Role-Based Semantics for Conceptual-Level Queries

David P. Silberberg
Ralph D. Semmel
The Johns Hopkins University
Applied Physics Laboratory
Johns Hopkins Road
Laurel, Maryland 20723-6099
{david.silberberg, ralph.semmel }@jhuapl.edu

Abstract

We are developing a system known as QUICK (for QUICK is a Universal Interface with Conceptual Knowledge) which provides simplified access to database systems. It allows users to develop applications and specify ad hoc queries without requiring them to understand the underlying schema. Users present high-level queries that specify only attributes to be selected and their constraints. In turn, QUICK infers corresponding SQL queries by using a knowledge construct called a *context,* which is derived from underlying conceptual schema. For most queries, the context provides enough information to insulate users from the underlying schema. The context does not contain sufficient knowledge to infer the corresponding SQL query from certain classes of high-level queries. Users must specify logical attributes and some of the joins already described by the schema. This paper identifies how relationship *roles* specified in the conceptual schema in conjunction with new knowledge representation constructs called *pseudo-schemas* and *super-contexts* can be exploited to generate reasonable queries on complex schemas for these classes of high-level queries. Both new constructs are automatically inferred from the original conceptual database schema.

Proceedings of the 5th KRDB Workshop
Seattle, WA, 31-May-1998

(A. Borgida, V. Chaudhri, M. Staudt, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/

1. Background

Query formulation on relational databases is a difficult task. Database programmers must understand the logical and conceptual database schemas to construct queries. The logical schema describes the structure of the database tables including their attributes, data types, keys, and indexes. The conceptual schema describes the semantic structure of a database including its entities, relationships, and generalizations (a.k.a. specializations). Database programmers that formulate queries to a database certainly know which attributes they want to retrieve and how to constrain the search. However, with conventional query languages, such as SQL, programmers that formulate multiple table queries must understand the semantic structure of the schema to properly formulate the joins and specify the tables. Programmers must also be familiar with the tables in which attributes are located, the table keys, and the associations described in the conceptual schema. If the database schema is discarded, which is often the case after a database is created, the programmer must perform significant research to reconstruct the semantics of the database. Furthermore, if the underlying data model evolves, all queries of all applications must be examined to determine if they are affected by the change. The affected queries must be updated to reflect the modified database model.

Ideally, the application programs and the underlying database environments should be decoupled. Programmers should expect that the database interface hides the underlying schema. Views often provide this capability to application programs. Unfortunately, views are static. When the underlying data model and database changes, all affected views must be updated manually. The use of views moves the onus of change from application programmers, who presumably know little about the underlying data environment, to database programmers, who are more familiar with the underlying data environment. Nevertheless, for a database with many

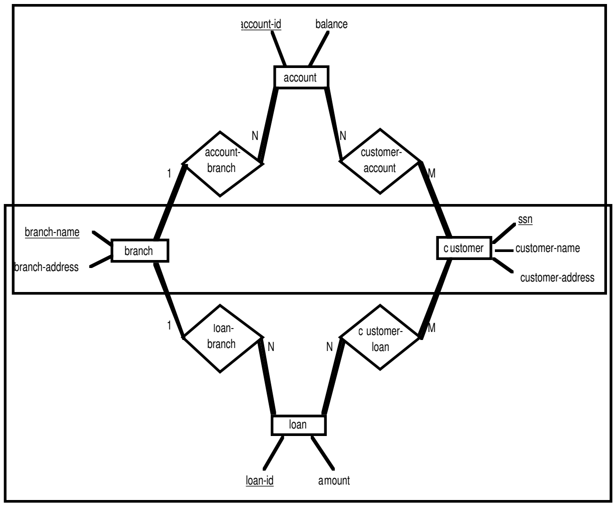

Figure 1. The Bank Schema

views, the effort to update all the appropriate views is still formidable. Furthermore, in an environment where ad hoc queries are performed, views are not generally helpful. Users must still formulate actual database queries and, thus, are still faced with the difficulty of having to understand the underlying database schemas to create the joins and specify tables.

2. Automated Query Formulation

Early approaches to solving this problem were explored in the development of the universal relation model [Kor84][Ley89][Ull89]. Unfortunately, the underlying representations used abstractions that are not typically used by database designers. Other approaches that use the more common entity-relationship (ER) models [Che76] either present a graphical user interface for selecting subgraphs which correspond to requests [Lie80][Zha83], or use universal relation concepts directly [Pah85][Wal84]. Others attempted to create automated query generators [Mar90], which resolve shortcomings of the universal relation model. Some commercial systems attempt to solve the same problem using ER constructs [Har98][Ora98]. All, however, still failed to provide the richness of expression needed for generalized queries.

