An interactive report generator language for a relational data base system allowing user defined aggregation operators

Tore Risch
4 March 1982

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computer Science University of Uppsala
750 02 Uppsala, Sweden

ABSTRACT

A query language handler for a relational data base system has been extended with a language for specifying the layout of the output from the queries, a report generation language. The features of the report generation language include handling of groups of attribute values in the result of a query and a facility for defining user defined aggregation operators; all built in aggregation operators are summation and average. They are defined using the facility. It is shown how the report generation language extends the usefulness of the query language. The implementation or the system is transportable. Queries can be run both interactively for testing report layouts and logic of queries and in a compiled mode where production programs in a conventional programming language (COBOL) are generated.

Key Words and Phrases: Query Language, Application Generator, Report Generator, Relational Database, Database Application Language, User Interface Language.

The research reported herein was supported by the *Swedish Board for Technical* Development (STU).

1. INTRODUCTION

Most data base systems include a query language handler by which the user can interactively manipulate the data base. Many systems, e.g. System R <Astrahan> and Ingres <Stonebraker> allow the embedding of query language statements in a conventional programming language. This is important when using the query language for practical applications since the query language is only used for specifying what data to transport to and from the data base. In practical applications one wants to be able to specify other things an well, e.g. detailed output layouts as well an integrations with other programs.

One often demanded type of operation on the result of a data base query is the possibility to specify the output format in detail. For this reason a report generator is an important extension to a query language handler.

In this paper a relational database system is presented, *which* in integrated with a language for specifying reports. This report generation language is useful not only for specifying report layouts; it will be shown how it extends the query language by introducing, e.g., operations over groups of attribute values.

A special feature of the report generation language is primitives for user defined aggregation operators. All built in aggregation operators, such as summation and the average are defined using these primitives.

The query and report generation language can be used either interactively or compiled. In the latter case a program generator is used for generating a program from a given set of query language statements.

Section 2 in this paper will describe briefly the relational query language on top of which the report generation language in built. The query language is influenced by QUEL <Stonebraker>.

Section 3 describes the report generator language and how it extends the query language. This paper concentrates on data base retrieval, even though the

system can also handle data base updates.

Finally, section 4 outlines the implementation. The design of the query and report generation language allows programs in a conventional programming_language (e.g. COBOL, FORTRAN or PASCAL) to be generated. Presently COBOL programs are generated. The design of the program generator is influenced by the program generator of the LIDAM system <Risch80a,Risch80b>.

The entire system is implemented in standard Fortran with a few machine dependent subroutines coded in assembly language. This makes the system machine independent to a great extent.

The system is called Midam (Machine Independent DAta Manager).

2. THE OUERY LANGUAGE

A typical retrieval query has the format

```
ALIAS DEPT(F);     /* F is a shorthand for DEPT */
GET F.MGR WHERE F.NAME EQ "TOYS";
                   /* Get the manager(s) of the TOTS dept. */
```

The ALIAS statement way be omitted if the attributes of the GET statement are qualified with their relation names (DEPT).

A data base retrieval always returns a sorted set of tuples, ordered ascending with duplicate tuples removed.

The first key word (GET) in the statement tells what to do with the result. GET prints it in a standard format. In order to store the result of the query in a relation, MANAGERS, and just print the number of tuples in it, one writes:

```
COUNT MANAGERS(DEPT.MGR) WHERE ZEPT.NAME EQ "TOYS";
```

The query language also contains statements for updates of the data base, and for data dictionary handling. The data dictionary is fully described by some relations which are also manipulatable by the query language.

In order to delete some records from a file one way write, e.g.

```
DELETE DEPT WHERE DEPT.MOR EQ “TOYS”;
```

To add a tuple to a relation one writes# e.g.

```
INSERT DEPT (MGR=”JONES”, DEPT=”TOYS”);
```

The system allows the user to define procedures of query language statements where compare values can be read from the terminal.

