Governors State University

Software Warehouse Report Management Project

Combined and Design

Version 1.0

**Team 4**

**CPSC 4338-SP-2018**

**Governors State University**

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Rev | Date | Purpose | Originator |
| Draft | 02/07/2018 | First draft | Team 4 |
| Draft | 03/25/2018 | Second draft | Team 4 |
| Draft | 4/17/2018 | Third Draft | Team 4 |
| Draft | 4/22/2018 | Fourth Draft | Team 4 |
| 1.0 | 4/26/2018 | Version 1.0 | Team 4 |

Table of Contents

[Revision History 1](#_Toc512487604)

[Project Description 3](#_Toc512487605)

[Definitions and Acronyms 3](#_Toc512487606)

[Project Requirements 4](#_Toc512487607)

[Design Description 5](#_Toc512487608)

[Nightly Processing/Indexing 6](#_Toc512487609)

[Query Module 7](#_Toc512487610)

[Design Units 10](#_Toc512487611)

[Nightly Processing 10](#_Toc512487612)

[*Functional Overview* 10](#_Toc512487613)

[*Associated Requirements* 10](#_Toc512487614)

[Query Module 11](#_Toc512487615)

[*Functional Overview* 11](#_Toc512487616)

[*Associated Requirements* 16](#_Toc512487617)

[Acknowledgements 16](#_Toc512487618)

[References 17](#_Toc512487619)

[Appendices 18](#_Toc512487620)

[Appendix A – Data Sets 18](#_Toc512487621)

[Appendix B – Data Sets 20](#_Toc512487622)

# Project Description

This project is to create an application by which a company can access its information better. It will allow its sales, marketing, and management teams to query information easily and in real-time.

Currently, there is no way for this to happen, and as a result information isn’t shared readily across departments, and it’s extremely difficult to organize and sift through data.

There is no system currently in place for this, so it will be a brand-new feature.

## Definitions and Acronyms

**GSU** – Governors State University

**App** – Software Application

**LME** – Last Minute Endeavors, who this project is for

**Customer** – Customers of Last Minute Endeavors

**Company –** Our customer’s clients

**Query** – A standardized syntax for users to request data from the application

**Field** – Any of the searchable pieces of data, i.e., “city”

**Expression** – Syntax to specify values within a field, i.e., “city=Chicago”

**Record** – The data set for an individual company, stored as a file

**Stack** – A data structure where the last item added is the first one processed (LIFO)

**Pop** – Remove an item from the top of a stack

**Push** – Add an item to the top of a stack

**Queue** – A data structure where the first item added is the first one processed (FIFO)

**Enqueue** – Add an item to the end of a queue

**Dequeue** – Remove the first item from a queue

# Project Requirements

**<LME-Nightly-Data-Processing-01000>(Team 4)**

Process the main data once a night creating sequential files for the indexed fields. It has to organize the data into appropriate data structures, so queries can be processed efficiently and in real-time.

**<LME-Queries-02000>(Team 4)**

**<LME-Queries-Syntax-02100>(Team 4)**

Queries must be entered using specific syntax.

An example query is (salesman=”joe”|salesman=”bill”)&city=”chicago”. Spaces are only to be used within double quotation marks. What’s on the left side of the equality operator (=,>, and <) is the field name to be used. The exact field names to use are listed in Appendix B.

Set operators are |,+ (or), & (and), ! (not), and – (subtraction). Parentheses are also allowed, to specify an exact order of operations. They and are evaluated first, from innermost and working outward.

**<LME-Queries-Output-02200>(Team 4)**

Query will return all company IDs that match the criteria specified by user. The queries must be perfectly accurate to what is typed in by the user. More information on query syntax is listed in <LME-2500>.

Output will be a tab-separated list. In the case of mismatched parentheses appropriate error messages will be displayed.

**<LME-Queries-Runtime-02300>(Team 4)**

Queries must be executable in real-time, and efficiently so multiple users can make use the system at the same time without having delays of more than three seconds.

**<LME-Queries-Searchable-Fields-02400>(Team 4)**

The query will be able to search based off the fields listed in appendix B.

**<LME-Queries-User-Interface-02500>(Team 4)**

User will have a command line interface to enter queries. This will be a console window where queries are entered on the command line, with the results scrolling vertically through the window.

**<LME-Queries-Parentheses-02600>(Team 4)**

User will be able to specify order of operations using parentheses: (). These are evaluated from the innermost pair going outward. An example expression is (A+B-(C+D)). In this case the C+D is evaluated first.

