A retrieval model for common textual database management systems[*]

Yves MARCOUX

EBSI, Université de Montréal
C.P. 6128, Succ. A, Montréal, Québec, Canada, H3C 3J7
E-mail: marcoux@ere.umontreal.ca

Abstract

The social mission of information professionals is to provide society with high quality information storage and retrieval services. In order to fulfill this mission, the professionals need to have an understanding of the tools they use that is sufficiently thorough for predicting the behavior of these tools in all normal circumstances.

One important class of tools used by information professionals are the *textual database management systems* (TDBMS's). At present, the retrieval capabilities of these systems are almost without exception incompletely described, a situation which sometimes renders the accurate prediction of their behavior difficult. Thus, the quality of information storage and retrieval services that the body of information professionals can provide society is not as high as it could be. To correct this situation, the retrieval behavior of the TDBMS's available to information professionals must be precisely and exhaustively described. In other words, an abstract retrieval model has to be elaborated for them.

[*]This research was done as part of IDRC Project 3-P-88-1010-3, *Réseau pédagogique pour l'enseignement de l'informatique documentaire*, jointly supported by IDRC (International Development Research Center of Canada) and CIDA (Canadian International Development Agency), with additional funding from UNESCO (United Nations Education, Science and Culture Organization). The project involved the *École des sciences de l'information* of Rabat (Morocco), the *École des bibliothécaires, archivistes et documentalistes* of Dakar (Senegal) and the *École de bibliothéconomie et des sciences de l'information* of Montréal (Canada). A computer aided instruction course based on the ideas in this paper was developed as part of the project.

In this paper, we present what we believe to be the first formally defined abstract retrieval model especially designed for describing the retrieval behavior of common, everyday TDBMS's. The model is rigorously defined, and can be used as a basis for describing the retrieval behavior of most of the existing TDBMS's that use boolean logic and so-called "repeating" values.

The process of modeling the retrieval behavior of TDBMS's shows that the form of query expressions accepted by existing systems is fairly restricted, and suggests a possible (and easily implementable) generalization. We show that this generalization would not only allow the formulation of interesting and meaningful requests that are *impossible* (or very difficult) to formulate in the present systems, but would in fact grant *logical completeness* to retrieval languages, a form of completeness analogous to relational completeness in the relational model.

1 Introduction

The social mission of information professionals is to provide society with high quality information storage and retrieval services. In order to fulfill this mission, the professionals need to have an understanding of the tools they use that is sufficiently thorough for predicting the behavior of these tools in all normal circumstances.

One important class of tools used by information professionals are the *textual database management systems* (TDBMS's). These include many packages available on various platforms for local development of textual databases, as well as many online database servers. At present, the retrieval capabilities of the existing TDBMS's are almost without exception incompletely described, as presented to their users (i.e., the information professionals), a situation which renders the accurate prediction of their behavior difficult, if not downright impossible, except by trial and error, which is hardly an acceptable method. Thus, the quality of information storage and retrieval services that the body of information professionals can provide society is not as high as it could be. To correct this situation, the retrieval behavior of the TDBMS's available to information professionals must be precisely and exhaustively described. In other words, an abstract retrieval model has to be elaborated for them.

Because the existing TDBMS's were developped in the absence of a formal retrieval model, they virtually all work differently as far as the details of retrieval are concerned, and their collective modeling poses a problem. One possible approach would be to define a huge, very general model, of which all existing systems would be particular cases (obtained by parametrization). This approach is likely to yield an unusable model, too general for being an effective thought-organizing tool, and to leave out some systems anyway. Thus, we do not consider this approach to be practical.

A probably better approach is to have a fairly simple model which, even though it does not describe any particular system exactly, can serve as a common basis for describing the behavior of *any* existing system. This is the approach we followed.

In this paper, we present what we believe to be the first formally defined abstract retrieval model especially designed for describing the retrieval behavior of common, everyday TDBMS's. The model is rigorously defined, and can be used as a basis for describing the retrieval behavior of most of the existing TDBMS's that use boolean logic and so-called "repeating" values (a "mild" case of non-atomic values).

A tremendous number of retrieval models and database models have been defined in the past, including "Non-Normal-Form" models (see [NMU91]), which bear ressemblance with the model defined here. However, these are not suitable for our purposes, because they are far too general. Working with a very restricted and specialized model allows us to include such detailed features as the allowed forms of query expressions, and investigate the possibility and consequences of relaxing the constraints on these forms.

