Multidatabase Querying by Context[1]

University of Calgary 2000-663-15
University of Manitoba TR-00-16

Ramon Lawrence
Advanced Database Systems
Laboratory
Department of Computer Science
University of Manitoba
umlawren@cs.umanitoba.ca

Ken Barker
Advanced Database Systems
and Applications Laboratory
Department of Computer Science
University of Calgary
barker@cpsc.ucalgary.ca

Abstract

The overwhelming acceptance of the SQL standard [10] has curtailed continuing research work in relational database query languages and processing. Since all commercial relational database systems conform with the SQL standard, there is little motivation for developing new query languages.

Despite its benefits and wide-spread acceptance, SQL is not a perfect query language. Complex database schema challenge even experienced database users during query formulation. As increasing numbers of less sophisticated users access numerous data sources within an organization or across the Internet, their ability to accurately construct queries with the appropriate structure and semantics diminishes. SQL can be hard to use as it provides only physical access transparency not logical transparency. That is, a user is responsible for mapping the semantics of their query to the semantics and structure of the database. Although graphical tools for query construction and high-level programming languages mask some of the complexity, the notion of querying by structure is intrinsic to most forms of data access.

In this work, we overview a new query language developed in conjunction with our integration architecture for automatically integrating relational schema. Although the major focus of this work is on database interoperability, the contribution of this paper is a language for specifying queries on the integrated view produced. The complexities of querying across database systems and resolving conflicts are too numerous to be fully described here, so this paper will discuss querying the integrated view of a single database.

The integration architecture integrates database schema information into a context view (CV). The context view is a high-level view of database semantics which allows logically and physically transparent access to the underlying data source(s). Since this context view is an entirely new way of organizing and categorizing database information, a new query language is developed. However, we demonstrate that the context view has similar properties as the Universal Relational Model and thus can benefit from its associated algorithms and ideas.

By allowing the user to query by context and semantic connotation, a whole new level of query complexity arises. Mapping of queries from semantic concepts to physical tables, fields, and relationships must be automatically performed. We will demonstrate that specific relational calculus expressions or SQL queries can be generated from abstract concepts which are rigorous enough for use in industrial applications and systems. Specifically, SQL generation and join discovery are overviewed. Thus, the query language can be mapped to SQL allowing backwards compatibility with existing systems.

1 Introduction

Despite dramatic changes in database size, complexity, and interoperability, SQL has remained fundamentally unchanged. The wide-variety of applications, users, and independent systems accessing databases rely on Structured Query Language (SQL) [10] to retrieve the required information. Although the complexity of SQL generation has been partially hidden by graphical design tools and more powerful programming languages, the fundamental challenges of SQL remain.

The fundamental problem of SQL is also one of its greatest benefits. SQL allows a database to be queried by a clearly defined structure which is a vast improvement over hierarchial methods and direct access technologies that require explicit navigation between records. Unfortunately, a SQL user is responsible for understanding the structure of a database schema, the names

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

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.

SQL is an extremely powerful language when used by sophisticated users who understand its use and the database queried. However, larger numbers of users, many with little database experience, may now interact with multiple database systems. These users have limited SQL understanding and limited understanding of database structure and semantics. In such an environment, SQL is insufficient for these users.

On a wider scale, organizations are attempting to achieve database interoperability by combining database systems into a more unified organization. The goal is to access all data sources transparently. Variants of SQL for multidatabase querying allow SQL-type queries to multiple data sources but suffer from the same limitations as SQL. These systems force users to understand the structure and semantics of all databases which introduces exponential complexity as the number of databases accessed increases.

To address these shortcomings, we have designed and implemented a new intergration system. Our integration architecture automatically integrates diverse relational schema into an unified view of concepts called a context view (CV). Relevant information on the integration system is presented in Section 2. The context view is designed to isolate the user from structural and semantic details by organizing database information into a hierarchy of concepts. 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. Although the context view and its associated query system were not developed to model the Universal Relation, they display many similar properties which can be used to better understand the foundations of the context view and may be used to develop similar query algorithms. We present algorithms in Section 4 for automatically mapping queries posed on the context view to SQL queries and propose extensions to the original model 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 [19] 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 issues of data integration including entity matching, field type and size reconciliation, and data scaling are not currently considered. Hence, discussions of such data integration problems related to querying 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 structure, types, sizes, names, and relationships) and saves the information into a XML document called a X-Spec. A X-Spec is a XML document which encodes schema structure and semantic information for a single data source. Semantic information is encoded as semantic names for each table and field in the data source. Semantic names are stored as XML tags and are designed to capture the full semantics of a schema element including contextual information.