Problems with the previous systems were overcome by the development of QUICK [Sem94], which is an intelligent data access system that simplifies the process of query formulation for application programmers and users that require an ad hoc query interface to a database. It extracts domain knowledge from a database conceptual schema and uses it to automatically formulate SQL queries from high-level queries. Thus, QUICK insulates the users from having to understand the semantics of the underlying database schema and from having to formulate complicated join clauses.

QUICK accepts high-level queries expressed in the Universal SQL (USQL) language [Dia95], which is a simplified version of SQL. The USQL interface requires users to specify only attributes and their constraints. Users do not have to specify table names and most join clauses described in the underlying conceptual schema.

QUICK uses an extended entity-relationship (EER) model [Bat92][Teo86] to represent conceptual database schemas. It then recognizes *contexts* of the schema, which are maximal sets of strongly related entities, attributes, relationships, and generalizations. Contexts attempt to capture the local and global semantic inter-relationships among entities, relationships and generalizations of a schema as intended by its designer. QUICK interprets a high-level USQL query with respect to the set of schema contexts to infer the corresponding SQL query likely intended by the user.

Figure 1 shows an example schema for a Bank database. Rectangles with enclosed text describe the entities of the system. This schema includes the customer, account, loan, and branch entities. Diamonds with enclosed text describe the relationships between entities. The numbers and letters associated with the relationships indicate the cardinality of the relationship. For example, the account_branch relationship indicates that there is a one-to-many (1:N) relationship between branch and account, while the customer_account relationship indicates that there is a many-to-many (M:N) relationship between customer and account. Double lines that emanate from relationships to entities indicate that the entities fully participate in the relationship. For example, all branches must be associated with accounts, and all account must be associated with a Branch. Attributes are represented by free text connected to an entity by a single line. For example, the loan entity is associated with the loan_id and amount attributes. Underlined attributes are the conceptual keys of the entities.

QUICK uses heuristics to analyze the schema and determine that there are two contexts associated with this schema. These contexts are represented by the large boxes surrounding multiple schema objects. The first context set contains the customer, customer_account, account, account_branch, and branch schema objects, while the second context set contains the customer, customer_loan, loan, loan_branch, and branch schema objects.

A high-level query presents a set of attributes that are associated with schema objects. If the set of schema objects form a subset of a context, QUICK formulates the corresponding SQL query using the join paths defined by the context. Thus, in response to the high-level USQL query:

```
SELECT customer_name,                    Query 1
       balance,
       branch_name
```

QUICK generates the corresponding SQL query:

```
SELECT                                   Query 2
      A.customer_name,
      C.balance,
      D.branch_name
FROM
      customer AS A JOIN customer_account AS B
          ON A.ssn = B.ssn
      JOIN AS account C
          ON B.account_id = C.account_id
      JOIN branch AS D
          ON C.branch_name = D.branch_name
```

The context provides the information necessary to infer the user's likely intention as formulated in high-level query. However, if attributes from both loan and account are requested, QUICK recognizes that no one context contains these attributes. Thus, QUICK infers that the database designer did not intended for this query to be formulated on the schema. If attributes are requested that belong to multiple contexts, such as customer_name and branch_name, QUICK cannot discern if either one or both contexts are desired by the user. Therefore, by default, QUICK generates the query which is the UNION of SELECT statements of both contexts. The inability for the user to indicate the intended context and for QUICK to generate the intended query is one of the limitations of the context paradigm. More limitations will be described in the next section.

Certainly, there is no guarantee that a generated query matches the user's intent. However, real-world experience indicates that QUICK does an excellent job at generating reasonable queries [Sil94][Sem95]. Thus, contexts appear to be an appropriate paradigm for automatic query generation.

