ConQuer: A Conceptual Query Language

A. C. Bloesch and T. A. Halpin

Bloesch, A.C. & Halpin, T.A. 1996, 'ConQuer: a conceptual query language', Proc. ER'96: 15th Int. Conf. on conceptual modeling*, Springer LNCS, no. 1157, pp. 121-33. Reprinted by permission.*

Relational query languages such as SQL and QBE are less than ideal for end user queries since they require users to work explicitly with structures at the relational level, rather than at the conceptual level where they naturally communicate. ConQuer is a new conceptual query language that allows users to formulate queries naturally in terms of elementary relationships, and operators such as "and", "not" and "maybe", thus avoiding the need to deal explicitly with implementation details such as relational tables, null values, and outer joins. While most conceptual query languages are based on the Entity-Relationship approach, ConQuer is based on Object Role Modeling (ORM), which exposes semantic domains as conceptual object types, thus allowing queries to be formulated in terms of paths through the information space. This paper provides an overview of the ConQuer language.

Introduction

It is now widely recognized that information systems are best designed first at the conceptual level, before mapping them to an implementation target such as a relational database. A conceptual schema expresses an application model in terms of concepts familiar to end users of the application, thus facilitating communication between modeler and subject matter experts when determining the schema. Once declared, a conceptual schema can be mapped in an automatic way to a variety of DBMS structures. Although use of CASE tools for conceptual modeling and mapping is widespread, very little use is currently made of tools for querying the conceptual model directly. Instead, queries are typically formulated either at the external level using forms, or at the logical level using a language such as SQL or QBE.

Form-based queries are typically very limited in expressibility, and can rapidly become obsolete as the external interface evolves. SQL queries, and to a lesser extent QBE queries, can be more expressive, but quickly become too complex for the average end user to formulate once non-trivial queries are considered. Even queries that are trivial to express in natural language (e.g. who does not drive more than one car?) can be difficult for non technical users to express in these languages. Moreover, an SQL or QBE query often needs to be changed if the relevant part of the conceptual schema or internal schema is changed, even if the conceptual version of the query still applies. Finally, commercial query optimizers for relational languages basically ignore the further semantic optimization opportunities arising from knowledge of conceptual constraints.

Query languages for object-oriented DBMSs suffer the same problems, and languages for pre-relational systems are even lower-level.

For such reasons, considerable research has been undertaken to provide a conceptual query language that enables users to formulate queries directly on the conceptual schema itself. For example, the SUPER project [1] has a graphical conceptual query language based on ERC+ (a variant of Entity Relationship (ER) modeling) [14]. Essentially, users copy the relevant portions of a conceptual schema into the SUPER query editor. They may then add further conditions to the query in a first-order like language. The principal advantage of the SUPER query editor is that essentially the same graphical language is used to both model and query a database. Unfortunately, the query language would be hard to grasp, for naïve users, without significant training.

SUPER's query editor keeps the schema browser separate from the query editor. By contrast the Hybris project [16] integrates the query editor and schema browser. Hybris's approach reduces the user's cognitive load but at the cost of reducing the expressivity of the query language.

ERQL [11] is a conceptual query language for an EER (extended ER) conceptual modeling language. It differs from Super and Hybris's query language in that it is textual. Essentially, ERQL is an SQL-like language modified to support EER. ERQL has the advantage over SQL in that relational details are hidden from the user.

Not all conceptual query languages are based on ER. ConceptBase models a deductive object database with a semantic net like modeling language Telos [13]. CBQL is a first-order like query language where users specify the attributes they wish to know and then constrain the result set with logical constraints. Like ERQL, ConceptBase's [10] query language CBQL [17] is textual. Unlike many conceptual query languages, CBQL supports the useful notion of parameterized queries. But once again, the language would be hard to grasp for naïve users.

