Preferences: Putting More Knowledge into Queries *

M. Lacroix and P. Lavency

Philips Research Laboratory, Brussels

Abstract

Classical query languages provide a way to express mandatory qualifications on the data to be retrieved. They do not feature facilities for expressing preferences or desirable qualifications. The need for preferences is illustrated in a software engineering framework. A preference mechanism is then presented as an extension of a language of the Domain Relational Calculus family, and the expressive power of the resulting language is discussed. The proposed mechanisms are shown to effectively allow the use of queries for supporting software configuration management functions.

*This work is funded in part by the "Services de Programmation de la Politique Scientifique" under Contract KBAR/SOFT/4.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

1 Introduction

The database query languages have established a style of putting queries where mandatory characteristics of what is to be retrieved have to be specified in a rather rigid way. For some applications however, one is generally ready either to weaken the initially required characteristics if there is no object satisfying them, or to strengthen them if there are too many answers. For example, one might query a program database for module versions whose target is a 16 bit machine, and whose status is "released", but one might accept module versions whose status is only "tested" when there is no released version satisfying the target characteristics. Similarly, one is likely to specify further characteristics if the set of answers to the query happens to be large.

Our approach was developed as an answer to the difficulty of expressing in traditional query languages desirable characteristics of what has to be retrieved. Those difficulties were particularly experienced in the retrieval of versions of objects from the database of an experimental programming environment. The proposed solution is thus illustrated and discussed in a software engineering database framework; it should however be relevant to more traditional applications as well.

The present paper is organized as follows. In Section 2, we illustrate the role of a query language in

a programming environment. In this setting, the "intensional" aspect which normal queries already present is stressed. Preferences can then be viewed as a device for allowing the users to express more thoroughly their intents to the system. In Section 3, we present our query language and the constructs to express preferences. In Section 4, we discuss the expressive power of the language. In Section 5, we illustrate with some examples how queries with preferences can be used in software engineering applications. Finally, in Section 6, we compare our approach with other approaches proposed in the literature.

2 Denoting Objects with Queries

A standard way to explain the function of queries is to present them as a mean of retrieving from a database information that is needed for taking some decision or performing some action. A typical query fulfilling this kind of function is e.g.: "Get the status of version 2.3 of module get_data".

In engineering applications, however, another function of queries appears to be prevalent. Queries are essential because they are in fact "intensional" denotations of objects in a database, i.e. they denote objects by their properties rather than by their name. A typical query fulfilling this kind of function is e.g.: "Get the versions of module get_data whose status is released". This is to be contrasted to the above query where one is interested in properties of an object which is explicitly named.

The latter kind of query is used in our programming environment for constructing members of large program families by selecting particular versions of the component modules. The denotation of versions by their properties rather than by a mere list of identifiers presents definite advantages. These advantages, which stem from their being a system interpretable specification of the user's intents, are twofold:

- Queries can be re-evaluated when the database state changes. This provides a basic mechanism for change propagation — a key feature in programming environments.
- Queries also constitute a basic self-documentation of *why* particular versions are chosen. This is to be contrasted with the selection of particular versions by their name, which in terms of documentation can just provide a record of *which* versions went into a program, thus loosing the user's intent.

The machinery of the traditional query languages unfortunately proves in practice not to be fully adequate for the scheme sketched above. The reason is that one is most often only interested in one answer to a query, and not in a set of them. Manually designating the preferred answer would make one loose the key advantages of "intensional" queries: the reasons behind a manual selection would remain opaque to the system. Facilities allowing one to make explicit to the system the knowledge one would use for manually selecting the preferred answers are thus essential. Such knowledge is expressed as preference clauses in the query language described in the next section.

3 A Language with Preference Clauses

The query language is basically a language of the Domain Relational Calculus family (DRC, see [LAa77,ULL82]). It is an application oriented simplification of DRC in that the form of queries is geared towards the retrieval of versions of software components. These modifications are not essential for the topic of the present paper.

