

# No-Schema SQL (NS-SQL): Querying Relational Databases Independent of Schema

Adam Boot    Maryam Karimzadehgan    Keith Kelly    Michael Maur    Jing Yu  
University of Illinois at Urbana-Champaign  
CS511: Advanced Database Management Systems

## Abstract

SQL is a declarative language that is dependent on the underlying schema of a database. This means that a casual user must know the underlying schema in order to retrieve relevant data from a database. However, dependency on the schema is undesirable for several reasons. It places a burden on the user to understand the schema, and it requires that the input queries must be updated whenever the schema changes. In this paper, we provide a more relaxed form of formal SQL syntax, No-Schema SQL (NS-SQL). NS-SQL frees user from knowing the underlying schema but retains the expressive power of a structured query language. The user need only know the basic syntax of SQL. Since we are relaxing formal SQL away from schema-specific content, certain kinds of ambiguity become possible, specifically column name and join path ambiguity. Providing an intuitive interface allows this new system to be effectively used by the casual user without knowledge of the target database's underlying schema.

## 1. Introduction

### Motivation

One of Codd's main stated goals of proposing the Relational Model as a new paradigm was to decouple data access from physical representation. This allows a declarative language like SQL to be independent of ordering, indexing, or access path dependencies. Unfortunately, SQL is still dependent on the underlying database schema. SQL currently places the burden of navigating the database schema on the user, and many casual database users find SQL onerous for this reason alone. There is a need for a less restrictive and more intuitive syntax for specifying a query. Specifically, this relaxed language should involve two things:

- **Automatically resolve column and table names.** The entire FROM block may be omitted. Further, oftentimes column names are unique in a database, so explicitly specifying their table is unnecessary. If the database uses intention-revealing column names, the user need never be aware of which columns belong to which tables; they need only know what type of data the database holds.
- **Automatically infer foreign-to-primary key mappings.** The application should be able to deduce when two tables involved in a query should adhere to a key mapping without the user explicitly saying so. If a relationship exists between two tables, then this is nearly always what the user intends.

The proposed query language looks very much like SQL, but its lack of formal rigor makes its use more pleasant to the casual user. The challenge of our project is to discover a target database's schema for use in query inferences and to disambiguate and translate between the NS-SQL input query and a formal SQL statement. Because the proposed query language allows implied table names and join paths, it may be impossible for a given query to be entirely disambiguated. In such a case, the application should identify the problem and suggest valid alternatives to the user.

### Example

A brief note on our color convention: throughout this paper we use font color and typeface in order to help highlight the purpose for or relationship among various entities. Of particular importance is the use of the color **blue** to indicate NS-SQL input or referenced column names and the color **green** to indicate formal SQL output or physical columns. A referenced column is simply a partially-qualified column name which is referenced somewhere in an input NS-SQL query. A physical column is an entity representing a fully-qualified column extracted from the database schema. One of the main challenges of our system is in determining the proper mapping between referenced columns and physical columns.

Consider a database with the tables **Employee**, **Job**, and **Facility**. Assume the user is looking for the name, city, and state of all employees whose salary is greater than \$70,000 and whose job title is “Database Designer.” In order to specify this query using formal SQL syntax, the following statement would be required:

Employee	Job	Facility
id	id	id
name	title	city
salary	description	state
job_id		
facility_id		

```

SELECT Employee.name, Facility.city, Facility.state
FROM Employee JOIN Job ON (Employee.job_id = Job.id)
JOIN Facility ON (Employee.facility_id = Facility.id)
WHERE (Employee.salary > 70000) AND (Job.title = “Database Designer”);
    
```

Much of this statement is concerned with the proper reassembly of the relational database’s normalized tables. We believe the syntax could be much more intuitive and pleasant to use. Consider the same query using NS-SQL:

```

SELECT name, city, state WHERE (salary > 70000) AND (title = “Database Designer”);
    
```

### Related Work

Related research has focused largely on the removal of structure from the input query, either in the form of a natural language query language or a keyword search. NS-SQL shares the motivation of a desire to provide a more intuitive interface to the structured data in a relational database. However, NS-SQL differs in that it does not attempt to achieve this by removing the structure of the query language itself, but rather it removes the necessity of knowledge about the structure of the underlying database. This is useful in that it retains the expressive powers of a structured query language without the burden associated with understanding the database’s schema. For a more thorough survey of related research papers see **Appendix A: Survey of Related Work**.

### Organization

This paper is organized as follows. In **Section 2: System Architecture**, we provide a high-level overview of our project. In **Section 3: Subsystems**, we describe the project subsystems in detail. In **Section 4: Future Work**, we discuss direction for expansion of NS-SQL and our supporting application. In **Section 5: Summary**, we analyze the contribution of our project and conclude the paper.

## 2. System Architecture

Our project is an ambitious one, as it involved the invention and vetting of a modified query language along with all of the implementation involved in constructing a reference system. The system architecture is depicted in **Figure 1**.

