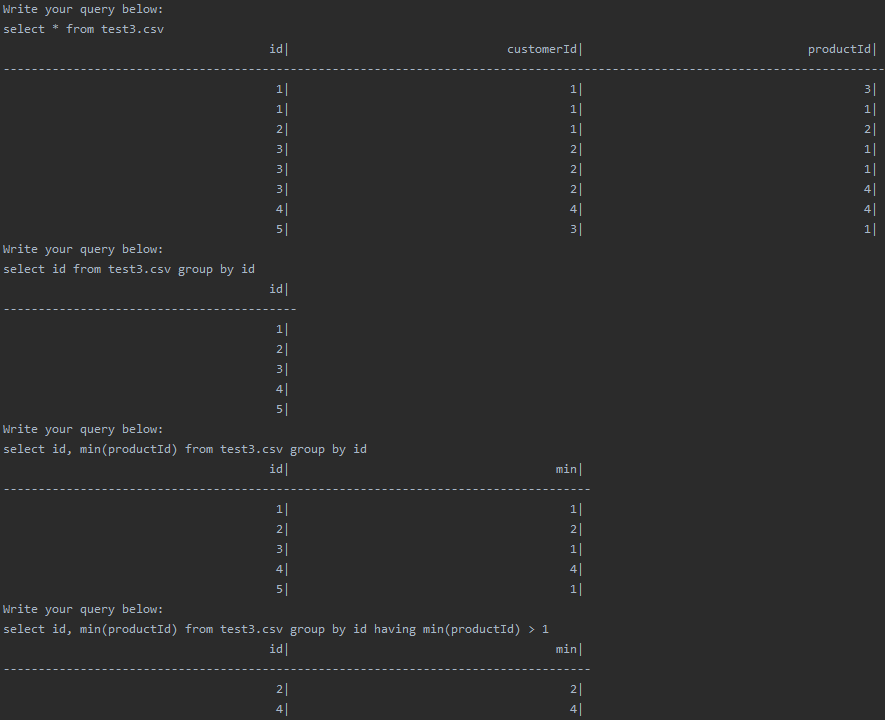
За допомогою мап ми проходимось по нашій таблиці та для кожного рядка викликаємо функцію checkConditionForCurrRow в яку ми передаємо наш рядок та таблицю, яка містить всі рядки в яких є наш рядок(так само як ми робили для агрегатних функцій в group by). Отож подивимось як ми провіряємо нашу умову:

(defn checkConditionForCurrRow [tableContainsRow condition commands]  
 (let [aggregateFunc (if (or (= (str/upper-case (first (str/split (first condition) #"\("))) (nth commands 11))  
 (= (str/upper-case (first (str/split (first condition) #"\("))) (nth commands 12))  
 (= (str/upper-case (first (str/split (first condition) #"\("))) (nth commands 13)))  
 (str/upper-case (first (str/split (first condition) #"\(")))  
 (str/upper-case (first (str/split (last condition) #"\("))))  
 indexOfAggFunc (if (str/starts-with? (str/upper-case (first condition)) aggregateFunc) 0 2)  
 rowToAggregate (subs (nth condition indexOfAggFunc)  
 (inc (.indexOf (nth condition indexOfAggFunc) "("))  
 (- (count (nth condition indexOfAggFunc)) 1))]  
 (if (= "=" (nth condition 1))  
 (cond  
 (= aggregateFunc (nth commands 11))  
 (if (=  
 (if (= indexOfAggFunc 0)  
 (read-string (nth condition 2)) (read-string (nth condition 0)))  
 (get (first (countFunc (getColumn rowToAggregate tableContainsRow))) *:count*))  
 true false)  
 (= aggregateFunc (nth commands 12))  
 (if (=  
 (if (= indexOfAggFunc 0)  
 (read-string (nth condition 2)) (read-string (nth condition 0)))  
 (get (first (avgFunc (getColumn rowToAggregate tableContainsRow))) *:avg*))  
 true false)  
 (= aggregateFunc (nth commands 13))  
 (if (=  
 (if (= indexOfAggFunc 0)  
 (read-string (nth condition 2)) (read-string (nth condition 0)))  
 (get (first (minFunc (getColumn rowToAggregate tableContainsRow))) *:min*))  
 true false))  
 (cond  
 (and (= aggregateFunc (nth commands 11)) (= 0 indexOfAggFunc))  
 (if (> (get (first (countFunc (getColumn rowToAggregate tableContainsRow))) *:count*)  
 (read-string (nth condition 2))) true false)  
 (and (= aggregateFunc (nth commands 12)) (= 0 indexOfAggFunc))  
 (if (> (get (first (avgFunc (getColumn rowToAggregate tableContainsRow))) *:avg*)  
 (read-string (nth condition 2))) true false)  
 (and (= aggregateFunc (nth commands 13)) (= 0 indexOfAggFunc))  
 (if (> (get (first (minFunc (getColumn rowToAggregate tableContainsRow))) *:min*)  
 (read-string (nth condition 2))) true false)  
 (and (= aggregateFunc (nth commands 11)) (= 2 indexOfAggFunc))  
 (if (> (read-string (nth condition 2))  
 (get (first (countFunc (getColumn rowToAggregate tableContainsRow))) *:count*))  
 true false)  
 (and (= aggregateFunc (nth commands 12)) (= 2 indexOfAggFunc))  
 (if (> (read-string (nth condition 2))  
 (get (first (avgFunc (getColumn rowToAggregate tableContainsRow))) *:avg*))  
 true false)  
 (and (= aggregateFunc (nth commands 13)) (= 2 indexOfAggFunc))  
 (if (> (read-string (nth condition 2))  
 (get (first (minFunc (getColumn rowToAggregate tableContainsRow))) *:min*))  
 true false)))))

Мал 69. Головна функція, яка перевіряє нашу умову в having

З самого початку ми визначаємо, яку ж агрегатну функцію нам передали. Потім визначаємо на якому місці стоїть дана агрегатна функція, щоб при порівнянні правильно її поставити. Далі виділяємо колонку, яку треба передати в агрегатну функцію. Коли вже визначили всі локальні значення, йдемо до нашого if. Тут ми дивимось чи наша операція “=”, якщо так то далі дивимось яка саме в нас агрегатна функція і зрівнюємо результат нашої агрегатної функції з заданим в умові значенням. А саме індекс агрегатної функції, тобто на якому місці в умові вона стоїть, потрібний для того щоб правильно поставити наш результат агрегатної функції в аргумент операції “>”, адже (= value\_1 value\_2) = (= value\_2 value\_1); (> value\_1 value\_2) != (> value\_2 value\_1). Ну і результат цих операцій ми передаємо в нашу булеву функцію. Думаю логіка ясна)

Давайте трішки відпочинемо від читання документації і потестуємо нашу програмку(звісно я зроблю це за вас, можете піти поїсти).



Мал 70. Результати

І перейдемо до останнього етапу, до ключового слова CASE. Якщо повернутись до функції getAllColumns, то можна побачити, що там є перевірка на це ключове слово. Для цього ми в цій функції і парсимо наші колонки. Там відразу ми знаходимо чи є ключове слово AS, щоб дати назву нашій колонці. Якщо немає, то буде вона названа досить креативно – case. Звісно це жарт, ахаха. Жарт, що досить креативно. А названа вона дійсно буде case за замовчанням. І там визивається така функція як checkCaseCondition, яка є головною для формування новою колонки по запиту користувача.

