# Querying Relational Databases without Explicit Joins*

Ramon Lawrence[1] and Ken Barker[2]

[1] University of Iowa, Iowa City, IA, USA
ramon-lawrence@uiowa.edu
[2] University of Calgary, Calgary, Alberta, Canada
barker@cpsc.ucalgary.ca

**Abstract.** Despite its benefits and wide-spread acceptance, SQL [5] is not a perfect query language. Although graphical tools for query construction mask some of the complexity, complex database schema challenge even experienced database users during query formulation because a user is responsible for mapping the semantics of their query to the structure of the database. In this work, we propose a semantic query language for graphically querying relational database systems that allows a user to query the database by semantics instead of structure. Database semantics are described using a global dictionary and semantic specifications that are combined to form an integrated, context view. Users query the semantic view by concept name, and the query processor translates semantic queries to SQL. This translation involves automatically determining attribute and relation mappings and join conditions.

## 1 Introduction

Despite changes in database size, complexity, and interoperability, SQL [5] has remained fundamentally unchanged. Although the complexity of SQL generation has been partially hidden by graphical design tools and more powerful programming languages, 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.

To simplify querying, we have implemented a new semantic query system. First, a database administrator semi-automatically produces a semantic view for querying using a global dictionary and a semantic specification called an X-Spec. A description of the capture process is in Section 2. Then, users graphically browse the integrated view to formulate queries. The query environment guides the user during query formulation allowing the user to unambiguously define the semantics of their query without explicitly referencing database structures or joins between relations. The query processor described in Section 3 is capable

* This research is sponsored by NSERC Research Grant (RGP-0105566) and TRLabs.

of automatically translating from the semantic context view to SQL queries by determining correct attribute and relation mappings and appropriate join conditions. The paper closes with related work and conclusions.

## 2 Semantic Capture Process

Before semantic querying can be performed on a relational database, the database administrator must explicitly define its semantics in a specification document. In this *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 an XML [13] document called an X-Spec. An X-Spec provides information for semantic querying.

An example order database (given in Figure 1) is used throughout the paper. We denote attribute and relation names as *attribute_name* or *relation_name*, and semantic names as `semantic_name`.

| **Relations** | **Attributes** |
|---|---|
| Categories | CategoryID, CategoryName |
| Customers | CustomerID, CompanyName |
| Employees | EmployeeID, LastName, FirstName, ReportsTo |
| OrderDetails | OrderID, ProductID, UnitPrice, Quantity |
| Orders | OrderID, CustomerID, EmployeeID, OrderDate, Shipvia |
| Products | ProductID, ProductName, SupplierID, CategoryID |
| Shippers | ShipperID, CompanyName |
| Suppliers | SupplierID, CompanyName |

**Fig. 1.** Order Database Schema

### 2.1 Global Dictionary Definition

The foundation of the query system is the definition of a global term dictionary that provides terms to represent concept semantics. The dictionary is organized as a hierarchy of concept terms. Concept terms are related using 'IS-A' relationships for modeling generalization and specialization and 'HAS-A' relationships to construct component relationships. Each concept term has a name, an unique key, and an unique definition.

We have built a term dictionary starting from the top-level ontological categories proposed by Sowa [10], and evolve it using a set of rules. However, the exact terms and their placement is irrelevant. The dictionary can be modified to capture database semantics for a particular environment. The global dictionary is not a complete English language dictionary such as WordNet [8], so its size and complexity are reduced.

### 2.2 Semantic Names

A *semantic name* captures system-independent semantics of a relational schema element including contextual information by combining one or more dictionary terms. A semantic name is a **context** if it is associated with a relation and a **concept** if it is associated with an attribute. A context contains no data itself and is described using one or more concepts. Similar to an attribute in the relational model, a concept represents atomic or lowest-level semantics.

A semantic name consists of a context and concept portion. The **context portion** is one or more terms from the dictionary that describe the context of the schema element. Adjacent context terms are related by either IS-A (represented using a ",") or HAS-A (represented using a ";") relationships. The **concept portion** is a single dictionary term called a concept name and is only present if the semantic name is a concept (maps to an attribute). The formal specification of a semantic name is as follows:

$semantic_name ::= [CT_Term] \mid [CT_Term]\ CN$

$CT_Term ::= CT \mid CT\ ; CT_Term \mid CT\ , CT_Term$

$CT ::= <\mathit{context\ term}>,\ CN ::= <\mathit{concept\ name}>$

A *dictionary term* is a single, unambiguous word or word phrase present in the standard 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 used in a semantic name that describes the context of the schema element associated with the semantic name. A *concept term* is a single dictionary term used in a semantic name that provides the lowest level semantic description of an attribute.