Object Role Modeling (ORM) is a generic term for a conceptual modeling approach which pictures the application world in terms of objects that play roles (individually or in relationships), thus avoiding the notion of attribute. It originated as a semantic modeling approach in the 1970s and has a number of closely related versions (e.g. NIAM [18], FORM [5], NORM [4] and PSM [9]). ORM facilitates detailed information modeling since it is linguistically based, is semantically rich and its notations are easily populated. An overview of ORM may be found in [6], a detailed treatment in [5] and formal discussions in [7; 8].

The use of ORM for conceptual and relational database design is becoming more popular, partly because of the spread of ORM-based CASE tools, such as Asymetrix's InfoModeler. However, as with ER, the use of ORM for conceptual queries is still in its infancy. The first ORM-based query language was RIDL [12], a hybrid language that combined both declarative and procedural aspects. Although RIDL is very powerful, its advanced features are not easy to master, and while the modeling component of RIDL was implemented in the RIDL* tool, the query component was not supported. Another ORM query language is LISA-D [9], which is based on PSM and has recently been extended to Elisa-D [15] to include temporal and evolutionary support. LISA-D is very

expressive but it is technically challenging for end users, and is currently supported only as an academic prototype.

Since InfoAssistant is a commercial product, more care has been taken with its user interface than would be normal in a research tool. As well as complying with Microsoft's user interface standards, InfoAssistant provides an intuitive interface for constructing queries that has met with positive industry reviews and user feedback. Typical queries can be constructed by just clicking on objects with the mouse. User interface deficiencies in the current version have been identified and will be corrected in the next version. For example, it is planned to make queries appear more like English sentences and provide support for unlimited undo/redo.

The rest of this paper provides an overview of ConQuer (CONceptual QUERy), a new ORM conceptual query language designed for ease of use, an early version of which has been released in the InfoAssistant product from Asymetrix (see Figure 1). The next section explains how the language is based on ORM, and illustrates how queries are formulated and mapped to SQL. The following section discusses the formal semantics. The conclusion summarizes the main contributions and outlines future research.

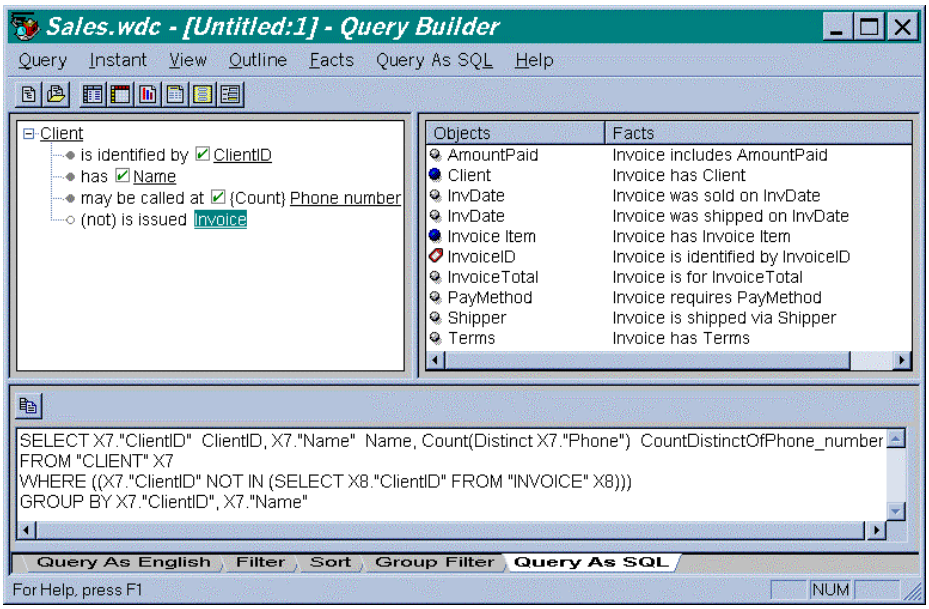

Figure 1: Screen snapshot of InfoAssistant's query editor.

ORM-based Conceptual Queries

Figure 2 is a simple ORM schema. Object types are shown as named ellipses. Entity types have solid ellipses with their simple reference schemes abbreviated in parenthesis (these references are unabbreviated in queries). For example, "Employee (nr)" abbreviates "Employee is identified by EmployeeNr".

