Department of Computer Science

1 Castle Point Terrace
Hoboken, NJ 07030
stevens.edu

CS562 – The Project

"The Idea"

This document is designed to help you understand the scope of the project and also demonstrate what your project code is supposed to accomplish (at a high level). It contains a sample of what your project will need to generate, corresponding to the ESQL and the equivalent Phi expression in the beginning of the document. It is important to note that this is an output of your project code (not the project code itself). So, first and foremost, please pay close attention to how the Phi expression "maps to" the code below.

Though it looks like a real code (C syntax based), it's a pseudo code, and as for the syntax of certain parts, please take them with a big grain of salt.

I also included a small code segment on the last page illustrating the use of 'dictionary' data structure of Python which can be helpful to use if you are using Python.

```
#------------------------------------------------------------------------------
# EMF query in ESQL
#------------------------------------------------------------------------------
ESQL Query:
select cust, count(ny.quant), sum(nj.quant), max(ct.quant)
  from sales
group by cust; ny, nj, ct
such that ny.cust = cust and ny.state = 'NY',
          nj.cust = cust and nj.state = 'NJ',
               ➔ . . . example of a predicate for a "dependent aggregate" . . .
               ➔ 2.cust = cust and 2.state = 'NJ' and 2.quant > avg_1_quant
          ct.cust = cust and ct.state = 'CT'

#------------------------------------------------------------------------------
# EMF query in Phi operation
#------------------------------------------------------------------------------
Phi Expression:
# 1. S - projected columns / expressions
cust, count_1_quant, sum_2_quant, max_3_quant
# 2. n - number of grouping variables
3
# 3. V - grouping attributes
cust
# 4. F-VECT - vector of aggregate functions
count_1_quant, sum_2_quant, max_3_quant
# 5. PRED-LIST - list of predicates for grouping var's
1.state = 'NY'; 2.state = 'NJ'; 3.state = 'CT'
# 6. HAVING
NONE
```

```
#------------------------------------------------------------------------------
# PART 1 of 2: define the data structure (mf_struct)
#------------------------------------------------------------------------------

struct {
        char    cust[50];               #-- 3. V (grouping attrib)
        int     count_1_quant;          #-- 4. F-VECT (list/vector of agg. Func's)
        int     sum_2_quant;            #-- 4. F-VECT (list/vector of agg. Func's)
        int     max_3_quant;            #-- 4. F-VECT (list/vector of agg. Func's)
} mf_struct [500];

/*---
        Example of your project code generating the code to define the 'mf_struct'
        for the ESQL above:

        To get the details of the schema of the 'sales' table, you have 2 options:

        1.  Have hard-coded schema data for 'sales' table in your project code,

                e.g., schema[("cust", "varchar(50)"), ("prod", "varchar(50)"), … ]
        -or-

        2. Use 'information_schema.columns' table to get the schema information…

        . . . . .

        #-------------------------------------------------------------------------------------
        # sample code of how you to generate the code for mf_struct
        # the following code assumes that you're storing the 6 operands of Phi
        # in separate variables: e.g., you can have a variable V (of array or list type)
        # to store the list of group-by attributes and their schema data and F_VECT
        # (of array or list type) to store the list of aggregate functions and the corresponding
        # types, etc.
        #-------------------------------------------------------------------------------------

        printf ("struct {\n");
        printf ("       %s %s[%d];\n", V[0].type, V[0].attrib, V[0].size); # cust
```

```
        # printf ("    %s %s[%d];\n", V[1].type, V[1].attrib, V[1].size); # prod (if for 2nd g.v.)
        printf ("      %s %s;\n", F_VECT[0].type, F_VECT[0].agg);
        printf ("      %s %s;\n", F_VECT[1].type, F_VECT[1].agg);
        printf ("      %s %s;\n", F_VECT[2].type, F_VECT[2].agg);
        printf ("} mf_struct[500];\n");
---*/
int     NUM_OF_ENTRIES = 0;

#------------------------------------------------------------------------------
# PART 2 of 2: processing logic
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
lookup (cur_row)
#------------------------------------------------------------------------------
#-- search for a given "group by" attrib. value(s) in mf_struct
{
        for (i = 0; i < NUM_OF_ENTRIES; i++)
        {
                if (mf_struct [i].cust == cur_row.cust)
                        return i;
        }
        return -1;
}

#------------------------------------------------------------------------------
add (cur_row)
#------------------------------------------------------------------------------
#-- adds a new entry in mf_struct, corresponding to a newly found "group by" attrib. value
{
        mf_struct[NUM_OF_ENTRIES].cust = cur_row.cust;

        mf_struct[NUM_OF_ENTRIES].count_1_quant = 0;
        mf_struct[NUM_OF_ENTRIES].sum_2_quant = 0;
        mf_struct[NUM_OF_ENTRIES].max_3_quant = -1;

        NUM_OF_ENTRIES++;
}

#------------------------------------------------------------------------------
output ()
#------------------------------------------------------------------------------
#-- adds a new entry in mf_struct, corresponding to a newly found "group by" attrib. value
{
        printf (". . . . .\n");       # header of the output (from operand S)

        for (int i=0; i<NUM_OF_ENTRIES; i++)
        {
                printf ("%s    %d      %d      %d\n",
                mf_struct[i].cust,
                mf_struct[i].count_1_quant,
                mf_struct[i].sum_2_quant,
                mf_struct[i].max_3_quant);
        }
}
```

```
/*--------------------------------------------------------------------------------------
The following is the diagram from the research paper #2 ("Ad-hoc OLAP Query Processing.PDF")
which outlines the processing logic for ESQL queries - the pseudo code below the diagram is based
on the logic described in the diagram.
--------------------------------------------------------------------------------------*/
```