The version of the model we present here does not include word-oriented operations (adjacency, proximity, etc.) The modeling of these operations poses specific problems, mainly because the implementation of these operations in existing systems does not give rise to an *extensional* semantics of query expressions (a semantics in which any subexpression is assigned a unique value, regardless of where in a query expression it occurs). We are currently working out the details of an extension to our model in which word-oriented operations are integrated and use an extensional semantics.

One side effect of using an abstract model is that it could serve as a basis for an eventual standardization of retrieval functionalities of TDBMS's (including a common retrieval language). The advantages of such a standardization for the community of information professionals (and, by transitivity, for society) are obvious, and attempts to perform such a standardization have been made in the past, with only relative success. We believe the lack of a formal model may have been partly responsible for the limited success of these endeavours.

The process of modeling has the other side effect of shedding new light on the behavior of the modeled objects. In our case, it revealed that the query languages of existing TDBMS's are fairly restricted, and suggested an easily implementable generalization. Namely, after the presentation of the model, we investigate a slight generalization of the form of query expressions accepted by existing systems, and show that it allows the formulation of (interesting and meaningful) requests that are *impossible* (or very difficult) to formulate in the present systems. In fact, we show that the proposed generalization of retrieval languages would grant them *logical completeness*, a form of completeness analogous to (although weaker than) relational completeness in the relational model.

2 Scope of the model

Our goal is to come up with a model that is as close as possible to the retrieval behavior of the "average" TDBMS. We include in the model data structuring capabilities and retrieval operations. However, we do not include such aspects as the concrete syntax of retrieval language, data manipulation operations (including data entry), reporting facilities, display of search results and search history management.

Usually, a retrieval or database model is introduced to investigate how retrieval or database management *could* be done (in the future). Because the semantic aspects of information retrieval are so important, most retrieval models include features or mechanisms that aim at capturing at least part of the semantic contents of documents (see for instance [BC76] or [Bla90]).

In contrast, the model proposed here attempts to describe what the mechanics of retrieval in TDBMS's *are now*. We deliberately leave out of the model all semantic aspects of retrieval. Note that, in real life, TDBMS's *are* used to support various indexing or representation schemes that do capture some semantic aspects of documents, however, we choose not to include these schemes in our model.

We certainly do not claim that our model represents the ideal way of doing retrieval in TDBMS's; however, we claim that it does represent fairly accurately the retrieval capabilities of TDBMS's today, and that it can suggest realistic (i.e., easily implementable) improvements.

3 The formal model: $\mathcal{FF}_0$

3.1 Name and notation

The name of a model is important, because it should convey as much of the nature of the model as possible. We chose the name "Flat-File" for our model, because this is usually the term used to refer to data structures created by common TDBMS's (see for instance [TL88]).

Because our model is just a first proposal, we expect that there will be many versions of it. Thus, we will use a subscripted notation to designate our model. We shall denote it by $\mathcal{FF}_0$.

3.2 Structure of a database

For the sake of clarity, we intersperse the definitions that directly relate to the model with auxiliary definitions and explanatory comments.

The basic data structure in $\mathcal{FF}_0$ is the *textual table*. It is a two dimensional array, the lines of which are called *records*, and the columns of which are called *fields*. A database in $\mathcal{FF}_0$ contains exactly one textual table, no less, no more. Thus, $\mathcal{FF}_0$, regarded as a database model, is a single-table model.

The number of fields in a textual table is a positive integer and is denoted by n. This number is fixed and will not vary during the lifetime of the database. The number of records in a textual table at any particular time is always a non-negative integer and is denoted by m. This number can vary during the lifetime of the database, according to the data manipulation operations carried out on the database. By "records and fields of a database", we shall mean the records and fields of the textual table associated with that database.

An alphabet is a non-empty finite set of symbols (characters). For any given alphabet Γ, Γ^* is the set of all finite (possibly empty) strings that can be made up using the symbols in Γ.

Associated with a database is a *base alphabet* Σ. All entries in that database's textual table are finite (possibly empty) *sequences of strings* from Σ^*. Note that a textual table is an homogeneous structure, because all entries have the same "data-type": sequence of character strings. Also note that the textual table is not in "first normal form" (relational terminology) because its entries are not atomic values.

