

# A Universal Relation Database Interface for Knowledge Based Systems\*

Heikki Hyötyniemi  
Helsinki University of Technology  
Control Engineering Laboratory  
Otakaari 5 A, Espoo, Finland

Aarno Lehtola  
Technical Research Centre of Finland  
Laboratory for Information Processing  
Lehtisaarentie 2 A, Helsinki, Finland

## Abstract

A universal relation query formalism (UQL) is presented and the transformation process from UQL into a standard relational database query language (SQL) is discussed. UQL has been designed to support knowledge based applications that need to access databases dynamically. UQL hides details of the underlying relational database and offers means for achieving independence of the data representation. On the UQL level the target database is seen as a universal relation. Preserving the semantical meaning in the transformation from UQL to SQL is vital, no matter what the actual implementation of SQL is like. Finding a general mapping between these formalisms is the key topic in the paper. UQL query processing system has been tested as a part of a portable natural language database interface.

## 1 Introduction

The integration of databases and knowledge based systems is a field of growing interest. The view of monolithic, self-contained knowledge based systems is becoming old-fashioned and there is need to couple intelligent systems with conventional software and raw data. This tendency can be seen in the expanding research work within, for example, *expert database systems* [8].

During the last years, database research has mainly been focused on different kinds of special database structures—semantical, deductive, or object-oriented databases, for example. However, large amounts of data have already been stored in traditional databases. In this paper we present a tool that can be used in advanced applications for dynamic access of information from currently available relational databases. It supports generic queries that are not constrained to any predetermined classes. Data retrieval is transparent, no matter what the actual implementation of the DBMS is like.

\*This research was funded by Sitra Foundation and carried out in Kielikone project.

This tool constitutes a link between the abstract procedures and raw data, hiding the real logical and physical organization of the database. The application client generates queries on a conceptual level, and our adapter system converts these expressions into database queries. The conceptual level queries are expressed in a formalism called UQL or 'Universal relation Query Language'. The transformation from the intermediate query level to database queries is flexible and easily modifiable, making the system portable. The UQL tools have been tested in a natural language database interface [4].

## 2 UQL query language

UQL hides the physical and logical database structure. In a UQL query only the relevant domain attributes and logical constraints are needed. The syntax of UQL is represented in the Appendix and some examples are shown in this section. The following simple UQL query could be paraphrased as "Output the turnovers of Nokia after the year 1985!":

```
(OUT company_name turnover year
(> year 1985)
(= company_name "Nokia"))
```

Prefix-form list structures are the main elements of the query. There are two basic subexpression types, *values* and *constraints*. Values may be plain numbers or strings, references to virtual database attributes or macros, or functions of various other values, arithmetical or statistical. Constraints include comparisons of two values, existence conditions, and logical combinations of other constraints. UQL structures are recursive and they can be arbitrarily deeply nested.

The UQL query consists of one or more *blocks*. These blocks are Lisp-like list structures beginning with the keyword 'OUT' or 'FROM-ALL'. Each block is internally homogeneous, or all constraints within a block have an equal effect on all the attributes, no matter where the constraints are located or how they are ordered. Superfluous parentheses are neglected. The ordering of values and constraints within a block is arbitrary to make the structure more declarative.

Blocks are the relational basic structures in UQL. The universal relation is generated between the database columns that are referred to in that block. A block is an independent substructure within a UQL query, and the evaluation of a block cannot prin-

cipally be affected by the surrounding UQL structures. From outside, a block is seen as a single attribute, and despite its return value, there are no other mechanisms for affecting the evaluation of the outer block.

The following conceptual query searches for the company that has the all time minimum turnover vs. personnel ratio and outputs the company name together with the corresponding year:

```
(OUT company_name year
  (= (DIV turnover personnel)
    (MIN (FROM-ALL (DIV turnover personnel))))))
```

The flow of information is also basically one-directional, from inner blocks to the outer ones. There is a *variable mechanism* for making bi-directional control flow possible. A variable declaration binds an arbitrary group of scalar values to a symbol. For example, if we are interested in the years, when the database companies had their turnovers at the maximum values, the query could be expressed as follows:

```
(OUT (A company_name) turnover year
  (= turnover
    (MAX (FROM-ALL turnover
      (= company_name A)))))
```

It has been noticed that a great deal of semantically meaningful unconstrained questions can be coded in UQL. However, the possibilities of the conceptual level query are constrained by the limitations of the object language SQL and that is why the expressive power of UQL syntax has been restricted. For instance, real iteration is not realizable in SQL or UQL. Because SQL handles tables as unordered sets, another basic type of difficult but semantically justified queries that cannot be coded in UQL is *"Give ten biggest companies."*

UQL is intended for information retrieval only, and there are no tools for database updating or manipulating available. This reflects the intended purpose of UQL as being the gateway to data banks for high-level applications having no idea of the real data structures.