3. Limitations of the Context Paradigm

The context paradigm is not always sufficient to determine the user's intention because it does not allow users to fully describe the intention or meaning of their queries. For instance, consider a query to the Bank database which requests information about customers and their branches. Two separate contexts cover the requested attributes and thus, the intention of the user is not clear. USQL does not provide the syntax for the user to clarify the request.

```
SELECT customer_name, branch_name        Query 3
```

Three possible interpretations of the query include requests to retrieve customer_name and branch_name of all accounts, of all loans, or both all accounts and loans. Currently, QUICK generates the following SQL query by default:

```
SELECT                                   Query 4
      A.customer_name,
      D.branch_name
FROM
      customer AS A JOIN customer_account AS B
          ON A.ssn = B.ssn
      JOIN AS account C
          ON B.account_id = C.account_id
      JOIN branch AS D
          ON C.branch_name = D.branch_name
UNION
SELECT
      A.customer_name,
      D.branch_name
FROM
      customer AS A JOIN customer_loan AS B
          ON A.ssn = B.ssn
      JOIN AS loan C
          ON B.loan_id = C.loan_id
      JOIN branch AS D
          ON C.branch_name = D.branch_name
```

But clearly, this may not match the user's intention.

Another class of queries that USQL does not provide the syntax and that contexts do not provide the semantics to discern the user's intention include queries over partial participation relationships. Consider the example Health Club database schema of Figure 2. The member and non_member entities are related by a one-to-many, partial participation invites relationship. The double line from the relationship to the non_member entity indicates that non_member fully participates in the relationship. A non_member can and must be invited by only one member. Thus, the non_member assumes the *role* of guest with respect to the member entity. The single line from the relationship to the member entity indicates that members partially participate in the relationship. Members may invite multiple non_members, but do not

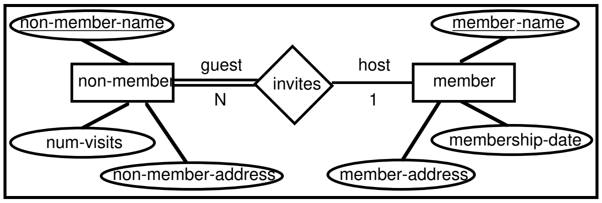

Figure 2. Health Club Database Schema

necessarily invite any. Members that invite non_members assume the role of host.

Ambiguity occurs when a query requests attributes from both entities. For instance, consider the following USQL request:

```
SELECT  member_name,                    Query 5
        non_member_name
```

Perhaps the intention of the user is to request non_members and their hosts. If so, the query is *non-member-centered* and a natural join query should be generated by QUICK.

```
SELECT                                  Query 6
      member_name,
      non_member_name
FROM
      member JOIN non_member
          ON member_name = member_name_fk
```

This query does not return those members who do not have guests. Or, perhaps, the intention of the user is to request members and their guests, if they have any. This query is a *member-centered* query and an outer join should be generated.

```
SELECT                                  Query 7
      member_name,
      non_member_name
FROM
      member LEFT OUTER JOIN non_member
          ON member_name = member_name_fk
```

This query returns all members. In either case, the context construct does not have the necessary information for an automatic query generator to discern the user's intention.

Another class of queries that do not provide enough information to formulate the appropriate SQL is queries over recursive relationships. For instance, the Employee database schema in Figure 3 describes an employee entity with a one-to-many, double-partial-participation manages relationship. Employees that manage other employees assume the role of manager. Employees that report to other employees assume the role of report. Employees need not manage any reports and may manage multiple reports. Employees report to at most one manager, but at least one employee (e.g., the CEO) does not report to any employees.

The corresponding logical schema is given below.

employee <name, manager_name_fk, salary, age>

The employee relation contains a manager_name_fk to store the employee's manager's name. This field is used to self-join the employee relation via the manages relationship.

USQL provides no mechanism to express a request for employee names and their manager names using only conceptual attributes. Furthermore, the context paradigm does not have the ability to automatically generate the join through the manages relationship. The logical attribute manager_name_fk must be added to the conceptual schema. Then, the user must explicitly specify the join between the name key and the manager_name_fk foreign key, as in the following USQL query.

```
SELECT  A.name, B.name                  Query 8
WHERE   A.name = B.manager_name_fk
```

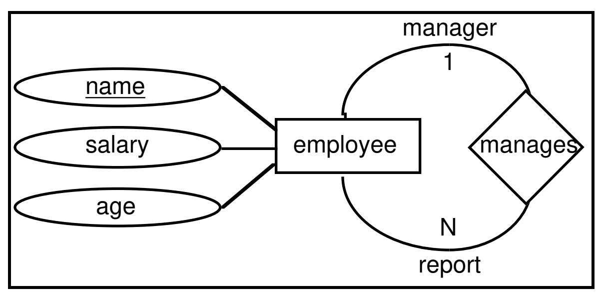

Figure 3. Employee Database Schema

The first problem with this approach is that users must step outside of the framework of the conceptual schema and draw upon the logical schema to express the join between name and manager_name_fk. Secondly, the user must explicitly express a join that is already represented in the conceptual schema. This violates the desire to insulate users from understanding the structural semantics of the schema. Thirdly, there are three interpretations of the query, all of which are correct given the information provided by the user. Since both sides of the manages relationship contain partial participation constraints, the interpretation based on the schema representation can be one of the following:

1. Select <u>all</u> employee names and their reports names, if they have any reports. This is expressed by the SQL query:

    SELECT Query 9
        A.name, B.name
    FROM
        employee AS A LEFT OUTER JOIN
        employee AS B
            ON A.name = B.manager_name_fk

2. Select <u>all</u> employee names and their manager names, if they have one. (The CEO does not have a manager.) This is expressed by the SQL query:

    SELECT Query 10
        A.name, B.name
    FROM
        employee AS A LEFT OUTER JOIN
        employee AS B
            ON A.manager_name_fk = B. name

3. Select only managers and their reports. The results only include rows that list managers in the first column and reports in the second column.

    SELECT Query 11
        A.name, B.name
    FROM
        employee AS A JOIN employee AS B
            ON A.name = B.manager_name_fk

In this case and in all the previous cases, the syntax of USQL does not allow users to express more powerful queries. Furthermore, QUICK does not have the underlying constructs to interpret more powerful queries. Users are required to understand the underlying schema to formulate complex queries. However, the schema does contain sufficient information to allow users to specify high-level queries without having to understand the schema. The next section will describe the syntactic additions to USQL and the semantic additions to QUICK that allow these classes of high-level queries to be interpreted according to the query formulator's intention.

4. Role-Based Semantics

By properly exploiting the defined roles of a conceptual schema, the limitations inherent in the context-based approach can be surmounted. To remedy the ambiguity of partial participation relationships, a new construct called a *pseudo-schema* is introduced, which uses the roles of the relationship to augment the EER representation. The pseudo-schema, which is automatically generated by QUICK, augments the original schema by introducing both *pseudo-entities* and *pseudo-relationships.* These explicitly represent the multi-fold semantic meaning of partial participation relationships. For example, in the Health Club database, QUICK recognizes that the member entity represents two separate concepts with respect to the invites relationship. The first is its role as a member and the second is its roles as a host. The pseudo-schema of Figure 4 shows the host pseudo-entity that represents this concept. The arrow with a circle from host to member indicates that the host entity is a specialization of the member entity. (If there were another specialization of the member entity, another line would emanate from the circle to the other specialized entity.

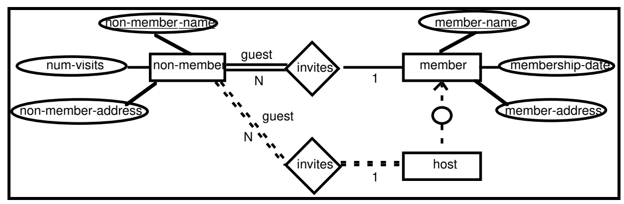

Figure 4. Health Club Pseudo-Schema

Either the circle would contain a "D" to indicate that the specialization is disjoint, or it would contain an "O" to indicate that the specialization is overlapping. In the case of a single child of an entity, the distinction between overlapping and disjoint specialization types are irrelevant.) The single-lined arrow from the circle to the member entity indicates that a member can exist independently. (If there were a double-lined arrow, members would be required to have at least one associated child entity.) Finally, the pseudo-relationship between non-member and host is a fully participatory and 1:N.

The syntax of USQL is also augmented to allow users to formulate role-based queries. Informally, the grammar of a USQL query was:

<USQL query> ::= SELECT <attribute list>
        [WHERE <where list>]
<attribute list> ::= <attribute>[, <attribute>]*
<attribute> ::= [<attribute prefix>]<attribute name>
<attribute prefix> ::= <tuple variable>.
<where list> ::= <where expr>
        [<AND-OR> <where expr>]*
<where expr> ::= <attribute> <op> <compare value>
<compare value> ::= <attribute> | <value>

The new syntax of USQL modifies the definition of <attribute prefix> as follows:

<attribute prefix>::= <tuple variable>.[<role name>.]*

Note that multiple role names can precede the attribute name to achieve effects that will be discussed later.

When presented the following USQL query:

```
SELECT  host.member_name,                 Query 12
        non_member_name
```

QUICK interprets the pseudo-schema to generate the SQL query that contains a natural join, as in Query 6. When presented with the following USQL queries:

```
SELECT  member.member_name,               Query 13
        non_member_name
SELECT  member_name,                      Query 14
        non_member_name
```

QUICK interprets the pseudo-schema to generate the SQL query that contains an outer join, as in Query 7.

