Schema-free SQL[*]

Fei Li
Univ. of Michigan, Ann Arbor
lifei@umich.edu

Tianyin Pan
Univ. of Michigan, Ann Arbor
ptianyin@umich.edu

H. V. Jagadish
Univ. of Michigan, Ann Arbor
jag@umich.edu

ABSTRACT

Querying data in relational databases is often challenging since SQL requires its users to know the exact schema of the database, the roles of various entities in a query, and the precise join paths to be followed. On the other hand, keyword search is unable to express much desired query semantics.

In this paper, we propose a query language, Schema-free SQL, which enables its users to query a relational database using whatever partial schema they know. If they know the full schema, they can write full SQL. But, to the extent they do not know the schema, Schema-free SQL is tolerant of unknown or inaccurately specified relation names and attribute names, and it also does not require information regarding which relations are involved and how they are joined. We present techniques to evaluate Schema-free SQL by first converting it to full SQL. We show experimentally that a small amount of schema information, which one can reasonably expect most users to have, is enough to get queries evaluated as if they had been completely and correctly specified.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; H.1.2 [**Information Systems**]: User/Machine Systems—*Human factors*

Keywords

Query Language; Relational Databases; Usability;

1. INTRODUCTION

Querying data in relational databases is often challenging. SQL is the standard method to query relational databases. While expressive and powerful, SQL requires its users to know the exact schema of the database, the roles of various entities in a query, and the precise join paths to be followed. This difficulty is exacerbated by normalization, a process that is central to relational database design, which brings benefits including saved space and avoided update anomalies, at the cost of spreading data across several relations and thus making the database schema more complex. Where queries are predictable, forms-based interfaces and other application-mediated mechanisms can be used. However, these techniques do not support ad hoc queries.

To address these difficulties, there has been a stream of research towards querying information from databases using keywords [6, 4, 8]. In keyword search, the users need to know neither a query language nor the underlying logical structure of the database. All they need to do is to type in some keywords. While very friendly to use, there are intrinsic ambiguities in using an unstructured set of keywords to convey complex information needs. Without properly specified structure information, it is very hard for a system to return prefect query results. Furthermore, keyword search loses many commonly used functions provided by SQL, such as comparison, computation, ordering and aggregation.

Our goal in this paper is to provide means for a user to specify queries without the burden of precise schema knowledge while at the same time not burdening the system with having to guess the desired result of keyword search. Before we describe our approach towards this goal, let us examine a motivating example to appreciate the issues.

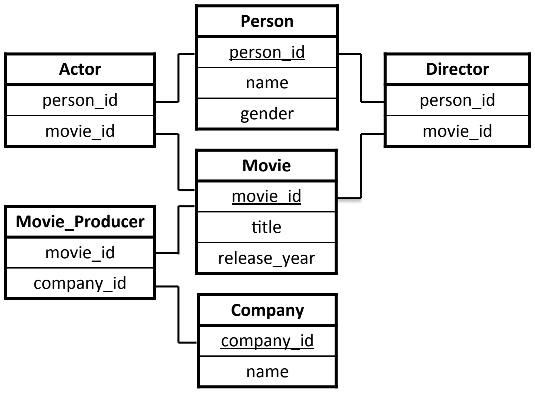
Consider a movie database with a simplified schema shown in Figure 1. When trying to use keywords to express the query example, many difficulties arise. First, traditional keyword search is designed to find keywords in database values rather than in metadata. Keywords cannot convey the information that "male" relates to "actor" and that "James Cameron" relates to "director". Fielded keyword queries have been suggested as a means towards addressing this difficulty part-way, but typically require that field names be specified precisely. Moreover, keywords cannot express comparisons and computations such as the clause "from 1995 to 2005". Similarly, "the number of", which is an aggregation function, is not supported in keyword search.

In short, an unstructured set of keywords is not capable of capturing the rich semantics of query intent required to produce good answers. Surely, we can all remember situations where we were frustrated because we couldn't specify some crucial structure in a query (e.g. movies before 2005). Though such distinctions are easy to make in SQL, writing SQL has its own challenges, as we see next.

In Figure 1, the movie information is split into six relations after normalization. As a result, even queries over such

[*] Supported in part by NSF grants IIS 1250880 and IIS 1017296

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588571.



Query Example: Return the number of male actors who have cooperated with director "James Cameron" in a production by "20th Century Fox" from 1995 to 2005.

Figure 1: Running Example.

a tiny schema may be hard to compose in SQL. For example, a simple query for (the name of) the director of "Star Wars" requires a join of three tables (Person, Director, and Movie). The reader can verify that the query example requires joining 7 relations: all 6 relations in the figure and one of them (Person) twice. The corresponding SQL query has over a dozen clauses. Furthermore, it involves tables (such as Person) and columns (such as Person_id) that are not present in the English language statement of the query intent. Only someone knowledgeable about the schema can figure out that these are required. The situation gets much worse in real-world databases when there are large numbers of relations (The Yahoo! Movie database has 43 relations). We have personal experience with scientist collaborators struggling with such schema issues as they attempt to conduct "e-science".

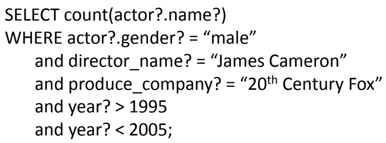
Writing correct SQL is challenging not only for new users, but also for professional DBAs who have to deal with complex enterprise schemas. Even technically competent programmers in MIS shops would appreciate being freed from having to remember large schemas and specify error-prone long sequences of joins. It would be beneficial if users can use SQL structure to express their query logic without spending time to understand the schema in detail.

The basic idea in Schema-free SQL is to reduce the user's burden in writing SQL queries through two important relaxations:

1. *Schema Relaxation*: users do not have to specify schema elements exactly, including relation names and attribute names.
2. *Join Path Relaxation*: users do not need to specify join paths, including which intermediate relations are involved and how they are joined with one another.

If no schema element or join path is specified, observe that the SQL query degrades to a set of attribute values (keywords), but with SQL structure. For example, it is still possible to query for an attribute that is greater than 1995. (In contrast, a pure keyword system could only look for attribute values equal to 1995). However, our relaxation is not that schema information must not be specified, but rather that it *may* not be specified. In other words, to the extent that the user is able to guess (or know) element names and join paths, the user should be able to include it. In the extreme case, the user may even specify a full SQL query, with no relaxation at all. However, in a typical situation, the user will specify as much as they can with ease, and the system should take care of the rest.

Example 1. *Figure 2 shows an example of Schema-free SQL for the query example in Figure 1. Observe that many relation names and attribute names are guessed wrong, but are nevertheless intuitively informative. Observe also that the query assumes a database structure different from the actual. For example, it refers to actor.name even though name is not an attribute of the relation Actor: it is the system's responsibility to understand that this refers to Person.name through a join between Person and Actor. Observe, further, that there isn't even a requirement that the user be consistent. For example, the query has in it actor.name and also director_name. Finally, note that the join through Movie remains implicit in this query just as it was in the English language version: while the user may have been able to specify this crucial element, this is exactly the sort of hidden assumption that requires a high level of understanding of the underlying database schema to make it explicit.*



```
SELECT count(actor?.name?)
WHERE actor?.gender? = "male"
    and director_name? = "James Cameron"
    and produce_company? = "20th Century Fox"
    and year? > 1995
    and year? < 2005;
```

Figure 2: Schema-free SQL.

