Title

Applications of a Natural-Style Database Query Language to Statistical Database Operations

Authors

Laubenheimer, William J
Rosenberg, Steven

Publication Date

1981-09-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

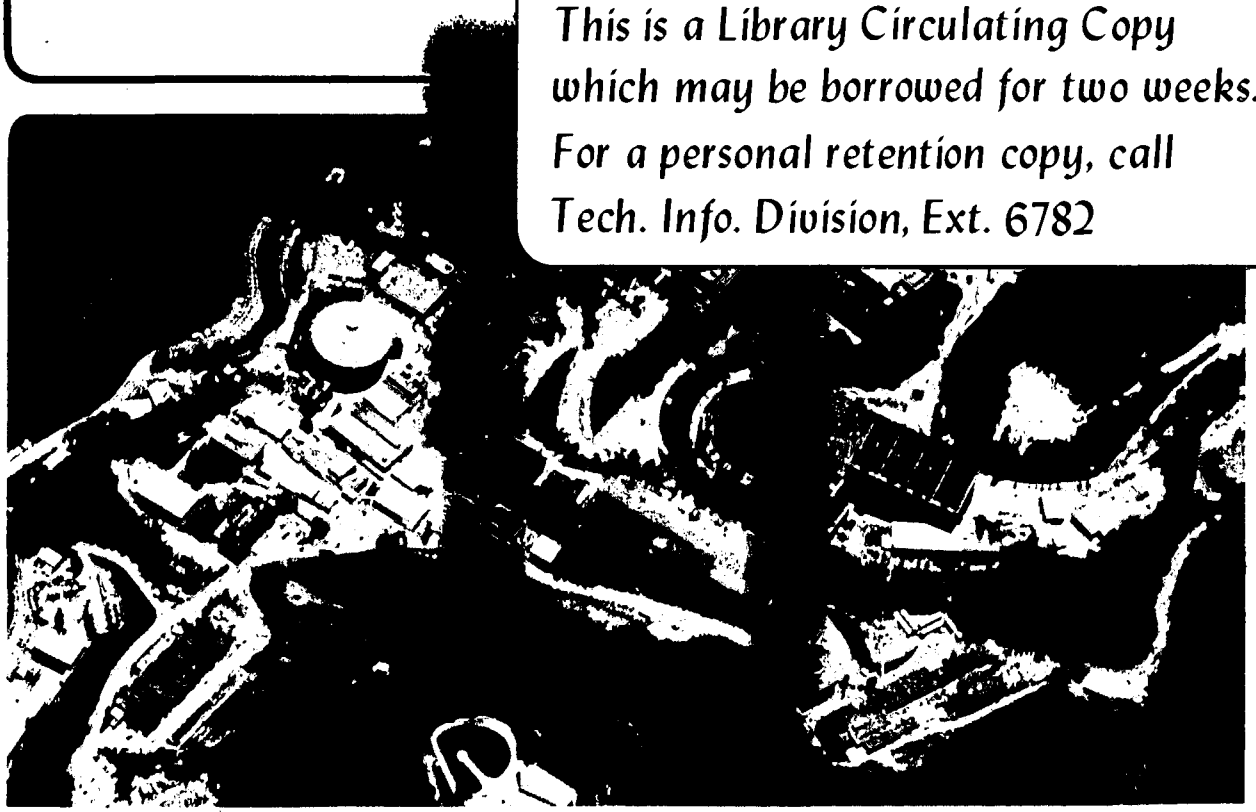
Physics, Computer Science & Mathematics Division

Submitted to Statistical Database Management Workshop, Menlo Park, CA, December 2-4, 1981

APPLICATIONS OF A NATURAL-STYLE DATABASE QUERY LANGUAGE TO STATISTICAL DATABASE OPERATIONS

William J. Laubenheimer and Steven Rosenberg

September 1981



Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48

DISCLAIMER

LBL-13348

Applications of a Natural-Style Database Query Language to Statistical Database Operations

William J. Laubenheimer
Steven Rosenberg

Computer Science and Mathematics Department
Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

ABSTRACT

FRL (Frame Representation Language) is a knowledge representation system suitable for use in database management systems. One drawback of FRL for such uses is its lack of a convenient mechanism for expressing queries, particularly for the naive user. A language which alleviates this difficulty by allowing queries to be expressed in a natural-sounding (although not actually natural) form is presented, and its uses and advantages in a statistical database environment are explored.