Semantic names capture system-independent semantics by combining terms from a standardized term dictionary. This dictionary is more than a simple set of standardized XML tags. Rather, it is a hierarchial, organization of concepts (represented by English words) which can be re-used in semantic names in different contexts. Thus, the dictionary is a base-set of concepts which can be applied and combined to present both the required contextual and concept information.

A *semantic name* is a XML tag constructed from one or more dictionary terms which provides

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

Figure 1: Northwind Database Schema

the complete context and concept information across systems. A semantic name has the form:

$semantic\_name = ''['' \; CT \; [\,[;CT] \mid [,CT]\,] \; '']'' \;\; [CN]$

$CT =< dictionary\_term >, CN =< dictionary\_term >$

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 represents a concept that is part of term A. The context terms provide a context framework for the concept that describes them. Every semantic name has at least one context term. The concept name is a single, atomic term describing the lowest level semantics. Fields have concept names to represent their base meaning void of any context information.

The integration process combines the X-Specs from the individual data sources into an integrated view of concepts by matching semantic names. Since a semantic name is itself a hierarchy of concepts (and contexts), the resulting integrated 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 throughout this paper. We have omitted some descriptive fields which are not important to the discussion including addresses and phone numbers. In the text, table and field names appear in *italics* and semantic names appear in `true-type`. The Northwind schema is given in Figure 1, and the associated mapping from system names to semantic names is in Figure 2. Finally, Figure 3 contains the integrated view produced for the Northwind database.

Issuing queries on the integrated view of concepts (Figure 3) involves choosing which fields to display in the query result. The query generator must use X-Spec information to generate the proper joins and mappings from semantic to system names. A "global key" such as UPC code is important in query generation as it is guaranteed unique across databases similar to a social security number. Such keys allow the system to perform joins across databases.

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

- The context view consists of a hierarchy of contexts and concepts organized to model the inherent semantics and relationships of the underlying data, not its physical representation and structure.
- A given term in the CV may map to zero or more schema elements in the underlying data sources. (Zero mappings occur if a context term is a component term of a larger context and is not used independently.)
- Mapping from a given CV term to its associated schema elements in underlying data sources (field and table names) is possible using information stored in X-Specs. (X-Specs store join information and all keys, foreign keys, and their associated attributes.)

Given this environment, the goal of this work is to provide an *ad-hoc* query facility allowing users to query diverse systems by semantic context. The query processor is responsible for the necessary translation between semantics and structure and the integration and presentation of

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 2: Northwind Semantic Name Mappings

V (view root)
- [Category]
    - Id
    - Name
- [Customer]
    - Id
    - Name
- [Employee]
    - Id
    - Last Name
    - First Name

V (cont.)
- [Order]
    - Id
    - Date
    - [Customer]
        - Id
    - [Employee]
        - Id
    - [Product]
        - Id
        - Price
        - Quantity
    - [Shipper]
        - Id

V (cont.)
- [Product]
    - Id
    - Name
    - [Supplier]
        - Id
    - [Category]
        - Id

V (cont.)
- [Shipper]
    - Id
    - Name
- [Supplier]
    - Id
    - Name

Figure 3: Northwind Integrated View

results as required. In this paper, we examine query generation for a context view constructed from a single database.

2.1 Previous Work

The ability to extract structured information is fundamental to database use. The SQL standard [10] allows users to query different database platforms using one language. This provides a degree of interoperability between systems and prevents users from learning query languages for each database platform. SQL provides an efficient and structured way for accessing relational data. However, specifying complex SQL queries with numerous join conditions and subqueries is too complex for most users [3]. 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 on database structure is often complex.

There are graphical query tools [7, 6, 30] to aid in the formulation of SQL queries and proposed extensions to the SQL language [31]. Many commercial databases use similar tools to aid the user in query construction. However, at the lowest level, a user is still responsible for mapping the semantics of their query into a structural representation suitable for the database. Unfortunately, this semantics-to-structural mapping is non-trivial, especially in large databases.

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 [22] and its successor IDL [17] 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. Metadata includes database names, relational names, and attribute names. The language allows queries across database systems in addition to regular relational expressions. Other MDBS query languages include DIRECT [24] and SchemaSQL [12].

The fundamental weakness in multidatabase query languages is the reliance on the user's knowledge of the database structure and semantics to construct queries. Further, data organization is optimized for efficiency not understanding. 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. Although IDL allows for the construction of multidatabase queries, it does nothing to reduce the need for the user to thoroughly understand the semantics.