### 3 Universal relation assumption

The *universal relation assumption* [7,2] offers the query system a high level of abstraction when automating the 'logical navigation' in addition to the 'physical navigation' among the database tables. The basic idea is to show the whole database domain as a huge single relation, where every piece of information can easily be selected. Universal relation is just an abstraction—as a real database structure it would not be feasible because of the massive amount of redundant, denormalized data. This universality assumption approach offers a reasonable data model for advanced applications.

Although universal relation approach offers data independence, there has been some criticism against the universal relation assumption [9]. The basic point is that no static universal relation can express all possible join paths in a semantically rich database, and some 'less probable' relation instances have to be neglected. In the UQL environment the absolute universality is abandoned. The universal relation is generated in various situ-

ations dynamically and independently, taking into account only the attributes present. Resulting 'local' universal relation structure may change non-monotonically: new information or additional attributes can drastically alter the relevant join paths in the database. Having only relevant attributes accompanying, relation instances containing null values can immediately be rejected.

### 4 Description of the database

For interpreting UQL queries a description of the target database is needed. It contains logical information about the database structures: available tables and columns, mnemonic alias names, data types, scaling factors, dimensions, functional dependencies, and null values. The following expression could be used to define attribute *turnover* to UQL:

```
ATTRIBUTE turnover:
  DB-COLUMN   annual_data.turnover
  HEADER      "Turnover"
  DATATYPE    float4
  NULL-VALUE  -1.0
  MULTIPLIER  million
  DIMENSION   Fmk
  ASSOCIATE   company_name year
```

The database description also contains basic information of the *keys*. There is a key for every semantically justifiable join, and all semantically equivalent database columns are associated with the corresponding key names. These keys in the UQL system are not identical with the traditional concept of a key. For example, every key consists always of only one database attribute. This is not a limitation, because many keys can be selected simultaneously. The keys may also be overlapping, so that some columns may be associated to several keys that are active in different semantical contexts. These keys are a means of realizing structures similar to *maximal objects* [6].

Also domain specific semantic information is described to UQL. Meaningful joins between database tables cannot automatically be found without knowledge of mutual interconnections among domain attributes. There is a rule-system that is used to select the relevant key combinations in different situations. Each rule consists of a group of preconditions and operations. The preconditions are composed of conditions mainly concerning the attributes of the UQL structure in that block, and the operations are usually used to add new key candidates or attributes into the block environment. As long as there are rules that are satisfiable, corresponding operations are executed.

Join paths are generated by connecting the database tables containing common keys. All references to an attribute are identified with each other. This makes it possible to search for join paths in a modular fashion, or to study different semantic entities separately on an arbitrary level of precision. Afterwards, these separately analyzed discourse entities are automatically connected with each other. Cyclic database schemas may be tackled in UQL by declaring enough key candidates, and heuristically choosing the most relevant key combination.

The simple rule below functions with a database containing in-

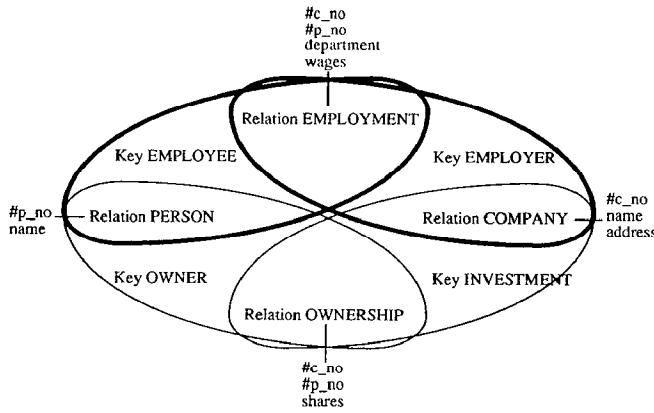


Figure 1. An example of a cyclic relation schema

formation about companies and persons that may either work for the company or own its shares (Fig. 1). The key candidates EMPLOYER and EMPLOYEE have been defined to include company and person identification codes, respectively. The rule connects relations COMPANY and PERSON using these key candidates, if there exist references to database attribute DEPARTMENT:

```

COMPANY PERSON in relations
DEPARTMENT in attributes
-->
relations <- union(relations EMPLOYMENT)
keys <- union(keys EMPLOYEE EMPLOYER)

```

## 5 Processing UQL queries

UQL presupposes that the target database query language is *relationally complete* [5]. To take full advantage of the power of UQL there should be a mechanism for expressing, for example, aggregation functions (maximum, minimum, average, etc.). The transformation process from UQL to a database query language SQL [1] has been tested. Implementations of SQL tend to have various kinds of limitations. One of the main objectives using UQL is to offer the higher level query generating system an implementation independent view of the database. That is why the transformation process uses only those features that are common to practically all SQL versions.