4.1 Partial Participation Cyclic Schemas

A more efficient design, perhaps, of the Health Club database is presented in Figure 5. The person entity is introduced to represent the common attributes of members and non-members. The name attribute represents the information from both the non_member_name and member_name attributes. The address attribute represents the information from both the member_address and non_member_address attributes. Both the non_member and member entities are specializations of the person entity. The circumscribed "D" indicates that both entities are disjoint. The double line emanating from the circle to the person entity indicates that a person must be associated with either a non-member or a member.

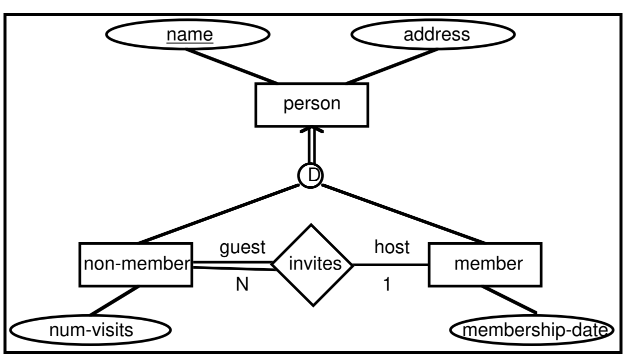

Figure 5. Alternate Health Club Schema

The underlying logical schema is represented as:

person <name, address>
member <person_member_fk, membership_date>
non_member <person_non_member_fk,
member_non_member_fk, num_visits>

The logical attribute person_member_fk is the foreign key on which the person and member tables are joined. The person_non_member_fk is the foreign key on which the person and non_member tables are joined. The member_non_member_fk is the foreign key on which the member and non_member tables are joined.

Certainly, the original USQL does not provide the syntax to express a query requesting both member and non-member names using only conceptual constructs. The best option is to modify the conceptual schema to describe all the logical foreign keys as conceptual attributes and express the USQL query as follows:

```
SELECT  A.name, B.name                    Query 15
WHERE   A.name = person_member_fk AND
        person_member_fk
            = member_non_member_fk AND
        B.name = person_non_member_fk
```

where A and B are tuple variables representing member and non-member, respectively. Unfortunately, users must specify logical attributes and the joins that are already represented in the schema. In addition, there is no mechanism to specify whether member names should assume the role of host or member. Thus, QUICK cannot discern whether to generate natural joins or outer joins in the resulting SQL query.

The corresponding pseudo-schema is shown in Figure 6. The member entity remains a partial participant in the one-to-many invites relationship while the host role is represented as a pseudo-entity. The host entity is a specialization of the member entity and it fully participates in the invites relationship. Using the pseudo-schema, the following role-based query:

```
SELECT  member.name,                      Query 16
        non-member.name
```

is interpreted by QUICK to generate the corresponding SQL query:

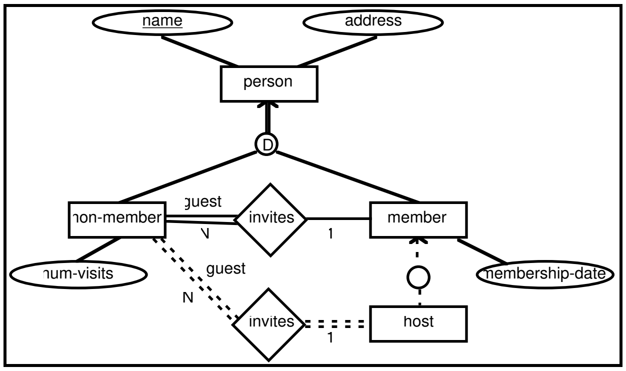

Figure 6. Alternate Health Club Pseudo-Schema

```
SELECT                                  Query 17
        P1.name, P2.name
FROM
        person AS P1 JOIN member
           ON P1.name = person_member_fk
        LEFT OUTER JOIN non_member
           ON person_member_fk
                = member_non_member_fk
        JOIN person as P2
           ON person_non_member_fk = P2.name
```

Similarly, QUICK uses the pseudo-schema to interpret the role-based USQL query:

```
SELECT  host.name,                      Query 18
        non-member.name
```

and generate the corresponding SQL query:

```
SELECT                                  Query 19
        P1.name, P2.name
FROM
        person AS P1 JOIN member
              ON P1.name = person_member_fk
        JOIN non_member
              ON person_member_fk
                    = member_non_member_fk
        JOIN person as P2
              ON person_non_member_fk = P2.name
```

In both cases, only conceptual objects are specified in the USQL query and QUICK generates the corresponding SQL query. The user did not have to be familiar with the underlying structure of the schema or logical foreign keys and the user did not have to specify joins in the high-level query.