The operation of concatenation of string sequences is denoted by $\oplus$. The length of a sequence σ (i.e., the number of string occurrences in σ)

is denoted by $|\sigma|$. The various string occurrences in σ are denoted with subscripts: $\sigma = (\sigma_1, \ldots, \sigma_{|\sigma|})$. We shall sometimes treat a sequence as a set; in those cases, the sequence shall be considered identical to the set containing exactly those strings that occur in the sequence. For example, $x \in \sigma$ will mean that x is a string that occurs in the sequence σ.

The fact that the entries of a textual table are *sequences* of strings, rather than strings, is to allow for so-called "repeatable fields". *A priori*, all fields in a textual table are repeatable, but there will be a way to restrict this feature to specific fields only.

The textual table associated with a database is denoted by T. The records and fields of T are indexed with natural numbers starting with 1. The entry in the ith record, jth field is denoted by $T_{i,j}$.

An entire record of T (seen as a one-dimensional array) is denoted using a single subscript on T; for instance, the ith record in T is denoted by T_i. If $R = T_i$, then R_j denotes $T_{i,j}$. We will often use the variable R to denote an arbitrary record in the database, and sometimes any record that could validly be in the database.

A number of various objects are associated with each field. For each j $(1 \leq j \leq n)$, the following objects are associated with field j:

1. A *field name* FN_j, which is an arbitrary symbol. In a database, the field names must be pairwise distinct.

2. An *optionality indicator* OPT_j, which is simply a boolean value (TRUE or FALSE).

3. A *repeatability indicator* REP_j, which is also a boolean value.

4. A *field definition* FD_j, which is a (possibly empty) subset of $\{1, \ldots, j-1\}$. If $FD_j = \emptyset$, then field j is said to be a *proper field*; otherwise, it is said to be a *calculated field*. Note that, since $FD_1 = \emptyset$, field 1 is always a proper field.

5. An *automatic transformation* f_j. An automatic transformation is a total function from Σ^* to Σ^*. If σ is a sequence of strings, then the notation $f_j(\sigma)$ denotes the sequence $(f_j(\sigma_1), \ldots, f_j(\sigma_{|\sigma|}))$.

The field name is the symbolic name by which the field will be referred to in search expressions. The optionality indicator indicates whether the field can be "omitted" or not. Field j is considered to be "omitted" in record i iff $T_{i,j}$ is the empty sequence. Thus, if OPT_j is FALSE, then for all records

R in the database, $|R_j| > 0$. The repeatability indicator indicates whether a field can be "repeated" or not. Field j is considered to be "repeated" in record i iff $|T_{i,j}| > 1$. Thus, if REP_j is FALSE, then for all records R in the database, $|R_j| \leq 1$.

The automatic transformation f_j is a sort of "pre-processing" that is done on the strings appearing in field j, and we shall say more about it in a moment.

The field definition FD_j determines where the information in field j comes from. If $FD_j = \emptyset$, then the information that ends up in field j is expected to be entered "by the user" during data entry operations. Otherwise, the information in field j is "calculated", using f_j, from the information present in other fields, as indicated by FD_j. More precisely, for all records R in the database, $R_j = f_j(R_{d_1} \oplus \cdots \oplus R_{d_h})$, where $d_1, \ldots, d_h$ are the elements of FD_j, and $d_1 < \cdots < d_h$. The requirement that FD_j be subset of $\{1, \ldots, j-1\}$ is imposed only to avoid circularity in the definitions of the fields.

As can be seen from the above paragraph, the automatic transformation f_j will be applied to all data entering field j whenever field j is a calculated field. When field j is a proper field, then f_j is also expected to be applied to all data entering the field, in that it should be applied (by the TDBMS) to the strings entered "by the user". Thus, the database will satisfy the condition that for all records R, for all fields j, $R_j \subseteq \mathbf{image}(f_j)$.

Typical automatic transformations would include stripping of leading and/or trailing blanks, transforming letters to upper- or lower-case, eliminating specific substrings (e.g., punctuation, blanks, initial articles, stopwords, etc.) Note that an automatic transformation *can* be the identity.

The purpose of field definitions is to offer the same functionality as the "multi-field indexes" that can be defined in most existing TDBMS's. They are also intended to allow searching the same field with different automatic transformations (e.g., authors' names as entered in one field, and transformed to upper-case in another).