To simplify querying, systems [27, 8, 15] have been developed which allow users to query by word phrases. These systems are not powerful enough for a general multidatabase environment because they do not allow the user to precisely define the exact data returned. Unlike a SQL query which is deterministic and precise, query by word systems which simplify query formulation by ignoring structure sacrifice query precision. Other systems which augment a relational database with logical rules or knowledge [18, 26] or change or add to the database in some manner to enable advanced queries to be posed violate database autonomy and thus are not desirable. In a general environment, a query system must isolate the user from structure and system details while at the same time provide a query language powerful enough to produce precise, formatted results. SemQL [20] attempts semantic querying using semantic networks and synonym sets from WordNet [25]. Although their approach is similar to ours, using a large online dictionary such as WordNet increases the complexity of matching word semantics. Also, since no integrated view is produced, it is not clear to the user which concepts are present in the databases to be queried. Our approach improves on SemQL by providing a condensed term dictionary, an integrated view to convey database semantics to the user, and a systematic method for SQL generation.

A fundamental database model is the Universal Relational Model which 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 [23], 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 [5], and more generally in the theory of joins [1] and querying [29, 16].

It is also important to distinguish our integration architecture from wrapper and mediator systems. Mediator systems either assume an integrated view of the data sources is constructed *a priori* by designers or do not construct an integrated view. If an integrated view is constructed, it is a conventional, structural organization of the data into relations and attributes. This integrated view is then mapped to the local views of the mediators by logical rules or query expressions specified by the designer. Thus, these systems achieve database interoperability by providing an integrated view and its associated mappings to local systems, then automatically process a query specified on the integrated view into queries on the individual data sources. Numerous such systems [9, 14, 11, 21, 4, 2, 13, 28] have been developed.

Mediator systems do not perform schema integration. Schema integration, or the actual construction of the integrated view, is performed manually by designers. Our work is unique in that it automatically produces an integrated view from data source specifications developed independently of other data sources and the global view itself. Thus, the scalability of the system is improved. The integrated view does not display structure and organization to the user. Rather, it hides as much structural information from the user as possible and displays information in a semantically intuitive hierarchy of contexts and concepts. Since the integrated view is no longer queried by structure, the new 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) produced by the integration architecture models database schema knowledge as a hierarchy of contexts and concepts. In this section, we more formally describe the nature of the CV and its relationship to the Universal Relation. First, it is necessary to define the concepts of a standardized dictionary term, a semantic name, and the context view.

A *dictionary term* is a single, unambiguous word or word phrase present in the standardized dictionary of terms. Each term represents an 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 used in a semantic name which describes the context (entity or relationship) of the schema element associated with the semantic name. A *concept term* is a single dictionary term used in a semantic name which provides the lowest level semantic description of a database field. Basically, a concept is a semantic name which maps to a database field. A context is a semantic name which maps to a database table. For example, the semantic name `[Category] Id` is a concept because it maps to the database field *CategoryID*. The semantic name `[Category]` is a context because it maps to the database table *Categories*.

A *semantic name* S_i consists of an ordered set of dictionary terms $T = T_1, T_2, ...T_N$ where $N >= 1$ which uniquely describe the semantic connotation of a schema element. If $N = 1$, then T_1 is a context term. 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.

A semantic name is a hierarchy of contexts each of which have a meaning independent of the semantic name. When integrating semantic names into a context view, it is necessary to match semantic names based on their associated terms. For this purpose, it is useful to define the *context closure* of a semantic name.

Definition. The *context closure* of a semantic name S_i denoted S_i^* is the set of semantic names produced by extracting and combining ordered subsets of the set of terms $T = T_1, T_2, ...T_N$ of S_i starting from T_1. ■

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

With the preceeding definitions, we now 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 which associate a schema element E_j with S_i.

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

That is, for every semantic name that exists in the context view, all its associated semantic names formed by taking a subset of its terms are also in the context view. Each semantic name in the view can be mapped to physical fields and tables by the set of mappings provided by the system.

The integration architecture combines schema elements into the context view by merging their associated semantic names with the semantic names currently present in the CV. Matching proceeds term-wise until a complete match is found 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 full semantic name S_i consisting of one or more dictionary terms $T_1, T_2, ...T_m$. When a node is added, each of its corresponding terms are recursively added starting at the root.

