Multidatabase Querying by Context[1]

Ramon LAWRENCE

Advanced Database Systems Laboratory
University of Manitoba
umlawren@cs.umanitoba.ca

Ken BARKER

Advanced Database Systems Laboratory
University of Calgary
barker@cpcs.ucalgary.ca

Abstract. The overwhelming acceptance of the SQL standard [5] has curtailed continuing research work in relational database query languages. Since all commercial relational databases use SQL, there is little motivation for developing new query languages. Despite its widespread acceptance, SQL is not a perfect language. Complex database schema challenge even experienced database users during query formulation. As increasing numbers of less sophisticated users access numerous databases, their ability to construct queries diminishes.

In this work, we overview a new query language developed in conjunction with our integration architecture for automatically integrating relational schema into a context view. A context view (CV) is a high-level view of database semantics that allows logically and physically transparent access to the underlying data source(s). By allowing the user to query by context and semantic connotation, a whole new level of query complexity arises. However, we demonstrate that the context view has similar properties as the Universal Relational Model and thus can benefit from associated algorithms and ideas. We demonstrate that high-level semantic queries on the CV can be systematically mapped to SQL queries that are rigorous enough for use in industrial systems and are compatible with existing systems.

Keywords. multidatabase, query, SQL, context, semantic, dictionary, schema, integration, Universal Relation

1 Introduction

Despite dramatic changes in database size, complexity, and interoperability, Structured Query Language (SQL) [5] has remained fundamentally unchanged. SQL is an extremely powerful language when used by sophisticated users. However, a larger number of inexperienced users now interact with multiple databases. These users have limited understanding of SQL let alone database structure and semantics. Although the complexity of SQL generation has been partially hidden by graphical design tools and programming languages, the fundamental challenges of SQL remain. A SQL user is responsible for understanding the structure of a database schema, the names associated with schematic elements, and the semantics of the data stored. Query formulation involves mapping query semantics into the semantics of the database and then realizing those semantics by combining the appropriate database structures. On a wider scale, organizations require the interoperability of database systems. Variants of SQL for multidatabase querying suffer from the same limitations as SQL and force users to understand the structure and semantics of all databases which introduces exponential complexity as the number of databases increases.

[1] This research is partially sponsored by a Natural Science and Engineering Research Grant (RGP-0105566)

To address these issues, we designed and implemented a new integration architecture that automatically integrates diverse relational schema into a unified view of concepts called a context view (CV). Relevant information on the integration system is presented in Section 2. The context view isolates the user from structural issues, and the system provides mappings from semantic concepts to structural organizations. Unlike other SQL languages and query tools, our query system never requires the user to understand database structure. Queries are formulated using a graphical interface by manipulating elements of the context view. The context view is a special type of Universal Relation describing the data source (Section 3) and has features that resolve some of its problems. We present algorithms for automatically mapping queries posed on the context view to SQL statements in Section 4 and extensions to the architecture in Section 5. The paper closes with future work and conclusions.

2 Background and Previous Work

Querying by context is developed in conjunction with our integration architecture [11] which automatically integrates relational database schema to produce an integrated view of concepts called a context view (CV). We will use the terms context view and integrated view interchangeably. Our current focus is on schema integration and database semantics. The related data integration issues are not currently considered and will not be examined here.

The integration architecture consists of two phases: a capture process and an integration process. In the capture process, the database administrator (DBA) uses an automated tool to extract database schema information (including types, sizes, names, and relationships) and saves the information into a XML document called a X-Spec which encodes schema structure and semantic information for a data source. Semantic information is encoded as semantic names for each table and field. Semantic names capture system-independent semantics by combining terms from a standardized term dictionary. The dictionary is a hierarchical, organization of concepts which are re-used in different contexts. Thus, the dictionary is a base-set of terms which are combined to represent both contextual and concept information.