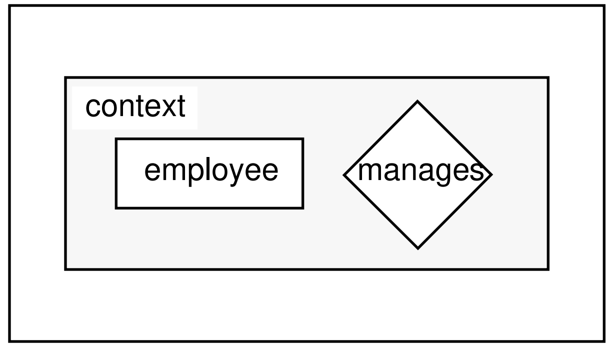

Figure 7. Simple Employee Schema Context

In general, role names of partial participation relationships are transformed into pseudo-entity names in the pseudo-schema. Entity names are then used as role names to disambiguate high-level queries. Similarly, any entity name can be used as a role name to disambiguate high-level queries. Therefore, queries to the Bank database may use account or loan as the role name to indicate the context desired. If roles are not specified in the query, the default query which spans both contexts is still generated, as in Query 4.

4.2 Self Relationships

The problems of the Employee Database in Figure 3 partially stem from the semantics of a context. A context is a maximal, *acyclic* set of entities, relationships, generalizations, and attributes of a schema. Thus, the singular context of the Employee Database contains only the employee entity and the manages relationship, as shown in Figure 7. Since contexts are acyclic, both senses of the employee entity (manager and report) cannot be represented by a single context. Therefore, contexts lack the information required to generate a self-join. Users are forced to explicitly specify joins in high-level queries as in Query 8.

To solve this problem, *super-contexts* are defined to represent the closure of all contexts and their recursive relationships as defined by their roles. The Employee database super-context, as shown in Figure 8, allows users to request an employee and his manager's manager, or an employee and his report's report, for example. QUICK interprets the super-context to generate the corresponding cyclic SQL query, as will be presented shortly.

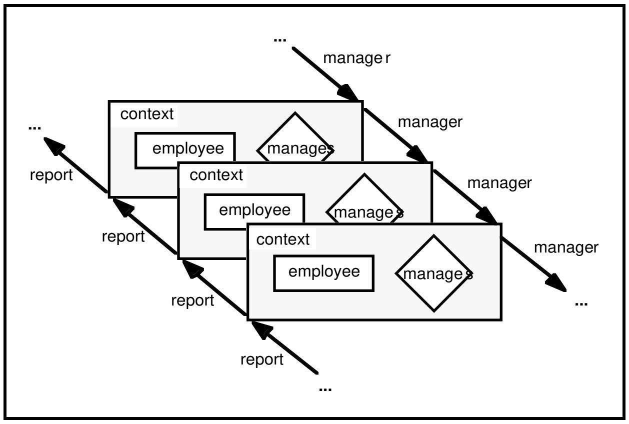

Figure 8. Employee Database Super-Context

The corresponding pseudo-schema of the Employee Database is shown in Figure 9. Since the manages relationship describes partial participation of the employee both in the role of manager and of report, report and manager pseudo-entities are represented in the pseudo-schema. The report and manager pseudo-entities are specializations of the employee entity. The double line from the overlapping generalization indicates that an employee must be either a manager or a report. The overlapping generalization indicates that an employee can be both. The full participation relationship between report and manager describes only the relationship between employees who are reports and employees who are managers. The partial participation relationship between employee and report describes the relationship between all employees and their reports, if any. Similarly, the partial participation relationship between employee and manager describes the relationship between all employees and their managers, if any.