3.1 Context View as a Universal Relation

There is an underlying similarity between a context view and an Universal Relation. An Universal Relation (UR) contains all the attributes of the database where each attribute has a unique name and semantic connotation. Although further extending assumptions on the Universal Relation are proposed by Maier *et al.* [23] related to access paths and attribute interrelationships, the fundamental feature of the UR is that all attributes are uniquely named with a unique 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. ■

Essentially, the automatic construction of a context view by combining semantic names builds an Universal Relation describing a data source. However, the Universal Relation is modeled as a hierarchy of contexts which serve to semantically subdivide the attributes of the relation.

Although a given semantic name occurs only once in a CV, it is entirely possible that there is more than one mapping to physical fields in even a single data source. For example, consider the Northwind database with tables *Orders* and *OrderDetails*. The field *OrderID* in both tables is assigned the exact same semantic name `[Order] Id`. This makes sense because each field has the same semantic connotation and is only represented as two fields due to the normalization of the tables. When these two tables are combined into a UR, only one instance is retained. However, the query system must decide on the correct and more efficient mapping when generating query access plans.

A context view examined as an Universal Relation addresses several of the problems of the UR model. First, the context view is automatically created by the system when the semantics of the database are systematically described by the DBA. In the construction of the semantic names, the DBA uniquely defines the semantics and name for each field. The system then uses the supplied semantics, schema and join information to automatically build the context view. As we will demonstrate, this process can be applied in reverse to extract query results from normalized database tables given a query expressed on the context view.

The context view also resolves the issues of large and complex Universal Relations. Since the context view is organized hierarchially by context, there is an explicit division of the context view into semantically grouped topics as opposed to one, flat relation containing all attributes. This reduces the semantic burden on the user when selecting query fields.

The context view is more than an Universal Relation. It is a hierarchially organized, integrated view of database knowledge in one or more systems. It is designed for easy integration of databases by capturing the semantics of their schema elements. Unlike a strict Universal Relation implementation, the context view is never physically constructed. Rather, like a view, it is an amalgamation of data stored in other structures which is built as needed. Thus, the focus of the rest of this paper is demonstrating how queries posed through the context view can be physically realized by an automatic algorithm which maps from semantics to structure and produces relational calculus (SQL) expressions on the underlying data source 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 most important property the query system must provide the user is consistency. The system must generate deterministic, repeatable, and semantically intuitive queries in all cases.

Given a context view (such as the integrated view of the Northwind database in Figure 3), users generate a query which contains a subset of its concepts. The selection criteria and result fields are then graphically selected by the user from the context view. Since a query is just a subset of the context view, the query can be examined similar to a context view.

After a query has been formulated by selecting the semantic names of the appropriate concepts and contexts, the query system is responsible for mapping the semantic query into a structural query for the underlying database. In this section, we present algorithms for mapping a semantic query to relational calculus and SQL.

There are two major requirements in mapping from semantic to structural querying. First, the system must select the appropriate fields to use for projection and selection. Since multiple mappings to the same semantic name are possible within a given data source, the system must select the most appropriate field mapping for each semantic name. The query result may be slightly different for different mappings to the same semantic name because new joins may be introduced if the field is in another table. Second, the join conditions must be automatically determined to combine the appropriate data source tables.

4.1 Determination of Data Source Fields and Tables

The query system determines which physical tables and fields to access in the data source based on the semantic names chosen by the user. In most cases, a semantic name in the integrated view has only one mapping to a physical field. However, in special cases, especially when considering key fields, a semantic name may map to several physical fields. Since the choice of field (and its parent table) may affect the semantics of the query, the query system must have well-defined rules which are logical and easily conveyed to the user.

Semantic names are selected by the user for display in the final result (projection) or for specifying selection criteria. Regardless, if the field is being used in a selection or projection operation, all fields are treated uniformly by the query system.

Determining the correct field instance to select if a given semantic name can be mapped to multiple fields in the underlying database is complex. Fortunately, it is unlikely that a semantic name has multiple field mappings when the database is normalized if the field is not a key field. However, the choice of a key field with multiple mappings is especially important as it affects the join semantics. Depending on the field mapping chosen, different tables are joined together. For example, the semantic name `[Order] Id` maps to two physical fields: *OrderID* in the *Orders* table and *OrderID* in the *OrderDetails* table. In both cases, the field has the same semantics. However, depending on which of the two mappings is selected, a new join may be introduced into the query if the table is not currently in the query.

For a key field occurring in more than one database table, there are four cases to consider based on the interrelationships between the parent tables for field mappings. That is, if the key field is present in two or more tables, the inherent interrelationships between these tables determine the complexity in selecting the correct mapping. These cases are presented below using examples from the Northwind database.