Basic principle in conversion is to generate one 'stand-alone' SQL query corresponding to the UQL expression. Many UQL constructs have their counterparts in SQL and can directly be syntactically transformed from a formalism to another. This kind of simple structures include numeric functions, the basic constraint structure, and some syntactic directives for output styling.

As query languages, UQL and SQL differ from each other not only because of the universal relation assumption of UQL. SQL has been designed to be computationally efficient and deterministic; the complexity of evaluation may not get incalculable when new substructures are added. The easiest way to prevent this is to limit the maximum amount of nested expressions, or to make substructures more or less context sensitive. On the other hand,

UQL is *orthogonal* in the sense that semantics of the substructures remains unchanged and unrestricted no matter how deeply nested they happen to be. To make UQL applicable to any available SQL implementation, the resulting SQL queries are made as 'flat' as possible, with no subqueries. For example, each of the independent blocks is realized as a temporary database table of its own.

In SQL, processing has to be *serializable*: there has to be an ordering between substructures, so that no operation refers to an element that has not yet been defined. This non-cyclicity is sure to result in realizable evaluation, but the expressive power is diminished. UQL allows one to write non-serializable queries. Although inner blocks are principally evaluated first, the *free variables* defined outside that block and not yet bound make this evaluation suspend. This way the UQL query processing is in a way like *unification process*: the variable definition block is gradually altered to make its variables match all of the constraints on any level.

To make the query processing realizable in a single pass, UQL creates temporary tables for the variables. The variable defining tables, or variable value sets, are constrained as much as possible, leaving only self-referential structures containing that variable unnoticed. Skipping constraints may cause too many values to remain in these variable tables, but the correct value set is a subset of all these candidates (Fig. 2). Concentrating on each of these candidate values at a time, the complete query can be evaluated and the feasible set of corresponding relation instances can be extracted—if that set is empty, that variable value candidate was not feasible in the first place.

Practically all legal UQL queries can be divided into simple standard SQL structures, however complex the UQL constructs are. The time complexity of the SQL queries may be very high, though. Blocks containing free variables are functions of all of those variables. Although superfluous variables and variable values are automatically deleted, this query processing mechanism is bound to result in exponential memory requirements. That is why it is wise to avoid variables having too many distinct values when utilizing this kind of approach.

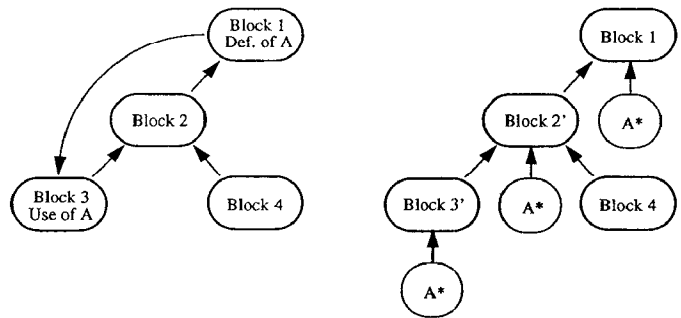


Figure 2. The solution to the non-serializability problem. The nodes in the trees correspond to individual database tables, and arrows show the dependencies. The set  $A^*$  is a superset of variable  $A$  value set

## 6 Limitations in transformation

In the transformation process, some pathological query structures exist. These structures are self-referential in some sense—for example, defining variables in a cyclical manner makes it impossible to generate the initial variable value tables in the database. This kind of shortcoming can often be avoided and should not limit the power of UQL essentially.

Aggregation functions MAX, MIN, AVG, etc. are another source of confusion. These functions extract information concerning the whole block structure, and if these functions further pose limitations on the same block, there is again a self-referential loop. In the following expression, the attribute `attr` may produce a loop:

```
(> attr (AVG attr))
```

Constraints like the one above are semantically questionable and can most often be considered as flaws of thought. A legal construct when limiting `attr` values to the greatest ones consists of an extra block:

```
(> attr (AVG (FROM-ALL attr)))
```

UQL considers blocks only as atomic relational entities. This limitation reduces computational complexity of the target queries considerably, but in some cases it may be restrictive.