Typically, a schema-free SQL query will be evaluated by an RDBMS after mapping each vague schema element to its similar schema elements in the database, and autocompleting the join path using the strongest join network that connects all the specified schema elements. Specifically, in the mapping process, we first preprocess all schema elements specified in a Schema-free SQL into a set of *relation trees*, in which each relation tree collects user specified schema elements relevant to the same relation. Then, we map relation trees to relations in the database based on a proposed similarity function. In the join path generation, we define the strength for join networks on *view graph*, which is a modification to schema graph. A view graph represents as views all known join path fragments, including those specified by the user and observed from query logs. Intuitively, the join networks constructed from these views are more likely to be "good" join paths than those constructed by connecting each single relation. Based on this intuition, in our system, a join network tends to be evaluated stronger if it contains views. Experimental results show that the strongest join networks generated on view graph are more likely to be the correct join path than those generated on schema graph.

The intellectual contributions of this paper are as follows:

1. *Schema-free SQL Framework.* We present Schema-free SQL in which users can query relational databases with whatever partial knowledge of the schema they have, all the way from full SQL to just keywords.
2. *System Architecture.* In Section 2, we describe a modular architecture that supports Schema-free SQL.

3. *Relation Tree.* In Section 3, we propose relation tree as a data structure to uniformly represent all specified schema elements. We then define an *l-relation trees query* as a generalization of an l-keyword search. The result of an l-relation trees query is a join network of relations, which contains all the information we need to transform a Schema-free SQL into a full SQL query. However, obtaining a good join network requires some creativity, as we discuss next.
4. *View Graph.* In Section 5, we define view graph as a modification to schema graph, which is widely used in keyword search. Using this concept, we propose algorithms for join networks generation in Section 6.

Other parts of the paper are organized as follows. In Section 4, we provide similarity evaluation functions to map relation trees to relations in the database. In Section 7, the usability and accuracy of our method is tested experimentally. We discuss related work in Section 8. In Section 9, we draw conclusions and point to future work.

2. OVERVIEW

In this section, we define the syntax and semantics of Schema-free SQL and describe the system architecture.

2.1 Syntax and Semantics

Schema-free SQL is an extension of SQL and hence can support all the functions provided by SQL. It follows the syntax of SQL with two relaxations:

1. **Schema Relaxation**: Users do not have to specify the exact schema elements including relation names and attribute names. They can express their uncertainty by means of a question mark in three ways:

   (a) "foo?" indicates that the user guesses the name of the element is foo, but is not sure about this. In the SELECT clause of the query in Figure 2, the user thinks there is a relation called "actor" with an attribute called "name".

   (b) "?x" indicates that the user has no clue about its name. She calls it x as a placeholder. The variable x can be used elsewhere in the query to indicate the same element and distinguish it from other elements, whatever be its name.

   (c) "?" also indicates an element for which the user has no clue of its name. The user doesn't have to bind it to a dummy variable name (such as x in the preceding case). The system generates a unique new dummy variable for each occurrence.

   Users may also choose not to mention the schema element at all. In the WHERE clause of the query in Figure 2, the user mentions an attribute called "year?" without saying which relation it comes from.

2. **Join Path Relaxation**: Users do not need to specify which relations are involved and how they are joined with each other. They can just leave the FROM clause blank (or partially populated), and leave out the foreign-key-primary-key (FK-PK) join path (in the WHERE clause). They can introduce relation names (exact or approximate) in other clauses (such as WHERE and SELECT) without these relations having been mentioned in the FROM clause.

A Schema-free SQL query is an under-specified SQL query, with under-specifications on account of only the two relaxations listed above. In consequence, Schema-free SQL supports all of SQL, including nesting, aggregation, and the many obscure functions provided in any particular vendor-specific implementation.

Once these relaxations are resolved, we obtain a fully specified SQL query, with the usual SQL semantics. Fixing the relaxations in Schema-free SQL is an inherently heuristic task, in which the goal is to infer the user's intent. Specifically, the schema relaxation is resolved by binding each vague schema element to its similar schema elements in the database, while the underspecified join path is completed by the strongest join network that connects all the bound schema elements.

2.2 Architecture

Our system first fixes the two relaxations in a Schema-free SQL query, then translates it into a standard SQL query and finally evaluates it to get accurate query results (it also supports returning top k translations directly to the user before evaluating the best one). A high level representation of the architecture is shown in Figure 3. The Relation Tree Mapper is used to fix the schema relaxation and the Network Builder fixes the join path relaxation. These two main modules are preceded by a Schema-free SQL Parser and succeeded by a Standard SQL Composer. Here we describe these four components of the system intuitively. A more precise and detailed description will have to wait until the following sections.

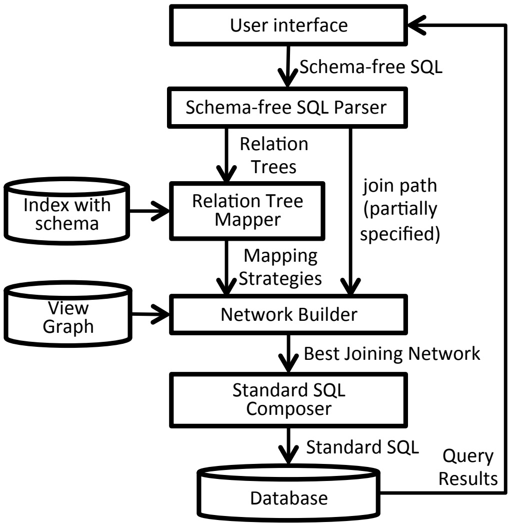


Figure 3: System Architecture.

2.2.1 *Schema-free SQL Parser*

The schema elements and join paths that users specify in Schema-free SQL can be vague and fragmentary. The Schema-free SQL Parser separates the schema elements and join paths from the rest of the query, which will remain unchanged. The schema elements are merged and uniformly represented as a set of trees called *Relation Trees* while the join paths (if partially specified) are represented by *views* as we will discuss below.

2.2.2 Relation Tree Mapper

For a traditional l-keyword search, it is easy to map a keyword to relations: if one of the tuples in the relation contains the keyword, the keyword can be mapped to that relation. But in the case of mapping relation trees, this is not that obvious since relation trees have a much more complex structure with multiple components to be matched. In the Relation Tree Mapper, we first evaluate the similarity between relation trees and relations in the database, then map them based on their similarity.

2.2.3 Network Builder

To get the full SQL query, we need to generate the correct join network of relations that contains all the relation trees. There is a combinatorial number of possible join networks, only one (or, occasionally, a few) of which is correct. The network builder generates join networks using a view graph, which models all supervised information (e.g. join path fragments specified by the user, query patterns in query logs) as views. Intuitively, join networks composed of these views are more likely to be correct than those constructed by connecting each single relation from scratch. We present ways to rank all join networks and develop algorithms to generate the top k join networks efficiently.

2.2.4 Standard SQL Composer

Each join network obtained from the previous step gives a possible interpretation of the Schema-free SQL query. Based on the join path in this network, and the mapping strategy used, the Standard SQL Composer instantiates exact schema elements in place of user's guesses, introduces appropriate join conditions in the WHERE clause, and fills in the FROM clause. The result is a correctly formed SQL query, which hopefully matches the user's intent. Note that k different SQL queries can be output, one for each join network returned from the preceding step. Typically, we may set k to 1, evaluate the full SQL translated and return the query results to the user. However, the system architecture is capable of returning multiple options when desired.

2.2.5 Nested Query

The sequence of steps described above applies only to a single-block SQL query. Given a nested query, it is processed one block at a time, starting from the outermost block, so that values for any correlated variables and other context is already set when inner blocks are processed.