- **1-1** - An one-to-one relationship between tables normally implies that the tables share some key. For example, assume the government has one large record on every person indexed by social security number. Then, semantic names of `[Person] SSN` and `[Record] SSN` are defined. If the 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 in Northwind 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 the tables *Orders* and *OrderDetails*. Since, an *OrderDetail* record cannot exist without an *Order* record, the *OrderDetails* table will have as part of its key the key for the *Orders* table. Both fields are assigned the semantic name `[Order] Id`. In this case, there are actually two field mappings to the same 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 will result in multiple field mappings to a single semantic name because the relationship is structured by constructing a joining table whose key is the combination of the keys of the two related tables. Consider, a database storing information on books and authors. Since a book may have multiple authors and an author may write multiple books, a joining table *BookAuthor* (`[Book;Author]`) is necessary to implement the M-N relationship between books and authors. However, this table effectively "disappears" into the integrated view. The *BookAuthor* table has mappings to both the Book table (`[Book] Id`) and Author table (`[Author] Id`) keys. The query system must select the appropriate instance.

There is one other special case when a semantic name may have multiple field mappings. When a database is not normalized, multiple fields in a single table may map to a semantic name. For example, if an order has up to three items all stored in the order table, then the semantic name `[Order;Product] Id` will have three mappings in the table. 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 which is currently present in the tables already in the query. Otherwise, it chooses the field whose parent table context matches the field context. This is done to identify the most logical semantic choice for the field. Presumably, this identifies the most common occurrences of the field and often is the primary key of the parent table. Finally, the system takes the first field mapping encountered if no other heuristic applies. The algorithm (see Figure 4) presented constructs a set of fields (F) and tables (T) which best map to the set of query nodes $Q = Q_1, Q_2, ...Q_n$ given by the user.

4.2 Determining Join Conditions

Given a set of fields and tables to access in the query, the query system must determine a set of join conditions between the tables. It is important to isolate the user from join construction while choosing appropriate joins to preserve the semantics of the user's query.

Define a *join graph* as an undirected graph where each node corresponds to 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 one or more joins interconnecting two nodes (tables) in the graph, and a *join tree* is a set of one or more joins interconnecting two or more nodes. Assume without loss of generality that the join graph is connected. (Otherwise, we apply the algorithm to each

```
For each term $Q_i$ in Q                                                        (1)
    $SN_i$ = semantic name of $Q_i$                                             (2)
    search_XSpec($X_j$,$SN_i$,num,R) // Search X-Spec for SN. Return results in R.   (3)

    If num = 1 // Only one occurrence of semantic name                          (4)
        Add field $R_k$ to F                                                    (5)
        Add parent table of $R_k$ to T                                          (6)
    Else                                                                        (7)
        If multiple occurrences but only in one table Then                      (8)
        For each result $R_k$ in R                                              (9)
            Add field $R_k$ to F                                                (10)
        Next                                                                    (11)
        Add parent table of $R_1$ to T                                          (12)
        End if                                                                  (13)
    End if                                                                      (14)
Next                                                                            (15)

// Second pass to resolve multiple occurrences
For each term $Q_i$ in Q                                                        (16)
    $SN_i$ = semantic name of $Q_i$                                             (17)

    If $Q_i$ has not been mapped                                                (18)
        search_XSpec($X_j$,$SN_i$,num,R) // Search X-Spec for SN. Return results in R.   (19)

        If Find any mapping $R_k$ of R with parent table $T_j$ already in T Then   (20)
            Add field $R_k$ to F                                                (21)
            Add parent table of $R_k$ to T                                      (22)
        ElseIf Find parent table $T_j$ of $R_k$ with context portion = context portion of $Q_i$ Then   (23)
            Add field $R_k$ to F                                                (24)
            Add parent table of $R_k$ to T                                      (25)
        Else                                                                    (26)
            Add field $R_1$ to F // Otherwise, add first mapping                (27)
            Add parent table of $R_1$ to T                                      (28)
        End if                                                                  (29)
    End if                                                                      (30)
Next                                                                            (31)
```

Figure 4: Field Selection Algorithm

connected subset and connect them using a cross-product.) The join graph for the Northwind database is in Figure 5.

Lemma 1. *If a join graph is acyclic, there exists only one join path between any two nodes.*

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_j to N_i. This implies that the graph has a cycle. ■

Lemma 2. *If a join graph is acyclic, there exists only one join tree between any subset of its nodes.*

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. If the join graph for a database is acyclic, there exists only one possible join tree for any of its tables. This implies that the query system does not have any decisions involving which joins to apply. We must only identify which joins are required to connect the required tables by constructing the join tree. The actual order in which the joins are applied is the join optimization problem which has been actively studied and will not be discussed here.