The database schema in which the versions are managed is quite simple: each software component is modeled by a relation whose key attribute identifies the versions, and whose non-key attributes model characteristics of the versions. For instance, an attribute DEFAULT can be used for indicating whether or not a version is a default one, the attribute STATUS for describing the development state of the version, and the attribute TARGET for indicating for which kind of target machine the version is developed.

The application oriented modifications of DRC are aimed at making more straightforward (i) the retrieval of versions of component modules by specifying constraints on their characteristics (in relational terms, retrieval of key attribute values of relations by stating conditions on the non-key attributes, without having to explicitly mention the

name of the relation linking them), and (ii) the retrieval of versions for all the component modules of a program without having to list the target variables corresponding to each component module.

Besides those modifications, a lot of syntactic sugaring has been used. The reader will however easily recognize DRC behind it; this should allow us to present the basic flavor of the language just on examples.

The query (Q1) selects the versions of the MAIN component which have the status 'coded' (indicating that the coding phase is finished). These versions are typically selected by the version tester.

```
select the versions of MAIN
 having STATUS = coded
```

Query Q1.

The query (Q2) selects the most recent version of MAIN. This is typically a selection made when developing new versions.

```
select the versions of MAIN
 having DATE = max (DATE of
            a version of MAIN)
```

Query Q2.

The query (Q3) builds the instances of the CONF program (having three components : MAIN, PROCESS-DATA, GET-DATA) which are necessary to perform the integration testing of the versions of MAIN. We assume that the versions must be tested with the default versions of the other components (developed for the same TARGET machine).

```
select the instances of CONF
 having
  the version of MAIN
   having STATUS = tested and
     same TARGET as the version
        of PROCESS-DATA and
     same TARGET as the version
        of GET-DATA;
  the version of PROCESS-DATA
   having DEFAULT = true;
  the version of GET-DATA
   having DEFAULT = true
```

Query Q3.

3.1 Simple Preference Clauses

Let us consider the versions of interest for the testers and let us assume that the testers should first deal with the versions developed for the 16 bit target machines. The query (Q1) should thus be refined with a qualification expressing that if there are versions which are 'coded' and developed for a 16 bit machine then these ones must be preferred to those whose status is 'coded' but not developed for such a machine.

With the help of a "prefer" clause appended to (Q1), this is expressed as follows.

```
select the versions of MAIN
 having STATUS = coded
from which prefer those
 having TARGET = 16
```

Query Q4.

The qualification in a preference clause is similar to the qualification of a standard query. The semantics of a preference clause can operationally be defined as follows (a more formal definition is given in Section 4.). The query without the preference clause is evaluated. The preference clause is then applied on the answer. It either turns the answer into an empty set in which case the preference clause is void, i.e. everything happens as if it were absent, or it hopefully reduces the cardinality of the answer.

The answer to query (Q4) above will thus be the coded versions developed for a 16 bit machine if there are such versions, otherwise it will be the coded versions developed for machine with a bit size different from 16.

3.2 Compound Preference Clauses

When several preferences are specified, some preferences are more important than others or they are equally important. In this section we present the constructs corresponding to these two possibilities.

3.2.1 Nested Preferences

Let us assume one wants to select the instances of CONF built with versions developed for the same type of machine, and prefers those built with tested versions of MAIN and PROCESS-DATA. If getting a tested version of MAIN is more important than getting a tested version of PROCESS-DATA, the query is

```
select the instances of CONF
 having
  the version of MAIN
   having
    same TARGET as the version
       of PROCESS-DATA and
    same TARGET as the version
       of GET-DATA
from which prefer those
 having
  the version of MAIN
   having STATUS = tested
from which prefer those
 having
  the version of PROCESS-DATA
   having STATUS = tested
```

Query Q5.

To express relative importance among preferences, we thus repeat the "from which" construct; the preferences following the "from which" keyword being considered less important than the preceding ones. This kind of multiple preference clauses can be operationally viewed as filters that are applied in the order they appear to what the preceding part of the query returns. Those that would reduce the answer to the empty set are ignored. For a more formal definition, see Section 4. In any case, the answer of (Q5), will be instances of the CONF program built with the versions developed for the same type of machine. If some of these instances have tested versions of MAIN and PROCESS-DATA the answer is restricted to these instances. If no such instances exist but some instances have a tested version of MAIN, these instances constitute the answer. If no such instances exist and if some instances have a tested version of PROCESS-DATA, the answer is restricted to these instances, otherwise the answer is not restricted.