The examples above should give some ideas of the main features of the query language. For a complete description of it, see <Carlsson>. The report generation language, which is described in the next section, makes it possible to further extend the basic set of commands.

3. THE REPORT GENERATION LANGUAGE

The report generation language extends the basic set of retrieval commands. In this way the output from a query can be controlled in detail. The reports use the fact that the result tuples are sorted. The reports are defined separately; one report definition can be used for several types of queries.

A new retrieval command, <name>, is defined with the command

```
  REPORT <name> <arglist> <definition>;
```

For example

```
  REPORT FOO(X,Y)=("X=";X;20;"Y=";Y);
```

defines a new report with name FOO, (<arglist> and <efinition> will be described later.)

The report FOO may be used, e.g., like this:

```
  FOO F.NO,F.THING WHERE F.NO GT 123;
```

That is, the word GET in a database retrieval statement may be replaced with the name of any user defined report. GET itself is defined by the system as a report. An example of the output from the above call to FOO with the above definition is

```
   X=315 Y=APE
   X=324 Y=CAR
```

In the example above a sorted set of tuples with attributes F.NO and F.THING is returned. The report definition specifies one output line for each tuple in the set. When the report is called, its formal parameters are successively bound to the values of the attributes of each tuple of the set. The formal parameters allow the definition of reports independent of the queries in which the reports are to be used.

The <definition> consists of a number of report notes, separated with semicolons.

The references to formal parameters or strings in FOO specify that their values shall be printed at the current position of the output line. The numbers indicate tabulation to the specified positions.

The types of notes supported are in summary:

1. Simple notes, the only type of notes used in the report FOO above, specify how to print one or several lines of output. Arithmetic expressions and function calls may be used in simple notes.

2. Special notes specify that some report notes are to be executed at some special position of the report. Headings and groupings are specified by using such special report notes.

3. Conditional notes specify that some report notes are to be executed only for those tuples where some specified predicate of variable values is true.

4. Assignment notes can be used for introducing local variables in the report definitions and for assigning them.

5. Several report notes may be syntactically grouped together into one report note by a report note list, where they are surrounded by parentheses and separated with semicolons (;), as in the definition of the report FOO above.

The next subsection describes more carefully the various types of report notes.

3.1 Simple notes

A simple note specifies that the value of a string, a variable, or an expression shall be written at the current position of the output line. By using simple report notes the output line is built from left to right. A slash (/) generates a linefeed, and the symbol EJECT generates a form feed.

A simple note may also be an arithmetic expression of formal parameters, **e.g.**

```
(...;X+52*(Y-3);...)
```

The value of the expression will be printed. In this case the system will check that all variables involved In the expressions are numbers.

It in allowed to refer to report functions in simple report notes. The system has a number of built in functions and this provides a mechanism for user defined functions. The use and definition of report functions will be described more carefully later. Functions returning numeric values can be called in arithmetic expressions.

3.2 Special notes

Special notes specify that some notes shall be executed at some specific position of the report, e.g. as headers. The notes which are not specified by a special note are hereafter called main notes. They are executed for each tuple. In this section the most important special notes available are described.

```
INIT <reportnote>;
```

INIT specifies report notes to be printed initially in the report. For example,

```
INIT ("This is a header";/;"This is the second line");
```

will print these two lines as the report header:

```
This is a header
This is the second line
```

FINISH <reportnote>; FINISH specifies notes to be executed at the very end of the report.

```
TOP    <reportnote>;
BOTTOM <reportnote>;
```

These two special notes specify notes to be executed at the top and at the Bottom, respectively, of each new page of the report.

```
CHANGE <attr>: <reportnote>
```

The report note(s) specified by CHANGE shall he executed only when the value

of the attribute, <attr>, is different from the value of that attribute in the previous tuple. A common type of layout specification is grouping of attribute values. For example, one way vent to make a report of a number of articles, their components, and their subcomponents having the layout and definition shown in figure 3-1. The group of components is a subgroup to the groups of articles.