(defn checkCaseCondition [file conditions nameOfMap commands]  
 (let [splitConditions (re-seq #"\"[^\"]+\"|[\S]+" conditions)  
 filteredConditions  
 (divideConditions (subvec (vec splitConditions)  
 (inc (.indexOf (map #(str/upper-case %) splitConditions) (nth commands 22)))  
 (.indexOf (map #(str/upper-case %) splitConditions) (nth commands 26))) commands)]  
 (map #(hash-map (keyword nameOfMap) (searchCorrCondCase filteredConditions %)) file)))

Мал 71. Головна функція ключого слова CASE

На вхід ми отримуємо готову таблицю, відфільтровану по всім параметрам запиту, отримуємо наш запит від CASE до END або до першого слова після AS, отримуємо case, або слово після AS та команди, звідки ми беремо ключові слова. Далі розбиваємо наші слова через пробіл, і фільтруємо їх за допомогою функції divideConditions наступним чином: [“CASE” “WHEN” “col1” “=” “2” “THEN” “It’s 2” “WHEN” “col1” “>” “2” “THEN” “It’s more then 2” “ELSE” “It’s less then 2”] => [“col1” “=” “2” “It’s 2” “col1” “>” “2” “It’s more then 2” “It’s less then 2”]. В функцію divideConditions перадємо слова від CASE до END (не включно). Подивимось на функцію:

(defn divideConditions [conditions commands & resultConditions]  
 (let [indexOfWhere (.indexOf (map #(str/upper-case %) conditions) (nth commands 23))  
 indexOfThen (.indexOf (map #(str/upper-case %) conditions) (nth commands 24))  
 indexOfElse (.indexOf (map #(str/upper-case %) conditions) (nth commands 25))]  
 (cond  
 (and (= 0 indexOfWhere) (= 4 indexOfThen) (= 6 (count conditions)))  
 (concat (if (empty? resultConditions) resultConditions (first resultConditions))  
 (subvec (vec conditions) 1 4) (conj [] (nth conditions 5)))  
 (and (= 0 indexOfWhere) (= 4 indexOfThen) (<= 8 (count conditions)))  
 (divideConditions (subvec (vec conditions) 6 (count conditions)) commands  
 (concat (if (empty? resultConditions) resultConditions (first resultConditions))  
 (subvec (vec conditions) 1 4) (conj [] (nth conditions 5))))  
 (= 0 indexOfElse)  
 (concat (if (empty? resultConditions) resultConditions (first resultConditions)) (conj [] (nth conditions 1)))  
 *:else* resultConditions)))

Мал 72. Функція для формування зручних умов

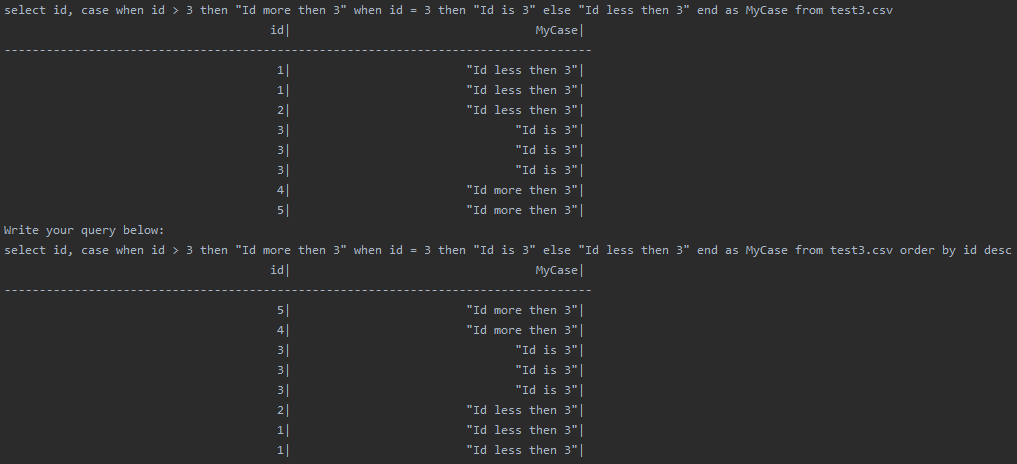
Це є рекурсивна функція, яка по частинкам парсить наш список. Спочатку знаходить перші входження таких ключових слів як: WHEN, THEN, ELSE. Якщо це запит і він містить всього 6 елементів і 1 елемент це WHERE, то відразу видаємо результат з’єднуючи 2, 3, 4, 6 елементи з вже ранніми результатами. Якщо ж розмір списка більше ніж 6, ми просто рекурсивно визиваємо цю функцію, з’єднавши 2, 3, 4, 6 елементи з ранніми результатами. Ну і якщо список починається з ELSE то просто з’єднуємо другий елемент з ранніми результатами. Ну якщо нічого не підходить, повертаємо значення яке скупчилось в аргументі resultConditions.