A *semantic name* is a XML tag constructed from dictionary terms which provides the complete context and concept information across systems. A semantic name has the form:

$$semantic\_name = "[" CT\ [[;CT]\,|\,[,CT]]"]"[CN] \ \text{where}\ CT =< term >, CN =< term >$$

That is, a semantic name consists of an ordered set of context terms (CT) separated by either a comma or a semi-colon, and an optional concept name term (CN). Each context and concept term is a single term from the standardized dictionary. The comma between terms A and B (A,B) represents that term B is a subtype of term A. A semi-colon between terms A and B (A;B) means that term A HAS-A term B, or term B is part of term A. The context terms provide a context for the concept that describes them. Every semantic name has at least one context term. The concept name is a term describing the lowest level semantics.

The integration process combines the X-Specs from the data sources into an integrated view of concepts by matching semantic names. The resulting view is a merger of all concepts across all data sources. A modified version of the Northwind database provided with Microsoft Access© is used as an example database for this paper. We have omitted some descriptive fields that are not important to the discussion such as address fields. In the text, table and field names appear in *italics* and semantic names appear in `true-type`. The Northwind schema is given in Table 1 and the associated mapping from system names to semantic names is in Table 2. Finally, Figure 1 contains the integrated view produced.

Table 1: Northwind Database Schema

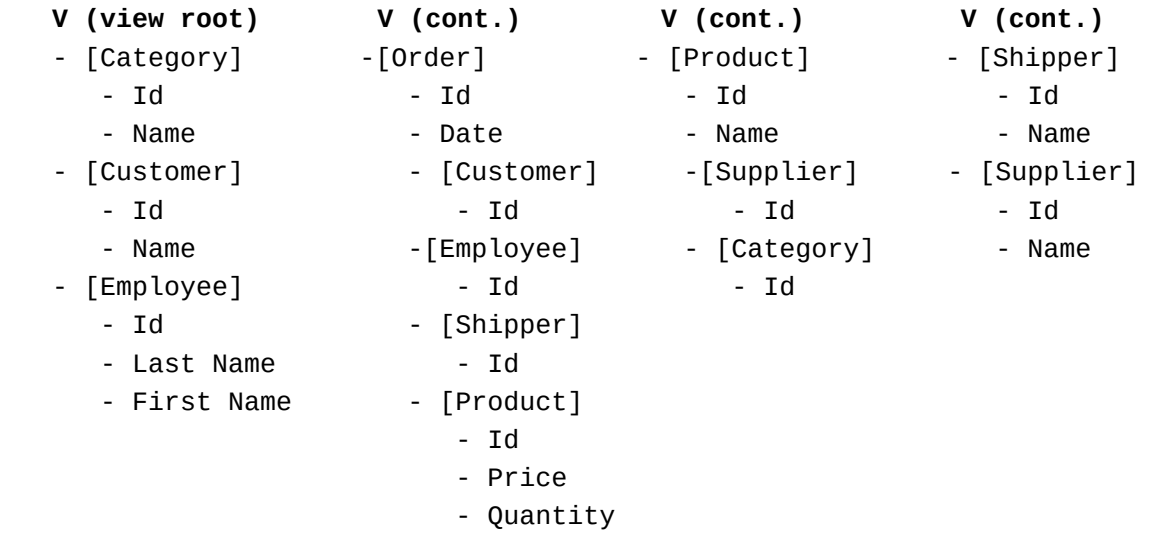
Tables	Fields
Categories	CategoryID, CategoryName
Customers	CustomerID, CompanyName
Employees	EmployeeID, LastName, FirstName
OrderDetails	OrderID, ProductID, UnitPrice, Quantity
Order	OrderID, CustomerID, EmployeeID, OrderDate, Shipvia
Products	ProductID, ProductName, SupplierID, CategoryID
Shippers	ShipperID, CompanyName
Suppliers	SupplierID, CompanyName