For ordered domains, the expression of preferences on the same attribute as in

```
select the versions of MAIN
 having AUTHOR = Pierre
from which prefer those
 having STATUS = integrated
from which prefer those
 having STATUS = tested
from which prefer those
 having STATUS = coded
```

Query Q6.

can be expressed as (Q7) below (assuming that integrated > tested > coded)

```
select the versions of MAIN
 having AUTHOR = Pierre
from which prefer those
 having a maximum STATUS
```

Query Q7.

3.2.2 Equally Important Preferences

Let us now consider a user of CONF who wants to select the instances built with versions developed for the same type of machine and who prefers those built with tested versions of MAIN and PROCESS-DATA. If having a tested version of PROCESS-DATA is as important as having a tested version of MAIN, the query is

```
select the instances of CONF
 having
 the version of MAIN
  having
    same TARGET as the version
       of PROCESS-DATA and
    same TARGET as the version
       of GET-DATA
from which
 prefer those
  having
   the version of MAIN
    having STATUS = tested
 prefer those
  having
   the version of PROCESS-DATA
    having STATUS = tested
```

Query Q8.

Equally important preferences are thus expressed by repeating the "prefer" construct. The answer to this kind of queries will be those satisfying a maximum number of preferences. In any case, the answer of (Q8), will be among the instances of the CONF program built with the versions developed for the same type of machine. If some of these instances have tested versions of MAIN and PROCESS-DATA the answer is restricted to these instances. If no such instances exist but some instances have a tested version of MAIN or some instances have a tested version of PROCESS-DATA, these instances constitute the answer, otherwise the answer is not restricted.

3.3 Second Order Constructs

To deal with very large programs with many components, second order constructs have been introduced for avoiding the tedious repetition of the same qualification for the different modules. For example, if the instances of the CONF program with all the tested versions must be built, one can specify

```
select the instances of CONF
 having
  the versions of all the modules
   having STATUS = tested
```

Query Q9.

Second order preference constructs to express equally important preferences on all the versions are also available. For instance, to build the instances of the CONF program with all the versions developed for a 16 bit machine while preferring those with tested versions, one can ask:

```
select the instances of CONF
 having
  the versions of all the modules
   having TARGET = 16
from which prefer those
 having
  the versions of a maximum number
        of modules
   having STATUS = tested
```

Query Q10.

4 Expressive Power of the Language

In this section we describe the correspondence between the different forms of preference clauses in our language and Domain Relational Calculus (DRC) expressions. For the sake of clarity, we introduce an intermediate step, a DRC extended with preference clauses. Another advantage of this intermediate step is to show how the notion of preference presented in this paper can be integrated in DRC. A query such as

```
select the versions of X
 having Q
from which prefer those
 having P1
```

Query Q11.

or

```
{x | Q(x)
     from which prefer P1(x)}
```

Query Q11'.

is equivalent to

$$\{x \mid Q(x) \wedge [\exists y\, Q(y) \wedge P1(y) \Rightarrow P1(x)]\}$$

Query Q12.

This kind of query is safe (see [ULL82]) since it can be expressed in relational algebra

$$(Q \times (P1 - R))[1] \cup ((Q \cap P1) \times R)[1]$$

where R is defined as

$$((Q \cap P1) \times P1)[2]$$

A query with nested preferences such as

```
select the versions of X
 having Q
from which
 prefer those having P1
from which
 prefer those having P2
```

Query Q13.

or

```
{x | Q(x)
     from which prefer P1(x)
     from which prefer P2(x)}
```

Query Q13'.

becomes in DRC

$$\begin{aligned}\{x \mid\ & Q(x) \\ \wedge\ & [\exists y\, Q(y) \wedge P1(y) \wedge P2(y) \\ & \Rightarrow P1(x) \wedge P2(x)] \\ \wedge\ & [\neg\exists y\, Q(y) \wedge P1(y) \wedge P2(y) \\ & \wedge \\ & \exists y\, Q(y) \wedge P1(y) \\ & \Rightarrow P1(x)] \\ \wedge\ & [\neg\exists y\, Q(y) \wedge P1(y) \wedge P2(y) \\ & \wedge \\ & \neg\exists y\, Q(y) \wedge P1(y) \\ & \wedge \\ & \exists y\, Q(y) \wedge P2(y) \\ & \Rightarrow P2(x)]\}\end{aligned}$$

Query Q14.