Отож, наші умови ми красиво поділили і можна переходити далі. В основній функції ми виконуємо наступну дію: мапимось по нашому файлі та для створюємо нові мапи з назвою case або ту яку ми передали та зі значенням, яке повертає функція searchCorrCondCase. Вона приймає поточний рядок та наші відфільтровані умови та повертає значення вказане в одній з умов, а якщо немає ELSE то пустий рядок. Розглянемо її:

(defn searchCorrCondCase [conditions currRow]  
 (let [indexOfValue (if (= -1 (.indexOf (keys currRow) (keyword (nth conditions 0)))) 0 2)]  
 (cond  
 (= 1 (count conditions))  
 (first conditions)  
 (= 4 (count conditions))  
 (if (= "=" (nth conditions 1))  
 (if (= 0 (compareForStringNumber  
 (if (= \" (first (nth conditions indexOfValue))  
 (last (nth conditions indexOfValue)))  
 (subs (nth conditions indexOfValue)  
 1 (- (count (nth conditions indexOfValue)) 1)) (nth conditions indexOfValue))  
 (get currRow (keyword (nth conditions (if (= 0 indexOfValue) 2 0))))))  
 (nth conditions 3) "")  
 (if (= 0 indexOfValue)  
 (if (> (compareForStringNumber  
 (if (= \" (first (nth conditions 0))  
 (last (nth conditions 0)))  
 (subs (nth conditions 0)  
 1 (- (count (nth conditions 0)) 1)) (nth conditions 0))  
 (get currRow (keyword (nth conditions 2)))) 0)  
 (nth conditions 3) "")  
 (if (> (compareForStringNumber  
 (get currRow (keyword (nth conditions 0)))  
 (if (= \" (first (nth conditions 2))  
 (last (nth conditions 2)))  
 (subs (nth conditions 2)  
 1 (- (count (nth conditions 2)) 1)) (nth conditions 2))) 0)  
 (nth conditions 3) "")))  
 *:else* (if (= "=" (nth conditions 1))  
 (if (= 0 (compareForStringNumber  
 (if (= \" (first (nth conditions indexOfValue))  
 (last (nth conditions indexOfValue)))  
 (subs (nth conditions indexOfValue)  
 1 (- (count (nth conditions indexOfValue)) 1))  
 (nth conditions indexOfValue))  
 (get currRow (keyword (nth conditions (if (= 0 indexOfValue) 2 0))))))  
 (nth conditions 3) (searchCorrCondCase (subvec (vec conditions) 4 (count conditions)) currRow))  
 (if (= 0 indexOfValue)  
 (if (> (compareForStringNumber  
 (if (= \" (first (nth conditions 0))  
 (last (nth conditions 0)))  
 (subs (nth conditions 0)  
 1 (- (count (nth conditions 0)) 1)) (nth conditions 0))  
 (get currRow (keyword (nth conditions 2)))) 0)  
 (nth conditions 3) (searchCorrCondCase (subvec (vec conditions) 4 (count conditions)) currRow))  
 (if (> (compareForStringNumber  
 (get currRow (keyword (nth conditions 0)))  
 (if (= \" (first (nth conditions 2))  
 (last (nth conditions 2)))  
 (subs (nth conditions 2)  
 1 (- (count (nth conditions 2)) 1)) (nth conditions 2))) 0)  
 (nth conditions 3) (searchCorrCondCase (subvec (vec conditions) 4 (count conditions)) currRow)))))))