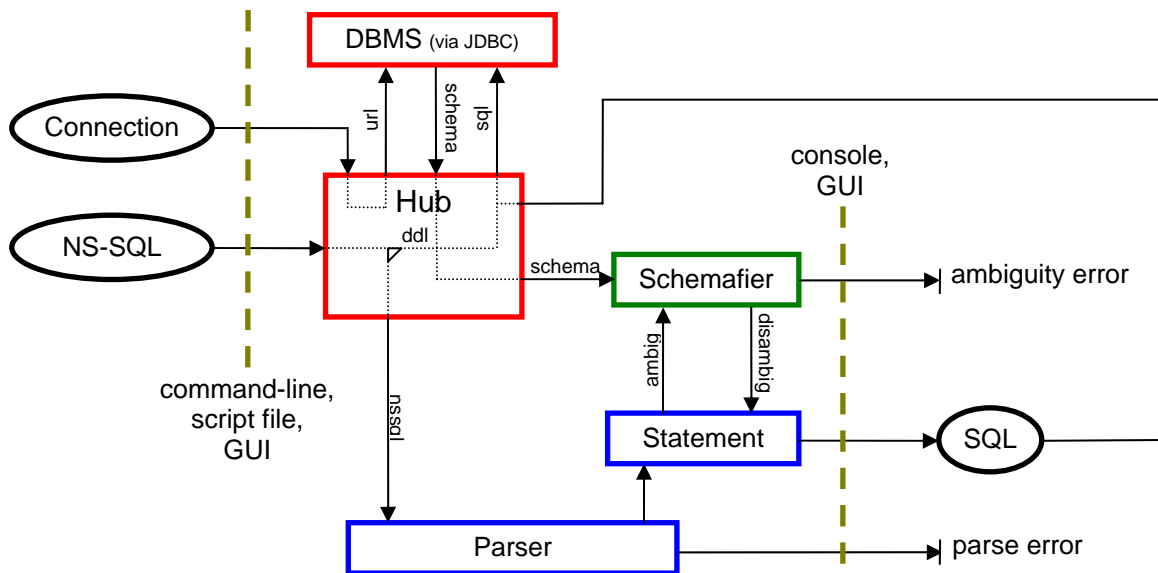


Figure 1

Our choice of programming language was Java, and we decomposed the project into three major subsections which each addresses a logically separate concern of running the source-to-source translation task:

- **Hub** – Manages input and output, establishes a DBMS connection, and infers the database’s schema via JDBC calls on its metadata. The Hub is also concerned with the tasks determining which statements need to be schemafied (and which may be passed straight through to the DBMS), as well as ensuring that any DDL statements processed during a session are reflected in the in-memory schema representation for subsequent DML statements.
- **Parser** – Parses the input NS-SQL query into an Abstract Syntax Tree (AST) by encapsulating tokens in separate node. The AST provides an interface to the important schema-sensitive clauses of interest to the Schemafier. The AST is designed to be mutable so that it may be augmented with column name qualification and join path information.
- **Schemafier** – Imposes the schema constructed in the Hub onto the NS-SQL statement parsed by the Parser. We use the invented verb *schemafy* to denote this process. The Schemafier must interpret the physical meaning of the column references from the input statement, and it must specify a join path to achieve a relationship amongst these physical columns. If no single unambiguous solution exists, the Schemafier must identify all feasible alternatives and advise the user of the possible solutions.

These three subsystems represent the majority of the effort we invested in this project. The algorithms and design decisions related to each are described in detail in the next section.

### 3. Subsystems

#### Hub

The purpose of the Hub is simply to tie the other subsystems together and provide them with input and register their output. The Hub is responsible for getting DBMS connection information from the user, specifically the DBMS type (so that the proper JDBC driver may be selected), server IP and port, connection username and password, and database name. Once the Hub has the DBMS connection information, it establishes a connection and polls the specified database for its underlying schema by way of standard JDBC calls. This schema information is encoded into an in-memory hierarchy including the physical tables and physical columns which will be used later by the Schemafier. A maximally-qualified column name is of the form **database.schema.table.column**. Note that the term schema here is used differently from the rest of this paper. Here *schema* refers to a specific element of a qualified table or column name. The actual implementation and effect of the schema element differs among DBMSs and we treat it as simply a containing namespace.

The Hub is also responsible for getting NS-SQL input queries from the user in order to parse and subsequently schemafy them. Once this process is complete, the Hub is responsible for registering the output to the user. This usually involves displaying the converted SQL statement or ambiguity error messages; however, the Hub may also apply the formal SQL statement to the DBMS across the existing connection and display the result set to the user.

The Hub accomplishes its input and output tasks through a pluggable user interface. We have provided two such interfaces: a Command-Line Interface and a Graphical User Interface. For more details on the use of these interfaces see **Appendix B: Application Interfaces**.

#### Parser

It is the Parser’s responsibility to isolate and extract column and table references to the Schemafier. The parsing engine must hide variations amongst DBMS such as SQL dialect from the rest of the system, so that the Schemafier may be implemented in a general fashion. Also, the Parser must support an AST which can be easily modified by the changes dictated by the Schemafier as it imposes the schema on the input query.

Unfortunately, existing Java-implemented SQL parsers either supported only a limited SQL grammar or their internal representation did not satisfy our needs. We instead decided to build our own SQL parser, and in this way we have fine control over the grammar we support and can tailor the internal representation to the needs of the Schemafier.

### Grammar Design

While there is specialty among dialects, most SQL grammars share many common features based on the published ANSI standard, and we base our grammar on SQL92. We then compared SQL92 with our expected NS-SQL inputs in order to “relax” the standard grammar such that its schema-specific elements become optional. For example, in an NS-SQL **SELECT** query, if a user does not know which table has the desired columns, he may omit the table name from each column reference and omit the **FROM** clause. The relaxed **SELECT** form may look like this:

**SELECT** [list of column references without table names]  
**WHERE** [search conditions]

Since only Data Manipulation Language (DML) queries may have their schema-specific content reasonably removed, we can focus on **SELECT**, **UPDATE**, **INSERT**, and **DELETE** statements, reducing the grammar size.

The primary concern when modifying the standard grammar is the potential to introduce conflicts. Fortunately, for the DML statements we are interested in, every clause has a predefined “token” at the beginning (such as **FROM**, **WHERE**, etc), which makes it easy to change one clause from fixed to optional.