**<LME-Queries-Union-Intersection-02625>(Team 4)**

User will be able to perform intersections using the & (and) operator, as well as being able to perform unions using the | (or) and + (addition) operators.

**<LME-Queries-Set-Subtraction-02650>(Team 4)**

Users can subtract the elements of one set from another by using the – (subtraction) operator. A-B would mean the elements of set B are subtracted from set A.

**<LME-Queries-Not-02675>(Team 4)**

By use of the ! (not) operator users can generate the complement of a set.

**<LME-Queries-Search-Expressions-02700>(Team 4)**

Users can specify which sets and subsets to perform operations on by using = for exact-match searches, and < > (less than and greater than) for ranges. The user will also be able to use regular expressions for the values (right side of the expression).

# Design Description

There are two main components of this application – the nightly processing unit and the query unit. These two pieces are independent of each other functionally, but the nightly processing is what makes it possible for the query processing to be able to execute in real-time.

The nightly processing goes through the client data and builds what is essentially an indexing system. The query system can use this indexing system to provide fast, accurate results for user queries.

These components are explained in detail below.

## Nightly Processing/Indexing

This application is going to collect data once a night, when all transactions and information are updated in the main system. The data is organized in a doubly linked list by customer, with pointers to the first and last company records. There are currently over 22,000 files (records), each its own sequential file, and in most cases links to additional linked lists.

The following flow chart explains the basic flow of this unit. The full explanation will appear later in this document, but this should give a basic idea of how it works.



## Query Module

This unit controls how queries are entered, parsed and processed and how the output is displayed.

When a user wishes to collect information, they create a query. They can look for a match or specify a range by using =, <, and >. They can enter fields to search on, and to use the following operations (in order of precedence in processing):

(, ) Parentheses – what’s inside is evaluated before anything else

! The not operator – returns the complement

&, | Intersection/union/addition, also referred to as and/or

+ Addition is the same thing as union, just easier for users

- Set subtraction

Example queries and their symbolic representations (A, B, etc) would be:

*All companies that owe more than 2500*

AAmountOwed>2500

*All the companies Larry handles except for the ones in Detroit*

A&!B Salesman=”Larry”&!city=”Detroit”

A-B Salesman=”Larry”–city=”Detroit”

*All companies in Chicago other than those in zip codes 60629 and 60634*

A&(B+C) City=”Chicago”&(zipcode=60629+zipcode=60634)

A-(B|C) City=”Chicago”–(zipcode=60629|zipcode=60634)

*All companies in Chicago and all companies in Detroit*

A+B City=”Chicago”+city=”Detroit”

A|B City=”Chicago”|city=”Detroit”

The example query that will be used throughout the rest of the document will be:

*All the companies that are not based in Detroit or Chicago that owe over $5000 that haven’t been called since 10/01/17.*

The actual query syntax for that will be:

!(city=”Detroit”+city=”Chicago”)&(AmountOwed>5000-LastContact>09302017)

Representing that symbolically we have:

!(A+B)&(C-D)



The above diagram illustrates some of the common uses for the query module.

Theusers of the query system will be salesmen, managers, accounts receivable employees, and various others that need to be able to access data about their clients.

Because there is such a variety of users, there’s an even greater variety in the types of queries that will be run, so flexibility are the two primary requirements.



# Design Units

## Nightly Processing

### *Functional Overview*

This part of the app must accumulate and index all data once a night. It has to reference the data from appropriate data structures so queries and reports can be processed efficiently and in real-time.

This application will basically create an indexing system using sequential files. Each file will be named after the field it is indexing and will store the corresponding company IDs. For example, city is an indexed field, so there will be a file for each city containing all company IDs for businesses in that city. So for the cities headquartered in Chicago, there will be a file named Chicago. That file will be in the ‘city’ directory.

Each night, before the data is processed, each sequential file that this application created the previous night will be moved to a backup file.

When the data is processed, as the linked list is iterated through, the indexed fields (see Appendix B) will have their values and the company ID appended to the file. All directories and filenames will be in lowercase.

