AN APPROACH TO THE USER INTERFACE AND SEMANTIC INTEGRITY FOR A RELATIONAL DBMS

Haim Kilov
K.Marx Str.75-13
Riga 11
USSR

Introduction. A recent analysis of feature catalogs of eleven prominent relational database management systems [1] defined 12 different interfaces. At least 6 of them (database schema definition, integrity constraints, database generation, report generation, special data entry, and database dictionary) involve users demanding - in a realistic environment - large amounts of information. Unfortunately, it became apparent from this analysis that exactly these interfaces were underemphasized: they were either not present at all in seven or more systems, or showed "considerable variability".

If, however, a database management system is designed and developed as a product rather than merely as an experiment, then user interface issues immediately become issues of utmost importance. After all, it is up to the users to decide whether or not to accept the system; inadequate interface capabilities lead to user non-acceptance, or, in the "best" case, frustration.

We attempted to design our relational DBMS [2,3,4,5] with these concerns in mind, and we tried to provide a proper user interface from the very beginning of the design (and delivery) of our product. It is well-known [6,7,8] that the relational model of data provides an adequate and convenient support for achieving the goals mentioned. In order to do this, the "basic" relational model should be semantically extended (various ways are possible, e.g., [5-8]); these extensions play an important role in creating both a (more or less) adequate model of reality and easier and more meaningful user interface.

Meta-database architecture and semantic integrity. It is widely accepted now [9] that a data dictionary/directory (a meta-database) must be designed as an integral part and, moreover, as the nucleus of a database management system. Data quality and data semantics support depends to a great extent upon the completeness and quality of information in this meta-database.

The main idea in the design of our meta-database is rather simple - all texts should be stored and handled separately from non-texts [2,4,5]. This approach was already proposed for relational DBMSs [10] - but only for efficient storage of information (in order to "squeeze" the database). Much more important, however, is the possibility of supporting essential semantic integrity aspects - which, as will be seen, contribute in an important manner to the creation of a friendly user interface as well.

Coding literals. We considered the idea of separate handlings of texts (also known as literals) as leading naturally to the more well-known and very important idea of the "crucial role of domains" [7] within a relational DBMS. Namely, all texts in our DBMS are organized into named text tables, and each such table corresponds to a separate textual database domain. "System" texts, i.e. names of relations, attributes, domains, etc., are grouped into system text tables.

The structure of all text tables as well as the handling of all literal values is identical. Thus, the domains in our database are not machine-oriented, i.e. are not defined in terms of character strings or integers - on the contrary: the domains are user-oriented, i.e. there are separate domains for, e.g., employee names, job titles, colors, heights, weights, etc. For each text domain an "explicit" act of creation" [8] is necessary for each of its text values. In such a way, text values

will be manually checked only at their creation - for the first and only time; later, during bulk data input (e.g., from user documents), each text value will be automatically (by the system) checked for existence in the proper domain [2,3,5]. (Compare [8]: "Symbols are all too often accepted as the names of people, companies, addresses, states, countries, and so on, with no test at all for their existence"). Moreover, all too often so called "user-defined codes" are (still) enforced by systems (actually, by data processing people), and so such unnatural things as - user defined! - department codes, employee numbers, and even color codes do appear. Evidently, "technical personnel" becomes trained in creating and using these usually numeric (and non-mnemonic) codes. However, this unnatural approach could and should be dropped - because names used in an interface with a DBMS should reflect "the conventions we use for naming people and things" [8]. The system itself should create and maintain an internal code for a name without the assistance of the user - whereas the user need not know any system-defined codes and could communicate with the system in his (or her) own terms. Evidently, the coding step could be done by the computer much better than by a human!