This work was supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under contract W-7405-ENG-48, and by the Defense Advanced Research Projects Agency.

1. Introduction

An important part of any database management system (DBMS) is the means with which it interacts with the user. This is particularly important if a substantial fraction of the user community is to consist of people who are not familiar with the intimate details of the DBMS, and indeed may not even be fluent in the implementation language of the DBMS (or any computer language). One approach to allowing such people to use the DBMS is to provide an interpreter which translates a natural (i. e. English-like) style of command into actual DBMS commands. Such a system, currently under development at the Lawrence Berkeley Laboratory of the University of California, is described below.

The particular implementation language on which the DBMS and query language are built is the FRL *(Frame Representation Language)* language [RG77a, RG77b], which itself is implemented in LISP. FRL was originally conceived as a knowledge representation system, but has many features which make it suitable for database implementation on a hierarchical model. In FRL, data objects are represented by *frames* [M75], which are nested lists of associations between a name, called an *indicator* in FRL, and a set of values. A frame is a collection of *slots*, which give the data structure of the frame. Slots are likewise collections of *facets*, which hold the data associated with the indicator of the slot, procedures to be executed when certain conditions regarding that slot are satisfied, and other items of relevance to the FRL system. Data are commonly either names of other frames, which allow the expression of relations, or outright values.

2. The FRL Interface

FRL provides procedures for creating and destroying frames, fields of frames, hierarchies of frames; for inserting and extracting data from

appropriate fields; for reading and printing frames; and for providing actions to be performed either when a certain operation is performed on a designated field or when a condition is satisfied by a particular field or combination of fields. A LISP programmer can use FRL by means of these procedures.

Another means of interaction with FRL is through a language called Framish, which allows certain frame manipulations to be performed by typing natural-sounding phrases as opposed to LISP code. Framish represents a far more pleasant method of interaction between a user unfamiliar with LISP and the FRL system, since many aspects of LISP (most notably quotation and parenthesis balancing) are to a large extent unnecessary in Framish.

Framish is based on the top-down operator precedence technique of Pratt [P73]. This technique creates parsers which can easily be extended in scope. An extension of Framish was made which gave it sufficient power to use FRL without having to understand LISP. This extension has resulted in a language which includes many features especially convenient for statistical database manipulations.

3. The Original Framish Language

As originally implemented, Framish supplied the following features:

- Arithmetic operations: sum, difference, product, quotient.
- Arithmetic relations: greater than, less than, equal to.
- Boolean operations: and, or, not.
- Membership operation: determine whether certain values are present in a given slot of a frame. Since hierarchichal information is maintained within the frame structure, this operation also determines hierarchical structure.

- Conditional evaluation: if-then, if-then-else.
- Sequencing and grouping of expressions.
- List building.
- The ability to call LISP or FRL functions using algebraic notation, or to revert temporarily to normal LISP syntax.
- Data operations: retrieval, insertion, deletion.
- Frame creation and deletion.

Once Framish has been loaded into an active FRL system, it may be used at any time. Framish commands may be mixed with LISP commands, and in fact, as indicated above, expressions in Framish may be intermingled with expressions in LISP.

Commands in Framish are expressed in a natural style, and it is not necessary to comprehend the structure of the frames to use Framish. For example, if, in a socio-economic setting, it was known that each state had a field (called a *slot* in FRL) called population which contained the population for that state, it would suffice to enter:

{ Get the population of California }

to retrieve the population of California. In order to find the total population of Federal Region IX, one can say:

{ the sum of the population of California, the population of Nevada, the population of Arizona, the population of Hawaii }

Turning to a business model, say, of a department store and its merchandise shipments, we might record a shipment of 500 television sets from a warehouse in San Francisco to a store in San Jose by the following sequence of steps:

(Create a new shipment)

(Its item is tv-sets)

(Its quantity is 500)

(Its warehouse is san-francisco)

(Its store is san-jose)

The FRL implementation of the DBMS would include instructions so that specifying the warehouse would subtract 500 from the number of television sets on hand in the San Francisco warehouse, and specifying the store would add 500 to the number of television sets on hand in the San Jose store.