Table 2 : Northwind Semantic Name Mappings

Type	Semantic Name	System Name
Table	[Category]	Categories
Field	[Category] Id	CategoryID
Field	[Category] Name	CategoryName
Table	[Customer]	Customers
Field	[Customer] Id	CustomerID
Field	[Customer] Name	CompanyName
Table	[Employee]	Employees
Field	[Employee] Id	EmployeeID
Field	[Employee] Last Name	LastName
Field	[Employee] First Name	FirstName
Table	[Order;Product]	OrderDetails
Field	[Order] Id	OrderID
Field	[Order;Product] Id	ProductID
Field	[Order;Product] Price	UnitPrice
Field	[Order;Product] Quantity	Quantity

Type	Semantic Name	System Name
Table	[Order]	Orders
Field	[Order] Id	OrderID
Field	[Order;Customer] Id	CustomerID
Field	[Order;Employee] Id	EmployeeID
Field	[Order] Date	OrderDate
Field	[Order;Shipper] Id	Shipvia
Table	[Product]	Products
Field	[Product] Id	ProductID
Field	[Product] Name	ProductName
Field	[Product;Supplier] Id	SupplierID
Field	[Product;Category] Id	CategoryID
Table	[Shipper]	Shippers
Field	[Shipper] Id	ShipperID
Field	[Shipper] Name	ShipperName
Table	[Supplier]	Suppliers
Field	[Supplier] Id	SupplierID
Field	[Supplier] Name	SupplierName

Figure 1: Northwind Integrated View



There are several features that characterize the context view (CV):

1. The context view consists of a hierarchy of concepts organized to model the inherent semantics and relationships of the data, not its physical representation and structure.
2. A term in the CV may map to zero or more schema elements in underlying data sources.
3. Mapping from a given CV term to its associated schema elements in the data sources (field and table names) is possible using information stored in X-Specs.

Given this environment, the goal of this work is to provide an *ad-hoc* query facility.

2.1 Previous Work

The SQL standard [5] allows users to query different database platforms using one language. However, specifying complex SQL queries with numerous join conditions and subqueries is too complex for most users [2]. Further, developing SQL queries requires knowledge of both the structure and semantics of the database. Unfortunately, database semantics are not always immediately apparent from the database schema, and mapping the required query semantics into a SQL query is often complex. Although graphical tools for query construction and high-level programming languages mask some of the complexity, structural querying is intrinsic to most data access.

SQL is unsuitable for querying multidatabases or federated databases. These systems are a collection of two or more databases operating to share data. Extensions of SQL such as MSQL [13] and its successor IDL [10] provide features for multidatabase querying. These languages allow the user to define higher order queries and views by allowing database variables to range over metadata in addition to regular data. Other MDBS query languages include DIRECT [15] and SchemaSQL [7]. The fundamental weakness of these languages is the reliance on the user's knowledge of database structure and semantics to construct queries. Understanding the structure and semantics of one data source is complicated in itself and the in-depth knowledge required to formulate queries on multiple databases is extremely rare.

A fundamental database model is the Universal Relation Model that provides logical and physical query transparency by modeling an entire database as a single relation. We will demonstrate the similarity of the context view with the Universal Relation Model [14], and thus argue that our system also provides logical and physical query transparency. There has been substantial work presented on querying in a Universal Relation environment [3], and more generally in the theory of joins [1] and querying [9,16].

It is also important to distinguish our architecture from wrapper and mediator systems. Mediator systems either assume an integrated view is constructed *a priori* by designers or do not construct an integrated view at all. The integrated view is then mapped to the local views of the mediators by logical rules or query expressions specified by the designer. These systems achieve database interoperability by providing an integrated view and its associated mappings to local systems and then automatically divide a query on the integrated view into queries on the individual data sources. Numerous systems [4,6,8,12] have been developed.

Mediator systems do not perform schema integration, which is performed manually by designers. Our work is unique because it automatically produces an integrated view from data source specifications developed independently of both other data sources and the global view itself. The integrated view hides structural organization from the user and displays information in a semantically intuitive hierarchy of concepts. Since the integrated view is not queried by structure, the query system presented in this paper is developed to compliment the unique nature of the architecture.