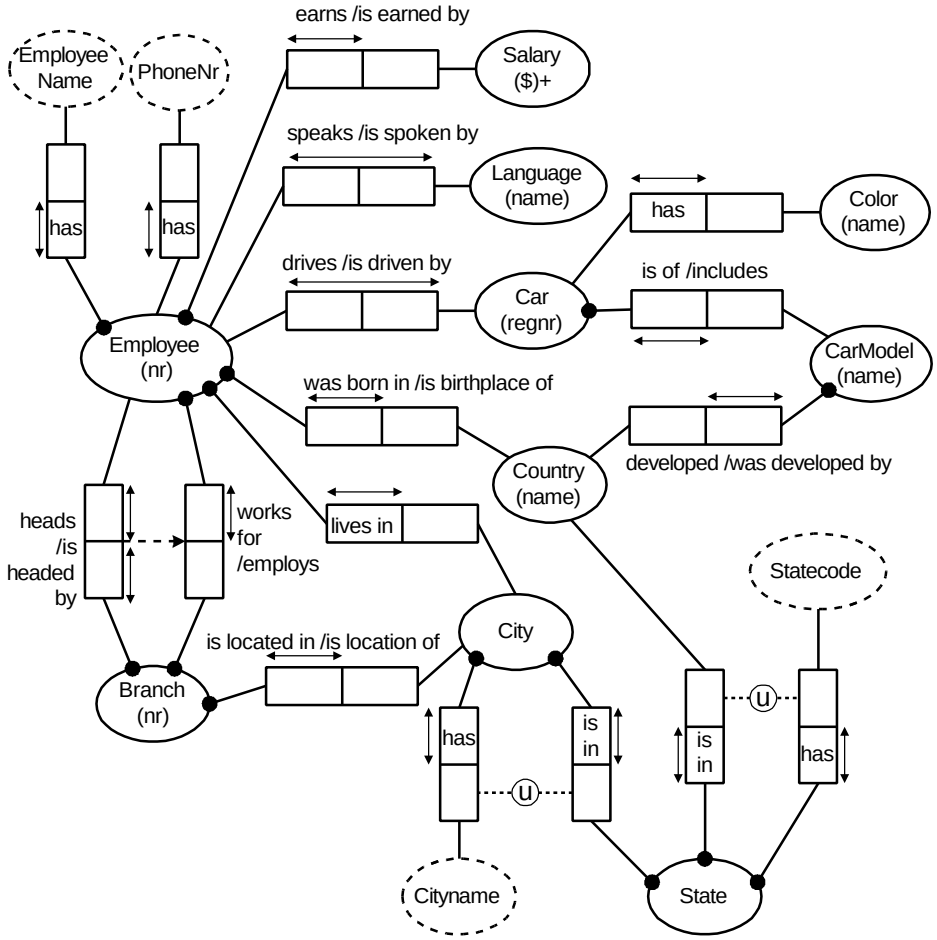

Figure 2: An ORM conceptual schema

If an entity type has a compound reference scheme, this is shown explicitly using an external uniqueness constraint (circled "u"). For example, a state is identified by combining its country with its statecode (e.g. Washington state is in the country named "United Sates of America" and has the statecode "WA", whereas Western Australia is in the country named "Australia" and has the statecode "WA"). Value types have dotted ellipses (e.g. "Statecode"). For simplicity we assume that cities may be identified by combining their name and state (apologies to inhabitants of Stockbridge Massachusetts USA).

Predicates are shown as named role sequences, where each role is depicted as a box. In the example all the predicates are binary (two roles). In the ORM version on which ConQuer is based, predicates may have any arity (number of roles) and may be written in mixfix form. A relationship type not used for reference is a fact type. An *n*-ary relationship type has *n!* readings but only *n* are needed to guarantee access from any object type. Figure 2 shows forward and inverse readings (separated by "/") for several binaries.

