NCEDC UCB/USGS

SeismiQuery

UC Berkeley Seismological Laboratory

I.- introduction:

1.- Overview:

SeismiQuery is a web tool that allows users to query the database for specific information about data availability as well as network, station and channel information. SeismiQuery currently runs on the IRIS Data Management Center Oracle database. The goal of this project was to modify SeismiQuery to make the installation seamless at other data centers, as well as introducing a new language that would be schema independent.

2.- Original Organization:

SeismiQuery used CGI scripts to query the database.

a) The HTML form was converted to a SQL query by a JavaScript function.
b) The SQL query was passed to a PERL script that calls a Java program to execute the query and got the results back from the database. The PERL script then formatted the results on the screen.

- Notes:

Some standardization problems appeared in the original configuration:

- When the SQL query was not complex, the PERL script built it itself instead of calling a JavaScript function.
- The SQL queries were based on the IRIS schema.
- To get response information, SeismiQuery executed remotely a program called *webquery*.
- It is unclear to me how the temporary HTML files were created for the daily and monthly requests. Standalone PERL scripts created these files but those scripts were never called. From the name of the directory where the PERL scripts were (***SeismiQuery/crons/***), I deducted that these HTML files must have been created periodically by cron jobs.

3.- New Organization:

The new organization of SeismiQuery doesn't diverge from the original one but instead tries to standardize the query process. The following states the main points of this organization:

a) A new language for the requests was introduced (MSQL: Meta SeismiQuery Language). This language has its own schema composed of generic data center views and is based on the SQL language.
b) For every request, the HTML form is converted to a MSQL request by a JavaScript function. This MSQL request is then passed as a parameter to a PERL script. This PERL script (**ProcessMSQL)** calls a parser (**MSQL2SQL**) that converts the MSQL request into a data center dependent SQL request. This SQL request is then passed to a program that get the results back from the database and displays them to the user in a convenient fashion. The role of the program **ProcessMSQL** is to process the MSQL query and output the results.

Please note that with this model, the parsing stage is the only step that a data center would have to update to be able to use SeismiQuery.

II.- MSQL Language Definition:

1.- Overview:

The MSQL's grammar was designed based on the one for the SQL command SELECT. However, please note that some aspects in the syntax are different from the original SQL command SELECT. Within SeismiQuery, MSQL queries can only be executed on its own schema, which is composed of the Generic Data Center Views.

2.- Grammar:

The following is a BNF grammar of MSQL.

```
select_command     ::=
                   "SELECT" ["ALL" | "DISTINCT"]
                   ("*" | (displayed_column {"," displayed_column}))
                   "FROM" (selected_table {"," selected_table})
                   ["WHERE" condition]
                   [group_clause]
                   [set_clause]
                   [order_clause]

displayed_column   ::=
                   (table_name "." "*")
                   | (expression [alias])

selected_table     ::=
                   (table_name [alias])

table_name         ::=
                   identifier

alias              ::=
                   identifier
                   | text

identifier         ::=
                   ("letter" {"letter" | "digit" | "_" | "$" | "#"})

text               ::=
                   "\"" string "\""
                   | "'" string "'"

date               ::=
                   "`" "digit" "digit" "digit" "digit" "/" "digit"
                   "digit" "/" "digit" "digit" " " "digit" "digit" ":"
                   "digit" "digit" ":" "digit" "digit" "`"

column_name        ::=
                   identifier
```

```
expression          ::=
                    [table_name "."] column_name
                    | text
                    | number
                    | date
                    | "NULL"
                    | (function ("(" ["ALL" | "DISTINCT"] expression {","
                      expression} ")"))
                    | "(" expression ")"
                    | ("+" | "-" | "PRIOR") expression
                    | expression ("*" | "/" | "-" | "+") expression

condition           ::=
                    expression ("=" | "!=" | "^=" | "<>" | ">" | "<" |
                    ">=" | "<=") (expression | "{" subquery "}")
                    | exp_list ("=" | "!=" | "^=" | "<>") "{" subquery
                      "}"
                    | exp_list ["NOT"] "IN" (exp_list | "{" subquery "}")
                    | expression ["NOT"] "BETWEEN" expression "AND"
                      expression
                    | expression "IS" ["NOT"] "NULL"
                    | ["NOT"] "EXISTS" "{" subquery "}"
                    | [table_name "."] column_name ["NOT"] "LIKE" text
                    | "[" condition "]"
                    | "NOT" condition
                    | condition ("AND" | "OR") condition

exp_list            ::=
                    "(" (expression {"," expression}) ")"

subquery            ::=
                    select_command

group_clause        ::=
                    "GROUP" "BY" exp_list ["HAVING" condition]

set_clause          ::=
                    ("UNION" | "UNION" "ALL" | "INTERSECT" | "MINUS")
                    select_command

order_clause        ::=
                    "ORDER" "BY" sorted_list

sorted_list         ::=
                    (sorted_item {"," sorted_item})

sorted_item         ::=
                    (expression | alias) [("ASC" | "DESC")]

function            ::=   date_fct
                    |     number_fct
                    |     char_fct
                    |     group_fct
                    |     conversion_fct
                    |     other_fct
```

```
group_fct           ::=   AVG
                    |     COUNT
                    |     MAX
                    |     MIN
                    |     STDDEV
                    |     SUM
                    |     VARIANCE

number_fct          ::=   ABS
                    |     CEIL
                    |     FLOOR
                    |     MOD
                    |     POWER
                    |     ROUND
                    |     SIGN
                    |     SQRT
                    |     TRUNC
                    |     ACOS
                    |     ASIN
                    |     ATAN
                    |     ATAN2
                    |     COS
                    |     COSH
                    |     EXP
                    |     LN
                    |     LOG
                    |     SIN
                    |     SINH
                    |     TAN
                    |     TANH

date_fct            ::=   ADD_MONTHS
                    |     LAST_DAY
                    |     MONTHS_BETWEEN
                    |     NEW_TIME
                    |     NEXT_DAY
                    |     SYSDATE
                    |     TRUE2STRING
                    |     STRING2TRUE
```

```
char_fct          ::=  CHR
                  |    INITCAP
                  |    LOWER
                  |    LPAD
                  |    LTRIM
                  |    REPLACE
                  |    RPAD
                  |    RTRIM
                  |    SOUNDEX
                  |    SUBSTR
                  |    TRANSLATE
                  |    UPPER
                  |    ASCII
                  |    INSTR
                  |    LENGTH
                  |    CONCAT
                  |    NLS_INITCAP
                  |    NLS_LOWER
                  |    NLS_UPPER
                  |    SUBSTRB
                  |    INSTRB
                  |    LENGTHB
                  |    NLSSORT

conversion_fct    ::=  CHARTOROWID
                  |    CONVERT
                  |    HEXTORAW
                  |    RAWTOHEX
                  |    ROWIDTOCHAR
                  |    TO_CHAR
                  |    TO_DATE
                  |    TO_NUMBER
                  |    TO_SINGLE_BYTE
                  |    TO_MULTI_BYTE

other_fct         ::=  DECODE
                  |    DUMP
                  |    GREATEST
                  |    LEAST
                  |    NVL
                  |    UID
                  |    USER
                  |    USERENV
                  |    VSIZE
                  |    EMPTY_BLOB
                  |    EMPTY_CLOB
                  |    BFILENAME
                  |    NLS_CHARSET_DECL_LEN
                  |    NLS_CHARSET_ID
                  |    NLS_CHARSET_NAME
```

3.- Generic Data Center Views:

Network Information

GDV_Network
Network_Code
Network_Abbreviation
Network_Full_Name

Station Information

GDV_Station
Network_Code
Station_Code
Station_Start_Date_Time
Station_End_Date_Time
Station_Latitude
Station_Longitude
Station_Elevation
Station_Site

Channel Information

GDV_Channel
Network_Code
Station_Code
Channel_Code
Location_Code
Channel_Start_Date_Time
Channel_End_Date_Time
Channel_Maximum_Drift
Channel_Sampling_Rate
Channel_Flags
Channel_Latitude
Channel_Longitude
Channel_Elevation
Channel_Depth
Channel_Azimuth
Channel_Dip
Channel_Sensor_Type
Channel_Site

Channel/Waveform Information

GDV_Channel_Summary
Network_Code
Station_Code
Channel_Code
Location_Code
Earliest_Waveform
Latest_Waveform

Waveform Information

GDV_Waveform
Network_Code
Station_Code
Channel_Code
Location_Code
Waveform_Start_Date_Time
Waveform_End_Date_Time

Station's Comment Information

GDV_Station_Comment
Network_Code
Station_Code
Station_C_Start_Date_Time
Station_C_End_Date_Time
Station_Comment_Text

Channel's Comment Information

GDV_Channel_Comment
Network_Code
Station_Code
Channel_Code
Location_Code
Channel_C_Start_Date_Time
Channel_C_End_Date_Time
Channel_Comment_Text

Event Information

GDV_Event
Event_Identifier
Event_Date_Time
Event_Latitude
Event_Longitude
Event_Depth
Event_Catalog
Event_Contributor
FE_Region_Code

Magnitude Information

GDV_Magnitude
Event_Identifier
Magnitude_Type
Magnitude_Contributor
Magnitude_Value

Region Information

GDV_Flinn_Engdahl
FE_Region_Code
FE_Seismic_Code
FE_Region_Text

4.- Examples:

Here are a few examples of valid MSQL queries.

- ```
  select distinct +a, -(a.b)*d - -e, (PRIOR (min (a.b, max (c))))
  "foo" from g g1, gh g3, p where not [min (a,b) >= {select min(a)
  from b}] and [[a,b+c,max(d) != {select a from b}] OR [a = (3+5)]]
  and v not like 'bar%' INTERSECT select * from t group by a+b,
  avg(sqrt(b-1)) having c != 3 order by a, b, "foo", (c + -d) desc
  UNION select a.* from b where z,a not in 1,"bar", (3-2) or p not in
  {select p from c} and z+a not between 1+2 and ("bar") and not [a is
  null] and not exists {select value from sinus};
  ```

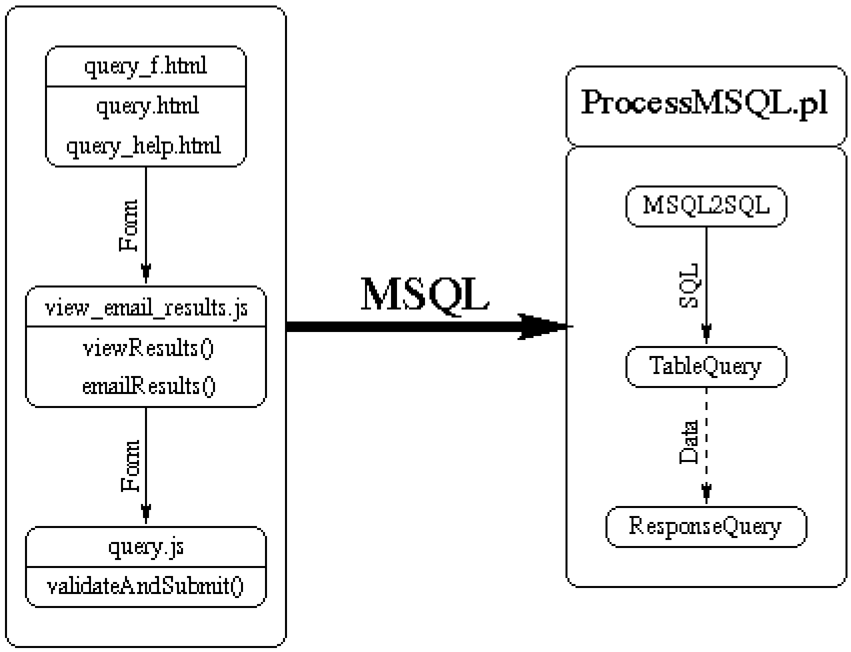
- ```
  select DISTINCT Network_Code, Station_Code, Channel_Code,
  w.Location_Code, TRUE2STRING (w.Waveform_Start_Date_Time),
  TRUE2STRING (Waveform_End_Date_Time) from GDV_Waveform w where
  Network_Code = 'BK' and Waveform_Start_Date_Time between
  STRING2TRUE('1980/01/01 00:00:00')-DELTA and STRING2TRUE('1981/01/01
  00:00:00') and Waveform_End_Date_Time between
  STRING2TRUE('1980/01/01 00:00:00') and STRING2TRUE('1981/01/01
  00:00:00')+DELTA and w.Station_Code like 'S%' and Channel_Code =
  'RB1' order by Network_Code, w.Station_Code;
  ```

- ```
  select DISTINCT c.Channel_Sensor_Type, c.Network_Code,
  c.Station_Code, c.Channel_Code, c.Location_Code, TRUE2STRING
  (Channel_Start_Date_Time), TRUE2STRING (c.Channel_End_Date_Time)
  from GDV_Channel c where Network_Code = 'BK' and
  c.Channel_Start_Date_Time between STRING2TRUE('1980/01/01 00:00:00')
  and STRING2TRUE('1990/01/01 00:00:00') and c.Channel_Sensor_Type
  like '%STS%' and Station_Code like 'S%';
  ```
  ```

III.-Implementation:

1.- Overview:

The following figure outlines the different processing stages of a SeismiQuery request.



1. An HTML file *query_f.html* contains the definitions of two frames; the 1st frame contains the HTML form and the 2nd one contains help for the user.
2. Once the user submits the form, a javascript function is called. Depending on the button the user clicked, this function can be *viewResults* or *emailResults*. These functions set up some variables and then call a function *validateAndSubmit*. This latest function converts the HTML form to a MSQL query and passes it to the Perl script *ProcessMSQL.pl*.
3. *ProcessMSQL.pl* calls the MSQL parser *MSQL2SQL*. This program parses the query and stores it in a parsing tree. A C function then walks through that tree and updates it with Data Center Specific actions. The output of *MSQL2SQL* is a SQL query on the Data Center own database schema.
4. Based on the output type variable defined by the javascript function, *ProcessMSQL.pl* calls the program *TableQuery* if the output type is **Table** or *TableQuery* and *ResponseQuery* if the output type is **Response**. *TableQuery* executes an SQL request on the Data Center database and outputs the resulting rows. *ResponseQuery* takes as arguments a network code, station code, channel code, location code, time span and writes out the instrument responses in the rdseed format. Please note that **Table** and **Response** are the only two output types currently used in SeismiQuery. However, new types could be added later on.
5. *ProcessMSQL.pl* then formats the results and displays them to the user.

2.- MSQL2SQL Parser:

The *MSQL2SQL* parser was developed with the unix tools lex and yacc. It parses any MSQL queries and stores it in a parsing tree. At the end of the parsing, *MSQL2SQL* calls a C function *UpdateTree*, which is part of the library *Tree.a*. This function walks through the tree and updates its nodes with Data Center specific information. The tree update functions are the only functions that a Data Center has to modify to be able to run SeismiQuery. Finally, *MSQL2SQL* calls the function *BuildSQL* that rebuilds the SQL statement from the parsing tree and writes it on the standard output.

3.- Parsing Tree:

The library *Tree.a* contains all of the functions needed to be able to use the parsing tree. Each node of the tree has the following structure:

```
typedef struct SNode Node;

struct       SNode {
              int  Type;                  /* Node’s Type */
              int  Subtype;               /* Node’s Subtype */
              int  Datatype;              /* Not Currently Used */
              int  Fixed;                 /* Not Currently Used */
              char Value[MAXVALUE];       /* Node’s Value */
              int  NbChild;               /* Number Of Children */
              Node *Children[MAXCHILD];   /* Children */
             };
```

- The main types of nodes are:

- **Statement**: Type = 1
  Number of Children = 2
  Children[0] = Select
  Children[1] = Select

- **Select**: Type = 2
  Number of Children = 6
  Children[0] = Attribute
  Children[1] = ExpressionList
  Children[2] = TableList
  Children[3] = [Condition]
  Children[4] = [Group]
  Children[5] = [ExpressionAttributeList]

- **Table**: Type = 3
  Number of Children = 1 | 2
  Children[0] = Table Name
  Children[1] = [Alias Name]

- **Expression**: Type = 4
  Number of Children = 1 | 2
  Children[0] = Expression
  Children[1] = [Expression | Alias Name]

- **Table.Column**: Type = 5
  Number of Children = 2
  Children[0] = Table Name
  Children[1] = Column Name

- **Group**: Type = 6
  Number of Children = 1 | 2
  Children[0] = ExpressionList
  Children[1] = [Condition]

- **Function**: Type = 7
  Number of Children = 2
  Children[0] = Attribute
  Children[1] = ExpressionList

- **Condition**: Type = 8
  Number of Children = 1 | 2 | 3
  Children[0] = Expression | ExpressionList | Select | Name | Table.Column | Condition
  Children[1] = [Expression | Select | ExpressionList | Text | Condition]
  Children[2] = [Expression]

- **List**: Type = 9
  Number of Children = 1 | .... | N
  Children[0] = Table[Alias] | Expression | ExpressionAttribute
  ....
  Children[N-1] = Table[Alias] | Expression | ExpressionAttribute

- **ExpressionAttribute**: Type = 10
  Number of Children = 2
  Children[0] = Expression
  Children[1] = Attribute

- **Terminal Node**: Type = 200
  Number of Children = 0

- The following table describes with more details the parsing tree node parameters:

Type	Subtype	Value	Number of Children	Children (Comments if no Child)