3 Context View as a Universal Relation

The context view (CV) models database schema knowledge as a hierarchy of concepts. In this section we describe the nature of the CV and its relationship to the Universal Relation.

A *dictionary term* is an unambiguous word phrase in the standardized term dictionary. Each term represents a unique semantic connotation of a given word phrase, so words with multiple definitions are represented as multiple terms in the dictionary. A *context term* is a dictionary term that describes the context of a schema element. A *concept term* is a term that provides the lowest level semantic description of a database field. Overall, a semantic name is a *concept* if it maps to a database field and a *context* if it maps to a table.

A *semantic name* S_i consists of an ordered set of dictionary terms $\{T_1,T_2,...T_N\}$ where $N \geq 1$ which uniquely describe the semantic connotation of a schema element. The last term T_N is a concept name if S_i has a concept name, otherwise it is the most specific context of S_i.

Definition. The *context closure* of semantic name S_i denoted S_i* is the set of semantic names produced by taking ordered subsets of the terms of $S_i = \{T_1,T_2,...T_N\}$ starting with T_1. ■

Example 1. Given a semantic name S_i=[A;B;C] D, S_i*={[A],[A;B],[A;B;C],[A;B;C] D}.

Now we are able to formally define a *context view (CV*) as follows:

If a semantic name S_i is in *CV*, then for any S_j in S_i*, S_j is also in *CV*.

For each semantic name S_i in *CV*, there exists a set of zero or more mappings M_i that associate a schema element E_j with S_i.

A semantic name S_i can only occur once in the *CV*.

The integration architecture combines schema elements into the CV by matching semantic names term-wise until it is completely matched or no further matches are found. Thus, the *CV* is a tree of nodes $N=\{N_1, N_2,...N_n\}$, where each node N_i has a semantic name S_i.

3.1 Context View as a Universal Relation

There is an underlying similarity between a context view and a Universal Relation. A Universal Relation (UR) contains all the attributes of the database where each attribute has a unique name and semantic connotation.

Lemma. *A context view (CV) is a valid Universal Relation if each semantic name is considered an attribute.*

Proof. For a given data source, each field is assigned a semantic name. The semantic name defines a unique semantic connotation for the field. To violate the Universal Relation assumption, a given semantic name must either occur more than once in the *CV* (non-unique attribute names) or two or more semantic names have identical connotations (non-unique semantic connotations). A semantic name can only occur once in a *CV* by definition. Hence, each semantic (attribute) name is unique. The construction of a semantic name by combining terms defines its semantics such that two different semantic names cannot have the same semantic connotation. Thus, a context view is a valid Universal Relation. ■

A context view addresses several of the problems of the UR model. First, the context view is automatically created by the system after the database semantics are systematically described by the DBA using semantic names. The context view also resolves the issue of complex Universal Relations. Since the CV is organized hierarchically, it is explicitly divided into semantically grouped topics as opposed to one, flat relation containing all attributes. This reduces the semantic burden on the user.

The context view is more than a Universal Relation. It is a hierarchically organized, integrated view of database knowledge in one or more systems. Like a view, it is an amalgamation of data stored in other structures that is built as needed. Thus, we demonstrate how queries on the CV are realized by mapping to SQL queries to extract the relevant data.

4 Query Parsing and Join Tree Construction

By isolating the user from database structure, the system becomes responsible for correctly formatting the query based on the user's intended semantics. Thus, the system must generate deterministic, repeatable, and semantically intuitive queries in all cases. Given a context view, users generate a query by graphically selecting the selection criteria and result fields by semantic name. Then, the query system translates the query into a structural query (SQL) for the underlying database. In this section, we present these translation algorithms.