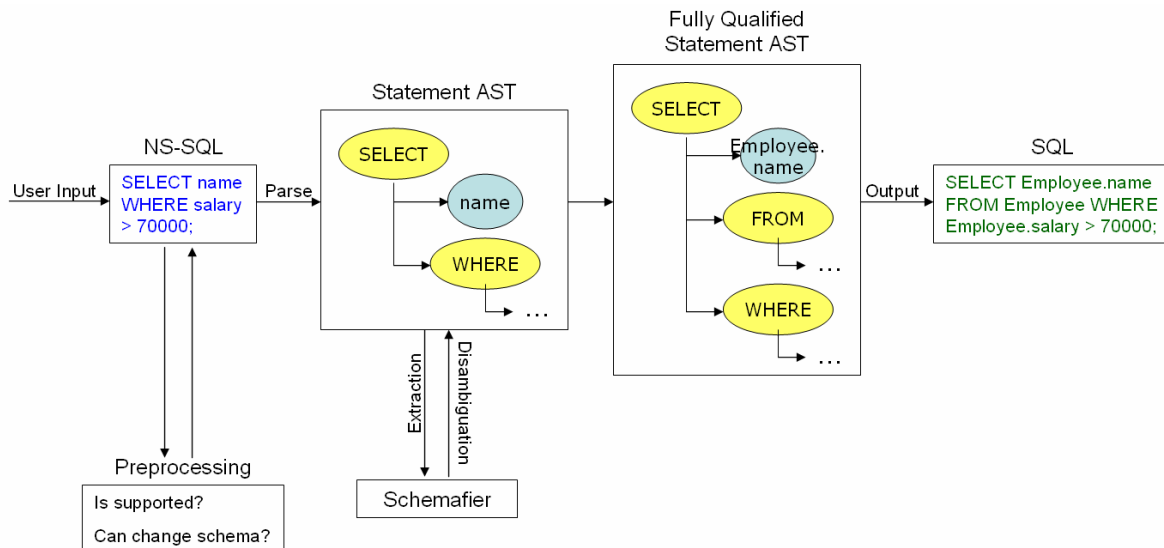
### Parser Generation

We used JavaCC [4] to generate the actual parsing program. JavaCC is a popular parser generator for Java which reads a grammar specification and converts it into a Java program that recognizes the grammar. While parsing and pattern matching, the Parser is at the same time setting up the AST, so that when it finishes the AST has been built.

JavaCC does have its limitations. It can only support LL(1) grammars, which means if a grammar has left recursion (such as  $Expr \rightarrow Expr + Term$ ), it is necessary to translate the left recursion to corresponding right recursion. Eliminating left recursion is not trivial, so translating the SQL92 standard BNF would be difficult. Fortunately, we were able to take advantage of a pre-translated SQL grammar from the Mckoi SQL database project [7].

### Parsing Engine Structure

The parsing engine is structured to run through several stages in the course of processing an NS-SQL input query as indicated in **Figure 2**.



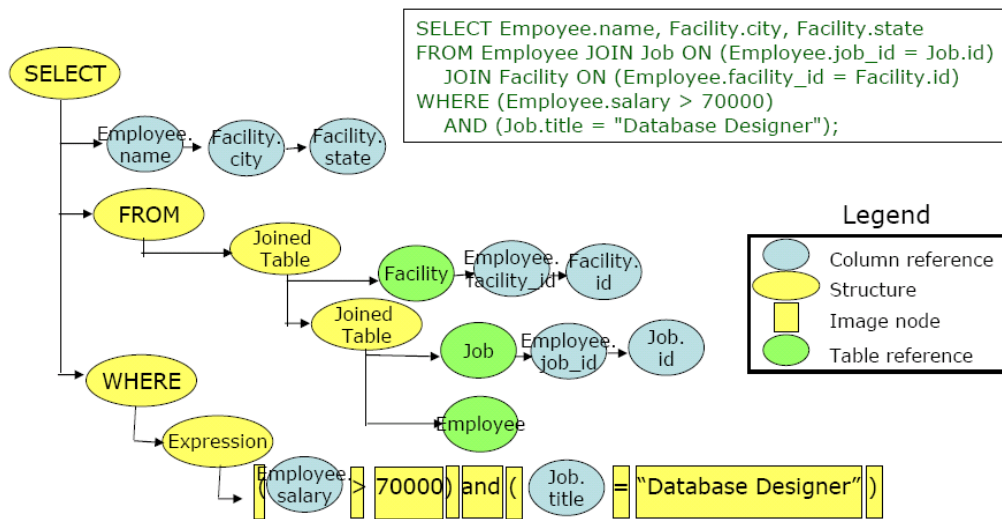
**Figure 2**

- **Preprocessing** – A given input query is checked to see if it is supported and whether it can change the schema of the database. Both of these checks can be performed before actual parsing by examining the first word of the query. The first word of an SQL query identifies the statement, and its properties can thus be determined. An unsupported query is simply passed through to the DBMS. If the query can change the database’s schema – such as a Data Definition Language (DDL) statement – then the in-memory representation of the schema must be refreshed, or future calls to the Schemafier may operate on stale schema knowledge.

- **Parsing** – The NS-SQL query is translated into a statement AST for manipulation within our system. The AST represents the interesting syntactical elements of the query as nodes in a tree. Specifically, column and table nodes are most interesting to us. The AST acts as an interface between the Parser and the Schemafier, allowing modification to certain nodes.
- **Extraction** – A list is built of all referenced columns and specified tables in the statement. This information is later extracted by the Schemafier as it does its job of imposing the schema onto the query.
- **Update** – As the Schemafier disambiguates the columns and tables with their fully qualified names, the AST allows directly modification of the appropriate mutable AST nodes. Also in this stage, the Schemafier provides a list of key mappings among tables which the AST uses to generate the necessary joins in the **FROM** clause.
- **Output** – The final resulting formal SQL statement is available by simply calling the **toString()** method on the AST root. The statement now generates the fully qualified query string representing itself and passes it back to the Hub to be run on the DBMS.

### Abstract Syntax Tree

The AST is the internal representation of a query in our system. As mentioned earlier, one reason we decided to build our own parsing engine is that existing parsers do not provide an AST that meets our needs. Most existing parsers focus on full representation of an SQL query, while we are only concerned with very specific elements. It is inefficient and error-prone to maintain these unnecessary AST nodes, so our version of an AST is tailored to the needs of the Schemafier. **Figure 3** depicts the fully-qualified structure of an example AST.



**Figure 3**

Oval nodes represent classes with significant content and accessor methods. For example, the content of a FROM node is a list of table name nodes, joined table nodes, or nested table nodes. Methods are provided to insert new nodes into the list. Rectangular nodes represent classes that are inconsequential to the Schemafier, so the content is only maintained to be output again later with the **toString()** method. All nodes are listed in the order of appearance in the input, which is different from the way most parsers operate.