## 7 Example of the transformation process

The target database consists of 60 columns in 15 database tables and contains information about Finnish stock market companies. It includes numeric annual facts as well as other public data. The semantical and logical knowledge about the relevant relations among the database structures has been declared accordingly.

```
(OUT (A company_name) turnover (B year)
(> turnover
(TIMES 1.5
(FROM-ALL turnover
(= company_name A)
(= year (MINUS B 1))))))
```

This non-trivial UQL expression contains two separate blocks, each of which affects the other. There are variables `A` and `B`, the former for company names and the other for year values. In the inner block the year value is always bound to be one less than the year value in the outer block while the company name remains the same. The query could be paraphrased as *"Search for such companies, the turnover of which has increased by 50 per cent in one year, and output the corresponding year."*

The resulting SQL query is now generated in a very general manner, not utilizing any possible means of optimization, to show the non-serializability resolution principle in practice. For example, each variable and block generates a temporary database table of its own, even if these could have been realized using subqueries:

```
CREATE TABLE Table1 AS
SELECT DISTINCT Head=basic_data.company_name
FROM basic_data,annual_data
WHERE (basic_data.c_no = annual_data.c_no
AND annual_data.turnover <> -1)

CREATE TABLE Table2 AS
SELECT DISTINCT Head=annual_data.year
FROM basic_data,annual_data
WHERE (basic_data.c_no = annual_data.c_no
AND annual_data.turnover <> -1)

CREATE TABLE Table3 AS
SELECT DISTINCT Head=annual_data.turnover,
Variable1=Table1.Head,Variable2=Table2.Head
FROM Table1,Table2,annual_data,basic_data
WHERE (basic_data.company_name = Table1.Head
AND annual_data.year = Table2.Head
AND basic_data.c_no = annual_data.c_no
AND annual_data.turnover <> -1)

SELECT Name=Table1.Head,
Turnover=annual_data.turnover,
Mult="mill",Unit="Fmk",
Year=Table2.Head
FROM Table1,Table2,Table3,basic_data,annual_data
WHERE (annual_data.turnover > 1.5 * Table3.Head
AND Table1.Head = basic_data.company_name
AND Table1.Head = Table3.Variable1
AND Table2.Head = annual_data.year
AND Table2.Head = Table3.Variable2
AND basic_data.c_no = annual_data.c_no
AND annual_data.turnover <> -1)
```

The universal relation is first simulated by connecting the `c_no` key columns of the database tables `basic_data` and `annual_data`. In addition to this, relation instances containing unknown turnover values are eliminated. Every variable definition inherits the inner structure and the universal relation definition of the corresponding block, excluding the unserializable constraints that are taken care of later. The structures `Table1` and `Table2` correspond to variables `A` and `B`, respectively, and `Table3` is generated from the independent inner block. Note, how the output values are grouped by the free variables.

In this case there will be three temporary database tables. The last structure of the SQL query corresponds to the resulting table that is output to the user. The query could be written more compactly and efficiently as a single query with a subquery, but this does not generally apply to more complex nested structures, or if the query were also left-recursive.

The specification of the transformation process is coded using a metacompiler formalism called R&CL [3]. R&CL allows one to write rule-based descriptions for defining mappings between syntactic structures of two formalisms in a declarative and domain-oriented way. High-level declarations are one means of achieving computing system independence and portability.

## 8 Discussion

To achieve database system independence and portability, only the most basic and reliable query constructs that are common to all SQL implementations are utilized. Some further optimiza-

tion would be useful when processing complex UQL queries. The transformation from UQL to SQL seems to be in any case so much faster than the final SQL query evaluation that more processing capacity could be spent during the transformation phase. All problems caused by non-serializability can be avoided if the query can be written without temporary database tables, and this is true for *most* queries in *most* SQL environments. In practice, computationally expensive queries are seldom encountered.

Another subject of future research could be making the database schema more dynamic. For example, tables that were output during previous UQL and SQL evaluation processes could be included among the stable database tables so that iterative query refinement and further result analysis could take place. One problem is that the generated result columns can be combinations of other attributes, like mathematical operations, and may have no corresponding property declarations in the database schema. This would result in a great amount of structureless data.

The principle is to generate a SQL query corresponding to a specific UQL structure once and for all. This approach is the easiest in a large database environment, where queries have to be prepared before being sent to the mainframe database. Iterative query processing in an embedded programming language environment would be another step forward, seen from the standpoint of the available expressive power. In general, iteration could easily result in an intolerable amount of processing time spent during an interactive session.

The amount of rows in the tables of the test database was relatively small. There was a plenty of numerical and alphanumeric data that made it possible to carry out complex mathematical operations and a wide scale of combinations among the database tables. This kind of database is really optimally suited for the UQL system: plenty of semantically interesting join paths and data combining, and little raw data manipulation that would demand a great deal of query optimization to make the interactive query process feasible.