1	1	"UNION"	2	Select, Select
1	2	"UNION_ALL"	2	Select, Select
1	3	"INTERSECT"	2	Select, Select
1	4	"MINUS"	2	Select, Select
2	1	"SELECT"	6	Attribute, ExpressionList, TableList, [Condition], [Group], [ExprAttrList]
3	1	"TABLE"	1	Table
3	2	"TABLE_ALIAS"	2	Table, Alias
4	2	"EXPR_ALIAS"	2	Expression, Alias
4	3	"PARENTHESIS"	1	Expression
4	4	"+" | "-" | "PRIOR"	1	Expression
4	5	"*" | "/" | "-" | "+"	2	Expression, Expression
5	1	"TABLE.COLUMN"	2	Table, Column
6	1	"GROUP BY"	2	ExpressionList, Condition
7	1	VALUE (Function Name)	2	Attribute, ExpressionList
8	1	VALUE (OPERATOR1)	2	Expression, Expression
8	2	VALUE (OPERATOR2)	2	Expression, Expression
8	3	VALUE (OPERATOR1)	2	Expression, Select
8	4	VALUE (OPERATOR2)	2	Expression, Select
8	5	VALUE (OPERATOR2)	2	ExpressionList, Select
8	6	"IN"	2	ExpressionList, ExpressionList
8	7	"NOT IN"	2	ExpressionList, ExpressionList
8	8	"IN"	2	ExpressionList, Select
8	9	"NOT IN"	2	ExpressionList, Select
8	10	"BETWEEN"	3	Expression, Expression, Expression

8	11	"NOT BETWEEN"	3	Expression, Expression, Expression
8	12	"IS NULL"	1	Expression
8	13	"IS NOT NULL"	1	Expression
8	14	"EXISTS"	1	Select
8	15	"NOT EXISTS"	1	Select
8	16	"LIKE"	2	NAME, TEXT
8	17	"NOT LIKE"	2	NAME, TEXT
8	18	"LIKE"	2	TABLE.COLUMN, TEXT
8	19	"NOT LIKE"	2	TABLE.COLUMN, TEXT
8	20	"NOT"	1	Condition
8	21	"AND"	2	Condition, Condition
8	22	"OR"	2	Condition, Condition
8	23	"PARENTHESIS"	1	Condition
9	1	"LIST_TABLE"	N	Table[Alias], ..., Table[Alias]
9	2	"LIST_EXPR"	N	Expression, ..., Expression
9	3	"LIST_EXPR_ATTR"	N	ExprAttr, ..., ExprAttr
10	0	"EXPR_ATTR"	2	Expression, Order Attribute
200	0	""	0	Select | Function Attribute
200	1	"ALL"	0	Select | Function Attribute
200	2	"DISTINCT"	0	Select | Function Attribute
200	3	"*"	0	Column *
200	4	VALUE	0	Table NAME
200	5	VALUE	0	Alias NAME
200	6	VALUE	0	TEXT
200	7	VALUE	0	Column NAME
200	8	VALUE	0	Column NAME (Table.Column)
200	9	VALUE	0	NUMBER
200	10	VALUE	0	DATE
200	11	"NULL"	0	"NULL"
200	12	"COUNT(*)"	0	Function COUNT(*)
200	13	"ASC"	0	Order Attribute
200	14	"DESC"	0	Order Attribute

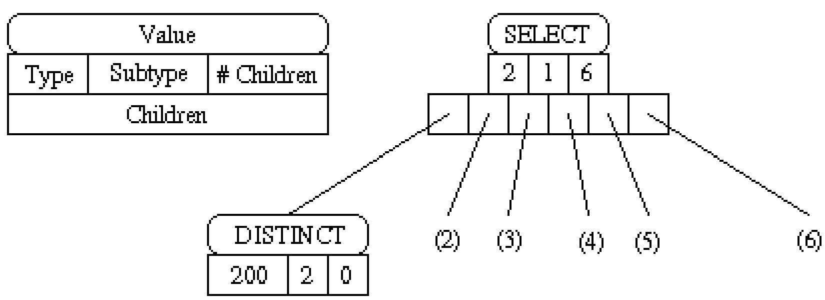
4.- Example:

This section gives an example of a parsing tree for a MSQL request.

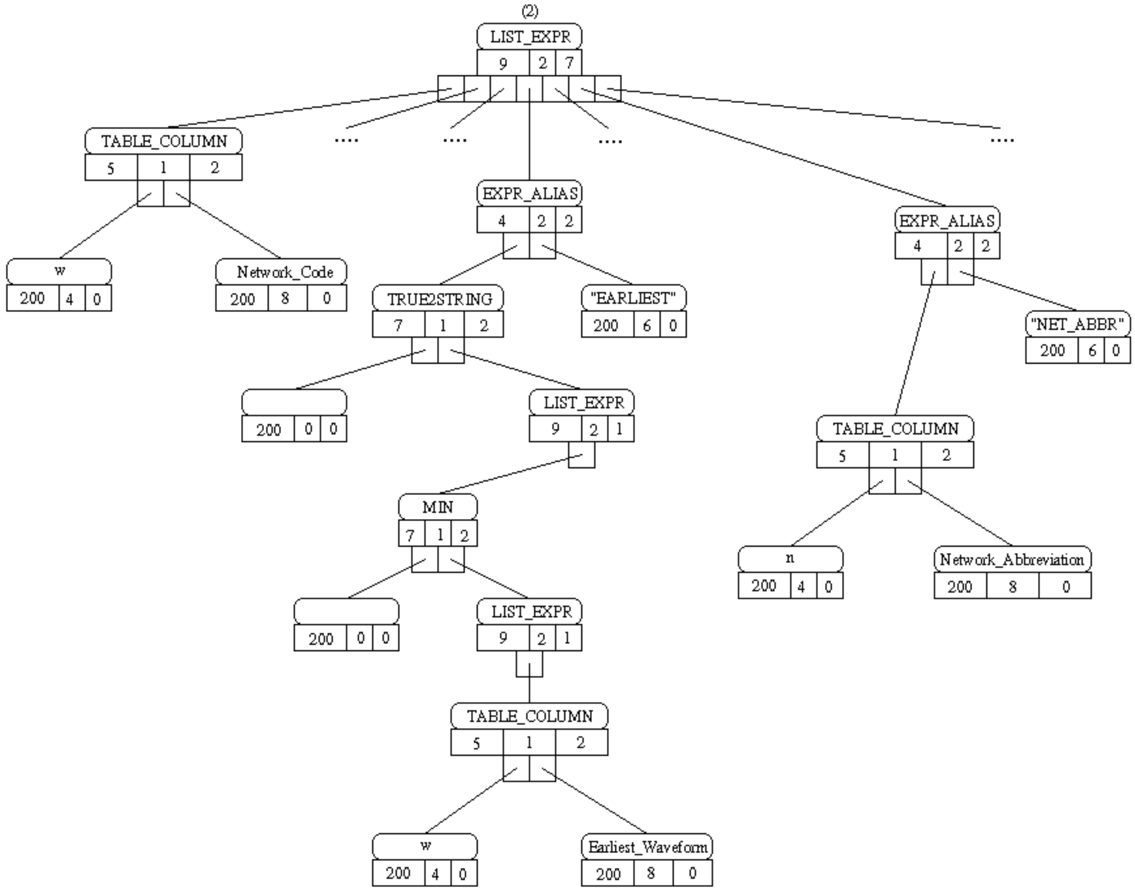
The MSQL query is:

```
SELECT DISTINCT w.Network_Code, w.Station_Code, w.Channel_Code,
      TRUE2STRING (min (w.Earliest_Waveform)) "EARLIEST",
      TRUE2STRING (max (w.Latest_Waveform)) "LATEST",
      n.Network_Abbreviation "NET_ABBR", n.Network_Full_Name
FROM  GDV_Channel_Summary w, GDV_Network n
WHERE w.Network_Code = n.Network_Code AND w.Network_Code LIKE 'BK'
      AND w.Station_Code LIKE 'ARC'
GROUP BY    w.Network_Code, w.Station_Code, w.Channel_Code,
            n.Network_Full_Name
ORDER BY    w.Network_Code, w.Station_Code, w.Channel_Code, "EARLIEST";
```

And the corresponding parsing tree is:

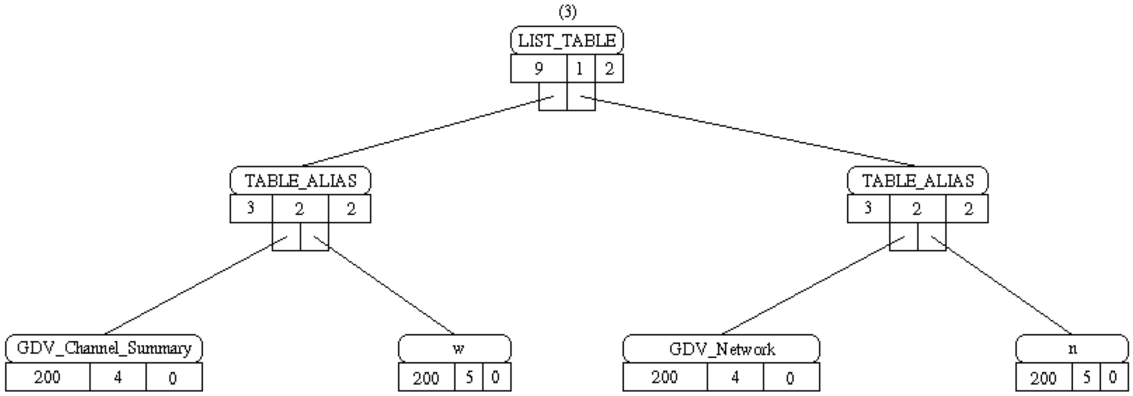


(2)



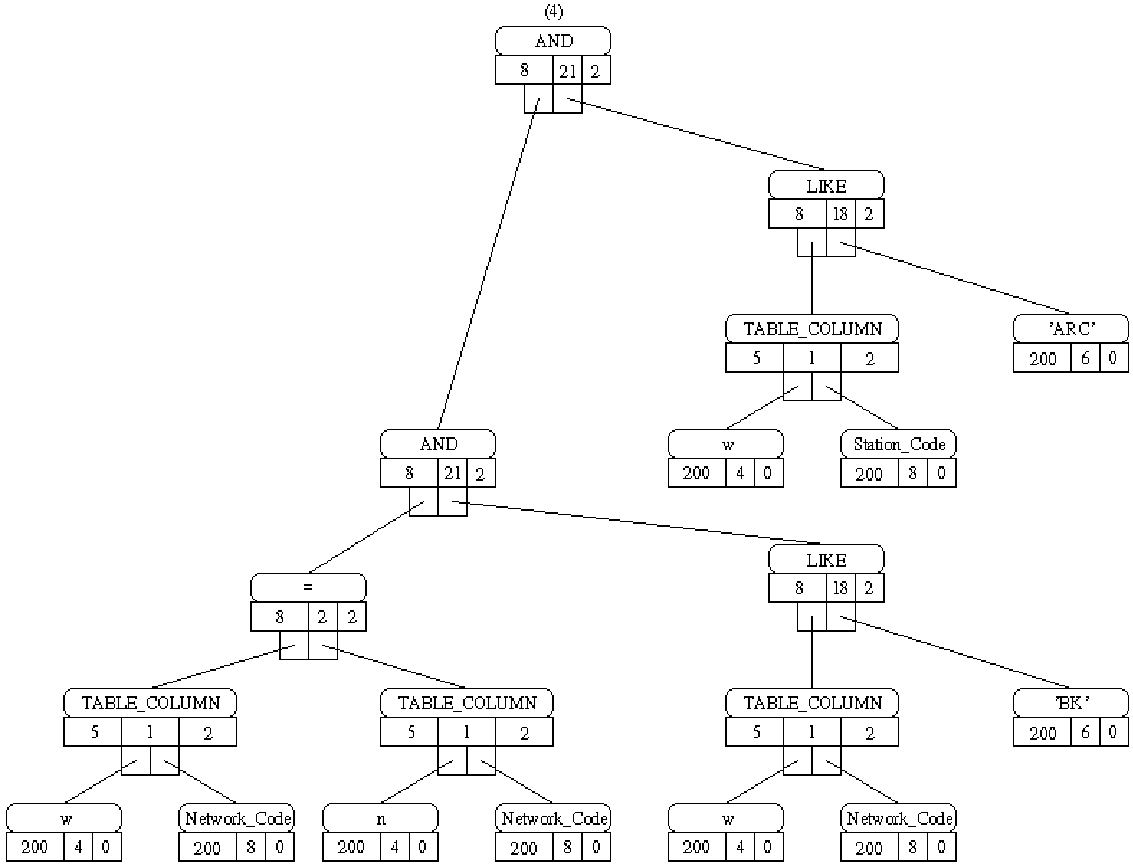
LIST_EXPR
9 2 7
....
....
....
....
TABLE_COLUMN
5 1 2
w
200 4 0
Network_Code
200 8 0
EXPR_ALIAS
4 2 2
TRUE2STRING
7 1 2
"EARLIEST"
200 6 0
200 0 0
LIST_EXPR
9 2 1
MIN
7 1 2
200 0 0
LIST_EXPR
9 2 1
TABLE_COLUMN
5 1 2
w
200 4 0
Earliest_Waveform
200 8 0
EXPR_ALIAS
4 2 2
"NET_ABBR"
200 6 0
TABLE_COLUMN
5 1 2
n
200 4 0
Network_Abbreviation
200 8 0

(3)

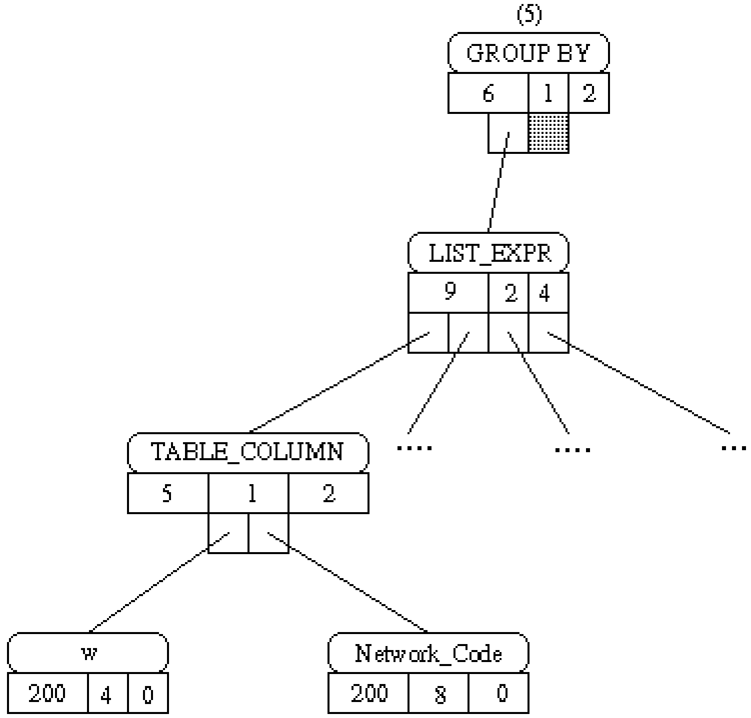


LIST_TABLE
9 1 2
TABLE_ALIAS
3 2 2
TABLE_ALIAS
3 2 2
GDV_Channel_Summary
200 4 0
w
200 5 0
GDV_Network
200 4 0
n
200 5 0

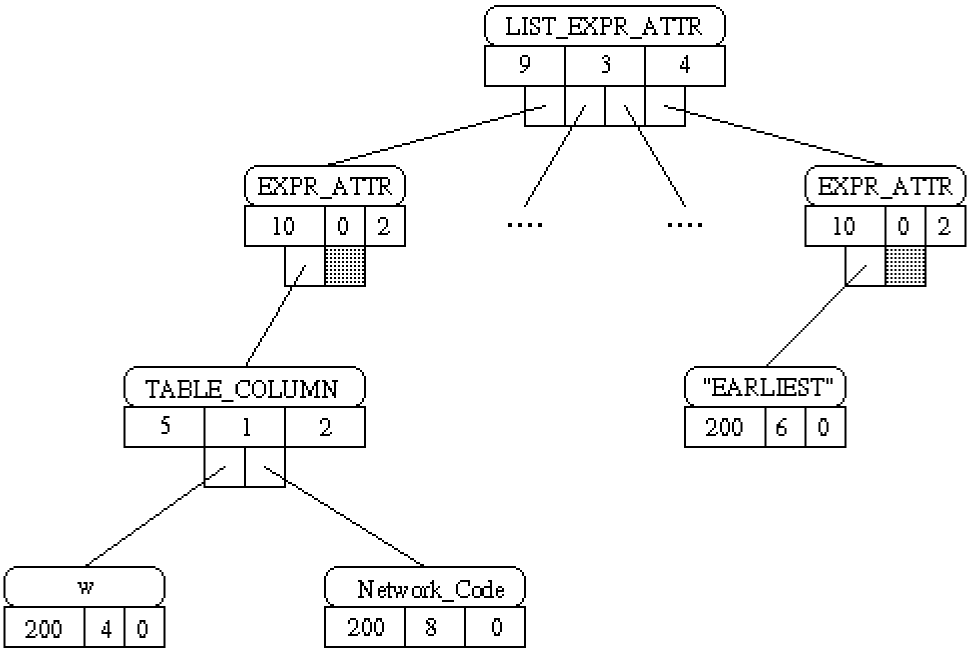
(4)


AND
8 21 2
LIKE
8 18 2
TABLE_COLUMN
5 1 2
'ARC'
200 6 0
w
200 4 0
Station_Code
200 8 0
AND
8 21 2
=
8 2 2
LIKE
8 18 2
TABLE_COLUMN
5 1 2
TABLE_COLUMN
5 1 2
TABLE_COLUMN
5 1 2
'BK'
200 6 0
w
200 4 0
Network_Code
200 8 0
n
200 4 0
Network_Code
200 8 0
w