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

*Example* If $S_i = [A; B; C]D$, $S_i^* = \{[A], [A; B], [A; B; C], [A; B; C]D\}$.

Overall, a semantic name is a combination of dictionary terms constructed by the DBA to represent schema element semantics. By combining terms, more complex semantics can be defined as compared to using terms in isolation.

### 2.3 X-Specs

The X-Spec is a database schema encoded in XML format and is organized in relational form with relations and attributes as basic elements. An X-Spec stores a relational database schema including keys, relationships, joins, and field semantics. Further, each relation and attribute in the X-Spec has an associated semantic name built using dictionary terms. Information on joins including their cardinality, join fields, and connecting tables is stored so that the query processor may identify which joins to apply during query formulation.

An X-Spec is constructed using a specification editor tool [6] during the capture process, where the semantics of schema elements are mapped to semantic names. X-Specs are constructed using XML because XML is an emerging semantic exchange standard. However, X-Specs may also be represented as formatted text files or structured binary files.

| Type | Semantic Name | System Name |
|---|---|---|
| Relation | [Category] | Categories |
| Attribute | [Category] Id | CategoryID |
| Attribute | [Category] Name | CategoryName |
| Relation | [Customer] | Customers |
| Attribute | [Customer] Id | CustomerID |
| Attribute | [Customer] Name | CompanyName |
| Relation | [Employee] | Employees |
| Attribute | [Employee] Id | EmployeeID |
| Attribute | [Employee;Name] First Name | FirstName |
| Attribute | [Employee;Name] Last Name | LastName |
| Attribute | [Employee;Supervisor] Id | ReportsTo |
| Relation | [Order;Product] | OrderDetails |
| Attribute | [Order] Id | OrderID |
| Attribute | [Order;Product] Id | ProductID |
| Attribute | [Order;Product] Price | UnitPrice |
| Attribute | [Order;Product] Quantity | Quantity |
| Relation | [Order] | Orders |
| Attribute | [Order] Id | OrderID |
| Attribute | [Order;Customer] Id | CustomerID |
| Attribute | [Order;Employee] Id | EmployeeID |
| Attribute | [Order] Date | OrderDate |
| Attribute | [Order;Shipper] Id | Shipvia |
| Relation | [Product] | Products |
| Attribute | [Product] Id | ProductID |
| Attribute | [Product] Name | ProductName |
| Attribute | [Product;Supplier] Id | SupplierID |
| Attribute | [Product;Category] Id | CategoryID |
| Relation | [Shipper] | Shippers |
| Attribute | [Shipper] Id | ShipperID |
| Attribute | [Shipper] Name | ShipperName |
| Relation | [Supplier] | Suppliers |
| Attribute | [Supplier] Id | SupplierID |
| Attribute | [Supplier] Name | SupplierName |

**Fig. 2.** Order Database Semantic Name Mappings

X-Spec information at the relation level stored in $X_T = (T_{SN}, T_Q, T_K, T_J)$ records consists of the relation semantic name $T_{SN}$, the SQL query $T_Q$ used to produce the relation/view, $T_K$ the set of keys for the relation, and $T_J$ the set of joins for the relation. For a view $T_Q$ is the SQL query executed to produce the view, or $T_Q$ is the relation name for physical relations.

Attribute level information in $X_F = (F_{SN}, F_Q, F_T, F_S)$ stores the attribute semantic name $F_{SN}$, the SQL query used to produce the attribute $F_Q$, the attribute type (integer, string, etc.) $F_T$, and the attribute size in bytes $F_S$. $F_Q$ is the system name for a physical attribute (e.g. `[OrderDetails].OrderID`) or a SQL statement used to construct a derived or calculated attribute.

The primary difference between an X-Spec and a relational schema is that each relation and attribute are assigned meaningful semantic names. Further, the model allows virtual constructs such as views and calculated attributes to be assigned semantic names. This allows a DBA to build more complex integrated views that consist of physical and virtually-derived constructs.