There are two major requirements in mapping from semantic to structural querying. First, the system selects the fields to use for projection and selection. Since multiple mappings to the same semantic name are possible, the system selects the most appropriate field mapping. Second, the join conditions are determined to combine the appropriate data source tables. Given the set of fields and tables to access and a set of joins to apply, it is straightforward to construct a relational calculus expression or SQL select-project-join query.

4.1 Determination of Data Source Fields and Tables

The system determines the tables and fields to access by the semantic names chosen. In most cases, a semantic name has only one mapping to a physical field. However, especially with key fields, a semantic name may map to several fields. Since the choice of field (and its parent table) may affect query semantics by introducing new joins, the system has well-defined rules which are easily conveyed to the user. Fields for selection or projection operations are treated uniformly by the system. For a key field occurring in two or more tables, the inherent interrelationships between the tables determine the complexity in selecting a mapping. The four cases are presented below with examples from the Northwind database.

1-1 - A one-to-one relationship between tables often implies they share a key. For example, assume a database has two tables indexed by social security number (SSN). If a user wants the SSN field, there are two possible mappings but with different semantic names (because their contexts are different). Thus, the mapping chosen is uniquely determined by the user's choice of semantic name.

1-N - One-to-N relationship between tables implies a foreign key from the N-side table to the one-side table. Consider, the *Orders* and *Shippers* tables with the semantic names `[Order;Shipper] Id` for the foreign key in *Orders* to *Shippers* and `[Shipper] Id` as the primary key of the *Shippers* table. Again, the query system has a unique mapping to the concept of a shipper id based on if the user selects the foreign key in the *Orders* table (`[Order;Shipper] Id`) or the primary key in the *Shippers* table (`[Shipper] Id`).

1-N dependent - When the N-side of the relationship is dependent on the one-side, a special case arises. Consider *Orders* and *OrderDetails*. Since, an *OrderDetail* record cannot exist without an *Order* record, the *OrderDetails* table has as part of its key the key for the *Orders* table. Both fields are assigned the semantic name `[Order] Id`. Thus, there are two field mappings to the semantic name. The general heuristic is to choose the primary key instance (*Orders*) unless the user selects attributes from the *OrderDetails* table.

M-N and **M-N dependent** - Any M-N relationship results in multiple mappings to a single semantic name because the relationship is structured by constructing a joining table whose key contains the keys of the two tables. Consider a database of books and authors. Since a book may have multiple authors and an author may write multiple books, a joining table *BookAuthor* (`[Book;Author]`) implements the M-N relationship. The *BookAuthor* table has mappings to the Book table (`[Book] Id`) and Author table (`[Author] Id`) keys.

In non-normalized databases, multiple fields in a table may map to a semantic name. For example, if an order has three items all stored in the order table, then the semantic name `[Order;Product] Id` will have three mappings. The semantically correct query should automatically normalize the data by splitting one order record into three normalized records.

To handle multiple mappings, the query system first selects a field that is currently present in the tables already in the query. Otherwise, it chooses the field whose parent table context matches the field context, or the first field mapping if no other heuristic applies. The algorithm (see Figure 2) determines a set of fields (*F*) and tables (*T*) from a X-Spec (X_j) which best map to the set of query nodes $Q = \{Q_1, Q_2, \ldots Q_n\}$ given by the user.