## 9 Conclusions

The main objective when realizing the transformation process from UQL to SQL was semantical equivalence between the query languages as well as the universality of the transformation. Optimization of the resulting database query has not yet been regarded as crucial.

UQL has proved to be powerful for expressing a wide scale of interesting expressions and queries. The transformation from UQL to SQL has been reliable and predictable. The non-convertible UQL structures seldom have any reasonable meaning, and the class of the pathological queries can be recognized automatically.

## References

- [1] Date, C. J.: *A Guide to the SQL Standard*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.
- [2] Leymann, F.: *A Survey of the Universal Relation Model*. Data & Knowledge Engineering 4, 1989, pp. 305–320.

- [3] Hyötyniemi, H., and Lehtola, A.: *A Metatool for Implementing Task-Oriented Formalisms*. Proceedings of the International Workshop on Tools for AI, 1989, pp. 182–188.
- [4] Jäppinen, H., Honkela, T., Lehtola, A., and Valkonen, K.: *Hierarchical Multilevel Processing Model for Natural Language Database Interface*. Proceedings of the Fourth Conference on Artificial Intelligence Applications, San Diego, 1988, pp. 332–337.
- [5] Maier, D.: *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983.
- [6] Maier, D., and Ullman, J. D.: *Maximal Objects and the Semantics of Universal Relation Databases*. ACM Transactions on Database Systems, Vol. 8, No. 1, 1983, pp. 1–14.
- [7] Maier, D., Ullman, J. D., and Vardi, M. Y.: *On the Foundations of the Universal Relation Model*. ACM Transactions on Database Systems, Vol. 9, No. 2, 1984, pp. 283–308.
- [8] Smith, J. M.: *Expert Database Systems: A Database Perspective*. Kerschberg, L. (ed.): Expert Database Systems. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1986.
- [9] Ullman, J. D.: *The U.R. Strikes Back*. Proceedings of the 1st ACM Symposium on Principles of Database Systems, 1982, pp. 10–22.

## APPENDIX: The syntax of UQL

The syntax of UQL is represented in a Backus-Naur-form (BNF) rule formalism. Non-terminal symbols are separated by '<' and '>' signs. Some non-terminals having an evident meaning (*name*, *number*, etc.) have not been declared explicitly. Terminal symbols, excluding parentheses, are quoted. Not all features of UQL have been mentioned—these include commands for altering the appearance of the output (headers, ordering instructions, etc.) and commands for scaling numeric expressions.

|              |     |                                         |
|--------------|-----|-----------------------------------------|
| <UQL>        | ::= | (<QUERY>*)                              |
|              |     | <QUERY>                                 |
| <QUERY>      | ::= | ('OUT' <VALUE> <EXPR>*)                 |
| <VALUE>      | ::= | <number>                                |
|              |     | <string>                                |
|              |     | <attribute-name>                        |
|              |     | <variable-name>                         |
|              |     | <DEFINITION>                            |
|              |     | <MACRO>                                 |
|              |     | <BLOCK>                                 |
|              |     | (<AGGREG> <VALUE>)                      |
|              |     | (<FUNCTION> <VALUE>)                    |
|              |     | (<OPERATION> <VALUE> <VALUE>)           |
|              |     | (<VALUE> <EXPR>*)                       |
| <EXPR>       | ::= | <VALUE>                                 |
|              |     | <CONDITION>                             |
|              |     | (<EXPR>*)                               |
| <DEFINITION> | ::= | (<variable-name> <VALUE>)               |
| <MACRO>      | ::= | <macro-name>                            |
|              |     | (<macro-name> <VALUE>*)                 |
| <BLOCK>      | ::= | ('FROM-ALL' <VALUE> <EXPR>*)            |
| <CONDITION>  | ::= | (<COMPAR> <VALUE> <VALUE>)              |
|              |     | ('AND' <CONDITION>*)                    |
|              |     | ('OR' <CONDITION>*)                     |
|              |     | ('NOT' <CONDITION>)                     |
|              |     | ('EXISTS' <VALUE>)                      |
| <COMPAR>     | ::= | '='   '<'   '>'   '>='   '<='           |
| <AGGREG>     | ::= | 'MAX'   'MIN'   'AVG'   'SUM'   'COUNT' |
| <FUNCTION>   | ::= | 'ABS'   'EXP'   'LOG'   'SQRT'          |
| <OPERATION>  | ::= | 'PLUS'   'MINUS'   'TIMES'   'DIV'      |