Synonyms. On the other hand, separate handling of text values provided a very important possibility to "explain" to the system that certain syntactically different texts are in fact semantically equivalent - because distinct names do not necessarily represent different objects. For example, a certain job title may be written on real documents (which are not manually copied on preprinted forms "suitable for input") as "senior programmer/analyst", "sen.progr/anal.", "senior progr.-anal.", "senior progr./anal.", "programmer/analyst, senior", "sr.prog/anal", "SAP", etc. Naturally, all of these texts must be announced to the system as equivalent - i.e. as synonyms, and the system will then assign to them the same internal code. Then, this internal code will be used as a representative of the whole set ("cluster") of synonyms in all of the database relation records. This means that data could be provided to the system in terms suitable and convenient for a user; the system does not enforce the use of one fixed, predetermined name for a "thing" - as various users (or various user classes) may be accustomed to different names and should be able to use them (compare [8]). Our system will understand all such names. One of them must be chosen from the synonym cluster and presented to the system as "standard" (e.g., "sen.progr./anal."), and, if desirable, another may be chosen as "expanded" (e.g., "senior programmer/analyst"). Only the standard or expanded name of a thing may be output.

The implementation of these ideas was especially attractive as it was built upon the B-tree based access method - "Dynamic Paged Memory" [2,11]. Each text table (i.e. each text domain) comprises a separate B-tree. In order to create an internal code for a text, this text is first of all squeezed (all non-significant blanks are removed), and then hashed into a 3-byte hash code. The fourth byte is used for collision handling, and so each text gets a unique key - its 4-byte code. (Incidentally, a test using 10000 real texts with lengths from 1 to 256 bytes showed no collisions at all! The texts were taken from our DBMS documentation files.) In such a way, every text could be easily located in its domain (and tested for existence). However, each cluster of synonyms should obtain a unique representative - a "logical" code which is to be stored in each relation record corresponding to a tuple which contains such a text. In the first version, the hash code of the "standard" text (considered to be the "main synonym") was taken as that representative. However, it was decided in the second version to create a special representative ("surrogate"), independent of the hash code, for each such standard text. Anyway, each synonym entry in the text table contains the corresponding "logical" code of its representative, and the standard text (i.e. main synonym) entry contains "physical" codes (i.e. keys) of all its synonyms as well as some other information.

Text tables for "system" texts, i.e. for domain names, relation names, attribute names, etc., are organized in the same way. However, for these tables each standard text value entry contains a reference to the corresponding entry in the system catalog of domains, relations, attributes, etc. In such a way it is easily possible to handle the catalog information using any synonym of the relevant schema name. (After all, a certain attribute may be called by various users "employee name", "emp-name", "empname", "empl.name", "name", and, maybe, even "n"). As a by-product, each relation record is of fixed length; naturally, each meta-database record is of varying length. Much more important, however, is the possibility to "talk about the symbols which name a thing separately from the representative of that thing [in the system]" [8].

Assuring Integrity. The semantic integrity of text domains can be easily assured in this system. Namely, first of all "all" text values (i.e. names) are created and announced to the database. During this stage - and only then - these values must be manually checked. Of course, if system names are created then standard system names must be accompanied by the relevant schema information about the corresponding domain, relation, attribute, etc. Afterwards, an existence test is automatically applied to every (user and system) text value which is to be input into the database. If this value exists in its domain then the corresponding logical code is located and taken as its representative in the relation record. Otherwise, the value is considered invalid - and may either actually be invalid, or show a deficiency in the "creation act" (either not all texts were created or a wrong name was created instead of the correct one). In the last case, it is easily possible to add a new text value or to replace an existing incorrect one. Of course, in order to successfully delete a standard text value, its representative must be absent in all records of all relations the attributes of which are defined on the corresponding domain (if permitted, however, such a representative may be replaced by a null value).

When text values are first entered into their domain, in order to be explicitly created before they are referenced, the amount of system-provided checks is not too much - as only the alphabet (set of permissible characters) and maximum length could be checked. On the other hand, numeric values are stored directly in the relation records (and not in a corresponding domain table); however, the amount of (syntax) check provided by the system is much greater for such values. For a numeric domain, it is possible to enter in its catalog record (and use in later checks of numeric values from this domain) such parameters as base type, range, divisibility (for integer base types), prohibited values, etc.