An arrow-tipped bar across a role sequence depicts an internal uniqueness constraint. For example, each employee earns at most one salary, but an employee may speak many languages and vice versa. A black dot connecting a role to an object type indicates that the role is mandatory (i.e. each object in the database population of that object type must play that role). The dotted arrow from the heads predicate to the works-for predicate is a pair-subset constraint (each employee who heads a branch also works for that branch). ORM has many other kinds of constraint, but these are not germane to our discussion.

Notice that no use is made of attributes. This helps with natural verbalization, simplifies the framework, and avoids arbitrary or temporary decisions about whether some feature should be modeled as an attribute. Moreover, since ORM conceptual object types are semantic domains, they act as semantic "glue" to connect the schema. This facilitates not only strong typing but also query navigation through the information space. We give an example later. When desired, attributes (e.g. birthplace) can be introduced as derived concepts, based on roles in the underlying ORM schema.

An Informal Discussion of ConQuer

ConQuer queries may be represented as outline queries, schema trees or text. Currently the InfoAssistant tool requires ConQuer queries to be entered in outline form, and automatically generates a textual verbalization. In this paper we discuss only the outline form, including some minor changes to the current version of the tool.

On opening a model for browsing, the user is presented with an object pick list. When an object type is dragged to the query pane, another pane displays the roles played by that object in the model. The user drags over those relationships of interest. Highlighting an object type within one of these relationships causes its roles to be displayed, and the user may drag over those of interest, and so on. In this way, a user may quickly declare a query path through the information space, without any prior knowledge of the underlying data structures.

Items to be displayed are indicated with a tick "☑": these ticks may be toggled on/off as desired. The query path may be restricted in various ways by use of operators and conditions. As a simple example, consider the query: Who lives in the city in which branch 10 is located? This may be set out as the following ConQuer outline:

```
Q1      ☑Employee
        +-- lives in City
                    +-- is location of Branch 10
```

This implicit form of the query may be expanded to reveal the reference schemes (e.g. EmployeeNr, BranchNr). Its verbalized form is: "List the employeenr of each employee who lives in the city that is location of a branch that is identified by branch nr 10". Notice how City is used as a join object type for this query. If attributes were used instead, one would typically have to formulate this is a more cumbersome way. For example, if composite attributes are allowed we might use: List Employee. employeenr where Employee.city = Branch.city and Branch.branchnr = 10. If not, we might resort to: List Employee.employeenr where Employee.cityname = Branch.cityname and Employee.statecode = Branch.statecode and Employee.country = Branch.country and Branch.branchnr = 10.

Avoidance of attributes also helps to lengthen the usable lifetime of a conceptual query. For example, suppose that after storing the previous query, we change the schema to allow an employee to live in more than one city (e.g. a contractor might live in two cities). The uniqueness constraint on Employee lives in City is now weakened, so that this fact type is now many:many. With most versions of ER, this would mean the fact can no longer be modeled as an attribute of Employee.

Moreover, suppose that we now decide to record the population of cities. In ER or OO this would require that City be remodeled as an entity type instead of as an attribute. Hence an ER or OO based query would need to be reformulated. With ORM based queries however, the original query can still be used, since changing a constraint or adding a new fact type has no impact on it. Of course, the SQL generated by the ORM query may well differ with the new schema, but the meaning of the query is unchanged.

The previous query formed a linear path. Tree-shaped queries may be formulated by use of the logical operators "and" and "or". When two or more predicates stem from the same object type occurrence, an "and" operator is implicitly assumed. For example, consider the query: List the employee number and salary of each employee who has a salary above $90000 and either speaks more than one language or drives a red car. This may be set out as Q2. Notice also the simple treatment of functions, which often prove difficult in SQL [2].

```
Q2      ☑Employee
            +-- has ☑Salary > 90000
            +-- either speaks count (Language) > 1
              ¦
                or-- drives Car
                            +-- has Color 'red'
```

Unless qualified by a "not" or "maybe", predicates are interpreted in the normal, positive sense. For example, the following query asks: "Who has a phone and drives a car?"

```
Q3      ☑Employee
            +-- has PhoneNr
            +-- drives Car
```

On our ORM schema it is optional for an employee to have a phone or to drive a car. To issue queries on optional roles in a relational language like SQL, we need to know where the facts are stored and to cater for null values. Neither of these needs has anything to do with conceptualizing the query. Null values and relational outer joins can prove very confusing for SQL users [3]. One of the benefits of ORM is that all its base fact types are elementary, and hence cannot have null values at all.