### Schemafier

The purpose of the Schemafier subsystem is to impose a database schema onto an input query in order to convert it into a fully-specified formal SQL query. There are two major stages in this process. First, we must determine a unique set of physical columns which correspond to the columns referenced in the input statement. Second, we must determine a unique join path by which to relate these columns. However, the flexibility afforded by the relaxation away from schema-specific information in the user's query carries with it the possibility of several types of ambiguity in resolving this query. In order to restrict the search space we consider – and thereby the range of possible ambiguities – we make two reasonable assumptions. First, all referenced columns must be relatable; that is, the user is never interested in a query that will produce a Cartesian product. Second, we will never consider join

paths that can only produce a restricted result set; that is, the inclusion of joins with unreferenced tables which do not add to the relationships amongst the referenced tables, but only limit them. These assumptions become important as we discuss the algorithms by which we achieve the selection of column names and join paths.

### *Column Name Selection*

The first major task in imposing a schema onto an NS-SQL statement is to associate column references from the query with physical columns from the schema. The full qualification of the physical column is known, of course, but the column reference in the input query will likely be only partially qualified. Specifically, it is expected that the user will commonly reference only the column names. Interpreting the physical column becomes non-trivial in cases when the column reference is not sufficiently qualified and there are multiple physical columns that share a partially-qualified name. For example, if the user only makes reference to a column name of something very common such as **id** or **name** chances are high that there is more than one physical column with this column name. It is therefore necessary to identify all possible interpretations of the set of column references. The challenge is in selecting a physical column consistent with the user's intent. One of the fundamental assumptions we make in this project is that column references always refer to a set of columns that are relatable to each other; that is, there is some set of join paths amongst the tables in the schema which establish a relationship amongst the columns (otherwise, the query would result in some form of Cartesian product). If a given interpretation of a column reference is not relatable, we do not consider it as a legitimate option. We evaluate this notion of "relatability" by way of a graph tagging algorithm.

The tagging algorithm associates a *tag* or unique identifier with all nodes which can be reached from each other. This is done by interpreting the database schema as a graph where each table is represented by a node and each key mapping is represented by a directed edge (in the direction from foreign to primary key). Conceptually, the tagging algorithm creates a new tag for each root (node without parents) and cycle. The tag is then expanded to encompass all descendants of the root or cycle. The intuition is that any set of columns contained entirely within a tag are fully relatable to each other and represent a valid solution.

Because we do not initially know which nodes are roots and because cycles are difficult to determine except through complete graph navigation, we chose to build up the tag set in arbitrary order and proceed until all nodes are tagged. The algorithm we use to establish the tag set for a given schema graph recursively queries each node for its parent's tags, and then returns this combined set of tags to its children. The following pseudocode demonstrates this concept:

```

build-tag-set() {
  for each node {
    if not already tagged {
      build-tags(node, {}) //no history
    }
  }
}

build-tags(node, call-history) {
  if node is a root {
    create a new tag and apply to all nodes in call-history
  } else {
    for each parent of this node {
      if parent is already tagged {
        apply the parent's tag to all nodes in call-history
      } else if parent is already in call-history {
        create a new tag for the cycle and include all nodes in call-history in the cycle
      } else {
        build-tags(parent, {call-history + node})
      }
    }
  }
}

```

For example, consider **Figure 4**. This schema is composed of four tables: **Student**, **Course**, **Enrollment**, and **Survey**. The foreign-to-primary key mappings are indicated and each table's columns are shown beside the table-node. Notice that there are two tables that contain a column named **credit\_hours**. **Course.credit\_hours** represents the possible credits available for a given course (as it might offer varying amounts of credit depending on coursework). **Enrollment.credit\_hours** represents the specific amount of credit a given student has enrolled for in the course. Depending on the other columns referenced in the query, this may or may not result in an ambiguity.

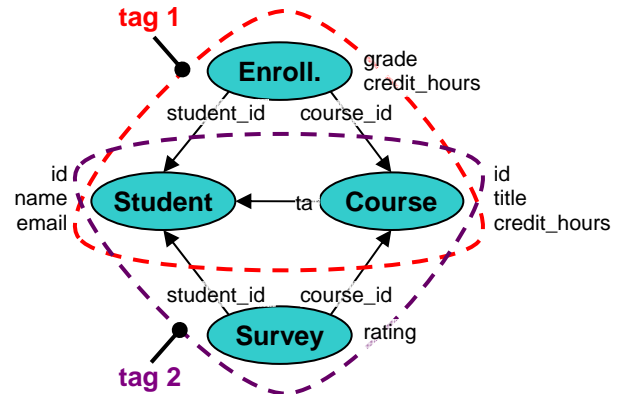


Figure 4

**Example 1:** If the input NS-SQL query were **SELECT rating, credit\_hours**; then there is no ambiguity. This is because we know that the only interpretation of **rating** is **Survey.rating** and the only tag that contains **Survey.rating** is **tag 2**. **Course.credit\_hours** shares **tag 2**, but **Enrollment.credit\_hours** does not. We can therefore eliminate **Enrollment.credit\_hours** as a possible interpretation, leaving only the solution where **credit\_hours** refers to **Course.credit\_hours**. The ultimate resulting SQL query for this example would be **SELECT Survey.rating, Course.credit\_hours FROM Survey JOIN Course ON Survey.course\_id = Course.id**;

**Example 2:** If the input NS-SQL query were **SELECT name, credit\_hours**; then there is ambiguity. This is because we know that the only interpretation of **name** is **Student.name** which shares **tag 1** with both **Course.credit\_hours** and **Enrollment.credit\_hours**. We cannot therefore eliminate either interpretation of **credit\_hours**, leaving two valid solutions and an ambiguity which we must rely on the user to resolve.

**Example 3:** If the input NS-SQL query were **SELECT grade, rating**; then there is no possible relation. This is because we know that the only interpretation of **grade** is **Enrollment.grade** in **tag 1** and the only interpretation of **rating** is **Survey.rating** in **tag 2**. We cannot therefore relate **Enrollment.grade** to **Survey.rating**, leaving no valid solutions.