Layout:

```
ARTICLE COMPONENT       SUBCOMPONENT
-----------------------------------
A1      C1              C3
                        C4
        C2              C5
                        C6
        C3              C6
                        C7
A2      C4              C5
        C5              C8
```

Definition:

```
REPORT ARTCOMP(ART,COMP,SUBCOMP)
       INIT(HEADER(ART);16;HEADER(COMP);28;HEADER(SUBCOMP);
            /;"-----------------------------------")
       CHANGE ART: ART;
       CHANGE COMP: (16;COMP);
       28;SUBCOMP);
```

Fig. 3-1: A report layout with a group and a subgroup and its definition

Several levels of groups and subgroups can be introduced by using CHANGE for more than one formal parameter. COMP is a subgroup to ART because COMP is before ART in the formal parameter list, and thus the result tuples are primarily sorted by ART and secondarily by COMP. The built-in function HEADER returns the name of the attribute to which a formal parameter in bound.

NOCHANGE <attr>: <reportnote>;

The specified report note(s) are executed only when the value of the attribute, <attr>, has not changed compared to the previous tuple.

SUMMARY <attr>: <reportnote>;

The specified report note(s) are executed only if the value of the attribute <attr> will change in the next tuple, i.e. at the end of a group. A typical use of SUMMARY is to print subtotals, by using the built in function SUM(<attr>), which returns the accumulated sum or the attribute <attr> within the current group. SUM is context sensitive so that it returns different accumulated sums depending on whether it is called from within a group or not. If SUM is not called in a CHANGE, NOCHANGE, or SUMMARY special note, the accumulated sum from the beginning of the report in returned. For example, figure 3-2 shows the layout and the definition of a report of ARTICLES, their components, and the quantities of each component.

Layout:

```
ARTICLE   COMPONENT QUANTITY
---------------------------
A1        C1        11
          C2        20
          C3         8
                    --------
                    39
A2        C4        31
          C5         3
                    --------
                    34
---------------------------
       GRAND TOTAL: 73
```

Definition:

```
REPORT ARTCOMP(ART,COMP,QUANT)=(

       INIT("ARTICLE";11;"COMPONENT";21;"QUANTITY";/;
            "--------------------------- ");
       CHANGE ART: ART;
       11;COMP:21;QUANT;
       SUMMARY ART: (21;"--------";/;
                     21;SUM(QUANT));
      FINISH ("---------------------------";/;
              "      GRAND TOTAL ";SUM(QUANT)));
```

Fig. 3-2: The layout and definition or a report with a group and summaries.

Notice the different types of accumulated sums returned. The built-in function SUM is an example of an aggregation function, which will be further discussed later.

3.3 Assignment notes

Local variables may be introduced and updated with the assignment note:

```
    <var>:=<expression>;
e.g. X:=X*32;
```

The assignment note does not generate any output. It <var> is not used as the left hand side of some earlier assignment note, it will be declared as a new local variable with the same data type as the data type or the expression on the right hand side.

For example, figure 3-3 shows a report which counts the number of tuples in a search.

```
REPORT COUNT()=(INIT(C:=0); /*Introduce the counter C */
                C:=C+1;     /* Increment C for each tuple*/
                FINISH(C;”TUPLES FOUND”));
```

Fig. 3-3: A report with local variables.

Figure 3-4 shows how to introduce page numbering.

```
REPORT ...(...)=(INIT(PG:=0;EJECT);
                 TOP (PG:*PG+1;60;"PAGE”;PG);
                 ...);
```

Fig. 3-4: Page numbering

3.4 Conditional notes.

```
IF <predicate> THEN <reportnote> ELSE <reportnote>;

e.g.
IF X GT 5 THEN X ELSE X-5;
```

The THEN note is executed if the predicate is true, otherwise the ELSE note. The same types or comparisons are allowed in the predicate as in the selection rule of the query language.