Users may formulate role-based queries using the

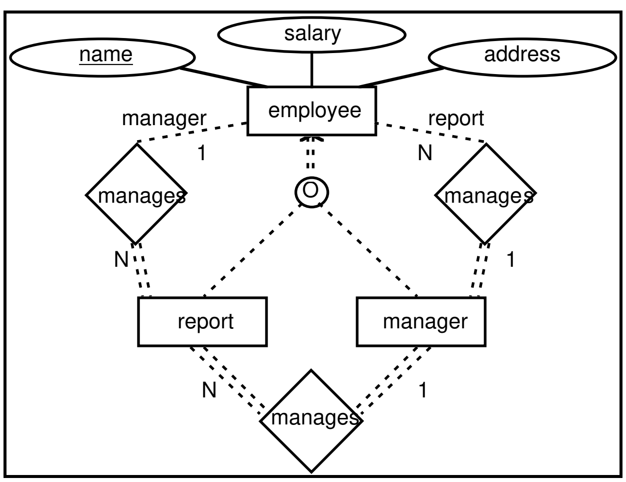

Figure 9. Employee Database Pseudo-Schema

augmented USQL without having to specify logical-level attributes or joins that are defined by the schema. QUICK interprets the pseudo-schema and super-context constructs to automatically generate corresponding SQL queries. For example, the query that requests salaries of all employees and their managers, if they have one, is expressed in USQL as:

```
SELECT salary, manager.salary                  Query 20
```

The corresponding generated SQL query is:

```
SELECT                                         Query 21
        E1.salary, E2.salary
FROM
        employee AS E1 LEFT OUTER JOIN
        employee AS E2
            ON E1.manager_name_fk = E2.name
```

The query which requests the salaries of all employees and their reports, if they have reports, is expressed by the USQL query:

```
SELECT salary, report.salary                   Query 22
```

The corresponding SQL query is:

```
SELECT                                         Query 23
        E1.salary, E2.salary
FROM
        employee AS E1 LEFT OUTER JOIN
        employee AS E2
            ON E1.name = E2. manager_name_fk
```

The query which requests the salaries of managers and reports is expressed in the USQL syntax as:

```
SELECT manager.salary, reports.salary          Query 24
```

The corresponding SQL query is:

```
SELECT                                         Query 25
        E1.salary, E2.salary
FROM
        employee AS E1 JOIN employee AS E2
            ON E1.name = E2. manager_name_fk
```

Finally, the USQL syntax permits users to express queries that leap layers of contexts via the super-context construct. Thus, the request for report salaries and manager's manager's salaries is expressed in USQL as:

```
SELECT report.salary,                          Query 26
       manager.manager.salary
```

The corresponding SQL query is:

```
SELECT                                          Query 27
        E1.salary, E3.salary
FROM
        employee AS E1 JOIN employee AS E2
            ON E1.manager_name_fk = E2.name
        JOIN employee AS E3
            ON E2.manager_name_fk = E3.name
```

5. Roles Versus Tuple Variables

On the surface, tuple variables and roles appear to be similar concepts. However, their semantic meanings are quite different and complementary. Roles clarify the meaning of the relationship between entity attributes. In addition, roles clarify the how contexts within a super-context must be joined when the schema includes self-relationships. Tuple variables, on the other hand, identify different instances of either query contexts or super-contexts within a single query. For example, a query that pair-wise compares all employees' salaries is expressed using tuple variables to describe each employee instance. The USQL query is:

```
SELECT  E1.name, E1. salary,                    Query 28
        E2.name, E2. salary
WHERE   E1.name != E2.name
```

The WHERE clause ensures that no result row contains a self-comparison.

Similarly, the complementary aspects of roles and tuple variables is demonstrated by the following request:

"List all employees and their reports, if any, whose salary is greater than Jones' manager."

The "employees and their reports, if any" and "Jones' manager" phrases identify two schema supercontexts. The full query is expressed by comparing the supercontexts by salary, either using tuple variables or embedded SELECT statements. The corresponding high-level query expressed with tuple variables and roles is:

```
SELECT  U.name, U.report.name                   Query 29
WHERE   V.report.name = "Jones" AND
        V.manager.salary < U.salary
```

An automated query generator will generate the following SQL query:

```
SELECT                                          Query 30
        A.name, B.name
FROM
        employee AS A LEFT OUTER JOIN
        employee AS B
            ON A.name = B.manager_name_fk,
        employee AS C JOIN
        employee AS D
            ON C.name = D.manager_name_fk
WHERE
        D.name = "Jones" AND
        C.salary < A.salary
```

Similarly, the high-level query expressed with an embedded SELECT statement and roles is:

```
SELECT  name, report.name                       Query 31
WHERE   salary >
            (SELECT manager.name
             WHERE  report.name = "Jones" )
```

The corresponding generated SQL query is:

```
SELECT                                          Query 32
        A.name, B.name
FROM
        employee AS A LEFT OUTER JOIN
        employee AS B
            ON A.name = B.manager_name_fk
WHERE
        A.salary >
            (SELECT
                C.salary
             FROM
                employee AS C JOIN
                employee AS D
                    ON C.name = D.manager_name_fk
             WHERE
                D.name = "Jones" )
```

In both cases, roles are used to identify entities and their relationships within a supercontext. Tuple variables and embedded SELECT statements identify how supercontexts are related.