And a query with equally important preferences such as

```
select the versions of X
 having Q
from which
 prefer those having P1
 prefer those having P2
```

Query Q15.

or

```
{x | Q(x)
     from which
      prefer P1(x)
      prefer P2(x)}
```

Query Q15'.

becomes in DRC

$$\{x \mid Q(x) \wedge [\exists y\, Q(y) \wedge P1(y) \wedge P2(y) \Rightarrow P1(x) \wedge P2(x)] \wedge [\neg\exists y\, Q(y) \wedge P1(y) \wedge P2(y) \wedge \exists y\, Q(y) \wedge (P1(y) \vee P2(y)) \Rightarrow P1(x) \vee P2(x)]\}$$

Query Q16.

It is thus possible to express preferences in any complete language [COD72]. Preferences clauses with a maximum or a minimum can similarly be expressed in DRC.

```
select the versions of X
 having Q
from which
 prefer those having a maximum T
```

Query Q17.

or

```
{x | Q(x)
     from which prefer those having
       a maximum {t | P(x,t)}}
```

Query Q17'.

Note that P in (Q17') is the predicate corresponding to the relation linking X to T in (Q17); P is implicit in (Q17). This query can be expressed in DRC as (Q18).

$$\{x \mid Q(x) \wedge [(\exists y\, \exists t\, Q(y) \wedge P(y,t)) \Rightarrow P(x,t1) \wedge t1 = \mathbf{max}\{t \mid \exists y\, P(y,t) \wedge Q(y)\}]\}$$

Query Q18.

The closed subformula $(\exists y\ \exists t\ Q(y) \wedge P(y,t))$ is necessary since there could be no t such that $(Q(x) \wedge P(x,t))$. Note that parts of the query are repeated in the scope of the maximum (here Q(x) is repeated). Furthermore it could be shown that when these preferences are nested, the equivalent DRC expression contains nested maximums.

As illustrated by (Q12), (Q14), (Q16) and (Q18), the relational calculus formula of a preference clause contains a closed subformula. A closed subformula is potentially dangerous: since the subformula is closed , there is no free variable that can be used to connect it with the other elements of the query and with the target variable. In languages allowing closed subformulas, there is thus no syntactic safeguards preventing one from stating completely unrelated propositions. Note however that preference clauses are not "disconnected", the connection is here made by repeating part of the closed subformulas as open subformulas.

To avoid "disconnected" queries, the syntax of languages such as ILL [LAb77] has been designed in such a way that closed subformulas cannot be specified. Since in its basic form, the language described here can be viewed as a variant of [LAb77] , it was impossible to express preferences. The introduction of the preference construct solves this problem, with the advantage of restricting the queries containing closed subformulas to "non-disconnected" ones.

Our approach allows one to easily express multiple preferences while the corresponding relational calculus formulas become more and more cumbersome as the number of preference clauses increases (compare our queries with the equivalent DRC queries). In fact, it could be shown that the number of conjunctions in the DRC expression is

a combinatorial function of the number of preferences. The complexity of such formulas might explain why traditional query languages have established a style of putting queries with mandatory qualifications rather than desirable qualifications.

5 Examples

In this section we illustrate the role of queries with preferences in a software development process.

As a first example, we consider queries defining on what objects a development task can be performed (i.e. the view of the database specific to this task). Considering the integration testing task, the instance which must be constructed to test a version of MAIN can be characterized by the following query

```
select the instances of CONF
 having
  the version of MAIN
   having STATUS = coded and
     same TARGET as the version
        of PROCESS-DATA and
     same TARGET as the version
        of GET-DATA;
   the version of PROCESS-DATA
    having DEFAULT = true;
   the version of GET-DATA
    having DEFAULT = true
```

Query Q19.