For example, figure 3-5 shows a report to list the sum of the salaries for each sex in some departments. The tuples are sorted by department, and the attribute SEX is equal to zero for males.

```
REPORT SEXSAL(DEPT,SEX,1NCOME)=
       (CHANGE DEPT: (MALE:=0; FEMALE:=0);
        IF SEX EQ 0 THEN MALE:=MALE+1;
                    ELSE FEMALE:=FEMALE+1;
        SUMMARY DEPT: ("SALARIES FOR DEPARTMENT ";DEPT;/;
                       "MEN: ";MALE;" WOMEN: ";FEMALE));
```

Fig. 3-5: Making summaries under conditions.

figure 3-6 shows a report which prints the maximum value or some attribute.

```
REPORT PRMAX(X)=
       (INIt(M:=X); /*Initialize to the first attribute
                      value In the set*/
        IF M LT X THEN M:=X;
        FINISH("The maximum of ";HEADER(X);" is ";M));
```

Fig. 3-6: The maximum of the values of one attribute.

Since the data type of M Is Inherited from the data type of X and since the predicate LT is defined for both strings and numbers, the report definition is

independent of the data types of the attributes on which it is applied.

3.5 Report functions.

The report generator language provides a function definition facility.

A new function is defined with the command:

```
REPORT <function> <arglist>=VALUE(<expression>);
```

for example

```
REPORT PLUS(X,Y)=VALUE(X+Y);
```

When the function is called the value of the expression is returned. Note that there are no declarations in function definitions, the functions are generic.

The system provides a mechanism for defining aggregation functions, such as SUM. SUM is no normal function, but rather a coroutine. SUM is defined as in figure 3-7.

```
REPORT SUM(X)=(INIT(S:=0);
               S:=S+X;  /* Executed for each tuple */
               VALUE(S));
```

Fig. 3-7: The definition of the aggregation function SUM.

The local variable S contains the accumulated sum. It In initialized to zero in the special note INIT. At any point the accumulated sum is returned through the variable 5. As earlier pointed out, different accumulated sums are returned depending on where SUM In called. Therefore the INIT statement will be executed at different places depending on where SUM in called. If SUM is called outside a group, INIT will be executed initially in the report, and, when called within a group, it will be executed in the beginning of that group. Analogous, FINISH will he executed at the end of the report or group, respectively.

The report AVG in figure 3-8 returns the average of some attribute values up to the tuple where it is called.

```
REPORT AVG(X)=(VALUE(SUM(X)/CNT());
```

where

```
REPORT CNT()=(INIT(C:=0);
              C:=C+1;
              VALUE(C));
```

Fig. 3-8: An aggregation function for average.

The aggregation function MAX in figure 3-9 returns the maximum value of some attribute value.

```
REPORT MAX(X)=(INIT(M:=X);
               IF X GT M THEN M:=X;
               VALUE(M));
```

Fig. 3-9: An aggregation function for maximum.

Analogous to the report PRMAX above, MAX can be applied on both numbers and strings. The data type of M is inherited from X. The data type of a local variable may be inherited from some other variable, or, if it is bound to an attribute of some relation, from the data type of the attribute as stored in the data dictionary. This illustrates the generic nature of report functions.

An aggregation function can also be used as a normal report. The result of such a retrieval is the value of a call to the aggregation function executed at the bottom of the report. For example, SUM can be used not only as an aggregation function, but also as a retrieval statement to print the sum of some column in some relation, as in figure 3-10.

```
SUM PERS.SALARY WHERE PERS.DEPT EQ “TOYS”;
```

Fig. 3-10: An aggregation function used as a report.

3.6 Variable number of arguments

Some reports are applied to a variable number of actual parameters, e.g. GET. Therefore the report generation language provides a method to loop over the formal parameters. This is done with the special note

FOREACHVAR <reportnote>

The report note will be executed for each actual parameter, thereby building an output line. A system variable, *VAR, can he referenced in the report note. It is bound to each actual parameter in turn. See <Carlsson> for a complete description of FOREACHVAR.