### 2.4 Semantic View Construction

An integration process formats the database X-Spec information into an integrated view of concepts by matching semantic names. Since a semantic name is itself a hierarchy of terms, the resulting integrated context view (CV) is a merger of all concepts across all attributes and relations in the database.

**Definition.** 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 that associate a schema element with $S_i$.
- A semantic name $S_i$ can only occur in the $CV$ once.

The integration process builds $CV = (CV_{SN}, CV_Q)$, where $CV_{SN} \in \{T_{SN} \cup F_{SN}\}^*$ and $CV_Q \in \{T_Q \cup F_Q\}$. That is, the $CV$ is the union of the semantic names for all relations and attributes along with their associated mappings. The $CV$ is a structurally-neutral, hierarchical representation of database concepts. The integrated view for the order database[1] is in Figure 3.

## 3 Query Processing

Users formulate queries by manipulating elements in the context view in a process called *querying by context*. The context view provides physical and logical access transparency. The user is not responsible for determining schema element mappings or joins between relations in the database.

A semantic query $Q = (Q_{SN}, Q_R, Q_O, Q_G, Q_C)$ where $Q_{SN}$ is the set of semantic names selected in the query, $Q_R$ is the set of relationship conditions between semantic names, $Q_O$ is the set of ordering criteria for the row results, $Q_G$ is the set of grouping criteria, and $Q_C$ is the set of selection criteria. Ordering and selection criteria are specified using semantic names rather than system names. Thus, for a given $Q$, a translation is required to a SQL query of the form $SQ = (SQ_T, SQ_F, SQ_J, SQ_O, SQ_G, SQ_C)$, that contains the required relation set ($SQ_T$), attribute set ($SQ_F$), joins ($SQ_J$), ordering criteria ($SQ_O$), grouping criteria ($SQ_G$), and selection criteria ($SQ_C$).

The query processor performs this translation during **dymanic view creation** that extracts relevant data from the database and requires the system:

- **Enumerate Semantic Names** - required by the user.
- **Determine Relevant Attributes and Relations** - for each semantic name mapping.
- **Determine Join Conditions** - to connect the relations.
- **Generate and Execute SQL Queries** - created in the previous steps.

[1] For simplicity, only the term at a given level is displayed. For example, the term in bold has full semantic name `[Order;Product] Id` as `Id` is under the `[Product]` and `[Order]` contexts.

| Integrated View Term | Data Source Mappings (not visible to user) |
|---|---|
| **V (view root)** | **N/A** |
| - [Category] | Categories |
| - Id | Categories.CategoryID |
| - Name | Categories.CategoryName |
| - [Customer] | Customers |
| - Id | Customers.CustomerID |
| - Name | Customers.CompanyName |
| - [Employee] | Employees |
| - Id | Employees.EmployeeID |
| - [Name] | |
| - First Name | Employees.FirstName |
| - Last Name | Employees.LastName |
| - [Supervisor] | |
| - Id | Employees.ReportsTo |
| - [Order] | Orders |
| - Id | Orders.OrderID, OrderDetails.OrderID |
| - Date | Orders.OrderDate |
| - [Customer] | |
| - Id | Orders.CustomerID |
| - [Employee] | |
| - Id | Orders.EmployeeID |
| - [Product] | OrderDetails |
| **- Id** | **OrderDetails.ProductID** |
| - Price | OrderDetails.UnitPrice |
| - Quantity | OrderDetails.Quantity |
| - [Shipper] | |
| - Id | Orders.Shipvia |
| - [Product] | Products |
| - Id | Products.ProductID |
| - Name | Products.ProductName |
| - [Supplier] | |
| - Id | Products.SupplierID |
| - [Category] | |
| - Id | Products.CategoryID |
| - [Shipper] | Shippers |
| - Id | Shippers.ShipperID |
| - Name | Shippers.ShipperName |
| - [Supplier] | Suppliers |
| - Id | Suppliers.SupplierID |
| - Name | Suppliers.SupplierName |

**Fig. 3.** Integrated View of Order Database

### 3.1 Enumerating Semantic Names

Semantic names are selected by the user for display in the final result (projection) or for specifying selection criteria. In some cases, a real-world concept may be represented multiple times in the database and possibly with different semantic names. The first task for the user is to determine the correct semantic name for querying. For example, in the example database there are two semantic names that map to the concept of a "Shipper ID": *ShipperID* (`[Shipper] Id`) in *Shippers* and *Shipvia* (`[Order;Shipper] Id`) in *Orders*. The choice of semantic name depends on the query requirements. Selecting `[Shipper] Id` will select all shipper ids whether or not they have transported an order, whereas `[Order;Shipper] Id` only returns shippers who have sent orders.