There can however be more than one version of MAIN having STATUS coded, corresponding for instance to the successive enhancements made by the developer. The method or strategy consisting in integrating only the last version of MAIN (i.e. containing the last improvements) satisfying the same characteristics as in (Q19) can be supported by the following query

```
select the instances of CONF
 having
  the version of MAIN
   having STATUS = coded and
     same TARGET as the version
        of PROCESS-DATA and
     same TARGET as the version
        of GET-DATA;
   the version of PROCESS-DATA
    having DEFAULT = true;
   the version of GET-DATA
     having DEFAULT = true
from which prefer those
 having
  the version of MAIN
    having a maximum DATE
```

Query Q20.

The method or strategy consisting in integrating successively all the versions can be supported too: "a maximum" must then be changed in "a minimum".

We can refine those queries and specify that the versions developed to fix bugs (the bugs being detected during the unit testing or during the integration testing) must be integrated before the others. We then have the following query

```
select the instances of CONF
 having
  the version of MAIN
   having STATUS = coded and
     same TARGET as the version
        of PROCESS-DATA and
     same TARGET as the version
        of GET-DATA;
  the version of PROCESS-DATA
   having DEFAULT = true;
  the version of GET-DATA
   having DEFAULT = true
from which
 prefer those
  having
   the version of MAIN
    having UNIT-BUGFIX = true
 prefer those
  having
   the version of MAIN
    having INTGR-BUGFIX = true
from which
  prefer those
   having
    the version of PROCESS-DATA
     having a maximum DATE
```

Query Q21.

These queries can be considered as basic specifications of the task management facilities provided by software engineering environments. They specify on what objects a task must be performed and the environment could for instance warn the user

when there is a new object on which the task must be performed.

As a second example, we consider queries specifying the instances of CONF to be used as components of a larger system. One can decide that only the instances built with integrated versions developed for the same kind of machine can be used in this context and this defines the mandatory qualification of the query. One can then choose a "cautious style" (see [LEB83]) and select those built with the default versions. The query is then

```
select the instances of CONF
 having
  the versions of all modules
   having DEFAULT = true and
    STATUS = integrated;
  the versions of all modules
   having same TARGET
```

Query Q22.

Or one can chose a "dynamic style" à la Make [FEL79], where the versions containing all the last improvements are used as soon as they become available. The query is then

```
select the instances of CONF
 having
  the versions of all modules
   having STATUS = integrated;
  the versions of all modules
   having same TARGET
from which prefer those
 having
  the version of MAIN
   having a maximum DATE
from which prefer those
 having
  the version of PROCESS-DATA
   having a maximum DATE
from which prefer those
 having
  the version of GET-DATA
   having a maximum DATE
```

Query Q23.

Note that the order of preferences in (Q23) indicates what changes must be taken into account when it is not possible to take them all into account at the same time. For example in (Q23), if the most recent integrated version of MAIN is not developed for the same kind of machine as the most recent integrated version of PROCESS-DATA (i.e. the last improvement of MAIN is not compatible with the last improvement of PROCESS-DATA), then the instance built with this version of MAIN and another version of PROCESS-DATA, will be returned since the preference on the DATE attribute of the version of MAIN is specified as more important than the preference on the DATE attribute of the version of PROCESS-DATA.

6 Comparison with other Approaches

Different approaches [CHA76,MOT86] have been proposed to express and handle desirable qualifications (preferences in [CHA76] , goal queries in [MOT86]). Unlike ours, these approaches assume that numerical information about values of the DB domains is available (membership functions in [CHA76] and "distances" between values in [MOT86]).

The approach of [MOT86] is interesting because the distance between two values is in many cases a relevant information which is in the DB either implicitly (e.g. absolute value of the difference of numerical values) or explicitly (e.g. a relation indicating the distance in miles between two location names). Basic desirable qualifications are then queries involving a distance to a target value which must be minimized. This is a quite natural approach as long as the distance is predefined. For instance it is quite natural to select the versions whose price is as close as possible to a target price.