### Join Path Selection

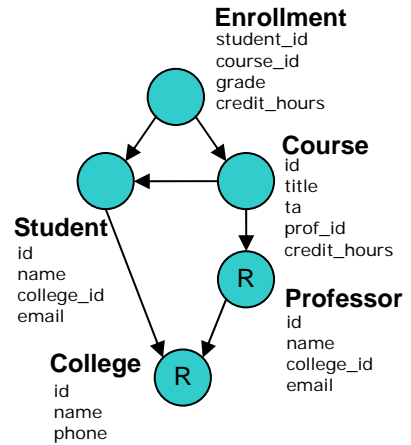
The second essential part of the Schemafier takes care of the join path selection. If the column name selection was able to match the referenced columns from the NS-SQL statement to a unique set of physical columns in the database, these physical columns serve as input here. Given the foreign-to-primary key mapping (fk-pk) from the underlying database, this part of the Schemafier tries to relate the tables containing the referenced columns in an unambiguous way. If one single join possibility is found, the input statement is updated to formal SQL and presented to the user who can execute it on the database. If multiple possibilities to relate the referenced tables exist, an ambiguity is reported.

As mentioned earlier, one of our fundamental assumptions is that the user does not want to relate tables using a Cartesian product. Allowing this would enable arbitrary joins in the database and would no longer allow us to determine a unique join path amongst the referenced tables. The join path selection algorithm will add two additional assumptions about the user's intention which help us reduce the number of ambiguous alternatives. We believe these assumptions are very reasonable because they ensure the result set is not unnecessarily restricted, and the assumptions need only be applied for very complicated schema.

The first additional assumption is that the user does not intend to include fk-pk mappings in a join that are not needed to relate the referenced tables. An example for such a case will be given in the outline of the algorithm below. The second assumption refers to cycles. While it is possible for a database to have tables that are connected by fk-pk relationships in a circular manner, we assume that the user even when referencing all tables in a cycle does not want to relate them using all mappings of the cycle at the same time. As there may be multiple mappings between tables this means that in a valid join at least two neighboring tables in each cycle are not connected by a direct fk-pk mapping between them.

We will now briefly outline the main steps of the join path selection algorithm. **Figure 5** shows the structure of a database slightly modified from the previous example. The tables of which columns were referenced in the NS-SQL statement are marked with the letter R.

1. In order to relate the tables **College** and **Professor**, determine their *common predecessors*: the set of nodes from which the referenced tables can each be reached by way of a series of fk-pk mappings. In this case, **Enrollment**, **Course**, and **Professor** itself fulfill this property, making each of them possible join roots.
2. For each join root determine the available paths to all referenced tables. For example starting with **Course**, **College** can be reached either via **Student** (path **C1**: **Course**→**Student**→**College**), or by way of **Professor** (path **C2**: **Course**→**Professor**→**College**). **Professor** is reachable from **Course** only directly (path **P1**: **Course**→**Professor**).
3. All possible combinations of the available paths are generated, each potentially forming a way to join the referenced tables. Every alternative contains exactly one path from the join root to each referenced table. The alternatives here are (**C1**, **P1**) and (**C2**, **P1**).



**Figure 5**

4. Path combinations that do not fulfill our assumptions about the user's intent are removed. Note that in (**C2**, **P1**) both paths share the mapping **Course**→**Professor**. The information necessary to relate the two referenced tables is sufficiently contained in the sub-path of **Professor**→**College**, and the additional join involving **Course** only restricts the result set of the query. While the edge **Professor**→**College** essentially returns the requested college values for all professors, the edge **Course**→**Professor** unnecessarily restricts the result set to those professors who are assigned to a course.
5. If processing all of the potential join roots in the manner described above yields only one possible way to relate the referenced tables, the fk-pk mappings are extracted and inserted into the AST provided by the Parser. This completes the transformation from the user's NS-SQL input into a fully-qualified formal SQL statement.

Two additional issues deserve treatment, as the example above was simplified to illustrate the basic concept of join path selection:

1. Whenever paths between a join root and a referenced table are retrieved, cyclic structures among the mappings may cause unexpected results. To avoid this, paths are constructed from the referenced table towards the join root, keeping track of a history of visited table-nodes and mapping-edges on this path. Whenever a node is encountered that is already contained in the call-history or the end of a branch is reached, the search is abandoned. Whenever the join root is found, a new path is created using the call-history of visited edges, and the search for more paths to connect the referenced table to the root continues.
2. Joins that are “unnecessary” are pruned. To determine such joins, we retrieve the set of nodes from the join root (exclusive) to the first referenced table (inclusive) from each path in the current set of candidate paths. The resulting sets of nodes are then intersected with each other. If the result of this intersection is not empty, this indicates that all paths cross at some node between the root and the first referenced table. Since the joins below this point of intersection are already sufficient to relate the referenced table, the current combination of paths can be discarded. Because the point of intersection must have been a common predecessor as well, the necessary information about the join from this point on is not lost, but processed separately.

The following examples further demonstrate the join path selection algorithm:

**Example 1:** For the query **SELECT title, grade;** the result is unambiguous as **Course** and **Enrollment** can only be related using the direct mapping (**Enrollment**→**Course**).

**Example 2:** For the query **SELECT college\_id, phone;** the result is ambiguous, as there are three valid ways to join **Professor** and **College** including:

- (**Professor**→**College**)
- (**Course**→**Professor**, **Course**→**Student**→**College**)
- (**Enrollment**→**Course**→**Professor**, **Enrollment**→**Student**→**College**)

The three discarded path combinations are:

- (Course→Professor, Course→Professor→College)
- (Enrollment→Course→Professor, Enrollment→Course→Professor→College)
- (Enrollment→Course→Professor, Enrollment→Course→Student→College)

**Example 3:** The query `SELECT Course.id, Student.name;` resolves two name ambiguities by qualifying the column references for `id` and `name` with a table name. The result of the join path selection is ambiguous however, as there are two valid ways to join `Course` and `Student`:

- (Course→Student)
- (Enrollment→Course, Enrollment→Student)

The one discarded path combination is:

- (Enrollment→Course, Enrollment→Course→Student)

## 4. Future Work

We believe our project was an ambitious one. While we feel we fulfilled the main research goal, there are still many areas of our work with may be extended to further improve the usefulness of the product. These areas of interest are focused on the two most challenging and extendable subsystems of the project.