Figure 2 : Field Selection Algorithm

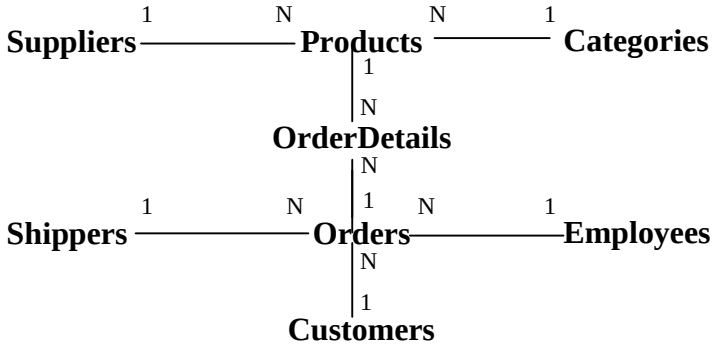
```
For each term Qi in Q                                                                    (1)
    SNi = semantic name of Qi                                                            (2)
    Search_XSpec(Xj, SNi, num, R)          // Search X-Spec for SN.  Return results in R. (3)
    If num =1 Then                         // Only one occurrence of semantic name      (4)
        Add field Rk to F and parent table of Rk to T                                    (5)
    Else                                                                                 (6)
        If multiple occurrences but only in one table Then                               (7)
            For each result Rk in R                                                      (8)
                Add field Rk to F                                                        (9)
            Next                                                                        (10)
            Add parent table of R1 to T                                                 (11)
        End if                                                                          (12)
    End if                                                                              (13)
Next                                                                                    (14)
// Second pass to resolve multiple occurrences
For each term Qi in Q                                                                   (15)
    SNi = semantic name of Qi                                                           (16)
    If Qi has not been mapped Then                                                      (17)
        search_XSpec(Xj, SNi, num, R)      // Search X-Spec for SN.  Return results in R. (18)
        If Find any mapping Rk of R with parent table Tj already in T Then              (19)
            Add field Rk to F and parent table of Rk to T                               (20)
        ElseIf Find parent table Tj of Rk with context portion = context portion of Qi Then (21)
            Add field Rk to F and parent table of Rk to T                               (22)
        Else                                                                            (23)
            Add field R1 to F and parent table of R1 to T                               (24)
        Endif                                                                           (25)
    Endif                                                                               (26)
Next                                                                                    (27)
```

4.2 Determining Join Conditions

Given a set of fields and tables to access, the system determines a set of joins between the tables to isolate the user from join selection and preserve the semantics of the user's query.

Define a *join graph* as an undirected graph where each node is a table in the database and there is a link from node N_i to node N_j if there is a join between the corresponding two tables. For this discussion, we ignore multiple joins between two tables on different keys. A *join path* is a sequence of joins interconnecting two nodes in the graph; and a *join tree* is a set of joins interconnecting two or more nodes. Assume without loss of generality that the join graph is connected. Otherwise, we apply the algorithm to each connected subset and connect them using a cross-product. The join graph for the Northwind database is in Figure 3.

Figure 3 : Join Graph for Northwind Database



Lemma 1. *There is only one join path between any two nodes in an acyclic join graph.*

Proof. Proof by contradiction. Assume that two join paths exist between node N_i and node N_j. Then, we could take the first path from N_i to N_j and return on the second path from N_i to N_j. This implies that the graph has a cycle. ■

Lemma 2. *There is only one join tree between any subset of nodes in an acyclic join graph.*

Proof. Proof by induction. The statement is true for two nodes as per Lemma 1. Given a subset of *m* nodes with only one join tree, add another node *N* to the set. Assume that by adding *N* there exists more than one join tree in the new subset of *m+1* nodes. Since, there was only one join tree for the previous *m* nodes, this implies that *N* must be connected to more than one node in the subset. If *N* is connected to two nodes N_i and N_j in the *m* nodes, then there must be a path from *N* to N_i, N_i to N_j, and N_j to *N* by Lemma 1. This produces a cycle. Thus, the statement holds for *m+1* nodes. The result follows by induction. ■

The consequences of Lemma 2 are important because no decisions on which joins to apply are required. We must only identify which joins are required to connect the tables by constructing the join tree. The order in which the joins are applied is the join optimization problem that has been actively studied and will not be discussed here. From this result, we construct an algorithm that builds a matrix *M* where entry $M[N_i,N_j]$ is the shortest join path between any pair of nodes N_i and N_j. By combining join paths, the query system identifies all the joins required to combine database tables by constructing the only possible join tree.