3. REPRESENTING SCHEMA-FREE CONTENT

Our first task is to instantiate the correct names for all relations and attributes that are insufficiently (or even incorrectly) specified in a given Schema-free SQL query. As a starting point, we have the guesses at names provided by the user in the query itself. Additionally, we may have guesses at structure, relating attribute to table, in the query. Finally, we may have values for some attributes in the query specification, giving us a hint of what those attributes may be. To put all of these constraints together in one framework, we introduce the concept of *relation tree* in this section. Then, in the next section, we consider how to map relation trees to the actual schema of the existing database.

As a first step towards obtaining relation trees, we denote each occurrence of schema-related content by an expression triple, comprising relation name, attribute name, and value condition, some of which may be undefined or inapplicable.

3.1 Expression Triples

There are altogether three kinds of schema-relevant expressions in Schema-free SQL: (a) the relation names in *FROM clause*, both original names and aliases, (b) the attribute names (together with relation names if specified) in all other clauses, and (c) the value constraint conditions (together with relation names and attribute names if specified) in *WHERE clause*. All other information in the query is considered schema-irrelevant, including (a) general keywords like: SELECT, FROM, WHERE, GROUP BY, ORDER BY, ASC, OR, (b) aggregation keywords like SUM, COUNT, MAX, (c) Computation symbols like +, -, *.

EXAMPLE 2. *In the query example in Figure 2, consider the clause "SELECT count(actor?.name?)". We do not need to know what "SELECT" and "count" mean. Instead, the only thing we need to do is to map "actor?.name?" to the attribute "Person.name" in the database. Then the clause is transformed to "SELECT count(Person.name)", which is the SELECT clause in the standard SQL.*

We uniformly represent all expressions as *expression triple*s with three entries. The three entries store the relation name, attribute name and condition constraint, respectively. If an expression does not specify the relation name, attribute name or condition constraint, the corresponding entry stores a star mark instead. For convenience, we represent these triples as trees of height three and call the three levels as relation level, attribute level and condition level respectively. The upper part of Figure 4 shows all the expressions in Figure 2. In the next subsection, these expression triples will be merged into relation trees.

3.2 Relation Trees

The expression triples are often not independent of one another. We merge related expression triples according to the following rules:

1. Expression triples with identical relation name (their aliases must also be identical if specified) are merged at the relation level.
2. Expression triples with both identical relation name and identical attribute name are merged at the attribute level.
3. Expression triples that have identical attribute name, but do not specify the relation name, are merged at the attribute level.

The merged results are called **Relation Tree**s since each of them collects the schema information related to one relation in the database. Similarly, subtrees at the attribute level are called **Attribute Tree**s.

EXAMPLE 3. *The merging process of expression triples in the query example is shown in Figure 4.*

After the preprocessing, all the schema-relevant information are transformed into a set of relation trees, denoted as $RT = \{rt_1, ..., rt_l\}$. We call this an ***l*-Relation Tree Query**, which can be considered as a generalization of a traditional l-keyword search, with two major differences:

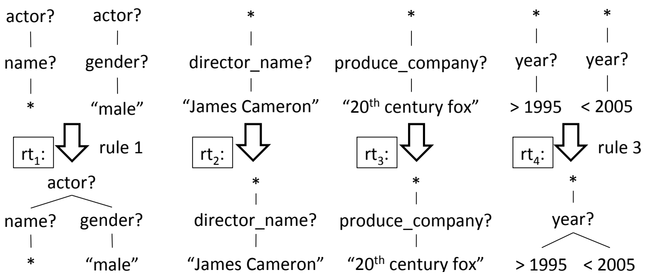


Figure 4: Merging Expression Triples.

- In l-keyword search, it is easy to tell if a keyword should be mapped to a relation using the *contains* function. Since a relation tree often has much more information than a keyword, we cannot use simple containment in an l-relation trees query. In the next section, we provide similarity functions to evaluate the similarity between relation trees and relations, and map them by their similarity.
- In l-keyword search, many join networks of relations which contain all keywords will be evaluated. Their evaluation results are then ranked and the most relevant ones are returned to the user. In contrast, in l-relation trees query, join networks of relations are ranked before evaluation. Only the best join network of relations will be evaluated to return the exact query results to the user. We provide mechanisms to rank join networks in Section 5 and generate the best (or top k) join network efficiently in Section 6.

4. MAPPING RELATION TREES

In this section, we discuss the mapping from relation trees to relations in the database, which is based on their similarity. There are many ways to define similarity based on previous efforts described in the literature. All are heuristic with no clear universal best. The goal of this section is not to find the best similarity evaluation function. Rather, we give a framework for the similarity evaluation function and recommend a "good" one we use in our system.

For each relation tree rt, we call the set of its mapped relations as the *Mapping Set* of r, and denote it by $MAP(rt)$. We define $MAP(rt)$ as follows:

Definition 1 (Mapping Set of a Relation Tree). *Let rt be a relation tree, $V = \{R_1, ..., R_n\}$ be the target database with n relations, $Sim(rt, R_i)$ be a similarity function between rt and relation R_i, and σ be a predefined relative threshold between 0 and 1. The Mapping Set of rt is defined as $\{R_i | Sim(rt, R_i) > \sigma * MAX\{Sim(rt, R_j)|1 \leq j \leq n\}\}$.*

Note that we do not simply pick the one relation with maximum similarity, since relation tree similarity alone may not give us the right answer. Instead, we would like to keep in play all relations with high similarity. We could define an absolute threshold for this purpose. However, we have preferred to do so in terms of a fraction σ of the maximum, for the following intuitive reason: when the user specifies the name(s) correctly, there is usually one relation with high similarity, and we can ignore all others with low similarity; on the other hand, if the user specifies the name poorly, there is no good match and all similarities are low, and in this case we want to keep around several likely candidates.

4.1 Similarity Evaluation

Let rt be a relation tree with relation name $n(rt)$ and attribute trees $\{at_1, ..., at_m\}$, R be a relation in the database. Intuitively, rt is similar to R if R contains similar information for $n(rt)$ (similar at root level) and each at_i (similar at attribute level). We use $Sim(n(rt), R)$ and $Sim(at_i, R)$ to denote the similarity at root level and attribute level respectively, which will be formally defined later. The similarity between rt and R is defined as follows:

$$Sim(rt, R) = Sim(n(rt), R) \prod_{i=1}^{m} Sim(at_i, R)$$

4.2 Similarity at Root Level

If there is a relation name at the root of a relation tree, similarity with this name is obviously a major indicator of a matching relation in the database. Frequently, the user is off in specifying the relation name because the user has a different schema in mind than the database, with normalization being a common culprit in this regard. To allow for this, we also match the name at the root of a relation tree with names of neighboring relations.

Formally, let $n(rt)$ be the relation name of relation tree rt, $n(R)$ be the relation name of relation R. Suppose that $\{R_{neighbor}\}$ is all the relations that R refers to or is referred by. rt is similar with R at root level if $n(rt)$ is similar to $n(R)$ or there exists a $R_i \in \{R_{neighbor}\}$ with $n(R_i)$ very similar to $n(rt)$. The similarity is formally defined as follows:

$$Sim(n(rt), R) = MAX(Sim(n(rt), n(R)), \{Sim'(n(rt), n(R_i))\})$$

In the equation, R_i iterates over all the relations in $\{R_{neighbor}\}$. $Sim(a, b)$ and $Sim'(a, b)$ are similarity functions between two strings with the constraint that $Sim(a, b)$ is larger than $Sim'(a, b)$. In our implementation, we use the Jaccard Coefficient between the q-gram sets of a and b as $Sim(a, b)$, and multiply it with k_{ref}, a predefined constant between 0 and 1, to compute $Sim'(a, b)$.