### Parser

It would be desirable to support a majority of the most popular dialects so that users of our system can leverage the simplicity of NS-SQL while retaining the full power of the target DBMS's extensions. To this end, the Parser may be extended to support dialects without affecting standard queries. The Parser was purposefully designed to be highly extensible (see **Appendix C: Parser Extension**). Example extensions include the following:

- Arguments
  - `SELECT TOP 10 ...` (SQL Server)
  - `SELECT ... LIMIT 0, 10` (MySQL)
  - `SELECT ... SAMPLE 10` (Oracle)
- Functions
  - `CUBE` and `ROLLUP` (SQL Server and Oracle)
- Statements
  - `RENAME` (MySQL)
  - `PARTITION` (Oracle)

### Schemafier

The Schemafier currently accepts input of the simple style we expect to be vastly more common amongst the target user population. However, for purposes of completeness, it is desirable that the system also support more advanced and complicated input statements. Such options include support for nested queries and column or table aliasing.

Another major area of expansion is support for non-`SELECT` statements. We believe that `INSERT`, `UPDATE`, and `DELETE` are excellent candidates for inclusion. However, we have identified some special considerations when it comes to statements that can manipulate data. In NS-SQL, a data manipulation statement might reference multiple tables since the user does not know the schema; whereas, standard SQL simply prohibits this. A simple solution would be to simply advise the user that manipulating multiple tables is not allowed. This solution is not very satisfying, however. Unfortunately, supporting multiple tables means a single NS-SQL statement may produce multiple formal SQL statements, and we suspect that maintaining data integrity in the face of uniqueness and fk-pk constraints will be very challenging. For example, if a referenced column is related by a key mapping (that is, in a normalized table), then is it appropriate to change the foreign key reference or to change the column data in the separate table?

We believe that one reasonable way to assist in column name selection is to consider column data types. If two columns with the same name have different data types, then they may be distinguishable. The challenge comes with

extracting data type information from input NS-SQL query. This may be possible if the column is referenced in the WHERE clause as part of a condition with a literal value.

Supporting column name synonyms so that the user need not know a column's exact name would be a very useful feature. Also, an improvement to the user interface is automatic ambiguity resolution so that a user need only choose among a set of alternative solutions to resolve an ambiguous NS-SQL statement. Finally, the system may learn from past cases when the user has manually disambiguated queries to learn which column names or join paths are preferred and automatically make assumptions when faced with such an ambiguity problem in the future.

## 5. Summary

NS-SQL is a unique approach to improving the user experience of querying of relational DBMSs by providing a schema-transparent perspective on the database. As opposed to natural language or keyword interfaces which remove structure from the input query, NS-SQL retains the expressiveness of a structured query while omitting the tedium of direct schema knowledge. In addition to being more intuitive to use, NS-SQL also provides the important additional advantage of schema independence for the input query. This means that as a database's schema changes, the input query need not.

The NS-SQL project accomplishes the main intent to provide a modified SQL syntax which omits schema-specific information, and proves the viability of this concept by providing a reference implementation that handles the major algorithmic challenges of flexible input parsing, column name selection, and join path selection. In effect, a user may make use of our system to query a database without specific knowledge of its exact schema. The system is also able to accurately report any type of unavoidable ambiguity encountered and assist the user in manually resolving it.

We have extensively tested the system to ensure its proper functioning (see **Appendix D: Testing**). Finally, our implementation is highly extensible to allow future work to continue to expand the functionality and applicability of the system.

## 6. References

- [1] G. Bhalotia, C. Nakhey, A. Hulgeri, S. Chakrabarti, and S. Sudarshanz. "Keyword Searching and Browsing in Databases using BANKS." *Proceedings of International Conference on Data Engineering*, 2002.
- [2] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. "Proximity Search in Databases." *VLDB*, 1998.
- [3] V. Hristidis and Y. Papakonstantinou. "DISCOVER: Keyword search in relational databases." *VLDB*, 2002.
- [4] Java Compiler Compiler (JavaCC). <https://javacc.dev.java.net/>
- [5] Yunyao Li, Huahai Yang, H. V. Jagadish. "NaLIX: an Interactive Natural Language Interface for Querying XML." *SIGMOD 2005*, June 14-16, 2005.
- [6] U. Masermann and G. Vossen. "Schema Independent Database Querying (on and off the Web)." *Proceedings of IDEAS2000*, 2000.
- [7] Mckoi SQL Database. <http://mckoi.com/database/>
- [8] Vesper Owei, Hyeun-Suk (Sue) Rhee, Shamkant Navathe. "Natural Language Query Filtration in the Conceptual Query Language." *IEEE 1997*, 1060-3425197.
- [9] J. Plesn'ik. "A bound for the Steiner tree problem in graphs." *Math. Slovaca 31*, p. 155-163, 1981.
- [10] SQL4J. <http://www.cs.toronto.edu/~jglu/sql4j/index.htm>
- [11] J. Van den Bussche, D. Van Gucht, G. Vossen. "Reflective Programming in the Relational Algebra." *Journal of Computer and System Sciences*, 52(3): 537-549, 1996.

## Appendix A: Survey of Related Work

Since database query languages like SQL can be intimidating to casual users, there have been some efforts towards natural language and keyword-based search in databases. Natural language interfaces are an attractive notion since the user will be much more comfortable expressing his query intentions by way of natural language instead of an unfamiliar structured language. To this end, NaLIX [5] is a generative interactive natural language query interface to an XML database. It reformulates the input query to XQuery expression and translates it by means of mapping grammatical proximity of natural language parsed tokens to proximity of corresponding elements in the result XML.

CQL/NL [8] allows users to formulate database queries in natural language. CQL/NL queries are filtered for search predicates derived from conceptual schema constructs. Based on the identified search predicates, CQL/NL uses a set of predefined natural language templates to compose a natural language explanation of the query. The explanatory statement is returned to the user for validation. It uses the association semantics of semantic data models to construct query solution paths.

In BANKS [1] and Proximity-Search [2], a database is viewed as a graph with objects or tuples as nodes and relationships as edges. Relationships are defined based on the properties of each application. For example, an edge may denote a primary to foreign key mapping. In BANKS, the user query specifies two sets of objects, the *Find* and the *Near* objects. These objects may be generated from two corresponding sets of keywords. The system ranks the objects in *Find* according to their distance from the objects in *Near*. An algorithm is presented that efficiently calculates these distances by building hub indices. In Proximity-Search, answers to keyword queries are provided by searching for Steiner trees [9] that contain all keywords. Heuristics are used to approximate the Steiner tree problem. These two systems rely on a similar architecture. A drawback of these approaches is that a graph of the tuples must be created and maintained for the database. Further, the important structural information provided by the database schema is ignored.