*Note that in the case of AmountOwed, or any currency field the data will be broken up into ranges. All companies owing less than or equal to 100 will go into a file named 100, all files under or equal to 200 and over 200 will be in their own file, and so on.*

Algorithm:

Backup files – move previous night’s files and directories to a backup

Create directories for each indexed field listed in Appendix B

For each company record in doubly linked list:

For each field, directory will be ~/fieldname (lowercase)

// For AmountOwed, where, this method isn’t practical

// the ranges will be broken up. <2500 will access

// files 500,1000,1500,2000,2500 and so on

Filename will be value (i.e., chicago) (in lowercase)

If filename doesn’t exist, create it

Append companyID to file

For records with additional linked lists (i.e. hardware):

Iterate list, create/append to files as necessary

### *Associated Requirements*

<LME-Nightly-Data-Processing-01000>(Team 4)

## Query Module

### *Functional Overview*

This unit will validate query syntax, process it, and return the result as a set of company IDs.

The validation will consist of making sure that all parentheses match up correctly.

In processing the query, the following order of operations will be used, with equal operators being evaluated from left to right):

( ) Parenthesis are always evaluated first, from innermost going out

! (not) Complement is evaluated next

& (and), |,+(or) Intersections and unions (and/or) are next, from left to right

- Set subtraction is last, from left to right

Search expressions (city=”Chicago”) will be represented symbolically by letters from this point going forward, for simplicity.

The example we’re going to be using and tracing through is:

!(A+B)&(C-D)

The basic algorithm we’re going to be using is

Validate parentheses

Process the query with correct order of operations

Output the list of company IDs returned by the query processing

The first two of those are detailed below.

*Note that for the sake of simplicity we’re dealing with symbols rather than going into the detailed parsing of the expressions or the sets themselves.*

**Validate parenthesis**

Here, we’re making sure first that there are equal numbers of left and right parentheses and that they line up correctly.

We use a counter to keep track of the left parentheses. When we encounter a right parenthesis we first check if the counter is positive, since a right parenthesis can only exist to close a left one.

If we encounter a right parenthesis without an existing left, or if we reach the end of processing and have unmatched left parentheses we inform the user of the problem, throw an exception and exit processing.

We keep track of how many of each parenthesis there are so if there is an error we can display the counts to the user.

The pseudocode for the validating the parentheses:

For each char in query

If char == ‘(‘

Increment counter

Increment left

Else if char == ‘)’

If counter <= 0

Throw exception: mismatched parentheses

Exit

// Matches a left parenthesis

Increment right

Decrement counter

If counter != 0

Throw exception with counts for left and right

Exit

// Parentheses match

Our example query: !(A+B)&(C-D) would go through with the variables changing as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Symbol** | **Actions taken** | **Counter** | **Left** | **Right** |
| ( | counter++, left++ | 1 | 1 | 0 |
| ) | counter--,right++ | 0 | 1 | 1 |
| ( | counter++, left++ | 1 | 2 | 1 |
| ) | counter--,right++ | 0 | 2 | 2 |

**Process the query with correct order of operations**

This query must now be processed using the correct order of operations. To do that, we’re going to use a shunting algorithm. The order of operations goes as follows:

* Parenthesis
* In the case of nested parenthesis, they are evaluated from innermost going outward
* Complement (!)
* Unions (| and +) and intersections (&) are evaluated next, from left to right
* Subtraction is evaluated last and from left to right

The algorithm itself uses a queue for processing, another for input, and a stack for the operators.

As the query is processed, symbols/search expressions are moved to the input queue.

Operators are pushed onto the operator stack, and if they are of lower precedence than the previous operator, then they are popped from the stack and added to the process queue.

Left parentheses are added to the operator stack, and when a right parenthesis is encountered pop operators from stack and enqueue them into process until a left parenthesis is reached. That parenthesis is popped, and the rest of the input queue is processed.

Once the entire input queue has been processed, any remaining operators on the stack are popped and enqueued in process.

For each element in the input queue:

If the element is not an operator:

Push it on process queue

Else (element is an operator):

While operator at top of stack has greater precedence

* Or equal precedence and is left-associative

Pop operator from operator stack and push into process queue

Enqueue operator into process

If element is ‘(‘:

Push onto operator

If token is ‘)’:

While top operator in stack != ‘(‘:

Pop from operator stack onto process queue

Pop left bracket from stack

When all elements have been read

For each operator on stack

Pop operator onto process queue

In our example query !(A+B)&(C-D) as we iterate through it we get the following values (note that both the stack and the queues are displayed from left to right in order of when the next item is pulled from it):

|  |  |  |
| --- | --- | --- |
| **Input** | **Operator** | **Process** |
| ! | ! |  |
| ( | (! |  |
| A | (! | A |
| + | +(! | A |
| B | +(! | AB |
| ) | ! | AB+ |
| & | & | AB+! |
| ( | (& | AB+! |
| C | (& | AB+!C |
| - | -(& | AB+!C |
| D | -(& | AB+!CD |
| ) | & | AB+!CD- |
| END |  | AB+!CD-& |

The original query !(A+B)&(C-D) has now become a queue with the elements AB+!CD-&.