From this result, it is possible to construct an algorithm which 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 can identify all the joins required to combine database tables by constructing the only possible join tree. The general algorithm is presented later in the section.

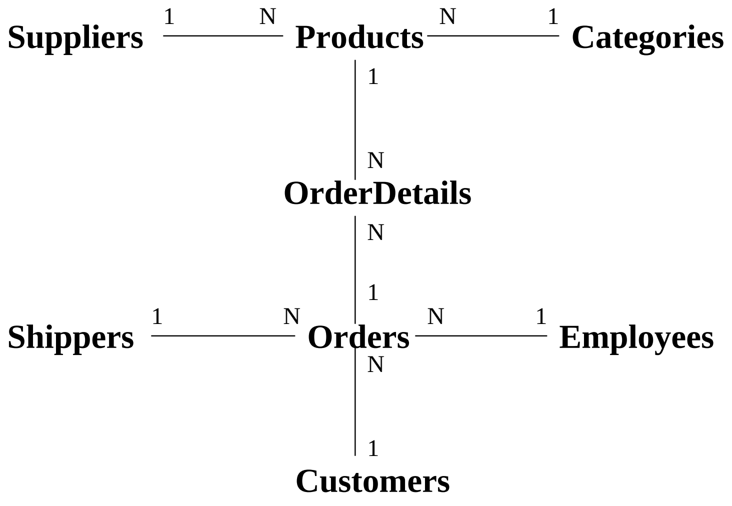

Figure 5: Join Graph for Northwind Database

Theorem 1. *Given a subset of nodes from matrix M, which stores 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 node N_i to N_j and from node N_i to N_k. Unioning these paths together results in 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. However, the general case of a cyclic join graph must be handled. A cyclic join graph arises typically when joins are added for query convenience and when tables serve multiple semantic roles in a database. A given table can assume multiple semantic roles in several ways. One way is by acting as a lookup table for several other tables. For example, assume the Northwind database (see Figure 6) also stored information on the employee who entered each order product in addition to what employee entered the overall order. In this case, *Orders* and *OrderDetails* have foreign keys to the *Employees* table storing the employee who entered the record. This produces a cycle between *Orders*, *OrderDetails*, and *Employees*. Notice that the join path chosen between the tables represent different semantic queries. The join path *Orders-Employees-OrderDetails* represents the orders entered by employee with their products; *Orders-OrderDetails-Employees* represents the orders with their products grouped by the employee entering the product; *Employees-Orders;Employees-OrderDetails* represents which employees entered both an order product and its overall accompanying order.

A second instance of cyclic join graphs appears when a table stores a generalized concept which may have multiple subconcepts. Using Northwind, assume the *Shippers* and *Suppliers* table are combined into one table called *Companies*. Effectively, the *Companies* table stores the general notion of a company and the specific types of companies: shippers and suppliers. Cycles occur when tables join to the different semantic instances (shipper,supplier) in the *Companies* table.

Finally, cycles occur when redundant joins are added to the database. For example, the *CategoryID* field could be added to *OrderDetails* for a direct link to *Categories* instead of joining through *Products*. This results in a cycle involving *OrderDetails*, *Products*, and *Categories*. Note that joins of this nature may be lossy when used in combination with other, valid lossless joins. An invalid lossy sequence of joins contains a join with a N-1 cardinality followed by a join with a 1-N cardinality. There may be other joins between these two joins. The result is a lossy join

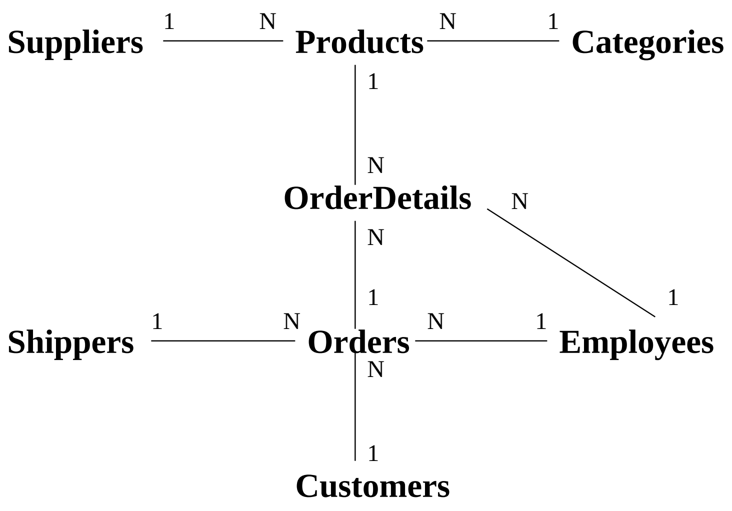

Figure 6: Cyclic Join Graph for Northwind Database

because it results in a M-N cardinality relationship between the merged tables. Effectively, this results in invalid information being created by using these joins. Also, a join of cardinality M-N between two tables is always lossy. Thus, the algorithm first attempts to find join paths without using these types of lossy joins.