The relational join operation is defined in our DBMS only for attributes having equal domains - and in such a way only meaningful joins are possible. It should be reiterated that domains are defined semantically, as user-oriented named objects, and therefore domains with equal properties and distinct names (from distinct synonym clusters) are different (e.g., "age in years" and "height in inches" both have a coinciding base type - "short integer" - and a coinciding range - "0 - 100"; however, it is impossible in our database to formulate a query like "print the names of all persons for whom age in years is equal to height in inches" - as corresponding domains are different). Evidently, all synonyms from a synonym cluster are considered as names of the same domain. (Domain names for each relation attribute are provided in the definition of the relation schema.)

It is interesting to note that the idea of a system-defined surrogate for each entity (see, e.g., [6]) may perhaps be considered as a logical development of the idea of creating a system-defined representative for each text. Moreover, in our implementation surrogates for (kernel) entities with a one-attribute key coincide with representatives for the text values from the domain corresponding to that attribute. On the other hand, if an entity has a multi-attribute key then an additional attribute

("artificial key") is created by the system, and the system-created surrogate of the corresponding entity is stored there. In such a way, access to entity properties may be made using its multi-attribute key, whereas each entity has a one-attribute representative (its surrogate) which enters associations, etc. Actually, we accepted main ideas of Codd's RM/T [6] as the basis for (a certain level of) our implementation; however, unlike [6], there exist different kinds of characteristic entities in our model (and we think about other types of entities like sets, group entities, etc.). Namely, characteristic entities of one kind - as job histories - disappear when their kernel entity - employee - disappears; however, characteristic entities of another kind - as rank salary ranges - do not disappear when their kernel entities - employees with this rank - disappear (as new ones with the rank may be hired). These two kinds of characteristic entities must be supported differently; the last one is better described as a "reference entity".

Data input. Evidently, facilities for bulk data input (as well as for bulk information output) should differ from facilities for query-language-type commands. As our system had to be designed and implemented for batch users, large amounts of paper documents had to be both input into and produced by the system.

Our main decision concerning data input was to completely eliminate the coding step (see above, and also [12]). Best of all the causes in our decision can be expressed using a quotation from [12]: "In a typical commercial installation, many input items are manually copied onto preprinted forms... Cost considerations alone dictate that we eliminate this medieval practice, leaving the formatting, sequencing, and coding to the computer itself". Our decision was very non-traditional, and the book [12] helped us to reaffirm our then somewhat heretic point of view [2,3] - both to ourselves and to our colleagues, though [12] treated interactive rather than batch input.

The goal of the "data input processor" was to permit input directly from as many existing paper document forms as possible. All these documents are traditionally filled in without any consideration of the "computer needs"; the system has to accept user-oriented documents, check them, and insert the correct information into the database. In order to accept information from such documents, variable-length fields, variable-length lists, etc. should be the norm, as text values appear much more often than do documents specially suited for keying. Moreover, these documents usually contain no "filling in leading zeros, lettering within boxes, and all the other insults poor input design has forced upon us" [12]. In order to handle such information, we consider all document fields as being of varying (and, maybe, large) length. Each such field is separated by a delimiter, and the default value of a delimiter (used wherever possible) is a blank (or several blanks). If, however, a multiple-word item must be entered then the delimiter for such a field must be pre-specified (e.g., semicolon; all blanks surrounding a delimiter are non-significant). Each document line is separated by a line delimiter, and if there are "not enough" values in a line than it is automatically filled in with null values. In such a way, we dropped not only the idea of manual coding, but also the idea of fixed-format input; only free-format input is used. There is a lot of other syntax sugar accepted by the data input processor (e.g., the use of ditto marks for carryover from the previous line, etc.) [2,3].

Document definition. Each document form must be made known to the system, and a "document definition form" is used in order to do this; such a form is also a document and is filled (and keyed) in using the same simple rules as for any other document. The document definition form is pre-processed by the system, and then is interpreted when real documents of the corresponding type are input. In such a way a certain degree of data independence is provided because the document definition does not change when corresponding schema (relation or domain) properties are changed, - unless they are changed too drastically, e.g., an attribute is dropped altogether.