Example 4. *Take the relation tree rt_1 in Figure 4 and all relations in Figure 1 as an example. The root of rt_1 is actor?. For relation Actor, the similarity at root level, $Sim(actor, Actor)$ is 1. Suppose k_{ref} is 0.7. rt_1's similarity with $Person$ and $Movie$ is 0.7 since they are both referred to by relation Actor.*

Sometimes, $n(rt)$ may not be specified. In this case, we try to find some clues in ats for the mapping. We first set the root similarity to, k_{def}, a small default value, then we use each attribute name in ats instead of $n(rt)$ to compute the similarity at root level. We update the root similarity each time when the computed similarity is higher than the current root similarity.

4.3 Similarity at Attribute Level

Let at be an attribute tree and R be a relation with attributes $\{A_1, ..., A_m\}$. We map at to the attribute A_i that is most similar to at. The similarity between at and R is formally defined as follows:

$$Sim(at, R) = MAX(\{Sim(at, A_i)|1 \leq i \leq m\})$$

Intuitively, an attribute tree at is similar to an attribute A if (a) their attribute names are similar, (b) the conditions (if specified) under the attribute tree are satisfied by the tuples in the attribute. Let $n(at)$ and $n(A)$ be the attribute name of at and A respectively. The similarity between at and A is defined as follows:

$$Sim(at, A) = Sim(n(at), n(A)) * \frac{m+1}{n+1}$$

In the equation, $Sim(n(at), n(A))$ denotes the string similarity between the attribute names of at and A. $\frac{m+1}{n+1}$ reflects whether the condition constraints are satisfied, in which n denotes the total number of condition constraints under at and m denote the number of conditions constraints which can be satisfied by the tuples in A.

5. VIEW GRAPH

The previous sections mapped relation trees to relations in the database. Our next task is to find a join network that contains these mapped relations, based on which we can specify the equivalent SQL query. A similar problem arises in schema-based keyword search (e.g. DISCOVER [8]). We cannot use their techniques directly for two reasons. First, in Schema-free SQL, the user may specify part of the join path, which should be carefully taken into account in the join path generation. Second, in keyword search, the output of the join path generation is usually all the possible join networks within a length threshold. In contrast, the output in our method is only the (few) best join network(s).

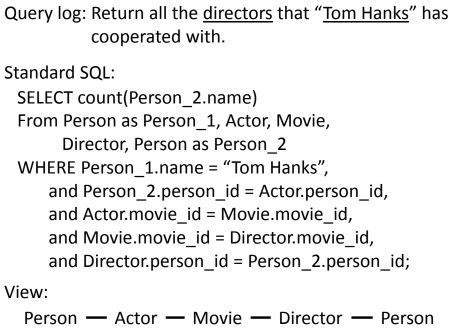
In this section, we introduce the notion of view graph, which captures partially specified join path and the join path fragments in the query log as views, which suggest join networks that are better than others.

5.1 Model

The schema of the database can be represented as an undirected graph $S(V, E)$, where V is the set of relations in the database $\{R_1, ..., R_n\}$, and there exists an edge (R_i, R_j) in E, if a foreign key defined on R_i refers to the primary key defined on R_j (FK-PK relationship). In addition, let $VIEW$ represent the set of predefined views $\{view_1, ..., view_m\}$. A *view* in $VIEW$ is a connected tree of relations with each edge being a join (not necessarily a FK-PK relationship join) in the view definition. We can add these views to the schema, to define an undirected **View Graph**, $G(V, E, VIEW)$.

Views can come from various sources. First, the join path may be partially specified by the user in the schema-free query. If the specified join path is not connected, each of its connected parts will be transformed to a view. Second, query patterns mined from query logs and forms designed by experts may also be suggestive of likely queries, and are therefore transformed into views. More sophisticated techniques, such as those suggested by [13], may also be used beyond this, if desired. Intuitively, join networks constructed from these views are more likely to be reasonable than those constructed by arbitrarily connecting single relations from scratch.

Example 5. *Consider the query log along with its corresponding standard SQL shown in Figure 5. We transform it to a view as shown. For simplicity, we take this view as the unique view in the view set. This view set and the schema graph in Figure 1 forms the view graph.*



Query log: Return all the directors that "Tom Hanks" has cooperated with.

Standard SQL:

```
SELECT count(Person_2.name)
From Person as Person_1, Actor, Movie,
     Director, Person as Person_2
WHERE Person_1.name = "Tom Hanks",
    and Person_2.person_id = Actor.person_id,
    and Actor.movie_id = Movie.movie_id,
    and Movie.movie_id = Director.movie_id,
    and Director.person_id = Person_2.person_id;
```

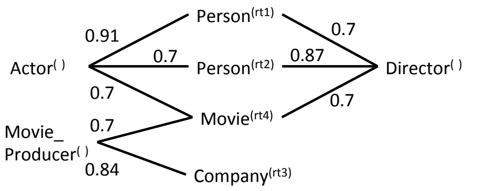
View:

Person — Actor — Movie — Director — Person

Figure 5: View Example.

Given a relation mapping for each relation tree, there are many possible join networks connecting them. In the next section, we will discuss how to score these networks to choose the best. But first we have to deal with the additional complication that there are several candidate mappings for each relational tree. To address this challenge, we introduce the notion of an **Extended View Graph**, denoted as $G_X(V_X, E_X, VIEW_X)$. Given a view graph $G(V, E, VIEW)$ and an l-relation trees query $RT = \{rt_1, ..., rt_l\}$, for each node $R_i \in V$ and each relation tree $rt \in RT$ that maps to R_i, there exists a node in G_X denoted as $R_i^{(rt)}$. There is also a node $R_i^{()}$ in G_X for each node R_i that has no relation tree mapped to it. There exists an edge $(R_1^{(rt1)}, R_2^{(rt2)})$ in E_X if there is an edge (R_1, R_2) in E. Similarly, for each view in G, we have to consider all possible mappings of relation trees to each node in the view. Each of these gives rise to a view in $VIEW_X$.

Example 6. *For the four relation trees in Figure 4, we assume that rt_1 and rt_2 map to Person, rt_3 maps to Company and rt_4 maps to Movie. This mapping gives rise to the extended view graph shown in Figure 6. Note that there are two ways to transform the view example in Figure 5, one replaces the leftmost Person by $Person^{(rt1)}$ while the other replaces it by $Person^{(rt2)}$. The values on the edges are their weights, which will be formally defined later.*



Views:

view$_1$: Person$^{(rt1)}$ — Actor$^{()}$ — Movie$^{(rt4)}$ — Director$^{()}$ — Person$^{(rt2)}$

view$_2$: Person$^{(rt2)}$ — Actor$^{()}$ — Movie$^{(rt4)}$ — Director$^{()}$ — Person$^{(rt1)}$

Figure 6: Extended View Graph.

With the extended view graph in place, we can restrict the join networks we consider to satisfy a given l-relations trees query as follows:

Definition 2 (Candidate Join Network). *Given an l-relation trees query $RT = \{rt_1, ..., rt_l\}$ and a relational database with extended view graph $G_X(V_X, E_X, VIEW_X)$, a* *candidate join network is a connected tree of nodes where for every two adjacent nodes $R_1^{(rt1)}$ and $R_2^{(rt2)}$ in the tree, $(R_1^{(rt1)}, R_2^{(rt2)})$ is in E_X or in a view in $VIEW_X$. Note that $(R_1^{(rt1)}, R_2^{(rt2)})$, $(R_1^{(rt1)}, R_2^{(rt3)})$ cannot exist in a join network at the same time if the primary key of R_2 in $R_2^{(rt2)}$ and $R_2^{(rt3)}$ is referred to by the same foreign key in R_1.*

Definition 3 (Minimal Total Join Network (MTJN)). *An MTJN is a candidate join network that satisfies the following two conditions:*

- *Total: the join network contains all relation trees in RT.*
- *Minimal: the join network is not total if any relation in it is removed.*

5.2 Ranking of Results

Given an l-relational tree query and an extended view graph, the MTJN is not unique. In fact, there will typically be many MTJNs, and only one of them corresponds to the correct interpretation of the query. To be able to tell which MTJN is good, we have to score it, and we do this scoring based on weights we assign to edges in the (extended) view graph. In other words, rather than just considering presence of edges in the view graph, we now weight this presence depending on how likely that edge is.