6. Conclusion

Simplified access to databases helps speed system development time, simplifies programming tasks, and increases application robustness with respect to underlying schema modifications. Contexts have been used as the knowledge-based construct for QUICK to provide these benefits to real-world applications. Unfortunately, the benefits of contexts are limited when multiple contexts apply to a query, when there are recursive relationships, and when there are partial participation relationships. This paper demonstrates that roles defined in conceptual schemas in conjunction with pseudo-schemas, super-contexts, and an augmented high-level query syntax can be used effectively in these cases. Ultimately, these new constructs are beneficial in that they greatly decouple applications and users from underlying databases.

Bibliography

[Bat92] Batini, C., Ceri, S., and Navathe, S.B. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, Redwood City, CA. 1992.

[Che76] Chen, P.P. "The Entity-Relationship Model - Toward a Unified View of Data, " *ACM Transactions on Database Systems 1,* 1, 9-36. 1976.

[Dia95] Diamond, S.D. The Universal SQL Processor, Internal Johns Hopkins University Applied Physics Laboratory communication. RMI-95-005. 1995.

[Har98] Harris, L., English Wizard 3.0, http://www.englishwizard.com/. 1998.

[Kor84] Korth, H.F., Kuper, G.M., Feigenbaum, J., Van Gelder, A., and Ullman, J.D. System/U: A Database System Based on the Universal Relation Assumption. *ACM Transactions on Database Systems 9,* 3, 331-347. 1984.

[Ley89] Leymann, F. "A Survey of the Universal Relation Model," *Data & Knowledge Engineering 4,* 4, 305-320. 1989.

[Lie80] Lien, Y.E. "On the Semantics of the Entity-Relationship Model," in *Entity-Relationship Approach to System Analysis and Design,* P.P. Chen, ed. North-Holland, Amsterdam, pp. 155-167. 1980.

[Mar90] Markowitz, V.M., and Shoshani, A. "Abbreviated Query Interpretation in Extended Entity-Relationship Oriented Database," in *Entity-Relationship Approach to Database Design and Querying,* Lochovsky, F.H., Ed. North-Holland, Amsterdam, pp. 325-343. 1990.

[Ora98] Oracle Corporation, Developer 2000, http://www.oracle.com/products/tools/dev2k/index.html. 1998

[Pah85] Pahwa, A., and Arora, A.K. Automatic Database Navigation: Towards a High Level User Interface. In *Proceedings of the 4th International Conference on Entity-Relationship Approach,* Chen, P.P., Ed., IEEE Computer Society Press, pp. 36-43. 1985.

[Sem94] Semmel, R.D. "Discovering Context in an Entity-Relationship Conceptual Schema," *Journal of Computer and Software Engineering*, 2:1, pp. 47-63. 1994.

[Sem95] Semmel, R.D. "Integrating Reengineered Databases to Support Data Fusion," *Journal of Systems Software,* 30:127-135. Elsevier Science Inc., NY, NY. 1995.

[Sil94] Silberberg, D.P., Semmel, R.D. "The StarView Flexible Query Mechanism." *Astronomical Data Analysis Software and Systems III*, Crabtree, D.R., Hanisch, R.J., Barnes, J. (eds.), Astronomical Society of the Pacific, Vol. 61., pp. 92-95. 1994.

[Teo86] Teorey, T.J., Yang, D., and Fry, J.P. "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *ACM Computing Surveys 18,* 2, 197-222. 1986.

[Ull89] Ullman, J.D. *Principles of Database and Knowledge-Base Systems,* Vol. 2. Computer Science Press, Rockville, MD. 1989.

[Wal84] Wald, J.A., and Sorenson, P.G. Resolving the Query Inference Problem Using Steiner Trees. *ACM Transactions on Database Systems 9,* 3, 348-368. 1984.

[Zha83] Zhang, Z., and Mendelzon, A.O. A Graphical Query Language for Entity-Relationship Databases. In *Proceedings of the 3rd International Conference on Entity-Relationship Approach,* pp. 441-448. 1983.