This queue expresses the original query in postfix notation, stored in the process queue.

To evaluate this query we’re going to process the queue in the following way, using a stack:

For each token in process

If token is operator

Temp2 := pop from stack

If token is !

Result := evaluate(temp2)

Else

Temp1 := pop from stack

Result := evaluate(temp1, temp2, token)

Push result on stack

Else

Push token on stack

Result := pop from stack

Output the results

The evaluate function, used in the pseudocode above calls the appropriate function to handle any necessary set operations. The sets that these operations are performed on are generated by another function that’s detailed a little later in this document.

Evaluate(op1, op2, operator)

// If no op2 operator must be !

// In this case, return complement of op1

// This would be implemented by overloading the function

If operator is !

return complement of operand1

else

switch (operator)

&: return intersection of op1 and op2

|: return union of op1 and op2

+: return union of op1 and op2

-: return op1 – op2

This will interpret our example !(A+B)&(C-D) in the following order:

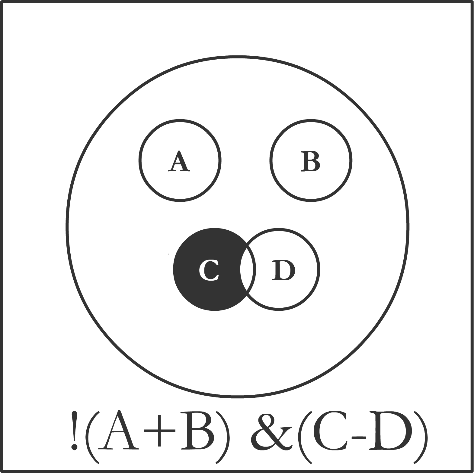
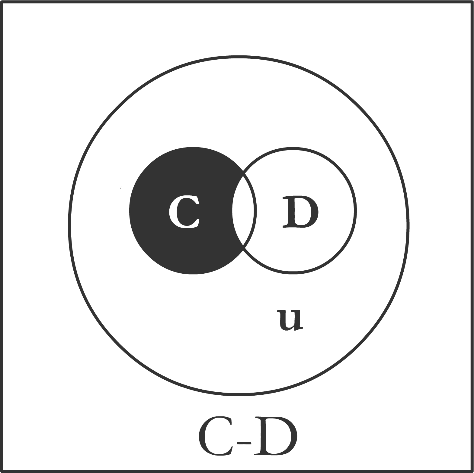
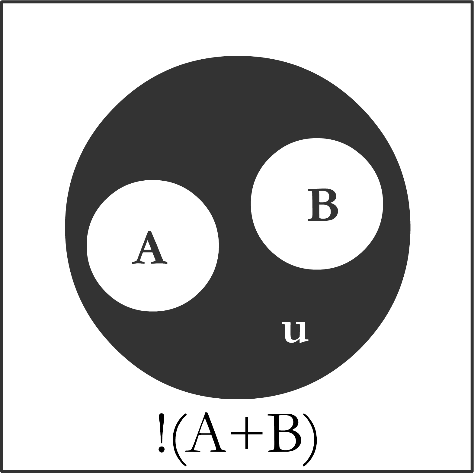
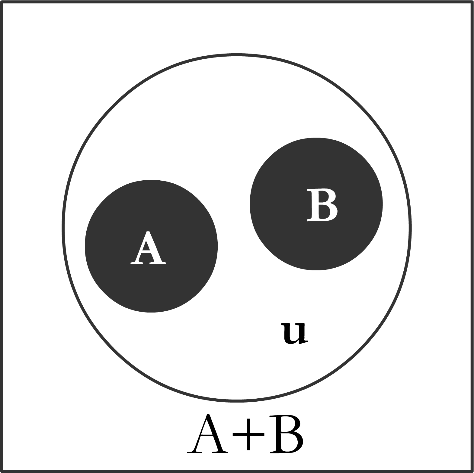
ResultAB := union of A and B

ResultAB := complement of ResultAB

ResultCD := C – D (set subtraction)

Result := intersection of ResultAB and ResultCD

The diagrams below illustrate the order of operations and how the sets change. First, A+B is evaluated, after which the complement of the answer is taken. The third step switches to the other set of parentheses, and the enclosed C-D. Finally, the intersection of the last two result sets is taken and we have our final set.