Finally, K, an integer between 1 and n, is the *key field designation.* Field K is the "primary key" field of the textual table, i.e., for any two records T_i and T_j, if $i \neq j$, then $T_{i,K} \neq T_{j,K}$. The condition that both OPT_K and REP_K be FALSE is also imposed.

Although the presence of a key field is not necessary in the model, we chose to include it because most existing TDBMS's support some form of "unique identification" of records. Multiple-field keys could have been included; however, they render the presentation of the model much more cumbersome, and we have thus chosen not to include them. Besides, very few existing systems allow multiple-field keys (a notable exception being the INMAGIC system [Inm92]).

This ends the structural part of the definition of a database.

3.3 Query language

We now introduce other elements that will allow the formulation of search expressions or queries. To begin with, we will discuss informally the kind of queries we want to be able to formulate. We use the example:

$$(\exists AU)[AU \neq \texttt{"}Smith\texttt{"} \ \& \ AU \neq \texttt{"}Royer\texttt{"}] \ \&$$

$$(\forall TI)[TI = \texttt{"}Spring\texttt{"} \vee TI = \texttt{"}Fall\texttt{"}]$$

The queries we want to formulate are sentences (formulas with no free variable) in some sort of logical language. In the example, AU and TI are field names (used as variable names), and $=$ and $\neq$ are *predicate names*, representing respectively string equality and inequality. Double-quotes (`" "`) delimit string constants, and the other symbols are the usual logical quantifiers, connectives, etc., with their usual meanings.

Intuitively, a query will retrieve from the database those records that "satisfy" the query. A record will be retrieved iff the query evaluates to TRUE when the field names it contains are allowed to range over the corresponding fields of the record (treated as sets for this purpose). The field names are quantified because of the possibility of "multiple occurrences", i.e., because a field in a record is a *sequence* of strings, and not just a simple string.

Thus, assuming field AU contains author names and TI contains titles, the above example would retrieve exactly those records with *at least* one author name other than Smith and Royer, and *no* title other than Spring or Fall.

In order to be able to formulate that kind of queries, we now introduce more objects in the definition of a database.

A finite number $q > 0$ of two-place predicates $P_1, \ldots, P_q$, each defined over $\Sigma^* \times \Sigma^*$, are associated with a database. Corresponding to each predicate P_p $(1 \leq p \leq q)$, is a *predicate name* PN_p, which is an arbitrary symbol. Like the field names, the predicate names must be pairwise distinct.

Before we add the last objects in the definition of a database, we need to define the *logical language* $\mathcal{L}$ associated with a database.

Let Λ be the set of symbols $\{(,),[,],\forall,\exists,\&,\vee,\neg,\prime,"\}$. Without loss of generality, we assume that the double-quote symbol (") is *not* in Σ, and that *no* field name FN_i or predicate name PN_p is in Λ.

We define $\mathcal{L}$ to be the set of all *sentences* that can be constructed with symbols from $\Lambda \cup \Sigma \cup \{FN_1, \ldots, FN_n, PN_1, \ldots, PN_q\}$, using the following rules:

1. A *variable name* is either a field name or a field name followed by one or more prime symbols ($\prime$). We say that the variable name *contains* the field name that is part of it.

2. A *string constant* is a string from Σ^* immediately preceded and followed by the double-quote symbol (").

3. A *positive atom* is an expression of the form "$v\ PN\ c$", where PN is a predicate name, v is a variable name, and c is a string constant (we use infix notation for predicates).

4. A *negative atom* is a positive atom preceded by the symbol $\neg$.

5. An *atom* is either a positive atom or a negative atom.

6. *Formulas* are built from atoms in the usual way.

7. A *sentence* is a formula with no free variable.

It is worthwhile noting that a predicate is always used with one variable name and one string constant. This is an essential feature of the model, and of the existing TDBMS's. Note that most TDBMS's have a "range" predicate which is three-place, but is always used with one variable name and two string constants. We shall say more about such predicates later.

Here is an example of a complex sentence:

$$(\forall AU') \neg (\exists TI)[(\neg\, TI = \texttt{"}\textit{Summer}\texttt{"}) \ \&$$

$$(\forall AU'')[AU'' > \texttt{"}\textit{Smith}\texttt{"} \vee AU' < \texttt{"}\textit{Bono}\texttt{"}]]$$

The way the sentences in $\mathcal{L}$ are interpreted is somewhat peculiar, and is based among other things on ideas of *many sorted logic* [YCC92].