It should be noted that although the style of statement used in Framish is natural, Framish is not, in fact, a truly natural language. In fact, the parsing technique used [P73] is neither geared towards nor particularly suited for natural-language parsing, and Framish statements may only deviate from the indicated form in such ways as have been anticipated by the language builder. Against this, though, must be weighed the immense overhead in space and time required by "true" natural-language database query languages and the general lack of ease with which such systems can be modified. It is generally quite easy to add additional Framish statement types to Framish. Although the intended purpose of Framish was to allow the LISP-naive user to avoid the more awful features of LISP, the extensions below turned it into a worthwhile tool for the experienced programmer as well.

The major defects of Framish before its extension center primarily on its incompleteness, the most notable aspects of which are:

- Lack of a convenient selection operator for identifying a subset of data satisfying a certain predicate;

- Lack of a convenient iteration mechanism;
- Lack of an aggregation operator over full or partial subsets of a given range;
- Restriction of frame manipulation operations to the facet (i.e. datum) level; it is impossible to alter the basic structure of a frame;
- Lack of access to the delayed-execution features of FRL, whereby a section of code is to be executed only at such time as a certain set of conditions has been met. Not only are these features a necessary part of knowledge-based reasoning systems which are often built on the frame databases of FRL, but they also allow the implementation of such common database operations as security and integrity constraints, triggers, and views. [SJR81]

4. The Query Extension of Framish

The first extensions to Framish were in the realm of query expression. An operation called "display" was added for those users who would rather type in

{ display the shipments of san-francisco }

in the earlier department-store example, than just

{ the shipments of san-francisco }.

To select certain items, any of the verbs "which", "where", or "with" were made available to introduce a predicate against which each item in the indicated slot could be tested. The modifiers "all" and "any" were also added in order to provide whichever quantification was desired. Thus, to find all shipments from the San Francisco warehouse to the Stockton store, any of the following statements would serve:

{ display all shipments of san-francisco where their store is stockton },

{ display all deliveries of stockton where their warehouse is san-francisco },

{ display all instance of shipment where its warehouse is san-francisco, its store is stockton } *

If only one such shipment is desired, the keyword "all" could be replaced by "any", or omitted. When applying the "where" construct, the data identified to the left of the "where" are assumed to be a set of frames. Predicates needing to access data in those frames may refer to the frame as "it", "them", or "they". Framish is capable of distinguishing certain possessive forms of these.

If a truth value is desired instead of the actual values satisfying the predicates, the verbs "have" or "has" may be used instead. For example, the sentence:

{ if any shipments of san-francisco have their item is tv-sets then display [tv sets have been shipped] else display [no tv sets have been shipped] }

will display a message indicating whether any television sets have been shipped from the San Francisco warehouse.

It is also possible to iterate a statement over a set of objects by enclosing the set to be iterated over in angle brackets "<", ">". The advantage of this notation over the standard means of expression used in algebraic query languages is that it is not necessary to introduce auxilliary

* This statement provides a clear example of the degree to which Framish fails to be natural. Since the FRL field which identifies hierarchical dependence is called "instance", it is not possible to say "display all instances of shipment..."

variables into the query expression. As an example, a query producing a list of locations to which refrigerators have been shipped from the Los Angeles warehouse is:

{ display the store of < all shipments of los-angeles where its item is refrigerator > }

which is expected to be much easier than an approximate algebraic equivalent of:

{ display all store of x where x is a shipment of los-angeles, the item of x is refrigerator }.

The iteration construct also serves as a set-former and aggregator. For example, to find the number of ladders shipped from all points to the Sacramento store, one need only write:

{ display the sum of < the quantity of < all deliveries of sacramento where its item is ladder >> }

The second set of angle brackets causes the summation to be an aggregation of the list formed by iterating the retrieval operation over the range specified by the inner set of angle brackets.