Keyword search is a popular information discovery method because the user does not need to know either a query language or the underlying structure of the data. DISCOVER [3] exploits the database schema, which leads to relatively efficient algorithms for answering keyword queries because the structural constraints expressed in the schema are helpful for query processing. DISCOVER returns qualified joining networks of tuples; that is, sets of tuples that are associated because they join on their primary and foreign keys and collectively contain all the keywords of the query. DISCOVER differs from NS-SQL in that it is a keyword-focused solution instead on a structured query solution.

SISQL [6] is a language based on Reflexive Relational Algebra [11] for querying databases whose schema may be unknown. It takes parameterized SQL queries as input and dynamically generates correct SQL statements. For example, a query in SISQL may be “SELECT [A] FROM [RI] WHERE [B] = ‘John’]” such that [A] is a placeholder for attributes in the database in question, [RI] is one of the relations in that database having [A] as an attribute, and [B] is another attribute in [RI]. If the exact relation (or attribute) names are unknown, *parameters* can be used instead. During query evaluation, those parameters are instantiated with the appropriate names taken from a database to which the respective query is directed. One problem with this work is that it does not have disambiguation strategies for queries generated for databases containing attributes with the same name in distinct tables which is a consideration that NS-SQL does address. Also, NS-SQL syntax may be seen as a more relaxed form of SISQL.

## Appendix B: Application Interfaces

Our project provides a flexible mechanism of specifying a user interface. We have implemented two such user interfaces with which to interact with the system including a Command-Line Interface (CLI) and a Graphical User Interface (GUI). The usage details of each are specified below:

### Command-Line Interface

The CLI may be accessed by providing a non-empty set of arguments to the Main class. The command line interface accepts a number of options in order to establish the DBMS connection as well as several methods of providing NS-SQL input. These options are available in either short or long form. The available options are these:

Option	Short	Long	Value	Required
DBMS Type	-t	--type	Currently one of {"mysql", "sqlserver", "oracle"}	yes
Server	-s	--server	Server IP (defaults to localhost)	no
Port	-P	--port	Server port (defaults to DBMS-specific default)	no
Database	-d	--database	Database name	yes
Username	-u	--username	DBMS user (with appropriate permissions)	yes
Password	-p	--password	Password for DBMS user	yes
In	-i	--in	Input script file for NS-SQL source	no
Out	-o	--out	Output log file for resulting formal SQL	no
Error	-e	--err	Error log file for encountered exceptions	no
Result Set	-r	--result	Output log file for DBMS result set	no
Error Tolerant	-E	--errortolerant	Whether to ignore non-terminal errors	no

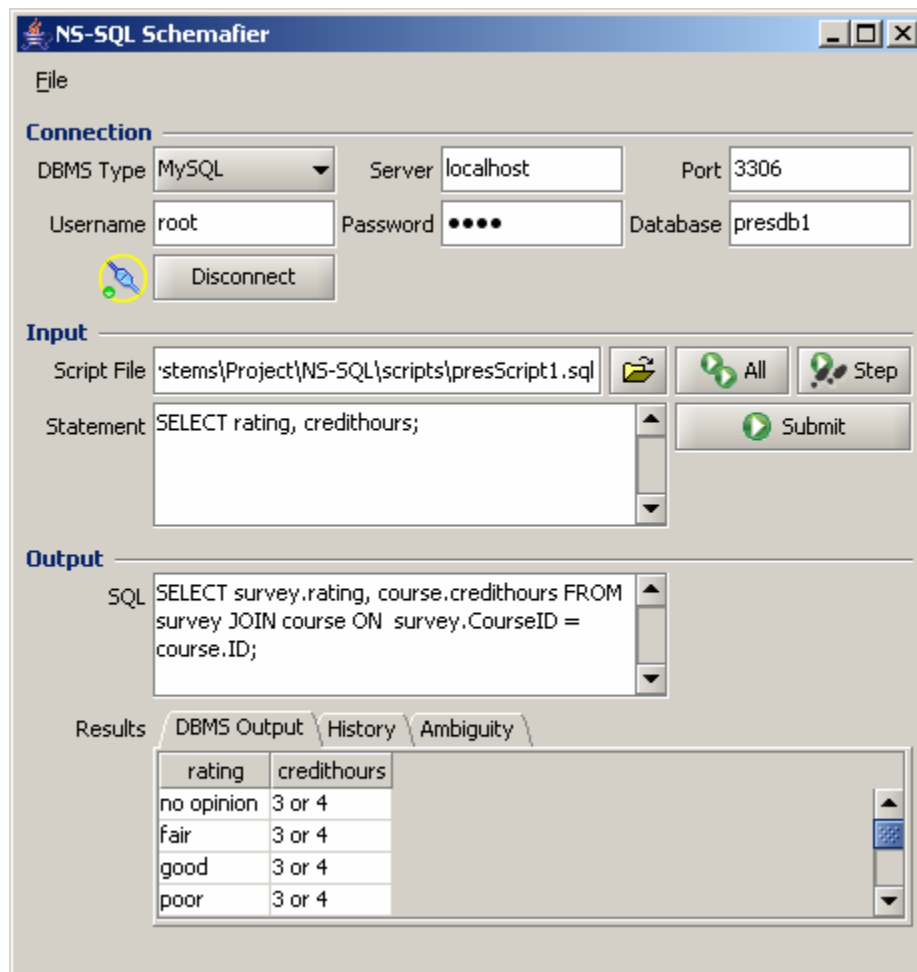
## Graphical User Interface

The GUI may be accessed by providing no arguments to the Main class. The **Connection** section collects information about the database connection. If they were previously empty, the **Server** and **Port** are automatically set to DBMS defaults upon **DBMS Type** selection. The **Username** and **Password** are automatically populated if corresponding NSSQL\_USER and NSSQL\_PASS environment variables are defined. Otherwise, they may be provided manually.