A database record R is said to *satisfy* (or *model*) a sentence $S \in \mathcal{L}$ iff S evaluates to TRUE under the following rules of interpretation:

1. Each variable name v in S is allowed to range over R_j, where FN_j is the field name contained in v.

2. A string constant c, when occurring in a positive atom "$v\ PN\ c$" in S, is interpreted as the string $f_j(x)$, where FN_j is the field name contained in v, and $x \in \Sigma^*$ is the string obtained by stripping off the double-quotes from c.

3. Each predicate name PN occurring in S is interpreted as the predicate P_p, where $PN = PN_p$.

4. All other symbols in S, including quantifiers and logical connectives, are interpreted as usual.

Naturally, any subexpression starting with $(\forall v)$ (respectively, $(\exists v)$), where v ranges over the empty set, evaluates to TRUE (respectively FALSE).

Note that, according to rule (2) above, the exact string denoted by a particular occurrence of a string constant in a sentence, depends on the field name contained in the positive atom that contains that particular string constant occurrence. This peculiarity is introduced here to reflect the fact that most TDBMS's perform on the search terms the same transformations that they perform on the inverted files entries: for instance, if authors' names are stored in upper-case in an inverted file, then the search terms are also converted to upper-case when searching on this inverted file.

The fact that a record R models a sentence $S \in \mathcal{L}$ is denoted by $R \models S$. Thus, a sentence S retrieves from the database exactly those records R such that $R \models S$.

The last objects we introduce as part of the definition of a database are a *query language* $QL \subseteq \mathcal{L}$, and a *validation expression* $VAL \in QL$. The query language represents all the queries that can be formulated (either directly or using some concrete syntax outside the model) for retrieving records from the database. In general (and certainly for the existing TDBMS's), it will be a proper subset of $\mathcal{L}$. The sentence VAL expresses a condition that has to be met by all records in the database; i.e., for all records R in the database, $R \models VAL$.

Two sentences S_1 and S_2 from $\mathcal{L}$ are said to be *query-equivalent* iff for all *possible* records R, $R \models S_1 \Longleftrightarrow R \models S_2$. Note that in this definition, we consider all the records that *could* validly be in the database, not just those that happen to be in the database at any particular point of time.

A sentence $S \in \mathcal{L}$ is said to be *expressible* in QL iff there exists a sentence in QL that is query-equivalent to S. The query language QL of a database is said to be *logically complete* iff for all $S \in \mathcal{L}$, S is expressible in QL.

3.4 Data manipulation operations

Although the data manipulation operations are not part of the model, we have to specify some aspects of how the TDBMS is expected to perform them. Namely, the TDBMS is expected to always leave the textual table associated with a database in a state such that all the conditions stated in the preceding sections are satisfied.

Thus, for instance, the TDBMS shall not allow a record R to be inserted in the textual table if it is not the case that $R \models VAL$. Conceptually, we can imagine that if the "user" ever tried to insert such a record, the TDBMS would remain in data-entry mode, or would terminate the operation with an error, leaving the database in its original state. Similarly, the TDBMS should deny performing any data manipulation operation that would result in any of the conditions stated in the preceding sections to be violated.

As far as data-entry is concerned, the TDBMS is of course expected to perform the automatic transformations and the calculations of the calculated fields, but this does not change in any way its responsibility of verifying that the database is left in an acceptable condition, and of denying the operation otherwise.

3.5 An example

We give an example of a database with 3 fields, i.e., $n = 3$. We take the base alphabet Σ to be the set of all roman letters (upper- and lower-case) and decimal digits (0 to 9).

We choose as field names $FN_1 = AU$, $FN_2 = YEAR$, $FN_3 = ANY$. We want fields 1 and 2 to be proper fields, and field 3 to be the "grouping" of fields 1 and 2 (unmodified), thus, we set $FD_1 = FD_2 = \emptyset$, $FD_3 = \{1, 2\}$, and f_3 as the identity. We want the key field to be field 1, so we set $K = 1$ and, thus, $OPT_1 = REP_1 = FALSE$.