The rules for keying in any document are simple enough - each "cell" must be followed by one or more blanks; each line must be followed by end-of-line delimiter, and each document - by end-of-document delimiter. Otherwise, it is necessary to key in exactly what is written on the form (so the form should be filled in legibly); however, each empty cell followed by at least one non-empty cell in its line must be keyed in as a certain pre-specified character, e.g., as a question mark "?". Naturally, alternate versions of these rules are possible. For example, if the default separator is a semicolon then question marks in empty cells are not needed; on the other hand, if the default separator is a blank then all explicit separators usually have to be explicitly written on the document. The system accepts any of such alternate versions - the corresponding information (a separator for each item, etc.) must be specified on the document definition form.

The use of forms and fill-in-the-blank approach is very helpful to the document users who as a rule are non-programmers. In batch mode, there are no other prompting means, whereas the value of such means cannot be underestimated. Of course, the document definition form does not specify any (syntax or semantic) item checks with respect to the database as all of them are already specified in the database schema and text tables. However, this form does specify, if necessary, checks within a document line (e.g., price=amount=cost) and within a document column (totals).

It should be noted, in passing, that the data input processor accepts correctly inevitable variations in expressing items with such base types as data and money; e.g., "NOV],]98]", "NOVEMBER 1, 1981", "1 NOV 81", "1 NOV, 81", "1.X1.81", etc., - all will be understood by the system.

Besides usual documents which are filled in by "non-privileged" users, there exist documents for specifying the database schema which are filled in usually by the database administrator and which actually represent the DDL. The most important of them are domain definition, relation definition, and text processing forms (in fact, some of the text processing forms may be filled in by non-privileged users). Naturally, the necessary prompting facilities are provided here as well. For DDL documents the fill-in-the-blank approach is of especial importance because the amount of information to be presented to the system (and to be input into the meta-database) during, e.g., a relation definition, is rather high, and the administrator (who need not be a programmer) needs some form of prompting simply in order not to forget anything when he or she specifies the database schema.

Error messages. Providing proper error messages to the user is of utmost value for a data input processor. These error messages should use the terminology of the document to be input, should not be cryptic, and should not use (alien to the user) computer terminology. Moreover, there should be a lot of detailed message texts (a well-known counterexample: the message "SYNTAX ERROR" in FORTRAN compilers), and the messages should be parameterized to the extent possible, i.e. they should contain the value(s) of the incorrect item(s), the context of the erroneous item within the document and within the database, etc. Naturally, all kinds of erroneous information are possible and should be anticipated. We used this approach in the development of the data input processor - and the whole DBMS - and so there are several hundred different error texts now within the system. Naturally, the complete error text is always presented to the user rather than a cryptic code plus a several inches thick manual containing these codes with corresponding texts. It is only too well-known how inadequate error messages (and/or system crashes because of user errors) irritate the users and possibly lead to the idea of dropping the system altogether; therefore instead of the obsolete - and incorrect - approach "garbage in - garbage out" we use "garbage in - error messages out".

After a "batch" of documents (considered as a transaction) is input into the system, the user gets a listing (echo) of all the documents (or, if he so desires, only of error-containing documents) together with the error messages, if any. Then the user corrects these errors directly on the listing, and the process of transaction input is re-iterated until no errors will be encountered anymore by the system. Only then does the actual database update take place; the intermediate data are stored in transient files and not in the database relation records. Of course, the system warns the user and the database administrator about any transient files whose lifetime became "too large".

It should be noted that the approach used in entering text items may be in some sense considered as adaptive - first of all, the "standard" texts together with their synonyms are made known to the system (using a text processing form). Then, if a new text (i.e. a synonym) is encountered during document input, the system prints a message telling that such a text is unknown; the user checks this text, and if it really is an acceptable synonym, adds this synonym to the corresponding text domain (or replaces an incorrect synonym text with the correct one), again from the text processing form. Afterwards the text in question will be successfully input (from the transient file), and in this way existence lists for text domains are dynamically augmented with relevant information; such a modification is neither exceptional, nor traumatic. Moreover, it is possible to replace an incorrect or obsolete standard name with a correct one even if its representative already exists in the database records - as this representative now is a logical code (rather than a hash code) and therefore does not change when the corresponding standard name changes.