Another statement which increases the power of the angle bracket construct in some cases is the "with" statement, which sets up a local context of variables. Typing:

{ with var_1 as exp_1, ..., var_n as exp_n do $stmt_1$; ...; $stmt_m$ }

causes the variables var_i to be assigned the results of evaluating the expressions exp_i while the statements s_j are executed.

A principal use of the "with" statement in statistical operations is in cross-products. To obtain a listing of total deliveries by warehouse, store, and item, it suffices to write:

{ with the-warehouse as < the instance of warehouse >, the-store as < the instance of store >, the-item as < the instance of item > display the-warehouse, the-store, the-item, the sum of < all instance of delivery where its warehouse is the-warehouse, its store is the-store, its item is the-item > }

The result of this will be a listing of the form:

san-francisco oakland toaster 50
san-francisco oakland ladder 10
san-francisco oakland bed 20
san-francisco stockton bed 4
san-francisco fresno hammer 68
los-angeles fresno ladder 10
los-angeles bakersfield dishwasher 14

This statement causes all shipments to be searched for each combination of warehouse, store, and item. A modification of "where" to select from an arbitrary list, which would allow a more efficient (although less clear) version of the above to be written, is planned.

An important feature of FRL which was made available in the extension of Framish was the ability to execute groups of operations only when a certain condition is met. This feature (called a *sentinel* [R79]) is useful in maintaining consistency relationships, such as, in the department-store example, ensuring that the store to which a shipment from a warehouse is sent is allowed to receive products from that warehouse. It also has some uses in the statistical realm. For example, suppose that quotas have been set for toasters, such as:

{ the toaster-quota of oakland is 1000 }

{ the toaster-quota of san-jose is 800 }

{ the toaster-quota of stockton is 300 }

{ the toaster-quota of sacramento is 500 }

We can arrange to keep an up-to-date list of those stores which have sold their quotas of toasters by having a frame called "toaster" with a slot called "made-quota", and then creating the following sentinels:

{ whenever a shipment of ‹oakland, san-jose, stockton, sacramento› having (the item of the current value is toaster) is added replace the toaster-quota of the current frame with the difference of the toaster-quota of the current frame and the size of the current value }

{ with the-store as ‹oakland, san-jose, stockton, sacramento› when a toaster-quota of the-store having not (the current value is greater than 0) is found put the-store in the made-quota of toaster }

Now, the San Francisco area manager can determine which of his stores is up to quota in toasters simply by typing:

{ display the made-quota of toaster }

He can find out how many more toasters the below-quota stores must sell to meet their quotas with the query:

{ display the toaster-quota of ‹ all store of san-francisco where not (its toaster-quota is greater than 0) › }

The type of sentinel is selected by the keywords *when*, *whenever* which appear at the beginning of the sentinel, and the keywords *added*, *removed*, *found* which separate the condition from the actions to be performed when the condition is satisfied. A sentinel containing the keyword *when* is activated only once and then is destroyed; if the keyword *whenever* is used

instead, it is activated whenever the condition is satisfied. The second keyword determines what operation on the slot shall cause the condition of the sentinel to be tested, with *found* meaning that the values on the slot when the sentinel is initiated will be tested.

Two common constructs in Framish also appear in the statements above. The second sentinel illustrates a common use of the *with* statement: to name a value from an iteration list so that it can be used in more than one spot (in this case, both to identify the data structure being altered and again to name the data structure to add it to a list of stores which have fulfilled their quota). Also, an iterative expression can contain a list of items to be iterated over should this be more convenient than expressing it from the database.

Since the last example, which is typical of statements involving iterative expressions and sentinels, is rather unwieldy to type in all at once, a means of breaking the task up into smaller segments using the underlying FRL system is available. There is a frame called a *context*, having slots called *environment* and *action*, which can be used in this fashion. A sample sequence of Framish statements presenting the previous example shows this feature:

{ create a new context called toaster-quota-context }

(Creates the context.)