To each edge e, we assign a weight $w(e)$, with a value between 0 and 1, where a larger weight indicates a stronger connection. While this is *not* formally a probability, it is intuitively useful to think of it as if it were the probability that this edge is in the user-desired query. $w(e)$ initializes all edges to a default constant c. Then for each edge $e = (R_1^{(rt1)}, R_2^{(rt2)})$, its connection is enhanced by the following equation, which follows the standard disjunction of probabilities for independent events: $w(e) = 1 - (1 - c) * (1 - MAX(Sim'(n(rt_1), n(R_2)), Sim'(n(rt_2), n(R_1))))$, where Sim' is defined in Section 4.2. Intuitively, rt_2 contains information specified by the user for R_2. High similarity between rt_2 and R_1 may mean some relationship between $R_1^{(rt1)}$ and $R_2^{(rt2)}$. Thus their connection is enhanced.

Example 7. *The edges in Figure 6 are marked with their weights. Let $c = 0.7$. Suppose that $Sim'(n(rt_1)$,$n(Actor))$ equals to 0.7. Then the edge of $(Actor^{()}, Person^{(r1)})$ is weighted $1-(1-0.7)(1-0.7) = 0.91$. The weights of other edges are computed similarly.*

The weight of a path is naturally computed as a product of the weights on its edges – intuitively multiplying probabilities of independent events to find the probability of the conjunction.

Definition 4 (Basic Weight of JN)). *Let jn be a join network constructed by edges $E' = \{e_i\}$. The basic weight of jn is: $w_{basic}(jn) = \prod_{e_i \in E'} w(e_i)$*

Definition 5 (Weight of View). *Let v be a view constructed by edges $E' = \{e_i\}$. The weight of v is: $w_{view}(v) = (\prod_{e_i \in E'} w(e_i))^{\frac{1}{2}}$*

In general, the weight assignments on views can be very elaborate. For example, views transformed from partial join path specified by the user should have very high weight. Similarly, query patterns mined from the query log can have different weights according to their frequency and other properties. Such careful tuning is beyond the scope of this paper.

Definition 6 (Construction Weight of JN). *Let jn be a join network built by views $VIEW' = \{v_i\}$ and edges E' $= \{e_j\}$ (not contained by any v_i). The construction weight of jn is: $w_{con}(jn) = (\prod_{v_i \in VIEW'} w(v_i))(\prod_{e_j \in E'} w(e_j))$*

The above formula gives the weight of a particular construction of a join network. In many cases, the same join network can be constructed in more than one ways, with one constructed only by edges and others containing views. In this case, we take the highest construction weight of the join network as its weight.

Definition 7 (Weight of JN). *Given a join network jn with all possible construction weights $\{w_{con}^1(jn)$,..., $w_{con}^n(jn)\}$, the weight of jn is defined as:*

$$w(jn) = MAX(w_{con}^1(jn), ..., w_{con}^n(jn))$$

Example 8. *One join network in Figure 6 is shown in Figure 7. If this join network is constructed from scratch, its construction weight is the multiplication of all its edge weights, which is 0.23. Another way to construct the example join network is to use the $view_1$ in Figure 6. $view_1$ itself is weighted $\sqrt{0.91 * 0.7 * 0.7 * 0.87}$, which is 0.62. Thus the construction weight in using $view_1$ is $0.62 * 0.7 * 0.84 = 0.36$. This is the highest construction weight of this join network. So its weight is 0.36.*

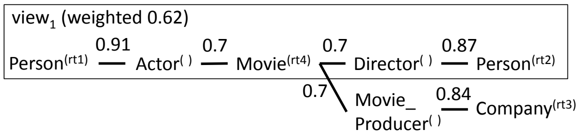


Figure 7: Weight of a Join Network.

6. GENERATING TOP K INTERPRETATIONS OF SCHEMA FREE SQL

The weight functions described above are heuristic, with no guarantee of the top ranked network being the one desired. For this reason, we would like to be able to generate the top several choices. Given the combinatorial number of possible join networks, it is important to develop efficient algorithms to accomplish this. In Section 6.1, we give algorithms to efficiently generate k MTJNs with the highest weights. In Section 6.2, we translate the input Schema-free SQL into a full SQL with the help of an MTJN.

6.1 Top k MTJNs Generation

The basic idea in our algorithm is to first select a set of nodes in the graph as initial JNs and then gradually expand this set until top k minimal total ones are generated. The resulting Algorithm 1 is shown in Figure 8. We choose the nodes that are mapped by the first relation tree as roots to expand. We rank these roots by their potential (line 2) and expand high potential ones first. (How to estimate potential will be discussed at the end of this subsection). KMTJNUpdate, shown in Algorithm 2, is the main workhorse of Algorithm 1. We call it to expand each root and update $kMTJN$

(line 4). To avoid generating isomorphic MTJNs from different roots, we delete each root from the extended view graph after it is expanded (line 5).

The efficiency of the algorithm mainly depends on the total number of JNs expanded. In DISCOVER [8], JNs are expanded arbitrarily. As a result, large numbers of isomorphic JNs are generated since the edges in a JN can be added in different orders. In our algorithm, we use *legality test* to ensure that each JN will be generated only once (line 7). But even without generating isomorphic JNs, the total number of JNs still increases exponentially with the length of JNs. So it is very beneficial if the JNs that do not have the potential to generate a top k MTJN can be pruned out early. In our algorithm, we use *potential estimation* to prune out JNs with no potential early (line 11-12). If a JN cannot expand to an MTJN with weight higher than the current kth MTJN's weight, it will be pruned out. To maximize this pruning benefit, we generate k MTJNs with high weights early by using a priority queue to store all partially expanded JNs ordered by their potential and expanding high potential ones first (line 3-12).

```
Algorithm 1: InitMTJNGen($G_X$($V_X$, $E_X$, $VIEW_X$), $RT$)

1: $kMTJN \leftarrow \phi$
2: rank all $R_i$ mapped by $rt_1$ by decreasing $R_i$.potential
3: for all $R_i$, do
4:     KMTJNupdate($R_i$, $G_X$, $kMTJN$)
5:     remove $R_i$ from $G_X$
6: return kMTJN;
```

Figure 8: Top k MTJNs Generation

```
Algorithm 2: KMTJNupdate($R_i$, $G_X$($V_X$, $E_X$, $VIEW_X$), $kMTJN$)

 1: PriorityQueue $\leftarrow \phi$
 2: jn = a tree of single node $R_i$
 3: PriorityQueue.push(jn)
 4: while PriorityQueue $\neq \phi$ do
 5:     jn = PriorityQueue.pop()
 6:     for all $e \in E_X$ and $view \in VIEW_X$ do
 7:         if e (or view) can be legally added to jn then
 8:             jn' = a tree expanded from jn by adding e (or view)
 9:             if jn' is both minimal and total then
10:                 update(kMTJN)
11:             else if jn'.potential $\neq 0$ then
12:                 PriorityQueue.push(jn)
13: return kMTJN
```

Figure 9: Top k MTJNs Update

Now we discuss the two important functions, legality test and potential estimation, in detail.