The user selects the concepts out of the integrated view they are interested in for their query, and may specify selection criteria, grouping criteria, and ordering

criteria using semantic names. The user also has control over the final display characteristics of each concept selected. Since the integrated view is organized hierarchically by concept, knowledge discovery is considerably simplified when browsing the integrated view.

### 3.2 Attribute and Relation Selection

The query system determines which relations and attributes 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 attribute. However, in special cases, especially when considering key attributes, a semantic name may map to several attributes. The choice of attribute (and its corresponding relation) may affect the semantics of the query. An overview of the attribute and relation selection algorithm is:

1. For each semantic name $F_{SN}$ with only one mapping, add attribute to $SQ_F$ and corresponding table to $SQ_T$.
2. For each semantic name $F_{SN}$ with multiple mappings all in one relation[2], add each attribute mapping to $SQ_F$ and the corresponding table to $SQ_T$.
3. For any semantic names remaining with ambiguous mappings, select an attribute mapping to $SQ_F$ for the semantic name if its corresponding table is already in $SQ_T$.
4. For any remaining semantic names with ambiguous mappings, select the mapping based on shortest join paths to current relations in $SQ_T$ (or other heuristics). This new mapping adds a new relation to $SQ_T$, so goto step 3.

In practice, the attribute mapping algorithm rarely uses the heuristics on join cost calculations in Step 4 because typically only key attributes will have multiple semantic name mappings that result from database normalization. When the user selects any attributes with unique mappings, the attribute selection algorithm has a starting set of relations to build a query from and determine which key mapping instance(s) to select based on the general heuristic of not introducing new joins into the query.

After completion of the attribute selection algorithm, the set of semantic names $Q_{SN}$ in $Q$ has been translated to the set of attributes $SQ_F$ and relations $SQ_T$ for the SQL query. Further, the system can replace the semantic names present in $Q_O$, $Q_G$, and $Q_C$ for ordering, grouping, and selection criteria with system attribute and relation names for $SQ_O$, $SQ_G$, and $SQ_C$ based on the discovered mappings.

### 3.3 Determining Join Conditions

Given a set of attributes and relations to access in the query, the query system must determine a set of join conditions between the relations. It is important

[2] This may occur when the database is not normalized. The system queries non-normalized records and normalizes the data when it is presented to the user.

to isolate the user from join construction while choosing appropriate joins to preserve the query semantics.

There is no concept of a "join" in the integrated view because it does not specify a structure for data representation. However, an equivalent operation is a context merger. A *context merger* is the combination of two contexts by applying a relationship condition. A relationship condition may be a join between relations if a join exists between the contexts or a cross-product if it does not.

In most cases[3], the user never directly specifies the relationship condition between two contexts in the integrated view, so the system must determine how to relate the two contexts. Since the user is able to query on any semantic name in the integrated view, the semantic concepts chosen may be related by being in the same relation, by joins between relations, or not related at all.

The system determines the joins to apply using join graphs. Define a *join graph* as an undirected graph where each node corresponds to a relation 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 relations. A *join path* is a sequence of one or more joins interconnecting two nodes (relations), and a *join tree* is a set of one or more joins interconnecting two or more nodes. Assume without loss of generality[4] that the join graph is connected. The join graph for the order database is in Figure 4.

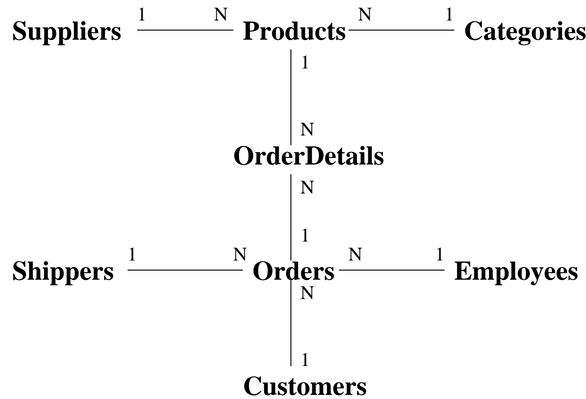


**Fig. 4.** Join Graph for Order Database

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

**Lemma 2.** *If a join graph is acyclic, there exists only one join tree for any node subset.*

[3] A user can override default system relationship conditions if the database contains multiple, ambiguous joins.