If we wish to exclude employees who play a given role, we add the "not" operator to that role. If we don't care whether they play a given role, we add the "maybe" operator to that role. For example, query Q4 asks "Who does not have a phone and does not drive a car?".

Q4 ☑Employee
+-- **not** has PhoneNr
+-- **not** drives Car

and query Q5 asks: List the employee number, employee name, phone (if any) and cars (if any) of each employee.

Q5 ☑Employee
+-- has ☑EmployeeName
+-- **maybe** has ☑PhoneNr
+-- **maybe** drives ☑Car

As the next section shows, the SQL code generated for the these two "maybe"s differs since the underlying uniqueness constraints are different for each predicate, so the relational columns appear in different tables. Conceptually however, the user should not have to be concerned with these issues.

ConQuer is capable of far more complex queries than cited here. However in this paper we are more concerned with a clear exposition of our basic approach and its rationale rather than with providing a complete coverage of the language.

Mapping to SQL

Using the Rmap algorithm [5], our conceptual schema maps to the relational schema shown in Figure 3 (domains are omitted). Keys are underlined, using a double underline for the primary key where more than one key exists. Optional columns are shown in square brackets (as in BNF). Subset constraints (e.g. foreign key constraints) are shown as dotted arrows.

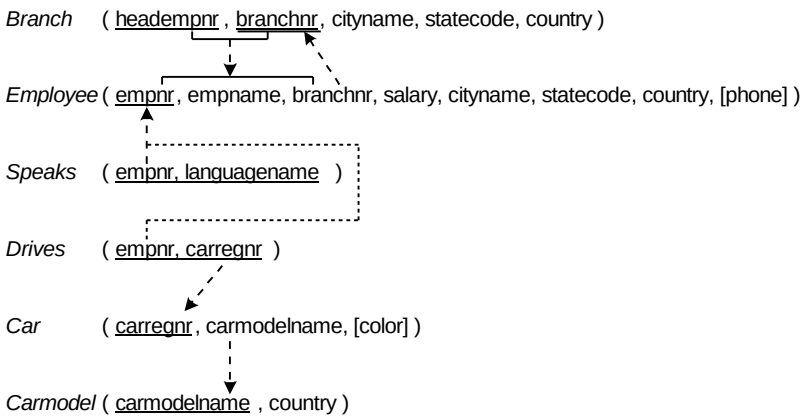

Figure 1: The relational schema mapped from the ORM conceptual schema in Figure 2.

InfoAssistant maps ConQuer queries to SQL for a variety of DBMSs, in the process performing semantic optimization where possible by accessing the constraints in the ORM schema. The SQL code for the earlier queries is shown below, S1 being the SQL for query Q1, and so on. A comparison with the ConQuer queries highlights the difference between the relational and the conceptual levels:

```
S1      select X1.empnr
        from  Employee as X1, Branch as X2
        where X1.cityname = X2.cityname and X1.statecode = X2.statecode
              and X1.country = X2.country
              and X2.branchnr = 10

S2      select X1.empnr, X1.salary
        from  Employee as X1, Drives as X2, Car as X3
        where X1.salary > 90000
              and (X1.empnr in
                      (select empnr from Speaks
                       group by empnr
                       having count(*) > 1)
              or (X1.empnr = X2.empnr and X2.carregnr = X3.carregnr
                   and color = 'red'))

S3      select X1.empnr
        from  Employee as X1, Drives as X2
        where X1.empnr = x2.empnr
              and X1.phone is not null
```

```
S4      select X1.empnr
        from  Employee as X1
        where X1.phone is null
            and X1.empnr not in
                (select empnr from Drives)

S5      select X1.empnr, X1.empname, X1.phone, X2.carregnr
        from  Employee as X1 left outer join Drives as X2 on X1.empnr = X2.empnr
```

The mapping algorithms automatically determine the appropriate join type (e.g. inner versus outer) or subquery and grouping action by exploiting the ORM constraints. The software can also reverse-engineer an existing relational schema to an ORM schema, so that at no stage does the user have to decide how relational tables are to be related in a query.