{ put ‹the-store as oakland, san-jose, sacramento, stockton› in its environment }

(Establishes the range over which the actions of the context will be iterated.)

{ put (the-update as: the item of the current value is toaster) in its environment }

(A sentence fragment, which in this case is a predicate.)

{ put (the-update-action as: replace the toaster-quota of the current frame with the difference of the toaster-quota of the current frame and the size of the current value) in its environment }

(Another sentence fragment: this time, an action.)

{ put (the-fulfillment as: the current value is less than 0) in its environment }

(Still another sentence fragment.)

{ an action is: whenever a shipment of the-store having the-update is added do the-update-action }

(Defines the first sentinel from the previous example.)

{ an action is: when a toaster-quota of the-store having the-fulfillment is found put the-store in the made-quota of toaster }

(Defines the second sentinel from the previous example.)

{ execute it }

The last Framish statement causes the context to be executed. The expressions in the environment are associated with the variables and then the actions are performed in sequence.

The true power of using the context as opposed to the previous means of setting up the sentinels becomes apparent when we take every occurrence of the word "toaster" and replace it with "object". A mechanism which will interrogate the manager about which item he wishes to have quota maintenance on can now be set up with the following statements:

{ create a new context called set-quota-context }

{ put (object as: request a response to !What do you wish to set quo-

tas on?!) in its environment)

(put (object-quota as: request a response to !What is the name of the quota?!) in its environment)

(its action is: execute the object-quota-context)

Typing:

(execute the set-quota-context)

will cause a dialogue between computer and user to take place as illustrated below, with the user responses to set up a quota system for ladders italicized:

What do you wish to set quotas on? *ladder*

What is the name of the quota? *ladder-quota*

Further additions to the object-quota-context could be made which would interrogate the user for the specific quotas to be established and place them in the appropriate spots.

5. Summary

In conclusion, it can be seen that the Framish language offers a means of expressing on a frame-based system many of the kinds of operations required in statistical database management. The syntax of the language is reminiscient of a natural language, and represents an effective intermediate stage between the computational power of the implementation language and the expressive flexibility of a natural-language approach, avoiding the esoteric nature of the first while not requiring the excessive time and space demands of the second. While this makes it attractive in the non-expert user environment typical of many statistical database systems, the features available make it a useful tool for the expert programmer and

DBMS designer as well.

The features of FRL and Framish suggest interesting possibilities for statistical database management system design. The natural style and relative power of Framish would allow a non-expert user to augment a "standard" DBMS in many useful ways. The capability of introducing rule-based reasoning could be used to create "intelligent" database systems which might blend database operations with artificial-intelligence reasoning. We hope to pursue these concepts in the near future.

6. Bibliography

[F79] Findler, Nicholas V. (ed.), *Associative Networks - The Representation and Use of Knowledge in Computers.* Academic Press, New York, 1979.

[M75] Minsky, Marvin, "A Framework for Representing Knowledge", in P. H. Winston (Ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975.

[P73] Pratt, Vaughan R., "Top-Down Operator Precedence". SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, 1973, 41-51

[RG77a] Roberts, R. Bruce and Ira P. Goldstein, "The FRL Primer". MIT Artificial Intelligence Laboratory Memo 408, 1977

[RG77b] Roberts, R. Bruce and Ira P. Goldstein, "The FRL Manual". MIT Artificial Intelligence Laboratory Memo 409, 1977

[R79] Rosenberg, Steven, "Reasoning in Incomplete Domains". Lawrence Berkeley Laboratory Memo LBL-8721, 1979

[SJR81] Stonebraker, Michael, Rowland Johnson, Steven Rosenberg, "Extending INGRES With a Rules System". Forthcoming.

[W77] Winston, Patrick H., *Aritficial Intelligence.* Addison-Wesley, 1977.

This report was done with support from the Department of Energy. Any conclusions or opinions expressed in this report represent solely those of the author(s) and not necessarily those of The Regents of the University of California, the Lawrence Berkeley Laboratory or the Department of Energy.

Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable.

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720