3.7 Calling external programs

The system has a possibility to call subroutines from reports. In this way other programs can be connected to programs generated by Midam. In order to be able to call the subroutines when using Midam interactively, the names and the parameters of the external subroutines have to be specified when the Midam system is loaded.

4. IMPLEMENTATION

MIDAM is a module in the MIMER software system <Mimer>, which also comprises a relational data base handler, MIMER-DB, a query language, MIMER-QL, and a screen handler system, MIMER-SH. The entire system, including MIDAM, is implemented in standard Fortran with a few system dependent subroutines in assembler. MIDAM consists or two subsystems: The MINER-DB system, the data base handler or MIMER, to which an interpreter for the programming language Lisp (LISP F3 <Nordström>) is connected. The Lisp interpreter is extended with functions to access MIMER-DB from Lisp.

The query language handler In MIDAM Is written in Lisp. A parsed version of each system command is given to a code generator written in Lisp, which generates an evaluatable Lisp program equivalent to the system command, the Macro form. When Midam is run in interactive mode, the Lisp Interpreter directly interprets the Macro form.

A special command is available for generating a COBOL program from a system command. The Macro form is then compiled into a COBOL program. The Macro form contains only low level Lisp functions, such as assignments, conditional expressions, and calls to MIMER-DB. Such a structure of the Macro form is chosen# so that the COBOL program generator becomes simple, and so that it is simple to generate programs in any programming language. A FORTRAN program generator is under construction.

Not every system command is compilable into COBOL; some commands manipulating the data dictionary cannot be compiled. One cannot generate a program which first reads the name of a relation, and then makes queries on this relation, since the file names must be known at compile time. System R <Chamberlin> solves this problem by making it possible to call the compiler from complied code, which is currently disallowed in Midam. Another missing feature available in System R is automatic recompilation of data base queries when the data base is substantially changed.

The report generation language is constructed so that the target program does not need any type information associated with each variable, even though the reports are generic. Thus the data tvpes of all variables of the target program are known at compile time.

The data types of attribute values of the data base are stored in the data dictionary. Thus, at any call to a report the data types of the formal parameters will be found in the data dictionary.

The types or local variables are found by looking at the expression from which their types are inherited. There is also a possibility to declare local variables.

The reports and the report functions are generic, i.e. they can be applied on

arguments of any data type supported by the system. Report calls therefore are compiled in-line, since the data types of the variables in the target program must be known at generation time. The different special notes of the reports will generate code at different places of the target program. This may sometimes generate a lot of code. However, some optimizations are used. For example, if an aggregation function is called several times with the same parameter in the same group, the calls will share the code. In come cases it should he possible to let the program generator generate one version of a report function for each combination of the data types of its actual parameter (see <Gries>).

5. DISCUSSION

A language has been described which is used for extending a relational query language so that output layouts from data base queries can be specified. Another way to specify such layouts is to use a command driven report generator, as, e.g., ASTRID <Gray>. The language presented here should not be regarded as a conventional report generator, but rather as a practical extension to a relational query language. Instead of providing a general report generator, the language provides some basic constructs for printing reports. Examples of such constructs include the possibility to specify where on the line a certain expression shall be printed, the possibility to execute report generation statements at specific positions of the report, and the use of groups of attribute values.

Most report generators and query languages have some built-in aggregation operators; normally the accumulated sum, average, and a few other are provided. In the language presented here no aggregation operators are initially built into the system; instead the language provides constructs for defining new aggregation operators. This makes the number or aggregation operators open ended. RIGEL <Rowe> has a similar construct called generators.

The report generation language also contains some conventional programming

langua constructs, such as assignments, conditionals, and functions. These types of constructs are important for specifying some types of database searches.

It is Important to be able to use a system like this interactively. Not only is it important to have an interactive query language for manipulating the data base and for testing logic of queries; it is also important to test the layout interactively. The language presented here can therefore be executed interactively during a testing phase. However, for production runs the interactive use of the language may be too expensive. The system therefore has a program generator which is able to generate data base manipulating programs In COBOL from statements in the language. Many production programs can be specified completely by this language, substantially decreasing the programming cost.