The **File** menu provides options to enable auto-apply and error tolerance or to exit the application. **Auto-apply** automatically applies a successfully converted NS-SQL statement to the underlying database (enabled by default). **Error Tolerant** allows script processing to proceed despite any encountered ambiguities when in batch mode (disabled by default).

The **Input** section allows the user to either specify a script file as input or provide individual NS-SQL statements. Individual NS-SQL statements may be specified in the **Statement** field and processed with the **Submit** button. If a script file is specified, it may either be submitted in batch with the **All** button or one at a time with the **Step** button. As each statement from the script is processed, the NS-SQL value is displayed in the **Statement** field, the converted query is displayed in the **SQL** field, and the output is appropriately displayed in the **Results** area. It may be useful to modify individual statements of a script file in the **Statement** field and resubmit, especially in case of ambiguity.

The **Output** section displays all relevant output for every user action. If an NS-SQL translation is successful, the resulting formal SQL statement is displayed in the **SQL** field. If auto-apply is enabled, the result set of a successful translation is displayed in the **DBMS Output** tab. If an ambiguity is encountered, the details of the problem are displayed in the **Ambiguity** tab. The results of all past actions are logged in the **History** tab.



## Appendix C: Parser Extension

### Dialect Compatibility

Every widely used DBMS platform extends SQL by introducing its own operators, keywords, arguments, clauses, and statements in addition to the standard syntax. Throughout the course of the project, we performed research into some of the dialects of SQL that exist – specifically those of MySQL, SQL Server, and Oracle – and evaluated them to determine the feasibility of making our system compatible with them. It would be desirable to support a majority of the most popular dialects so that users of our system can leverage the simplicity of NS-SQL on which ever DBMS platform they choose while retaining the full power of the target DBMS’s extensions.

### Parser Extensibility

We believe it is feasible to add dialect compatibility to our system in the future without breaking the existing mechanism because of the way we designed the Parser and its internal representation of an NS-SQL statement. The nature of our system only requires that we be able to extract references to columns and tables in a query and generate joins amongst tables when necessary. This leaves all other parts of the statement untouched. For a given dialect, all the Parser needs to know is how to extract the referenced columns and tables from the new syntax. In general this can be accomplished by modifying the SQL92 grammar definitions currently used. Though we did not find any concrete examples in our research, there may be some cases where certain dialects break compatibility with the SQL92 standard. This situation could require a change to the architecture of the Parser where it would need to use a slightly different grammar and AST representation depending on the platform being used.

Extending the Parser involves the following steps:

- Write additional AST node classes for the new statement.
- Implement a method that does the column/table references extraction.
- Implement a method that supports update given feedback from the Schemafier.
- Modify the grammar file to incorporate the new grammar changes, and add tree build-up instructions.
- Run JavaCC to generate the new Parser.

## Appendix D: Testing

### Parser Testing

Because the Parser is relatively independent from the rest of the system, we set up a separate test environment for it which feed test SQL queries into the Parser and let it build the AST. The AST's output SQL query was then compared with the input query by way of a Perl script. Matching queries prove that the AST was built correctly. We used a set of 561 sample SQL **SELECT** queries garnered from the SQL4J project [10] to test the Parser, which we eventually were able to completely support.

### Unit Testing

For unit testing, database-like structures were built in memory (not in a DBMS) to separately test different aspects of the system. JUnit test cases generated instances of the internal graph representations of databases to check essential subsystems such as JDBC connection, Parser, Column Name Selection, and Join Path Selection. Those test suites covered the basic functionality as well as more complex test situations such as multiple partly interwoven cycles.

### Integration Testing

For integration test we developed a number of MySQL test databases to verify the correct operation and integration of the system. Their structure is briefly outlined here. The following formatting is used for the indicated purpose.

Database Structure	<b>TableName</b>
Data Type	VARCHAR
	INT
	DOUBLE
Key Mapping	PrimaryKey
	ForeignKey
Name Ambiguity	RepeatedName

**DB01:** A simple database with just one table and no name ambiguity in order to basic functionality.

Employees
ID
FirstName
MiddleInitial
LastName
Salary

**DB02:** A slightly more complex database with several tables, ambiguous column names, but no foreign to primary key (fk-pk) mappings is the next step in database complexity. Supplier is intentionally left without a primary key to ensure correct operation for the unlikely case that no such key is defined on a table.

Customer	Employee	Inventory	Product	Supplier
ClientNr	ID	ID	ID	EnterpriseName
Company	EmployeeName	Description	Description	Town
Town	PhoneNr	Location	Category	PhoneNr
PhoneNr	email	Brand	Amount	FaxNr
email	Salary	YearOfPurchase	Price	Branch

## No-Schema SQL

**DB03:** In the third phase fk-pk mappings is added to a database with a limited amount of name ambiguity.

T1	T2	T3	T4	T5
C1	C3	C1	C7	C2
C2	C4	C3	C8	C5
				C6
				C10

not connected

**DB04:** Initial testing of composite keys is done with the following database.

T6	T7
C11	C13
C2	C14
C12	C1

**DB05:** A combination of DB03 and an extension of DB04 is used to test how to infer multiple composite key mappings from the JDBC connection. In addition, name ambiguity between the separate sets of related tables in the database is present.

T1	T2	T3	T4	T5	T6	T7
C1	C3	C1	C7	C2	C11	C1
C2	C4	C3	C8	C5	C2	C13
				C6	C12	C30
				C10		C20
						C40

not connected

**DB06:** A more complex database structure with a lot of name ambiguity as well as path ambiguity for in-depth testing. Due to the more complex structure of the database, the structure itself and the column overview are shown separately.

Building	Caretaker	College	Country	Course	Student	Professor
Name	ID	ID	Name	ID	UniversityID	Name
Caretaker	Name	Name	Continent	Title	Name	Department
College	PhoneNr			TA	email	Room
				Professor	StreetAddress	PhoneNr
Department	Enrollment	Room	Town	Size	Town	email
ID	EnrollmentID	ID	TownID	Room	HomeStreetAddress	StreetAddress
College	StudentID	Building	Name	CourseNr	HomeTown	Town
Name	CourseID	RoomNr	State	Year	PhoneNr	
Phone	Grade	Description	Country	Semester	Semester	
			ZipCode	Department	College	