200 4 0
Network_Code
200 8 0

(5)


GROUP BY
6 1 2
LIST_EXPR
9 2 4
TABLE_COLUMN
5 1 2
....
....
....
w
200 4 0
Network_Code
200 8 0

(6)


LIST_EXPR_ATTR
9
3
4
EXPR_ATTR
10
0
2
....
....
EXPR_ATTR
10
0
2
TABLE_COLUMN
5
1
2
"EARLIEST"
200
6
0
w
200
4
0
Network_Code
200
8
0

IV.- Remarks:

1.- Delta Value:

Within SeismiQuery and MSQL, a special keyword named **DELTA** has been introduced. This keyword is used to indicate a number of seconds and is replaced by a numeric value during the parsing stage of the MSQL query by *MSQL2SQL*. The numeric value is defined in the header file *Tree.h* by the field **DELTAVALUE**. An example of application for **DELTA** is when a user requests waveforms between times t_s and t_e. We would usually issue the following condition:

(Waveform_Start_Date_Time < t_e AND Waveform_End_Date_Time > t_s)

However, this condition means that the GDV_Waveform view needs a full scan. Since we know that waveforms in our database don't span more than **DELTA** seconds, we can optimize and rewrite the previous condition as:

(Waveform_Start_Date_Time BETWEEN t_s – **DELTA** AND t_e) AND
(Waveform_End_Date_Time BETWEEN t_s AND t_e + **DELTA**)

Depending on the value of **DELTA**, it may improve dramatically the performance of the query.

2.- Time Format Issue:

One of our primary concepts for SeismiQuery consisted of having dates represented as Oracle date datatypes. Further along the project, we started to encounter the limitations of this choice.

The Oracle Date datatype can only represent time down to the seconds and doesn't take account of the leap seconds. Some problems arose also when trying to parse a MSQL statement and convert fields back and forth between oracle date and true epoch time datatypes. Our waveforms times are currently stored in true epoch times in our database and converting them to strings on the fly was restricting us from using their indexes and therefore giving us very poor performance.

We came to the conclusion that having all dates within the Generic Data Views defined as true epoch time would be a more viable option. The conversion between a true epoch date and a string would then be achieved at an upper layer within SeismiQuery. However, the end user would only manipulate string dates, with the possibility of specifying a better granularity as well as leap seconds.

As part of MSQL, we would assume that there are functions True2String and String2True that would convert between True Epoch time and Oracle Date (string) formats.

3.- Calendar Queries:

Within the SeismiQuery model, the calendar queries (by station and by network) are handled differently than the other queries. A cron job runs daily and builds the calendars. When a user clicks on a day to retrieve data, the perl script *ProcessMSQL.pl* is called via a GET method (instead of a POST method) and its arguments are the MSQL query generated by a javascript function as well as variables defined by the HTML calendar itself (instead of the javascript function).