Legality Test: We could, by adding edges in different orders, create the same (isomorphic) join network (JN) multiple times from the same root. To avoid that, rightmost path expansion technique has been proposed in [12]. In their method, a unique numeric label is added to each node in the graph and a JN is modeled as a rooted, ordered tree, in which the children of each node are ordered by their numeric label. The intuition is that only the rightmost nodes, at any level, in the tree are allowed to expand and the newly expanded node must be the rightmost node at its level. In the case of extended view graph, the situation is more complex since a JN can be expanded by a view, which means a set of edges will be added to the JN at the same time. We adapt the rightmost path expansion to extended view graph.

Given a join network jn and a view $view$, $view$ can be legally added to jn only if they share exactly one node (the shared node must be a rightmost node in jn). Now, $view$ is also considered as a rooted, ordered tree, which is rooted at the shared node (denoted as $root(view)$) and orders each node's children by their numeric labels. The expansion is considered as the $view$ is added to jn at $root(view)$. $root(view)$ might have children both in $view$ and jn. In this case, the rightmost child of $root(view)$ must be in $view$, not in jn. Otherwise, the expansion will be considered illegal. Also, each view has a numeric label, if jn has already been expanded by some views, the label of the newly added $view$ must be larger than all previously added views. After expanding jn with $view$, all the previous nodes in jn, which are to the left of $view$ (the order of the node is smaller than the biggest order in the view in post-order traversal), are marked non-rightmost. While all newly expanded nodes are marked as rightmost no matter if they are in the rightmost root-to-leaf path. Following these rules, each JN will be generated at most once in the whole process[1].

EXAMPLE 9. *Take the JNs in Figure 10 as an example. We suppose that the numeric label of Movie_Producer is smaller than the numeric label of Actor and the numeric label of Actor is smaller than the numeric label of Director. So if they appear as siblings to each other, Movie_Producer is always to the left of Actor and Actor is always to the left of Director. Look at the JN (e), the nodes of* $Movie^{rt4}$*, Director and* $Person^{rt2}$ *are the rightmost nodes at each level. To distinguish rightmost nodes from others, all the rightmost nodes are marked red. JN (d) can never be expanded from JN (b), since the node Movie_Producer is not rightmost node and cannot be expanded any more. Similarly, the expansion from JN (d) to JN (e) is not an legal expansion, since the newly added node Actor is to the left of an existing child Director. Also, the JN (b) can never be an MTJN since the node Movie_Producer is not rightmost and hasn't been mapped by any relation tree, which will violate the Minimal Condition after it reach the Total Condition. So the expansion from JN (a) to (b) is illegal. JN (c) can be expanded to JN (f) directly by adding the* $view_1$ *in Figure 6, since JN (c) and* $view_1$ *share only one node* $Movie^{rt4}$ *and the rightmost child of* $Movie^{rt4}$ *is Director, which is in* $view_1$*, not in JN (c). Note that all the nodes of the newly added view are marked rightmost.*

Potential Estimation: Our pruning is based on a JN's potential to generate a top k MTJN. Given a JN jn, we approximate the upper bound of the weight of all the MTJNs that can be expanded from jn (denoted as $upper(jn)$) and use it as its potential. We set the potential to 0 if the approximated upper bound is lower than the weight of the current kth MTJN. The benefit of using $upper(jn)$ is two-fold. First, if $upper(jn)$ is lower than the weight of the current kth MTJN, jn can be pruned out almost safely. Second, the higher $upper(jn)$ is, the more likely jn can generate a top k MTJN.

[1] The only exception is when a JN can be constructed in more than one way using different components. For each construction, the JN may be generated once.

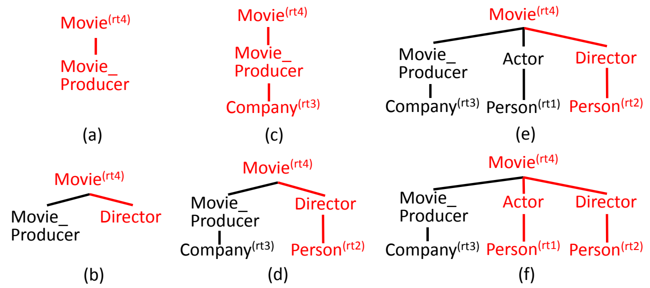


Figure 10: Rightmost Path Expansion

The potential estimation algorithm is shown in Figure 11. We first find all relation trees not contained by jn and then put the nodes mapped by these relation trees into a list L. For each round, we choose the node R^{rtj} that has the shortest (highest weight) path to jn and connect it through the path. After that, we delete all the nodes in L to which rt_j maps. We repeat this process until the jn is total. We use the weight of this jn as its potential. Later, If the potential is lower than the current kth MTJN's weight, it will be set 0. Consider jn may contain views, before we estimate the potential, we set the weight of each edge on the graph as its square root if the edge is contained in a view.

Algorithm 3: PotentialEstimate(*jn*, $G_x(V_x, E_x, VIEW_x)$, *RT*)

```
jn' = jn; L ; w = jn.weight;
for all $rt_i$ RT
    if jn' does not contain $rt_i$
        L.addAll(Map($rt_j$))
While jn' is not total do
    find $R^{rtj}$ in L which has shortest path p to jn'
    jn'.add(p)
    w *= weight(p)
    R.removeAll(Map($rt_j$))
return w;
```

Figure 11: Potential Estimation

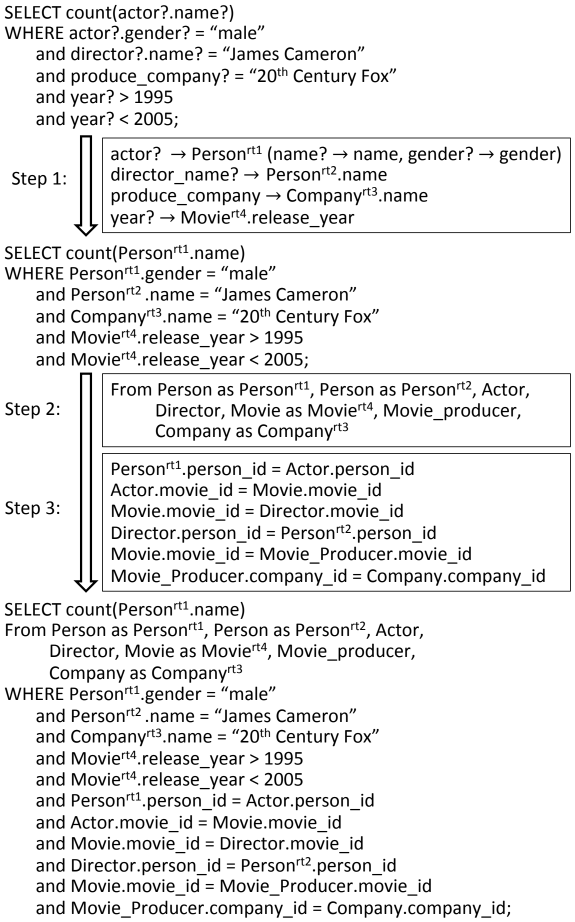
The most computationally complex part in this process is the computation of the shortest paths (highest weight paths) between each pair of nodes in the view graph. Since we can compute all shortest paths once and use them repeatedly, the whole potential estimation process can be efficient.

6.2 Standard SQL Composer

After the generation of the top k MTJNs, k full SQL queries can be generated, one corresponding to each MTJN. This is accomplished as a three-step translation of the given Schema-free SQL query.

1. All the uncertain/unknown relation (attribute) names in the relation tree are replaced by the relation (attribute) names in the corresponding relations in the MTJN.
2. All the relations in the MTJN are included in the *FROM clause*. If a relation appears more than once in the MTJN, the keyword *AS* is used to give an alias to each appearance.
3. All the edges in the MTJN are included in the *WHERE clause* as a join condition to join these relations.