The database records are expected to contain a single author name in AU, and the different years in which that author published something as multiple occurrences of $YEAR$. Since $YEAR$ and ANY must accept multiple occurrences, we set $REP_2 = REP_3 = TRUE$. We choose to declare $YEAR$ as optional, thus, we set $OPT_2 = TRUE$. The setting of OPT_3 has no effect, since ANY will always have something in it, because it "covers" the key field (AU).

We do not want any modification or special validation on the author's name, so we set f_1 as the identity. However, we would like the information entered as "years" to be checked to be numeric. We can do this by setting f_2 to be the function that leaves unchanged any string that is entirely numeric, and maps all other strings to the empty string. The validation expression (see below) will complete the validation "mechanism".

As predicates, we use $=$, $\neq$, $\leq$, $\geq$, $<$, and $>$, with precisely these symbols as predicate names. The exact meaning of $\leq$, $\geq$, $<$, and $>$ should be thoroughly described, but we shall only say that $x \geq$ `""` for all $x \in \Sigma^*$, where `""` denotes the empty string. We will suppose that the query language QL is equal to the logical language $\mathcal{L}$ of the database. Finally, we set the validation expression VAL to be the sentence "$(\forall YEAR)[YEAR \neq$ `""`$]$". Thus, only numeric information will be accepted in $YEAR$.

Note that the sentences "$(\exists YEAR)[YEAR =$ `""`$]$" and "$(\exists YEAR)$ $[YEAR =$ `"`*abc*`"`$]$" are query-equivalent in the context of this particular database, because no record that could validly be in the database can satisfy either sentence. The sentences "$(\exists AU)[AU =$ `"`*Smith*`"`$]$" and "$(\forall AU)[AU =$ `"`*Smith*`"`$]$" are also query-equivalent, because AU does not admit multiple occurrences and is not optional.

An example of a record that could validly be in the database is

$$((\text{"}\textit{Royer}\text{"}), (\text{"}\textit{1990}\text{"}, \text{"}\textit{1991}\text{"}), (\text{"}\textit{Royer}\text{"}, \text{"}\textit{1990}\text{"}, \text{"}\textit{1991}\text{"}))$$

The records that contain something (i.e., at least one occurrence) in $YEAR$ can be retrieved with the sentence "$(\exists YEAR)[YEAR \geq$ `""`$]$".

Note that, in general, any specific TDBMS will allow the definition of databases with only a restricted choice of base alphabet, automatic transformations, predicates, predicate names, and query language.

3.6 Discussion

We could have used sets of strings instead of sequences of strings for realizing multiple occurrences. However, we chose sequences for two reasons: (i) most existing TDBMS's use the equivalent of sequences, and (ii) in some cases, it could make sense to have twice the same string; for instance, in a bibliographic database, it is conceivable that two different authors of a same work end up with an identical controlled form of their name. We could also have used multi-sets, but chose sequences for reason (i).

The reader may be surprised by the absence of the inverted files (or "indexes") from the model. The reason is that inverted files constitute one implementation technique (albeit popular) which, in accord with the data independance principle, should not intervene in the abstract definition of the model. In this respect, inverted files are similar to indexes in relational databases: by necessity, the data definition language of any particular system allows defining them, but they are not part of the formal model.

Although there is only one data-type *per se* in the model, it should be clear from the example above, that there are sufficient validation capabilities in the model to amply realize all forms of "typing" found in existing TDBMS's.

A problem with TDBMS's is that sometimes, one would like to have "parallel fields", e.g., an author field and a date field such that the ith author corresponds to the ith date, and be able to formulate queries that take into account this correspondance. Some TDBMS's allow doing the equivalent of this with *subfields*. For the above example, we would have a single field split up into two subfields separated by a delimiter. The MARC format of bibliographic records uses this device extensively [Jac92].

Working with subfields involves locating and manipulating substrings within strings, and we have found that the best way to introduce them in a model is together with word-oriented operations. Thus, we shall not discuss subfields until we introduce word-oriented operations.

4 Logical completeness

4.1 Existing TDBMS's

With most TDBMS's, the query language of a database is limited to sentences with a very specific form and does not include of all $\mathcal{L}$.

A sentence $S \in \mathcal{L}$ is said to be in *first restricted form* (RF1) iff (i) no quantifier occurs in S in the scope of another quantifier, (ii) all variable names occurring in S are field names (no primes), (iii) all quantifiers occurring in S are $\exists$'s, and (iv) all subexpressions of S that are within the scope of a quantifier are restricted to be a conjunction of atoms. A sentence $S \in \mathcal{L}$ is said to be in *second restricted form* (RF2) iff it is in RF1, and all subexpressions of S that are within the scope of a quantifier are restricted to be atoms.