The layouts are specified independent of the queries on which they are to be applied. For this the reports are parameterized, and their formal parameters are bound to attribute values when the reports are applied to a data base search.

The data stored in the data base may have various data types. The reports can be defined independent of these data types. Therefore no declarations are needed of formal parameters of the reports; the reports are generic in a similar way as generic functions in, e.g., Ada <Ada>.

No data type cheeks are done in the target program; the types of all variables are known at generation time. The generic reports are therefore currently compiled in-line. The technique proposed by Gries and Gehani <Gries> to efficiently compile generic functions can sometimes be adopted in order to minimize the size (but not the speed) of the target program. Some other methods for code minimization have already been adopted, e.g., to share code pieces.

An important extension to the current language would be to allow for specification of the report layouts by using screen positions. This can be implemented as a front end to the current system. Work is currently under way of attaching an existing screen handler, MTMER-SH <Mimer>.

Another useful extension is to allow date base queries to be called from within reports. For example, a report printing personal data might need to access the data base for getting the address or each person. It should also be possible to create and extend data base relations from a report.

REFERENCES

<Ada> “The Reference Manual for the Ada Programming Language”, Superintendent of Documents, U.S. Government Printing Office, Washington D.C. 20402, Order No. L008-000-00354-8, 1980.

<Astrahan> M.M.Astrahan, M.W.Blasgen, D.D.Chamberlin, K.P.Eswaran, J.N.Gray, P.P.Griffiths, W.F.King, R.A.Lorie, P.R.McJones, J.W.Mehl, G.R.Putzolu, I.L.Traiger, B.Wade, and V.Watson: "System R: A Relational Approach to Database Systems", ACM Transactions on Database Systems, June 1976, pp. 97-137

<Carlsson> M.Carlsson, T.Risch: “Midam: A Data Language Handler for Pilot and Production Applications", Technical Report DLU 81/1, Uppsala Univ. Comp. Science Dept., UPMAIL, Box 2059, 750 02 Uppsala, Sweden, 1981.

<Chamberlin> D.D.Chairberlin, M.M.Astrahan, W.F.King, R.A.Lorie, J.W.Mehl, T.G.Price, M.Schkolnick, P.G.Selinger, D.R.Slutz, B.W.Wade, and R.A.Yost: "Support for Repetitive Transactions and Ad-hoc Queries In System R”, ACM Transactions on Database Systems, Vol. 6, No. 19 pp. 70-94, March 1981.

<Gray> P.M.D.Gray, R.Bell: “Use of Simulators to Help the Inexpert in Automatic Program Generation", EURO IFIP 79, P.A.Samet (editor), North-Holland Publishing Company, 1979.

<Gries> D.Gries, N.Gehani: "Some Ideas of Types in High Level Languages". CACM No. 69 1977.

<Mimer> MIMER Reference Manual, Uppsala University Data Center, Box 2103, S-750 02 Uppsala. Sweden.

<Nordström> M.Nordström: "LISP F3 User's Guide", Technical report DLU 79/19 Uppsala Univ. Comp. Science Dept., UPMAIL, Box 20599, 750 02 Uppsala, Sweden, 1979.

<Risch80a> T.Risch: "A Flexible and External Data Dictionary System for Program Generation", Proc. International Conf. on Data Bases, University of Aberdeen, Scotland, July 1980. Heyden, 1980.

<Risch80b> T.Risch: "Production Program Generation in a Flexible Data Dictionary System", Proc. Very Large Data Bases, Montreal, 1980.

<Rowe> L.A.Rowe, K.A.Shoens: "Data Abstraction and Updates in RIGEL", ACM-SIGMOD Conf., 1979.

<Stonebraker> M.Stonebraker, E.Wong, P.Kreps, G.Held: "The Design and Implementation of INGRESR", ACM Transactions on Database Systemas, september, 1976.