Product	Month	Year	sum(X.Quantity)	sum(Y.Quantity)

(a) mf-structure of Query Q4

Product	Month	Year	sum(X.Quantity)	sum(Y.Quantity)
A	1	1997		
A	2	1997		
A	5	1997		
B	2	1997		
B	3	1997		
B	6	1997		
B	9	1997		

(b) end of first scan

H :

Product	Month	Year	sum(X.Quantity)	sum(Y.Quantity)
A	1	1997	216	
A	2	1997	122	
A	5	1997	245	269
B	2	1997	455	
B	3	1997	196	
B	6	1997	386	
B	9	1997	265	

t :

Customer	Product	Day	Month	Year	Quantity
12443	A	11	5	1997	24

(c) during second scan

H :

Product	Month	Year	sum(X.Quantity)	sum(Y.Quantity)
A	1	1997	855	241 265
A	2	1997	587	241 265
A	5	1997	898	241 265
B	2	1997	785	411
B	3	1997	1221	411
B	6	1997	823	411
B	9	1997	562	411

t :

Customer	Product	Day	Month	Year	Quantity
12443	A	11	5	1997	24

(d) during third scan

```
#------------------------------------------------------------------------
main()
#------------------------------------------------------------------------
{
        current_row = connect_to_dbms();              # setting cursor to 1st row

        # TABLE SCAN 1: populate mf-struct with distinct values of grouping attribute (V)
        while(1)
        {
                if (end of table)
                        break;

                # look up current_row.cust in mf_struct
                pos = lookup (current_row);
                if (pos = -1)                         # current_row.cust not found in mf_struct
                        add (current_row);

                current_row.next();                   # to the next row
        }


        # grouping variable 1 (1.state = 'NY')
        while()
        {
                if (end of table)
                        break;

                if (current_row.state == 'NY')
                {
                        # look up current_row.cust in mf_struct
                        pos = lookup (current_row, mf_struct);
                        # current_row.cust found in mf_struct
                        mf_struct[pos].count_1_quant++;                    # COUNT()
                }

                current_row.next();                   # to the next row
        }


        # grouping variable 2 (2.state = 'NJ')
        while()
        {
                if (end of table)
                        break;

                if (current_row.state == 'NJ')
                {
                        # look up current_row.cust in mf_struct
                        pos = lookup (current_row, mf_struct);
                        # current_row.cust found in mf_struct
                        mf_struct[pos].sum_2_quant += current_row.quant;     # SUM()
                }

                current_row.next();                   # to the next row
        }

        # grouping variable 3 (3.state = 'CT')
        while()
        {
                if (end of table)
                        break;

                if (current_row.state == 'CT')
                {
                        # look up current_row.cust in mf_struct
                        pos = lookup (current_row, mf_struct);
                        # current_row.cust found in mf_struct
                        current_row.quant > mf_struct[pos].max_1_quant &&    # MAX()
```

```
                        mf_struct[pos].max_1_quant = current_row.quant
                }

                current_row.next();                    # to the next row
        }

        # output the result
        Output();
}
```

If you're using Python, the built-in data structure, "dictionary" can be useful in maintaining the internal data structures of your project code (e.g., for the Phi operands, contents of mf_struct, etc.) – one advantage would be the ability to access an element using more meaning names as indexes vs. numeric indexes.

The following is a simple code illustrating the use of dictionary data structre.

```
# Create a dictionary of customers and their corresponding quantities
cust_quant = {
    "Sam": 100,
    "Joe": 90,
    "Mia": 80,
    "Dan": 70,
    "Sue": 60
}

# Accessing and printing the quantities of individual customers
print("INITIAL LIST OF CUSTOMERS and QUANTITIES.............................")
print("Sam's quant:", cust_quant["Sam"])
print("Joe's quant:", cust_quant["Joe"])
print("Mia's quant:", cust_quant["Mia"])
print("Dan's quant:", cust_quant["Dan"])
print("Sue's quant:", cust_quant["Sue"])

# Modifying a customer's quantity
cust_quant["Dan"] = 90
print("CHANGING a customer's quantity (Dan)............................")
print("Dan's new quant:", cust_quant["Dan"])

# Adding a new customer and their quantity
cust_quant["Claire"] = 80
print("ADDING a new customer (Claire).............................")
print("Claire's quant:", cust_quant["Claire"])

# Removing a customer from the dictionary
print("REMOVING an existing customer (Joe).............................")
del cust_quant["Joe"]

# Printing all the customers and their quantities using a loop
print("LIST OF CUSTOMERS and QUANTITIES (after the
changes).............................")
for cust, quant in cust_quant.items():
    print(cust, ":", quant)
```