EXAMPLE 10. *Using the MTJN in Figure 7, we transform the Schema-free SQL in Figure 2 into its corresponding full SQL. The transformation process is shown in Figure 12.*



SELECT count(actor?.name?)
WHERE actor?.gender? = "male"
and director?.name? = "James Cameron"
and produce_company? = "20th Century Fox"
and year? > 1995
and year? < 2005;

Step 1:
actor? → Person^{rt1} (name? → name, gender? → gender)
director_name? → Person^{rt2}.name
produce_company → Company^{rt3}.name
year? → Movie^{rt4}.release_year

SELECT count(Person^{rt1}.name)
WHERE Person^{rt1}.gender = "male"
and Person^{rt2} .name = "James Cameron"
and Company^{rt3}.name = "20th Century Fox"
and Movie^{rt4}.release_year > 1995
and Movie^{rt4}.release_year < 2005;

Step 2:
From Person as Person^{rt1}, Person as Person^{rt2}, Actor,
Director, Movie as Movie^{rt4}, Movie_producer,
Company as Company^{rt3}

Step 3:
Person^{rt1}.person_id = Actor.person_id
Actor.movie_id = Movie.movie_id
Movie.movie_id = Director.movie_id
Director.person_id = Person^{rt2}.person_id
Movie.movie_id = Movie_Producer.movie_id
Movie_Producer.company_id = Company.company_id

SELECT count(Person^{rt1}.name)
From Person as Person^{rt1}, Person as Person^{rt2}, Actor,
Director, Movie as Movie^{rt4}, Movie_producer,
Company as Company^{rt3}
WHERE Person^{rt1}.gender = "male"
and Person^{rt2} .name = "James Cameron"
and Company^{rt3}.name = "20th Century Fox"
and Movie^{rt4}.release_year > 1995
and Movie^{rt4}.release_year < 2005
and Person^{rt1}.person_id = Actor.person_id
and Actor.movie_id = Movie.movie_id
and Movie.movie_id = Director.movie_id
and Director.person_id = Person^{rt2}.person_id
and Movie.movie_id = Movie_Producer.movie_id
and Movie_Producer.company_id = Company.company_id;

Figure 12: Full SQL Translation Process.

7. EXPERIMENTAL RESULTS

7.1 Evaluation Method

There are two crucial aspects we must evaluate: whether the system can correctly translate Schema-free SQL queries to full SQL queries (effectiveness), and how much work the user has to do in composing Schema-free SQL queries (usability). In addition, we must ensure that the running time of our algorithm is fast enough for interactive use.

Effectiveness: In keyword search systems, researchers often use IR metrics like precision and recall to evaluate the search quality. But these IR metrics would not work very well for Schema-free SQL since they will always be 100% if a Schema-free SQL is correctly translated into full SQL, and will be near 0% in most times when it is not. So, in our system, we evaluate the effectiveness by counting the fraction of Schema-free SQL queries that can be translated correctly in one of the top k translations. In our experiments, we test the cases when k equals 1 and 10.

User Burden: One way to quantify the user's burden in using a query system is to measure the cost (e.g. keystrokes, time) in the query construction. Recently, in [16, 14], researchers quantified the cost by counting the number of schema elements users specified in query construction. Following their lead, we use the concept of **information unit** as an objective metric to quantify the cost. In a query, any schema element, including a relation name and an attribute name, is an information unit. We evaluate the cost of a query by counting the number of information units used. In Schema-free SQL, many schema elements may be specified approximately, or partially. To avoid a complex model involving partial information units, we significantly overestimate the cost of our system by counting each of these as one full information unit.

EXAMPLE 11. *The cost of the example Schema-free SQL in Figure 2 is 6: actor, gender, name, director_name, year and produce_company.*

Many visual SQL query builders [1, 2, 3] interactively help the user to complete the join path when the user drags and drops a relation to the query construction window. As a representative of this class of interfaces, we use Flyspeed SQL Query [2] to conduct experiments for comparison.

For Schema-free SQL, separate evaluation of the effectiveness or usability is meaningless since our system can take in standard SQL, which has a 100% effectiveness but high user burden, or keywords (with SQL structure), which has low user burden but low effectiveness. The point of Schema-free SQL is not to be at either extreme, but rather to be somewhere in between: the user expresses partial schema knowledge they have and the system does the rest. Furthermore, it is reasonable to assume that users can express selections and projections, but may not be able to specify joins. Our experimental assessment focuses on just such a scenario, to assess the degradation in effectiveness and gain in usability compared to fully specified SQL.

There are several tuning parameters in our algorithms. We ran some initial experiments to determine good values for these parameters. Based on these experiments, not reported for lack of space, we set $\sigma = k_{ref} = c = 0.7$ and $k_{def} = 0.3$.

7.2 Movie Database

In this subsection, we evaluate the effectiveness and users' cost in queries over the Yahoo-Movie database, which has 43 relations and 71 FK-PK pairs. The generation of query sets is a challenging task itself. First, Schema-free SQL is supposed to be composed by users who can express query logic in SQL but do not know the exact schema of the database. Second, we would like the queries to cover most of the major functions of SQL to test whether Schema-free SQL can be correctly translated in different cases.

In order to satisfy these conditions, we chose two query sets. The first is composed of 17 example SQL queries[2] in a textbook [15], which are used to illustrate how to compose SQL queries. This query set includes single relation queries, multi-relation queries, queries with multi-level subqueries, and queries with aggregations. These queries were originally written for a mini database composed of 5 relations, not for Yahoo-Movies. We preprocess all these SQL queries in the following steps to remove some misleading information: delete all the FK-PK join paths in WHERE clause and the relation names in the FROM clause, then merge all the column names with their corresponding relation names. When using these queries to query the Yahoo-Movie database through our system, all 17 queries can be correctly translated to full SQLs in the top 1 translation. Note that in this experiment, no view graph is used to enhance effectiveness. The user's burden for these queries is shown in Figure 13. Generally, these Schema-free SQL queries specify only 35 (respectively, 55) percent as many information units as their corresponding full SQL queries (with a visual query builder). In short, a good visual query builder can reduce the query specification cost, compared to full SQL, but Schema-free SQL can reduce it further.

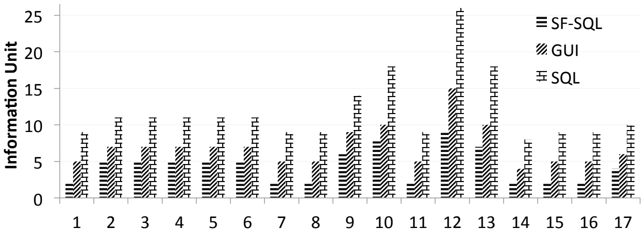


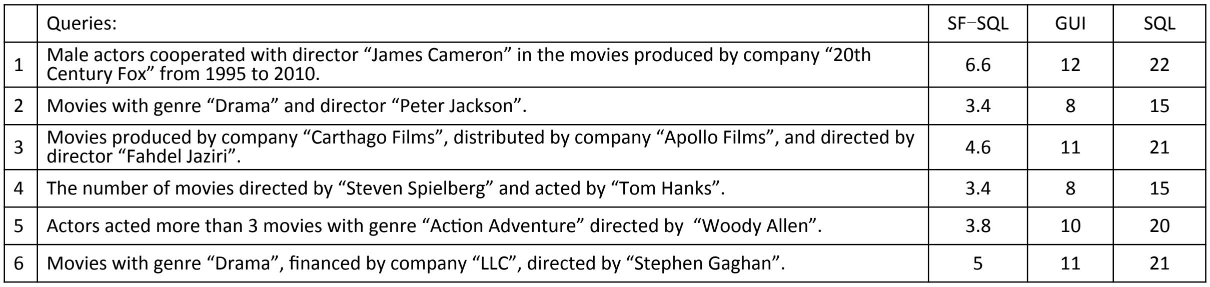
Figure 13: Information Unit Cost for the Queries in the Textbook.