Мал 73. Основна функція, яка вертає правильне значення виходячи з наших умов

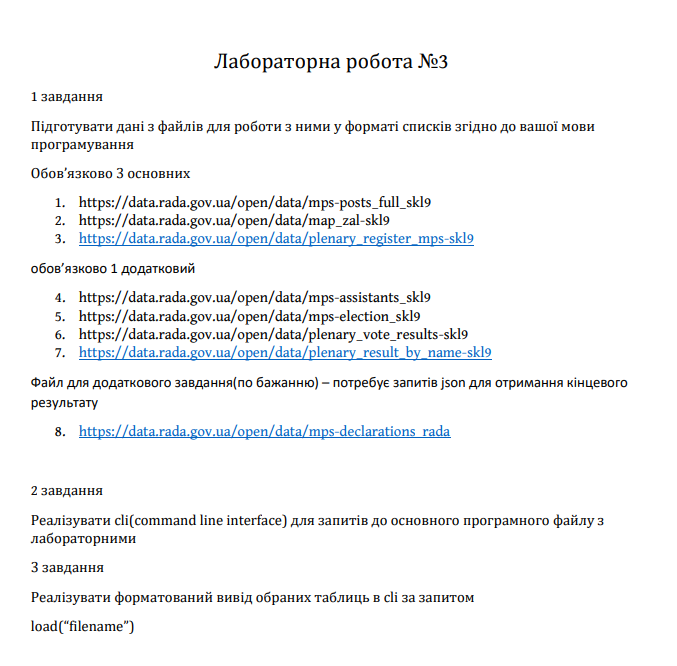
З самого початку ми знаходимо на якому місці знаходиться значення з яким потрібно порівнювати значення колонки. Взагалі ідея така: ми дивимось на кількість елементів в списку, якщо там рівно 4, то перевіряємо нашу умову, якщо true то вертаємо 4 елемент(той який йшов після then), якщо ж елемент тільки 1, це означає, що це елемент який йде після else, тобто просто вертаємо його, що і зроблено в 1 умові. Ну а якщо розмір більше, то дивимось на поточну умову, якщо вона true вертаємо 4 елемент, якщо false – обрізаємо ці 4 елементи та заново викликаємо цю функцію. Всередині цих великих останніх умов йде просте порівняння елементів. Спочатку дивимось яка операція “=” чи “>”. Якщо ж це “=” ми дивимось чи це на текст, якщо текст беремо значення всередині лапок, адже ви повинні передавати текст в лапках, і зрівнюємо, якщо ж це цифра то просто приводимо з тексту до цифри і зрівнюємо. З операціює “>” трішки складніше, адже потрібно брати до уваги на якому місці стоїть назва колонки. Ну з цим ми зустрічались коли розбирали WHERE. Тому перейдімо до тестування.