Theorem 1. *Given a matrix* M *storing shortest join paths for an acyclic join graph and a set of tables* T *to join, a join tree can be constructed by choosing any table* T_i *from* T *and unioning the join paths in* $M[N_i,N_1]$, $M[N_i,N_2]$, ... $M[N_i,N_n]$ *where* $N_1,N_2,..N_n$ *are the nodes corresponding to the set of tables* T.

Proof. Since the graph is connected, the matrix entries $M[N_i,N_1]$, $M[N_i,N_2]$, ... $M[N_i,N_n]$ represent join paths from N_i to all other nodes in the subset. Assume a join tree is not constructed. Thus, there is no path between some two nodes N_j and N_k. However, there is a path from N_i to N_j and from N_i to N_k. Then, there is a path from N_j to N_k. Thus all nodes are connected with the join tree and it is the only possible join tree as per Lemma 2. ■

Normalized databases often have acyclic join graphs. The system handles cycles by determining the best join paths using path length and join semantics/properties. The breadth-first search algorithm (see Figure 4) constructs the matrix *M* of best join paths by selecting the shortest join paths with no lossy joins. It works for all join graphs. Lossy joins are used only if there is no other path between nodes. A lossy join sequence contains a N-1 cardinality join followed by a 1-N cardinality join (produces a M-N join); and a M-N join is always lossy.

Figure 4: Algorithm to Calculate Join Paths

```
void calc_join_paths(ByRef M as matrix, G as graph)
// M is an N x N matrix where N is the number of nodes in the graph, NQ is a FIFO queue structure
// Note: For any matrix entries not assigned no join path exists between the pair of nodes (need cross-product to join)
    Node   F, N, LTN
    Link   L
    Integer  jtype                  // Type of join by cardinality: 1-1, 1-N, N-1, M-N
    For each node F in G                                                                          (1)
        M[F,F] = Null               // Empty join path to itself                                  (2)
        count = 0                                                                                 (3)
        accept_lossy = false                                                                      (4)
    Repeat_label:
        add F to NQ                                                                               (5)
        While NQ is not empty                                                                     (6)
            Remove first node N from NQ                                                           (7)
            For each outgoing link L of N                                                         (8)
                LTN = destination node of link L from N                                           (9)
                jtype = cardinality of join for L (from N to LTN)                                 (10)
                If LTN is not visited and (accept_lossy or                                        (11)
                  not ((M[F,N] has N-1 join and jtype=1-N)) or jtype=M-N) Then
                    Add LTN to NQ                                                                 (12)
                    Mark LTN as visited                                                           (13)
                    M[F,LTN] = M[F,N] + LTN                                                       (14)
                    count++                                                                       (15)
                ElseIf accept_lossy or not ((M[F,N] has N-1 join and jtype = 1-N)) or jtype = M-N) Then   (16)
                // Replace current join path (M[F,LTN]) if new join path is same length with better properties
                End if                                                                            (17)
            Next                                                                                  (18)
        End while                                                                                 (19)
        clear_flags()               // Clear all visited flags for all nodes in G                 (20)
        If count < # of nodes in G and accept_lossy = false Then                                  (21)
            accept_lossy = true                                                                   (22)
            Goto Repeat_label       // Repeat algorithm accepting all joins (even lossy)          (23)
        End if                                                                                    (24)
    Next                                                                                          (25)
```

Theorem 1 cannot be applied directly to produce a join tree for a cyclic graph because there will be multiple join trees that are all semantically valid. The system cannot differentiate them without knowledge about the intended query semantics. Although heuristic algorithms may choose a join tree, it is better to have a precise mechanism for the user to exploit. Thus, we define query extensions that allow the user to precisely define the semantics of the query so the system can uniquely determine the join tree required.

5 Query Extensions

To more precisely define query semantics, extensions to the integrated view are possible. The first extension allows the user to pick the root join table or row in the join matrix. Semantically, the root join table is the starting point of all join paths. This allows the system to unambiguously construct a join tree which matches the intended query semantics.