[4] Otherwise, we apply the algorithm to each connected subset and connect them using a cross-product.

Lemmas 1 and 2 follow directly from the properties of acyclic graphs. Importantly, if the join graph for a database is acyclic, there exists only one possible join tree for any of its relations. This implies that the query system does not have any decisions involving which joins to apply, as it must only identify which joins are required to connect the required tables by constructing the join tree.

From this result, it is possible to 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 can identify all the joins required to combine relations by constructing the only possible join tree.

**Theorem 1.** *Given a matrix $M$, which stores shortest join paths for an acyclic join graph, and a set of relations $T$ to join, a join tree can be constructed by choosing any relation $T_i$ from $T$ and combining 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 relations $T$.*

**Proof.** Proof by contradiction. 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 $N_k$ where $j \neq k$. However, there is a path from node $N_i$ to $N_j$ and from node $N_i$ to $N_k$. Combining these paths 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. ∎

For the general case of a cyclic join graph, we define a depth-first search based algorithm that constructs a matrix $M$ containing all join paths with no repeated links ranked by desirability. Join ranking is based on join properties such as cardinality, participation, and if they are lossy joins. Although the algorithm ranks join paths, in practice, only acyclic graphs can have their joins automatically determined without further user input by applying Theorem 1. For a cyclic graph, there will be multiple possible join trees each of which are semantically valid depending on the query semantics. The system cannot select a join tree without more knowledge about the intended query, although it is able to propose the "best" join tree based on the ranking performed. Thus, we define a graphical interface to the query model that allows the user to browse the integrated view, discover concept interrelationships, and more precisely define their query semantics which allows the system to uniquely determine the join tree required.

### 3.4 Graphically Guiding the User During Query Generation

The query system currently described is unable to determine join trees without further user input for cyclic join graphs. The challenge is to extract the required information from the user while still maintaining physical and logical query transparency as much as possible. The system allows the user to browse the context view to discover interrelationships between concepts. Some concepts are already inherently related by virtue of their hierarchical relationship. For example, the `[Order;Product] Id` is part of `[Order]` information. This relationship is explicitly captured in the integrated view by virtue of the product

information being nested under the order information in the hierarchy. However, other interrelationships cannot be captured using the strict hierarchy implicit in the semantic names. For instance, the join conditions relating the individual concepts are largely hidden to the user, even though these join conditions are automatically inserted by the query system during query execution. To make these interrelationships more apparent, the system automatically displays them. If a given semantic name (node) in the integrated view is actually a foreign key to another context (table) then when the user clicks on this concept, the attributes of the linked context are displayed.

For example, the field *EmployeeID* (`[Order;Employee] Id`) in *Orders* is a foreign key from *Orders* to *Employees*. When the user clicks on this semantic name, the system performs the join to the *Employees* table and displays the semantic names of the fields of *Employees* which can be added to the query.[5]

This approach has several benefits. First, it reduces the semantic burden on the user by automatically displaying concept interrelationships. Further, it reduces the query generation complexity for the system. By explicitly displaying the join information and associated attributes, the system now has an unambiguous reference from the user on which attributes to use, from what relations, and the corresponding join condition (attribute) to use to relate the two different contexts. Further, by allowing this virtual linking of concepts, the user is now able to express recursive relationships and queries on the integrated view.

To formalize the discussion, define a *context tree* as a tree of nodes where each node is a single semantic name, and at every node we have a list of attributes accessed from that context. The root node of the tree is $V$, the view root. There is a link from a parent node (context) in the tree to a child node, if the child node is a subcontext of the parent.

For example, let *OrderDetails* also have a foreign key *EmployeeID* to *Employees*. This produces a cycle in the join graph. Now, consider the query requiring the `[Employee] Name`, `[Order] Id`, and `[Order;Product] Amount`. The context tree for this query is given in Figure 5a. In the cyclic join graph, there are multiple possible join trees to connect these contexts. In general, if there is more than one child node of V and the graph is cyclic, there will be multiple join trees that relate the contexts.