As a simple example of semantic optimization, consider the following query: "Who works for a branch that is located in a city?". In ConQuer this is formulated as Q6:

```
Q6      ☑Employee
            +-- works for Branch
                        +-- is located in City
```

Using ORM mandatory role constraints, this query is automatically transformed into the simpler query "List all employees" and hence the following SQL is generated:

```
S6      select empnr from Employee
```

While the examples discussed here are simple, the benefits of the conceptual approach should now be clear. The reader interested in more complicated queries may wish to use the supplied ORM schema to formulate some harder queries first in ConQuer and then determine the corresponding SQL. It will become apparent that it is very easy to compose ConQuer queries that lead to extremely complicated SQL.

In section 2.1 we noted that ConQuer queries are minimally impacted by schema evolution. As a trivial example, suppose that originally our schema in Figure 2 required that employees drive at most one car. Query Q3 would now generate the following SQL:

```
S3'     select X1.empnr
        from  Employee as X1
        where X1.carregnr is not null
            and X1.phone is not null
```

Suppose that the schema now evolves into that of Figure 2, so that employees may drive many cars. The ConQuery query Q3 remains the same, even though the SQL generated changes from S3' to S3.

Formal Semantics

In this section, two alternative semantics are given for ConQuer queries: firstly a semantics based on conceptual versions of relational operators and, secondly, a semantics based on bag comprehension. The treatment is necessarily brief but it should be clear how the semantics could be completely formalized.

Conceptual Join Based Semantics

There is a certain similarity between relational tables and ORM fact types that can be capitalized on to give a relational model for ConQuer queries. In ORM, fact types such as 'Person works for Department' correspond to a *set* of tuples. In a well designed ORM model all fact types will be elementary (that is they cannot be split into simpler fact types without loss of information). However, in a well designed relational model the tables will not necessarily be elementary. Each row of a relational table corresponds to one or more facts. In general, each ORM fact type will correspond to a null free projection of certain columns in a relational table.

If we view fact types as relational tables then the *conceptual join* of two fact types in an ORM model corresponds to a relational join of the fact types (when viewed as tables). In general, a conceptual join of two fact types may or may not correspond to an actual relational join of tables in the relational schema to which the ORM schema maps. For example, if the two fact types map to the same table then an equijoin between the two fact roles will not require a relational join since, in a sense, the join has already been made. However, if two fact types map to different tables then an equijoin between the two fact roles will require a relational join. Note that if an object is compositely identified (e.g. a city may be identified by the combination of its name, state code and country) the fact roles it plays will correspond to more than one column in a table and conceptual joins involving it will correspond to relational joins over several columns.

Working from the root of a ConQuer query down, the query can be seen as a sequence of conceptual joins and conceptual operations forming a series of conceptual paths through the ORM model. Where the path passes through "and" and "or" nodes, conceptual inner equijoins are made between fact types and conceptual intersections or unions are made between the paths; where the path passes through "not" nodes the child role is conceptually subtracted from the parent role (here it is understood that the conceptual path below the "not" is evaluated first); where the path passes through "maybe" nodes conceptual left-outer equijoins are made between fact types (here also here it is understood that the conceptual path below the "maybe" is evaluated first).

Restrictions such as "Sales > 1 000 000" are placed on the population in fact roles before joins are computed. Conceptual selections (ticks) correspond to relational selections. Where an object type is compositely identified, a single conceptual selection will result in several relational selections.