It is easy to see that, with most TDBMS's, the query language of a database is restricted to the sentences in RF2.[1]

[1]In fact, it is usually further restricted to sentences in RF2 in which only positive atoms occur. This is not really a restriction if the set of predicates used is closed under complementation; however, in most TDBMS's, this is *not* the case, and thus, not all queries in RF2 are expressible. For example, the predicate $\neq$ is usually not present, with

A consequence of this restriction is that "range" queries (e.g., all authors between `"`A`"` and `"`L`"`) cannot be expressed by the conjunction of the predicates $\leq$ and $\geq$. What we want to do is "$(\exists AU)[AU \geq$ `"`A`"` $\&\ AU \leq$ `"`L`"`$]$". This is an RF1 query, but it can be shown not to be query-equivalent (in general) to any RF2 query, even when RF2 queries are allowed to include all of the predicates mentioned in our example database above. (The "obvious" RF2 formulation "$(\exists AU)[AU =$ `"`A`"`$] \vee \cdots \vee (\exists AU)[AU =$ `"`L`"`$]$" does not work, because, with the usual "alphabetical order" definition of $\leq$, there are infinitely many strings x such that `"`A`"` $\leq x \leq$ `"`L`"`.) Thus, some form of range predicate (a three-place predicate) is usually added to the query language, for the sole purpose of being able to express such queries.

4.2 Suggested generalization

Range queries are not the only ones that are not expressible with RF2 queries. The following RF1 query is another example: "$(\exists AU)[AU \neq$ `"`$Case$`"` $\&\ AU \neq$ `"`$Royer$`"`$]$". We claim that such queries are meaningful and interesting, and would justify generalizing the query languages of existing systems to include all RF1 queries. This would by the same token eliminate the necessity of adding specific predicates for range queries (although of course they could be left in for convenience as shorthands in the concrete syntax).

In fact, it can be shown that if the query language of a database includes all RF1 queries, then it is logically complete. The proof is similar to the standard proof that any pure monadic sentence is equivalent to a pure monadic sentence containing exactly the same predicate names and only one variable name [BJ80]. Thus, generalizing the query languages of existing systems to include all RF1 queries would greatly increase their expressive power.

As far as implementation is concerned, the proposed extension is fully compatible with the inverted file technique. An empirical "confirmation" of this fact is that most TDBMS's realize their range predicate by working on inverted files. Thus, the proposed generalization is practically feasible.

the consequence that sentences like "$(\exists AU)[\neg\ AU =$ `"`$Smith$`"`$]$" are not expressible. Note that the latter is *not* in general query-equivalent to "$\neg\ (\exists AU)[AU =$ `"`$Smith$`"`$]$".

Acknowledgment

We wish to thank Hervé Caussinus for many fruitful discussions and Cheval Cavelier for encouragement.

References

[Bla90] Blair, D.C. *Language and Representation in Information Retrieval.* Elsevier, Amsterdam, 1990.

[BC76] Bookstein, A. and Cooper, W. A general mathematical model for information retrieval systems. *Library Quarterly*, Vol. 46, No 2, 1976, 153–167.

[BJ80] Boolos, G.S. and Jeffrey, R.C. *Computability and Logic.* Second Edition, Cambridge University Press, 1980.

[Inm92] *INMAGIC Plus: User's Manual, Version 1.0*, INMAGIC Inc., Cambridge, MA, 1992.

[Jac92] Jacquesson, A. *L'informatisation des bibliothèques: historique, stratégie et perspectives.* Collection Bibliothèques, Éditions du cercle de la librairie, Paris 1992.

[NMU91] Ng, Y.K.; Melton, A. and Unger, E. A method for constructing generalized Non-Normal-Form models. *Proceedings, 19th Annual Computer Science Conf.*, ACM Press, 1991, 146–153.

[TL88] Tenopir, C. and Lundeen, G. *Managing Your Information; how to design and create a textual database on your microcomputer.* Neal-Schuman, New York, 1988.

[YCC92] Yang, J.S.H.; Chin, Y.H. and Chung, C.G. Many-sorted first-order logic database language. *The Computer Journal*, Vol. 35, No 2, 1992, 129–137.