The second optimization enhances the integrated view presentation by displaying join conditions to the user. If a semantic name is a foreign key to another concept, the system displays the attributes of the linked concept. For example in Northwind, the *EmployeeID* in *Orders* has a semantic name `[Order;Employee] Id` as it is the foreign key from *Orders* to *Employees*. When the user clicks on this semantic name, the system links to the *Employees* table and displays the fields of *Employees (EmployeeID, LastName, FirstName)*.

This approach has several benefits. It reduces the semantic burden on the user by automatically displaying concept interrelationships, and reduces the query generation complexity for the system. By explicitly displaying the join information and associated fields, the system now has an unambiguous reference from the user on which fields to use, from what tables, and the corresponding join condition to use to relate the two different contexts.

6 Future work and Conclusions

In this paper, we have demonstrated how the context view produced by our integration architecture is similar to the Universal Relation. Further, we demonstrated a method for mapping user queries on an integrated view of semantic concepts to SQL. The system handles complex join constructs and selects the appropriate fields, tables, and join conditions to preserve user query semantics. Finally, we propose extensions to the original context view to allow the user to more formally define the semantics of their query without explicit knowledge of the structure and interrelationships of database fields and tables.

References

[1] Aho A.V., Beeri C., Ullman J.: The theory of joins in relational databases. *ACM Transactions on Database Systems*. 4(3):297-314, September 1979.

[2] Bell J., Rowe L.: Human factors evaluation of a textual, graphical, and natural language query interfaces. Technical Report ERL-90-12, University of California, Berkeley, Feb. 1990.

[3] Brosda V., Vossen G.: Update and retrieval in a relational database through a universal schema interface. *ACM Transactions on Database Systems*. 13(4):449-485, Dec. 1988.

[4] Collet C., Huhns M., Shen W-M.: Resource integration using a large knowledge base in Carnot. *IEEE Computer*. 24(12):55-62, December 1991.

[5] Date C.J.: *The SQL standard.* Addison Wesley, Reading, US, third edition, 1994.

[6] Genesereth M., Keller A., Duschka O.: Infomaster: An information integration system. *SIGMOD Record*. 26(2):539-542, May 1997.

[7] Gingras F., Lakshmanan L., Subramanian I., Papoulis D., Shiri N.: Languages for multi-database interoperability. *SIGMOD Record*. 26(2):536-538, 1997.

[8] Kirk T., Levy A., Sagiv Y., Srivastava D.: The Information Manifold. In *AAAI Spring Symposium on Information Gathering* (1995).

[9] Korth H., Juper G., Feigenbaum J., Gelder A., Ullman J.: System/U: A database system based on the universal relation assumption. *ACM Transactions on Database Systems*. 9(3):331-347, 1984.

[10] Krishnamurthy R., Litwin W., Kent W.: Language features for interoperability of databases with schematic discrepancies. *SIGMOD Record*. 20(2):40-49, June 1991.

[11] Lawrence R., Barker K.: Automatic integration of relational database schemas. TR-00-15, University of Manitoba, Department of Computer Science, July 2000.

[12] Li C., Yerneni R., Vassalos V., Garci-Molina H., Papakonstantinou Y., Ullman J., Valiveti M.: Capability based mediation in TSIMMIS. *SIGMOD Record*. 27(2):564-566, June 1998.

[13] Litwin W., Abdellatif A.: An overview of the multidatabase manipulation language MDSL. In *Proceedings of the IEEE*, May 1987, 69-73.

[14] Maier D., Vardi M., Ullman J.: On the foundations of the universal relation model. *ACM Transactions on Database Systems*. 9(2):283-308, June 1984.

[15] Merz U., King R.: DIRECT: A query facility for multiple databases. *ACM Transactions on Information Systems*. 12(4):339-359, October 1994.

[16] Sagiv Y.: A characterization of globally consistent databases and their correct access paths. *ACM Transactions on Database Systems*. 8(2):266-286, June 1983.