There are however several complicating factors. Firstly, the population of the root object may not correspond to a single table column. If all the roles it plays are optional then the population may correspond to the union of several columns. Thus, for example,

the query "What are the names of all the countries?" (see Q7) would require a conceptual union of all the fact roles that Country plays.

Q7 Country
+- - has ☑ Country Name

Secondly, there are two plausible ways of handling conditions on object types within the scope of a "maybe"– ignore them or do not ignore them. If they are not ignored then all the queries that can be expressed ignoring them can also be expressed by just deleting the unwanted conditions. Pragmatically, the SQL generation is made much easier if they are ignored; thus, on pragmatic grounds alone, in version 1.0 of InfoAssistant these conditions are ignored.

Thirdly, bag functions are not easily expressible as relational operations. So we postpone the treatment of the semantics of bag functions until the next section.

Bag Comprehension Based Semantics

Alternatively, a ConQuer query may be interpreted as specifying the contents of a bag, via bag comprehension. For example, using "**[**", "**]**" as bag delimiters, the query "Which cars are red?" (see Q8) corresponds to the expression:

[x : Car | $\exists$ y : Color; z : ColorName (x has y $\wedge$ y has colorname z $\wedge$ z = 'red') **].**

In general, a ConQuer query corresponds to the straightforward translation of the query into first-order logic with the ticked object types quantified over by the bag comprehension operator and the non-ticked object types by an existential quantifier. Note that conceptual nulls can never occur so there is no need to use Lukasiewicz's (or any other) three valued logic.

Q8 ☑Car
+- has Color 'red'

Care must be taken in interpreting the "maybe" operator. Expressions of the form "maybe α", where α is some path expression, should be translated as:

$\alpha' \vee (\neg \alpha \wedge x_1 = \square \wedge x_2 = \square \wedge \ldots \wedge x_n = \square)$

where α' is the translation of α (with restrictions removed); x_1, x_2, …, x_n are the selected object types in α; and $\square$ is a blank in the result set (relationally a null).

A semantics for the bag (aggregate) functions of ConQuer can be given as follows. Construct a labeled bag expression corresponding to the ConQuer query where all expressions involving aggregate functions have been elided and object types with ticked aggregate function are treated as if they are themselves ticked.

For example, consider the query "What are the branches and total salary costs of branches with a total salary cost of more than $1 000 000?", which may be expressed in ConQuer as Q9:

Q9 ☑ Branch
+- employs Employee
+- earns ☑ **total**(Salary) > 1 000 000

Query Q9 would generate the following labeled bag:

let S = **[** x : Branch, y : $value | ∃ z : Employee; w : Salary (
x employs z ∧ z earns w ∧ w has $value y) **]**.

The effect of the aggregate functions can then be expressed by a bag comprehension like:

[x : Branch, u : ℜ | ⟨x⟩ ∈ S_1 ∧ u = ($\sum_{\langle x\,;\,y\rangle \in S} y$ | $x = x'$) ∧ u > 1 000 000 **]**

where ℜ is the set of reals; $S_{i, j, \ldots, k}$ is a set of tuples made up of the i'th, j'th, … , k'th entries in each tuple of the bag S; and ($\sum_{\langle x, y, \ldots, z\rangle \in S} \alpha$ | ρ) is, for each ⟨x, y, … , z⟩ in of the bag S, the sum of every α such that ρ holds.

In general, each ticked object type and ticked aggregate function will be quantified over by the bag comprehension, a conjunct will link selected object types to the corresponding elements of S, a series of conjuncts will specify the value of each aggregate function and a series of conjuncts will correspond to any conditions involving aggregate functions.

Conclusion

This paper has discussed ConQuer, a new conceptual query language based on ORM that enables end users to formulate queries in a natural way, without knowledge of how the information is stored in the underlying database. A basic version of ConQuer is supported in a commercial tool that reverse engineers existing relational schemas to an ORM schema, which can then be queried directly. The benefits of a conceptual, and more specifically an ORM-based approach, to queries were highlighted and a formal semantics provided.