To handle cycles the query system must make a determination of the best join paths between nodes. The query system uses join semantics, path length, and join properties (total participation, lossless vs. lossy) to determine best join paths. The breadth-first search based algorithm presented constructs the matrix M of best join paths. It works for both cyclic and acyclic join graphs. The algorithm selects the shortest join paths with no lossy joins (see Figure 7). Equal length join paths may be differentiated based on total participation or other join properties. Lossy joins are only used if there exists no other path between nodes (a cross-product would be necessary).

It would be ideal if we could use the previous algorithm of unioning join paths in the matrix to produce a join tree between any subset of nodes. However, if the graph is cyclic, there will be multiple join trees possible depending on the choice of starting node. These join trees are all semantically valid depending on the query. The system cannot differentiate for the user without more knowledge about the intended query semantics. Although it may be possible to define heuristic algorithms to choose the correct join tree based on the attributes chosen for the query, it is more desirable to have a precise mechanism for the user to exploit. Thus, we define query extensions to the original model which allow the user to more precisely define the semantics of the query such that the system can uniquely determine the join tree required.

Given the set of database 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 type query.

5 Query Extensions

To enable the user to more precisely define the semantics of their query, extensions to the integrated view are possible. The extensions allow the user to more accurately convey the semantics of their query or to override the system default semantics. The first extension is to allow the user to pick the root join table. Essentially, this gives the user the ability to choose which row in the join matrix to use. Semantically, the root join table chosen by the user is the starting point of all join paths. This allows the system to unambiguously construct a join tree which matches the users intended query semantics.

The second optimization is an enhancement of the integrated view presentation. Currently,

```
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

integer count
node F, N, LTN
link L
bool accept_lossy
integer join_type // 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:                                                               (5)
        add F to NQ                                                         (6)

        While NQ is not empty                                               (7)
            remove first node N from NQ                                     (8)
            For each outgoing link L of N                                   (9)
                LTN = destination node of link L from N                     (10)
                join_type = cardinality of join for L (from N to LTN)       (11)
                If LTN is not visited and (accept_lossy
                        or not ((M[F,N] has a N-1 join and join_type = 1 - N)) or join_type = M - N) Then   (12)
                    add LTN to NQ                                           (13)
                    mark LTN as visited                                     (14)
                    M[F,LTN] = M[F,N] + LTN                                 (15)
                    count++                                                 (16)
                ElseIf accept_lossy or not ((M[F,N] has a N-1 join and join_type = 1-N))
                    or join_type = M - N) Then                              (17)
                        // May want to replace a join path already constructed (M[F,LTN]) if
                        // - new join path is the same length as current one and
                        // - new join path has better properties (eg. total participation)
                Endif                                                       (18)
            Next                                                            (19)
        End while                                                           (20)

        clear_flags() // Clear all visited flags for all nodes in G         (21)

        If count ¡ # of nodes in G and accept_lossy = false Then            (22)
            accept_lossy = true                                             (23)
            Goto repeat_label // Repeat algorithm accepting all joins (even lossy)   (24)
        Endif                                                               (25)
Next                                                                        (26)

// Note: For any matrix entries not assigned no join path exists and need cross-product to join tables
```

Figure 7: Algorithm to Calculate Join Paths

the integrated view consists of a hierarchy of contexts and concepts. The join conditions relating the individual concepts are largely hidden to the user. However, these join conditions are automatically inserted by the query system as required. To make these interrelationships more apparent, the system can automatically display them. If a given semantic name (node) in the integrated view is actually a foreign key to another concept (table) then when the user clicks on this concept, the attributes of the linked concept are displayed.

For example, in Northwind the field *EmployeeID* in *Orders* has a semantic name `[Order;Employee] Id` corresponding to the foreign key from *Orders* to *Employees*. When the user clicks on this semantic name, the system automatically performs the join to the *Employees* table and displays to the user the fields of *Employees* (*EmployeeID*, *LastName*, *FirstName*) which can be added to the query.