Information output. Information must be presented to the user in an understandable and convenient form and, moreover, in accordance with certain aesthetic requirements to a formatted output document. These requirements often are defined not even by the (end) user, but rather by his customer who may wish to have the paper document provided to him by the system in exactly the same format as he was used to when all forms were prepared manually. It may happen that the person or group deciding whether to accept or reject the use of a particular DBMS evaluates the system using only the output documents - therefore the quality and flexibility of the output subsystem may be of critical importance. After all, it is only natural for a user to demand his information in such a format as he wants. If the system does not provide adequate output possibilities then a large amount of tedious, boring, and unpleasant application programming is necessary; good programmers are usually reluctant to write such programs.

Documents of very different kinds are to be output. It is necessary to output (echo) the input documents (the diversity of their forms is evident), to provide the user with the information he/she wants, perhaps, in the same way as he/she was used to earlier, and to display the database schema information to various classes of users and administrators. Naturally, bulk output facilities must differ from facilities for output of a few lines of a single query result. It is also evident that facilities for output in batch mode must differ from interactive output ones. It should be noted especially that because of user demands very extensive and perhaps somewhat exotic possibilities should be provided for, e.g., output of very wide documents, multi-level headings, etc.

The PRINT facilities of existing relational DBMSs are usually rather fixed and do not deal with a relatively large volume of highly formatted output. A notable exception is the NOMAD [13] DBMS which does have extensive output facilities - and does stress them! However, even there the structure of an output document is in some important aspects restricted.

It was decided, in the development of our information output system - the "printer", - to make an extensive use of the liberal text handling facilities which already existed. It is easy to see that situations when the width of an output document column is less than the text length may be encountered quite often. In this sense, a usual (e.g., relation) tuple to be output into a document line does not differ, from the printer's point of view, from a document header line. Each of these logical lines may consist of several physical lines. If a text value does not fit in one line in its (column) field then the system creates a "cell" for output of this value. The width of this cell is equal to the width of the field, and the height (number of lines, or "thickness") is defined by the printer depending depending on the text length and positions of delimiters (spaces, commas, etc.). In such a way, the thickness of the logical line to be output is equal to the thickness of the thickest of its cells. The printer itself does the whole painstaking job of accurate and pleasing formatting and layout (including justification, line positioning, etc.); multi-page output including heading layout, etc. is also dealt with automatically.

The printer provides possibilities both for output of (possibly, sorted) tuples of database relations and reference information in "standard" format - and for output of documents of almost any kind in any desirable format, in accordance with the user demands. The standard facilities provide a possibility to vary, to a considerable extent, the format of information output, and so in many cases these facilities are fully adequate. (Evidently, in order to output a relation in standard format it is sufficient to tell the system the name of this relation - and only if the default parameters are to be overrided, the user has to specify the overriding information.) However, there exist situations when standard facilities, even if overriding is provided, do not grant the service needed. In such a case it is easy to use from an application program a complex of printer's software tools which permits output of almost any desirable document in any desirable format. It is very important that a user interface with this facility be simple enough; in particular, the application programmer should not be required either to learn and use a (brand new) format language (like RPG), or to do any painstaking and dull job - which will be done by the printer.

The document to be output usually consists of a title (system + user), heading(s) (possibly, multi-level), and usual tuples. In order to output tuple elements, the printer must know the displacement and width for each column. These data are either calculated automatically (using both the database schema information and user's demands concerning page width, etc.), or may be explicitly provided by the user. If a page is "narrow" enough then two pages may be output alongside each other; on the other hand, if a page is "very wide" then a logical page will occupy several physical pages. In specifying the page width, the user should only remember that very narrow columns for (text) output will lead to successful output as well, but the tuples (logical lines) may then become very thick.