To get the sets for the operands the processor will open the appropriate file in the directory for that fieldname. In other words, for ‘Salesman=”Jim”’ the expression will be made lowercase, and what’s on the left will be the directory. What’s on the right will have its quotation marks stripped and will be the filename. So, the company IDs for that expression will be in ~/salesman/jim.

If the exact filename isn’t found, it will be treated as a regular expression and each file in the directory will be checked against the pattern and all results collated. This will be actually be a power set: all the subsets of the field where the fieldname matches the expression.

If the field is a currency field, where the files are broken up into smaller chunks to be looked up as ranges it will return all the files under the value specified. In other words, if the expression is for ‘AmountOwed>5000’ it will (after making AmountOwed lowercase), return a power set of all the IDs from files starting at 5000.

The pseudocode for evaluating the search expressions:

Expression := lowercase of expression

Field := left side of =,<, or >

value := right side of =, <, or >

Operator := what’s remaining

If file exists with name ~field/value

Return contents of file

// value is treated as a regular expression, or operator is < or >

Files = directory list of field

For each file in files

// Evaluate regular expression, using operator, for example:

// field is ‘salesman’, operator is ‘=’, value = ‘dav\*’

// contents of files that have names matching value’s pattern

// are added to a results set

evaluate if value matches operator

Return results

At this point the query has been completely processed and the results are displayed to the user.

*Note that the actual retrieval of the company IDs is based on the directory-file structure explained in the nightly processing section of this document. For example, City=”Chicago” would simply retrieve the contents of the file named Chicago in the directory named City.*

### *Associated Requirements*

<LME-Queries-Syntax-02100>(Team 4)

<LME-Queries-Output-02200>(Team 4)

<LME-Queries-Runtime-02300>(Team 4)

<LME-Queries-Searchable-Fields-02400>(Team 4)

<LME-Queries-User-Interface-02500>(Team 4)

<LME-Queries-Parentheses-02600>(Team 4)

<LME-Queries-Union-Intersection-02625>(Team 4)

<LME-Queries-Set-Subtraction-02650>(Team 4)

<LME-Queries-Not-02675>(Team 4)

<LME-Queries-Search-Expressions-02700>(Team 4)

# Acknowledgements

Authors: David Halek, Eric Gomez, Jelane Carroll, Drew Slack, Prof. Neng-Shin Chen

# References

Epp, S., 2011. *Discrete Mathematics with Applications*. 4th ed. Boston MA, USA: Brooks/Cole Cengage Learning.

# Appendices

## Appendix A – Data Sets

The data is currently organized as a doubly linked list with a record for each company.

Customer

Customer ID

Customer Name

Location Info (Address, City, State, Zip)

Current Balance

Salesperson *(Link to salesperson set)*

Customer Contacts *(Links to customer contact set)*

Tech (Hardware/software details) *(Links to TechInfo set)*

Notes *(Links to Notes set*

Invoices *(Links to Invoices set)*

Salesperson

Salesperson Name

Employee ID

Salesperson-Customer

This will be a linked list of all the salesman

Each salesman will be linked to a linked list of all their customers

Customer Contacts

Name: (Nickname, First, Last)

Date of birth

Note (whatever, wife, kid, likes sheep, etc)

Previous contact link

Next contact link

Tech

Hardware specifics – Vendor, Model, Make, Generation, Type (PC, Laptop…)

Software specifics – OS, OS version, Application Software, App version

Previous pointer

Next Pointer

Notes

Date

Next call Date

Customer Contact *(link to customer contacts)*

Note

Action Item

Previous pointer

Next Pointer

Invoices

Date

Sales Rep

Customer Contact (*link to customer contact)*

Sales orders

Sales amount

Paid

Invoice Balance

Note (10% discount applied)

Action item (Follow up after 2/10/18)

Previous pointer

Next pointer

## Appendix B – Data Sets

Each of the following represents a data set the users can query. These will each have their own directory, and within that directory will be files named after their value that will contain all the company IDs with that value.

For example – all the companies that are in Chicago will be have their IDs in ~/city/Chicago

City

State

Zip

CurrentSalesman

CurrentBalance

ContactDate

ContactedBy

LastCallDate

NextCallDate

ActionItem

InvoiceDate6

InvoiceAmount

AmountOwed

SaleAmount

SaleDate

SoldBy

Hardware

HardwareType

OperatingSystem

OSVersion

AppSoftware

AppSoftVersion