A different query can be derived by the user explicitly following the link to employee information through `[Order;Employee] Id` (Figure 5b). The user can specify another distinct join by selecting `[Employee] Name` after traversing the foreign key in *OrderDetails* to *Employee*. In this way, the user uniquely identifies the joins required by the system by graphically browsing the context view.[6]

Linking of contexts allows the system to handle recursive queries. *Employees* has a foreign key *ReportsTo* that links to *Employees* on *EmployeeId* in order to model that an employee can have a supervisor. The system can answer a query

[5] If multiple joins are possible on a given foreign key, the system allows the user to select which context (relation) they are interested in connecting to.

[6] Advanced queries involving right or left joins can be directly specified by the user by displaying the current context tree and allowing users to manipulate it.

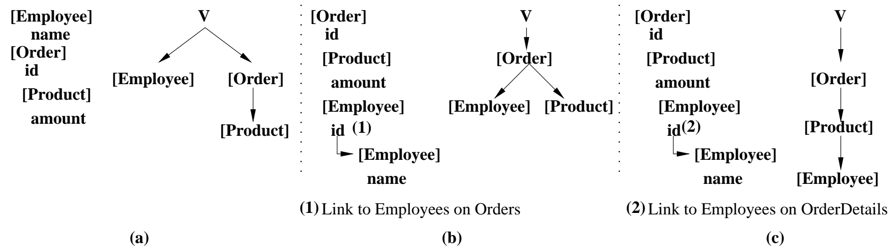


**Fig. 5.** Example Queries

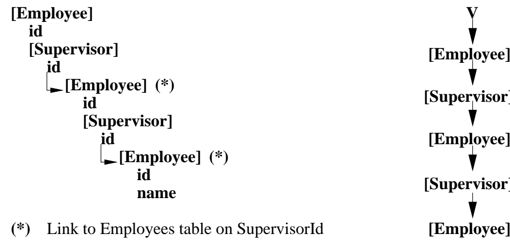


**Fig. 6.** Recursive Supervisor Query

like: "who is an employee's supervisor's supervisor?" by allowing the user to graphically traverse the context view (see Figure 6).

### 3.5 Generation and Execution of SQL Queries

Given the set of database attributes and relations to access and a set of joins to apply, it is straightforward to construct a SQL select-project-join query. After building a SQL query string, the query is transmitted to the database management system for execution. Although there are multiple open standards and proprietary protocols for each database and environment, the architecture is designed to utilize the ubiquity of the ODBC standard to access all major database systems. Results returned from each ODBC query are then processed by the client to perform global level formatting. Two example queries and their associated SQL mappings are below.

*Example 1 a)* The user requires all customer names (`[Customer] Name`).
*b)* The user requires all customer names with orders (`[Order;Customer] Name`).

| (a) | (b) |
|---|---|
| **Select** CompanyName | **Select** CompanyName |
| **From** Customers | **From** Customers, Orders |
| | **Where** Customers.CustomerID = Orders.CustomerID |

*Execution 1 a)* Retrieves customers whether or not they have an order. No context merging is required. *b)* The customer name is retrieved from the *Customers* relation. The query processor performs a context merger by combining

the `[Order]` context with the `[Customer]` context. A join is required between *Orders* and *Customers* to merge the two contexts.

## 4 Related Work

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 [1]. 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 [2, 11] to aid in the formulation of SQL queries and proposed extensions to the SQL language exist [12]. 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.

Query by example or QBE [14] was proposed to reduce this semantic burden on users. By providing a table-based query environment and a simple query syntax, QBE demonstrated that some of the burden of accessing relational databases can be mitigated by using advanced user interfaces that capture the semantics of a user's query, and automatically translate them to relational queries. Query by context (QBC) differs from query by example as QBC does not have a structural basis for the query. QBE's foundation rests on a table structure. Although QBE is effective at hiding much of the query complexity, QBC is a further advancement by hiding the structure as well. Further, this structural neutrality allows queries to be unaffected by database structural changes as long as the same semantic concepts are present, and will allow QBC to be extended to object-oriented databases and multidatabases.