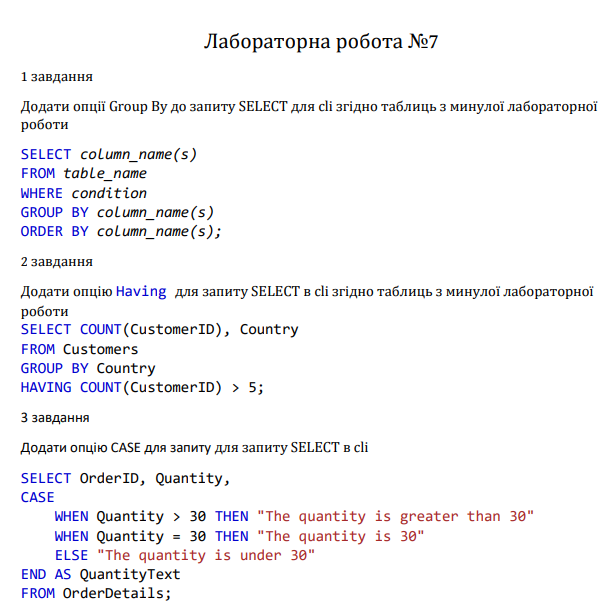
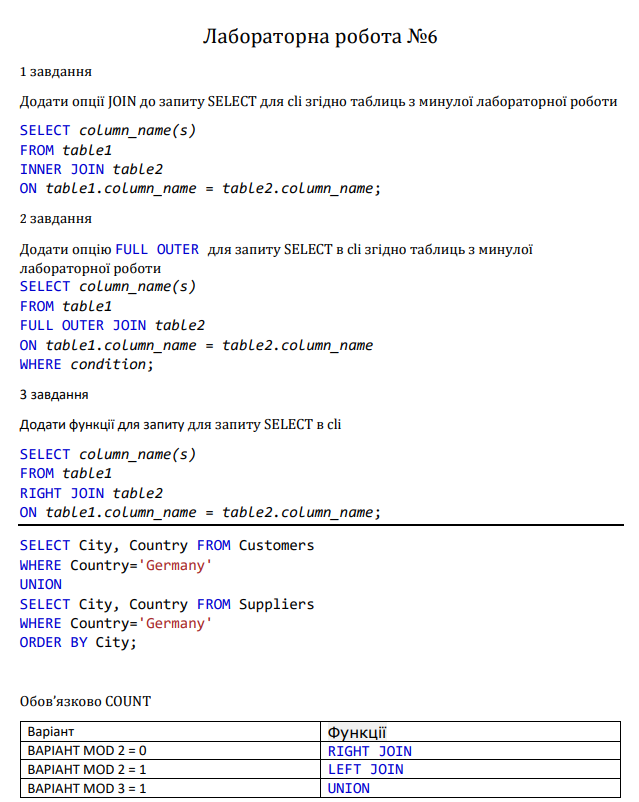
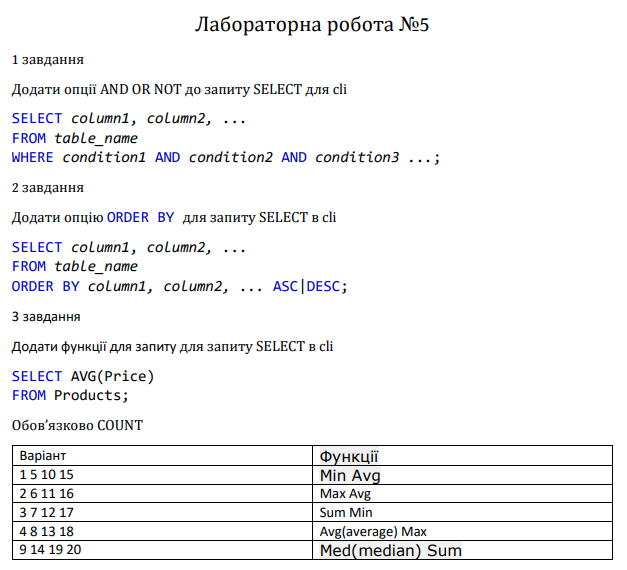
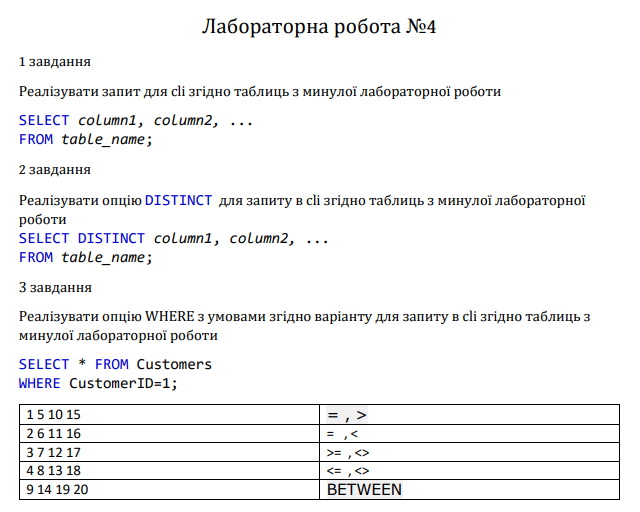


Мал 74. Результати

Ось і пролетіли всі лабки 😊

А я ж забув в звіт добавити самі умови лабораторних. Прикріплю їх нижче:





На цьому все. Дякую за увагу :)