This approach has several benefits. First, it reduces the semantic burden on the user by automatically displaying concept interrelationships. More importantly, it 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 (attribute) to use to relate the two different contexts.

For example, if the users selects *LastName* field for inclusion into the query directly from the *Employees* table, it may be ambiguous how to join *Orders* and *Employees* if there are multiple join trees. (This is not a problem in the original version of Northwind but could be an issue in other cases as discussed.) However, if the user selects the mapping to *LastName* which results from expanding the join through `[Order;Employee] Id`, the system and the user now know the exact join path.

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 simple but powerful method for constructing relational calculus expressions from semantic contexts provided in an integrated view. The query system is capable of handling complex join constructs and choosing the appropriate field, 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] A.V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database systems*, 4(3):297–314, September 1979.

[2] M. Barja, T. Bratvold, J. Myllymaki, and G. Sonnenberger. Informia: A mediator for integrated access to heterogeneous information sources. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM-98)*, pages 234–241, New York, November 3–7 1998. ACM Press.

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

[4] S. Bressan, C. H. Goh, K. Fynn, M. Jakobisiak, K. Hussein, H. Kon, T. Lee, S. Madnick, T. Pena, J. Qu, A. Shum, and M. Siegel. The COntext INterchange mediator prototype. *SIGMOD Record*, 26(2):525–527, May 1997.

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

[6] T. Catarci and G. Santucci. Query by diagram: A graphical environment for querying databases. *SIGMOD Record*, 23(2):515–515, June 1994.

[7] J. Chen, A. Aiken, N. Nathan, C. Paxson, M. Stonebraker, and J. Wu. Extending a graphical query language to support updates, foreign systems, and transactions. Technical Report S2K-93-38, University of California, Berkeley, 1994.

[8] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual/similarity. *SIGMOD Record*, 27(2):201–212, 1998.

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

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

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

[12] F. Gingras, L. Lakshmanan, I. Subramanian, D. Papoulis, and N. Shiri. Languages for multidatabase interoperability. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 536–538, May 13–15 1997.

[13] J.R. Gruser, L. Raschid, M. Vidal, and L. Bright. Wrapper generation for WEB accessible data sources. In *6th Int. Conf. on Cooperative Information Systems*, pages 14–23, New York, 1998.

[14] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*, 1995.

[15] D. Konopnicki and O. Shmueli. Information gathering in the World-Wide Web: The W3QL query language and the W3QS system. *ACM Transactions on Database Systems*, 23(4):369–410, December 1998.

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

[17] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):40–49, June 1991.

[18] E. Kuhn, T. Tschernko, and K. Schwarz. A language based multidatabase system. *SIGMOD Record*, 23(2):509, June 1994.

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

[20] J.O. Lee and D.K. Baik. SemQL: A semantic query language for multidatabase systems. In *Proceedings of the 8th International Conference on Information Knowledge Management CIKM'99*, pages 259–266, Kansas City, MO, November 1999.

[21] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability based mediation in TSIMMIS. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 564–566, June 1998.

[22] W. Litwin and A. Abdellatif. An overview of the multidatabase manipulation language MDSL. In *Proceedings of the IEEE*, pages 69–73, May 1987.

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

[24] Ulla Merz and Roger King. DIRECT: A query facility for multiple databases. *ACM Transactions on Information Systems*, 12(4):339–359, October 1994.

[25] G.A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Five papers on WordNet. Technical Report CSL Report 43, Cognitive Systems Laboratory, Princeton University, 1990.

[26] A. Motro and Q. Yuan. Querying database knowledge. *SIGMOD Record*, 19(2):173–183, June 1990.

[27] W. Ogden and S. Brooks. Query languages for the casual user: Exploring the ground between formal and natural languages. In *Proc. Annual Meeting of the Computer Human Interaction of the ACM*, pages 161–65, 1983.

[28] M. Roth and P. Schwarz. Don't scrap it, wrap it! A wrapper architecture for legacy data sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 266–275, 1997.

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

[30] M. Stonebraker, J. Chen, N. Nathan, C. Parson, A. Su, and J. Wu. Tioga: A database-oriented visualization tool. In Gregory M. Nielson and Dan Bergeron, editors, *Proceedings of the Visualization '93 Conference*, pages 86–93, San Jose, CA, October 1993. IEEE Computer Society Press.

[31] G. Vossen and J. Yacabucci. An extension of the database language SQL to capture more relational concepts. *SIGMOD Record*, 17(4):70–78, December 1988.