Another methodology for simplifying query complexity is to hide the structure of the database by allowing some form of querying by natural language. Systems [4, 9] have been developed that allow users to query by word phrases, but these systems are limited if 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 that simplify query formulation by ignoring structure, sacrifice query precision.

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 [7] provides semantic querying using semantic networks and synonym sets from WordNet [8]. 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 con-

densed term dictionary, an integrated view to convey database semantics to the user, and a systematic method for SQL generation.

A different approach to relational querying involves using an intelligent query processor that directs the user during query formulation. Kaleidoscope [3] provides a logic-based interface that allows a user to formulate queries using a restricted natural language syntax. As the user builds a query, Kaleidoscope uses rule-based and constraint knowledge to ensure a well-formed query, thus minimizing the chances for errors and easing the cognitive burden on the user.

A decent measure of user query performance is a count of the number of symbols or constructs required to formulate a query. Querying by context requires limited user input and interaction to formulate a query. For all systems, a user must select the concepts they are interested in. This process is performed in querying by context by allowing a user to graphically browse the context view. By organizing concepts by semantics instead of structure and using meaningful semantic names, the burden of concept selection and discovery is significantly reduced. Further, QBC attempts to minimize the user's specification of concept interrelationships. Concept interrelationships are automatically determined if possible, or can be easily specified by the user by traversing the integrated view. The concept of linking contexts by traversing the integrated view is similar to links connecting HTML pages. Even when the user must specify interrelationships, this is a simpler task than determining joins to connect physical relations as there will often be fewer of them. Overall, user query generation complexity is reduced by organizing and naming contexts and concepts semantically, isolating the user from database structures and naming (and the associated semantic to structural translation), and by providing a graphical query environment that allows the user to build queries by manipulating the simple semantic notions of concepts, contexts, and relationships between contexts.

## 5 Future Work and Conclusions

In this paper, we have demonstrated that by utilizing a global dictionary and semantic specifications an integrated, context view of a database can be constructed. This semantic view is easier for the user to query, as they are isolated from structural and naming considerations. We have discussed an automatic system that maps semantic queries to SQL and presents the results to the user. The query system is capable of handling complex join constructs and choosing the appropriate attributes, relations, and join conditions to preserve user query semantics. A graphical query environment is developed to allow the user to more formally define the semantics of their query without explicit knowledge of the structure and interrelationships of database attributes and relations. The query system described has been implemented in a software package called Unity [6].

Future work involves expanding the query processor to handle updates, and experimental comparisons of query by context with SQL, natural language querying systems, and other graphical query systems.

## References


1. 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.
2. T. Catarci and G. Santucci. Query by Diagram: A Graphical Environment for Querying Databases. *SIGMOD Record*, 23(2):515–515, June 1994.
3. S. Cha and G. Wiederhold. Kaleidoscope Data Model for An English-like Query Language. In *17th International Conference on Very Large Data Bases*, pages 351–361. Morgan Kaufmann, 1991.
4. W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. *SIGMOD Record*, 27(2):201–212, 1998.
5. C. Date. *The SQL standard*. Addison Wesley, Reading, US, third edition, 1994.
6. R. Lawrence and K. Barker. Unity - A Database Integration Tool. Technical Report 2000-664-16, Dept. of Computer Science, University of Calgary, July 2000.
7. J. Lee and D. 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.
8. G. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Five Papers on WordNet. Technical Report CSL Report 43, Princeton University, 1990.
9. W. Ogden and S. Brooks. Query Languages for the Casual User: Exploring the ground between Formal and Natural Languages. In *Proceedings of the Annual Meeting of the Computer Human Interaction of the ACM*, pages 161–165, 1983.
10. J. Sowa. Top-level ontological categories. *International Journal of Human-Computer Studies*, 43(5):669–685, 1995.
11. M. Stonebraker, J. Chen, N. Nathan, C. Parson, A. Su, and J. Wu. Tioga: A Database-Oriented Visualization Tool. In *Proceedings of the Visualization '93 Conference*, pages 86–93. IEEE Computer Society Press, October 1993.
12. 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.
13. W3C. Extensible Markup Langauge (XML) 1.0. Technical Report http://www.w3.org/XML/, February 1998.
14. M. Zloof. Query-by-Example: a data base language. *IBM Systems Journal*, 16(4):324–343, 1977.