Currently the language is undergoing major improvements to both the language architecture and the user interface, which will appear in a subsequent release. Moreover, the FORML language used for ORM modeling is being unified with ConQuer. While the current ConQuer architecture and mapping algorithms were developed by the authors, they would like to acknowledge the contributions of Erik Proper in formalizing an alternative version of the language and in suggesting the name "ConQuer".

References

1. Auddino, A., Amiel, E. & Bhargava, B. 1991 'Experiences with SUPER, a Database Visual Environment', *DEXA '91 Database and Expert System Applications*, pp.172-178

2. Date, C.J. 1996, 'Aggregate functions', Database Prog. & Design, vol. 9, no. 4, Miller Freeman, San Mateo CA, pp. 17-19.

3. Date, C.J. & Darwen, H. 1992, Relational Database: writings 1989-1991, Addison-Wesley, Reading MA, esp. Chs 17-20.

4. De Troyer, O. & Meersman, R. 1995, 'A logic framework for a semantics of object oriented data modeling', *OOER '95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, pp. 238-49.

5. Halpin, T.A. 1995, Conceptual Schema and Relational Database Design, 2nd edn, Prentice-Hall, Sydney, Australia.

6. Halpin, T.A. & Orlowska, M.E. 1992, 'Fact-oriented modelling for data analysis', *Journal of Inform. Systems*, vol. 2, no. 2, pp. 1-23, Blackwell Scientific, Oxford

7. Halpin, T.A. & Proper, H.A. 1995, 'Subtyping and polymorphism in Object Role Modeling', *Data and Knowledge Engineering*, vol. 15, Elsevier Science, pp. 251-81.

8. Halpin, T.A. & Proper, H. A. 1995, 'Database schema transformation and optimization', *OOER '95: Object-Oriented and Entity-Relationship Modeling*, Springer LNCS, no. 1021, pp. 191-203.

9. Hofstede, A.H.M. ter, Proper, H.A. & Weide, th.P. van der 1993, 'Formal definition of a conceptual language for the description and manipulation of information models', *Information Systems*, vol. 18, no. 7, pp. 489-523.

10. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M., Eherer, S., 1995, *ConceptBase– a Deductive Object Base for Meta Data Management*, Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases, vol. 4, no. 2, 167-192.

11. Lawley, M. & Topor R. 1994, 'A Query Language for EER Schemas', *ADC '94 Proceedings of the 5th Australian Database Conference*, Global Publications Service, pp. 292-304.

12. Meersman, R. 1982, 'The RIDL conceptual language', Research report, Int. Centre for Information Analysis Services, Control Data Belgium, Brussels, Belgium, 1982.

13. Mylopoulos, J., Borgida, A., Jarke, M. & Koubarakis, M., 1990, *Telos: a language for representing knowledge about information systems*, ACM Transactions Information Systems vol. 8, no 4.

14. Parent, C. & Spaccapietra, S. 1989, 'About Complex Entities, Complex Objects and Object-Oriented Data Models', *Information System Concepts– An In-depth Analysis*, Falkenberg, E.D. & Lindgreen, P., Eds., North Holland, pp. 347-360

15. Proper, H.A. & Weide, Th. P. van der 1995, 'Information disclosure in evolving information systems: taking a shot at a moving target', *Data and Knowledge Engineering*, vol. 15, no. 2, pp. 135-68, Elsevier Science.

16. Rosengren, P. 1994, 'Using Visual ER Query Systems in Real World Applications', *CAiSE '94: Advanced Information Systems Engineering*, Springer LNCS, no. 811, pp. 394-405.

17. Staudt, M., Nissen, H.W., Jeusfeld, M.A. 1994, *Query by Class, Rule and Concept*. Applied Intelligence, Special Issue on Knowledge Base Management, vol. 4, no. 2, pp. 133-157

18. Wintraecken, J.J.V.R. 1990, *The NIAM Information Analysis Method: Theory and Practice*, Kluwer, Deventer, The Netherlands.