This technique however requires that the user defines a distance between values when there is no natural distance on their domain. For instance to express preferences among authors, the user must define a distance between authors such that author A will be closer to the target author than author B if A is preferred to B. The definition of this distance is the indirect way provided to define an order, the values of the distances themselves being quite meaningless. Furthermore a new distance has to be defined each time the user's order of preferences changes.

Queries containing more than one distance in the qualification are then used to express compound desirable qualifications. They are handled by minimizing the (possibly weighted) sum of the distances. In this process the values of the distances

become critical and since some of these distances are rather artificial, weights (as well as other parameters such as scaling factors) are provided to let the user control this process and give more importance to some distances. These weights not only hide the real user intents but also are an indirect way to express them. In some case they must be "fine tuned" recalling the process of emulating desirable qualifications with mandatory qualifications.

The approach suggested in [CHA76] has the same drawbacks since defining the values of the memberships functions of some basic fuzzy predicates is as difficult as defining distances on some domains. These values are nevertheless very important since the evaluation of membership functions of compound fuzzy formulas is based on them.

Our approach avoids having to resort to metrics for supporting desirable qualifications. It should however be noticed that it is not incompatible with these other approaches. When distance operators are available, it is still possible to use them in mandatory qualifications as well as in preferences. As a matter of fact, the notion of priority goals in [MOT86] can be seen as a special case of nested (minimum) preferences (e.g. the preference qualifications may only involve distance operators).

7 Concluding Remarks

The notion of preference is motivated in this paper in the context of software engineering applications, and more precisely for configuration management functions. For this domain we contend that the reasons why some solutions are selected must be governed by explicit rules which directly capture the knowledge the developers would use for doing the same job manually. For this kind of applications, preferences are definitely to be preferred to a mechanism based on the minimization of distances.

The preference rules are part of queries rather than part of the database. In the context of software engineering applications, those queries are however not mere throw-away queries. They are used for determining the views of a database of module versions that are required for various tasks during a software development process.

We believe that the concept of preference is interesting in itself and should be useful in other application domains, especially when the stress is similarly put on the need for explicit preference rules. The proposed preference constructs can in principle be integrated in most existing query languages.

Since the transformation of preference clauses into DRC results in a combinatorial growth of the query, it seems worthwhile trying to directly support the preference construct in the query evaluation machinery. This is indeed what we have done in a prototype Prolog implementation of our language, which follows the operational semantics given in Section 3. Anyway, the optimization of queries with preferences has not been thoroughly studied and still largely constitutes an open problem.

References

[LAa77] M. Lacroix, A. Pirotte, "Domain-oriented Relational Languages", Proceedings of the 3rd International Conference on Very Large Data Bases, Tokyo, Japan, October 1977, 370-378.

[ULL82] J. Ullman, "Principle of Database Systems", Computer Science Press, 1982.

[COD72] E. F. Codd, "Relational Completeness of Data Sublanguages", Data base systems, Courant computer science symposium 6, Rustin (ed.), Prentice Hall, Englewood Cliffs, NJ, 1972, 65-98.

[LAb77] M. Lacroix, A. Pirotte, "An English Structured Query Language for Relational Databases", Proceedings Information Processing TC-2 Working Conf. on Modeling in Database Management Systems, Nice, January 1977, Nijssen (ed.), North-Holland, New York, NY, 1977.

[LEB83] D.B Leblang, R. P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment", ACM Software Engineering Notes 9, 3, May 1983, 104-112.

[FEL79] S. I. Feldman, "Make — A Program for Maintaining Computer Programs", Software — Practice and Experience 9, 4, April 1979.

[CHA76] C. L. Chang, "Deduce — A deductive query language for relational data base", Pattern Recognition and Artificial Intelligence, C. H. Chen (ed.), Academic Press, Inc, New York, 1976, 108-112.

[MOT86] A. Motro, "Supporting Goal Queries in Relational Databases", Proceedings of the First International Conference on Expert Database Systems, Charleston, South Carolina, 1-4 April 1986, 85-96.