In considering the application programmer's interface with the printer, it should be stressed that the system codes are never exposed to the user, and so the user has no access to the relation record (unlike the relation tuple). Therefore in order to obtain information from a tuple, it is necessary to know the name(s) of the attribute(s). The set of printer's software tools provides possibilities for creating the necessary environment, calculating the column widths and displacements, various kinds of output of a usual tuple, output of a heading tuple or its part, creation of a multi-level heading and output of a level from such a heading, output of a line or its part, various kinds of layout, etc. The user interface with this lower level of the printer is organized in such a way as to be very short and simple for often used facilities and become gradually more complex for less often used and more refined ("exceptional") ones.

Implementation remarks. It was decided to provide "trial" users with some database facilities as soon as they were operational. In such a way, a certain "privileged" class of users was able to test the system in a real-life situation. This approach of evolutionary design and delivery is discussed in [14]. Of course, a database management system differs from an application as it is necessary to develop quite a large part of the system in order to provide anything to a user; on the other hand, it does not all mean that the whole DBMS must be completed before its delivery. Some parts of the system, including the user interface, were designed with actual user feedback which provided much help in testing and evaluating the system and sometimes in its design as well. These privileged users were very eager in evaluating the system, and when the results were positive they provided a kind of publicity very desirable in accepting a new product.

Acknowledgements. I'd like to mention the following colleagues who contributed in a major way to the system: M. German, I. Popova, A. Tovstoles, G. Sheinkman. Thanks go also to our then students A. Savin and M. Vainshenker for taking an active part in the early stages of the implementation. And the book by T. Gilb and G. Weinberg provided important moral support when we began.

References.

1. D. Ries. DBMS architecture and the relational data model. A report of the Relational Task Group of the ANSI/SPARC Database Study Group. Draft. March 6, 1981. Computer Corp. of America, Cambridge, MA.
2. E. Kaplan, H. Kilov, A. Tovstoles, I. Popova, G. Sheinkman. Relational hybrid database management system. Concepts and ways of implementation. Riga, latNIINTI, 1978.
3. H. Kilov, I. Popova. Data input processor. In: Proceedings of the 2nd National Conference of ES Computer Users, Moscow, 1979, pp. 201-204.
4. H. Kilov, I.Popova. Text organization in a DBMS. Proc. of the 2nd National Conf. of ES Computer Users, Moscow, 1979, pp.105-108.
5. H. Kilov, I. Popova. Meta-database architecture for a relational DBMS. Programmirovanye, 1981, No.1, pp. 83-88.
6. E.F. Codd. Extending the database relational model to capture more meaning. TODS, 4 (1979), 4, 397-434.
7. E.F. Codd. Data models in database managment. In: Proc. of the Workshop on data abstraction, databases, and conceptual modelling, Pingree Park, Colo., June 23-26, 1980. SIGPLAN Notices, 16 (1981), 1, 112-114.
8. W. Kent. Data and reality. North-Holland Publ. Co., 1979.
9. The ANSI/SPARC DBMS framework report of the Study Group on database management systems. Ed. by D. Tsichritzis and A. Klug, University of Toronto. Information Systems, 3 (1978), 173-191.
10. Won Kim. Relational database systems. Comput.Surv. 11 (1979), 3, 185-211.
11. A. Tovstoles, E.Kaplan, G.Sheinkman. A dynamic paged memory system in OS/ES. In: Proc. of the 2nd National Conf. of ES Computer Users, Moscow, 1979, pp.297-300.
12. T. Gilb, G. Weinberg. Humanized Input. Winthrop Publ., 1977.
13. D. McCracken. A guide to NOMAD for application development. National CSS, Inc. Wilton, Conn. February 1980.
14. T. Gilb. Evolutionary development. ACM SIGSOFT Software Engineering Notes, 6 (1981), 2, p.17.

Note! This paper was edited lightly and submitted by Ben Shneiderman, Department of Computer Science, University of Maryland, College Park, MD 20742.