To test the accuracy and usability of our method in sophisticated queries, which refer to many relations and contain complex join paths, we chose six complex queries as our second query set, in which each one has a join path with more than five relations. We recruited five students who are major in information science, all familiar with SQL but not with the schema of Yahoo-Movie, and asked them to specify these six queries, based only on the natural language description shown in Figure 14. The result is that all the users are able to specify all queries in Schema-free SQL and get the correct full SQL queries in the top 1 translation. The average of their information unit cost is only 24 (resp. 45) percent of the full SQL queries (with a visual query builder). Details are shown in Figure 14. Once again, view graphs are not used to enhance effectiveness.

7.3 Course Database

To challenge our system with more sophisticated queries, we obtain a set of 48 complex SQL queries against the course database used in CourseRank [5] comprising 53 relations. We generate Schema-free SQL by deleting all the FK-PK join paths in the WHERE clauses and all the relations in the FROM clauses excepting the relations at the ends of each join path, which are typically used for selection or projection: information which a typical user should be able to specify. We also asked a student with experience in database application development to create his own schema that covers the query intent in all the 48 queries. He designed a schema with only 21 relations, very different from the CourseRank schema. We ran the same Schema-free SQL queries over the two schemas. For each query, we test if it can be translated correctly in the best translation or at least in one of the top 10 translations over the two schemas. In our first test, we use schema graph as the underlying data model abstraction. The result is shown in Figure 15. The

[2] There are altogether 27 complete SQL queries in [15]. We remove 10 of them that contain information outside Yahoo-Movie.

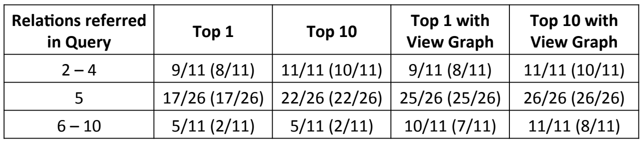


	Queries:	SF-SQL	GUI	SQL
1	Male actors cooperated with director “James Cameron” in the movies produced by company “20th Century Fox” from 1995 to 2010.	6.6	12	22
2	Movies with genre “Drama” and director “Peter Jackson”.	3.4	8	15
3	Movies produced by company “Carthago Films”, distributed by company “Apollo Films”, and directed by director “Fahdel Jaziri”.	4.6	11	21
4	The number of movies directed by “Steven Spielberg” and acted by “Tom Hanks”.	3.4	8	15
5	Actors acted more than 3 movies with genre “Action Adventure” directed by “Woody Allen”.	3.8	10	20
6	Movies with genre “Drama”, financed by company “LLC”, directed by “Stephen Gaghan”.	5	11	21

Figure 14: Sophisticated Queries for Yahoo-Movie Database.

translation quality decreases significantly when the queries become more complex. That is because when a query refers to more relations, the number of possible join paths grows sharply. Unsupervised join path generation cannot accurately tell which join paths are better than others.

In our second test, we use the view graph instead of schema graph as the underlying data model abstraction. We order all the queries by the number of relations they refer to. After each query is tested, we transform it to a view for future use. By doing so, the construction of complex queries can benefit from the previous simple queries by using them as building blocks. The result is shown in Figure 15. The translation quality is improved significantly by using view graph. We see that working with a different schema has virtually little impact on effectiveness, at least for simpler queries. There is a slight effect for complex queries (with 6-10 relations).



Relations referred in Query	Top 1	Top 10	Top 1 with View Graph	Top 10 with View Graph
2 – 4	9/11 (8/11)	11/11 (10/11)	9/11 (8/11)	11/11 (10/11)
5	17/26 (17/26)	22/26 (22/26)	25/26 (25/26)	26/26 (26/26)
6 – 10	5/11 (2/11)	5/11 (2/11)	10/11 (7/11)	11/11 (8/11)

Figure 15: Effectiveness with/without Query Logs (the numbers in parentheses are for the schema with 21 relations)

The information unit costs are shown in Figure 16. In general, Schema-free SQLs only specify 33 (reps. 62) percent as many information units as full SQL queries (with a visual query builder).

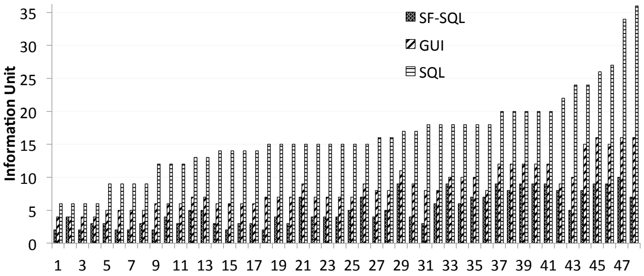


Figure 16: Information Unit Costs in Queries on the Course Database

Finally, we note that the bulk of the execution time is consumed by the Join Network generation. Therefore, we test the efficiency of the algorithms in top k MTJNs generation. We compare the algorithm described in Section 6.1 with the algorithms modified from [8] (Regular) and [12] (Rightmost). The modifications are (a) the expansion stops when the top k MTJNs are generated, and (b) the expansion of a JN can be either adding an edge or adding a view. We test their running time for k equals to 1 and test our algorithms for various k. The results are shown in Figure 17. Note that the time is the average query time for queries with the same number of relations involved (denoted as $size$). The algorithm modified from [8] slows down with $size$ quickly since too many isomorphic JNs exist. The algorithm modified from [12] behaves much better since it ensures that each JN will be expanded at most once. For our algorithm, we compute the upper bound of the weight for each partially expanded JN and prune out many JN before their size reach $size$. For these reasons, our algorithm runs substantially faster (notice that the Y-axis is on a log scale). We also see that there is a noticeable, but modest, cost to generating multiple MTJN, and this cost grows with query complexity.

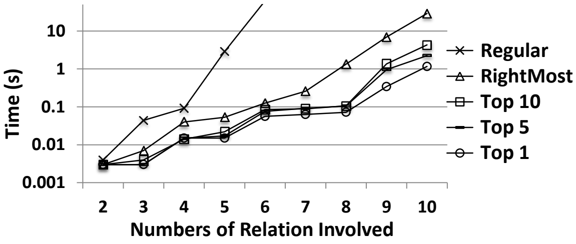


Figure 17: Efficiency Test

8. RELATED WORK

The goal of Schema-free SQL is to make sophisticated queries easier to compose over relational databases. There are various efforts toward making database systems more usable and [9] provides a good review for existing methods.

One strategy towards making relational database usable is using keyword search [17]. Two main families of methods are used: schema-based approaches (e.g. DISCOVER [8]) and graph-based approaches (e.g. BANKS [6]). Schema-based approaches first translate the keywords into a set of minimal total join networks (MTJN) and then evaluate them. Our MTJNs generation follows DISCOVER, but we focus on ranking MTJNs accurately through view graph and design algorithms to generate only the top one or few top k of them efficiently. In graph-based approaches, the database is modeled as a data graph, in which each node is a tuple [6]. Steiner trees of tuples that contain all the keywords are constructed directly on the data graph and outputted as results.

Recently, variations of keyword-based search have been proposed. SQAK [14] supports aggregation functions in keyword search. In [7], keyword search and form-based search are combined. In their system, a set of forms is pre-specified. When a user types in some keywords, the system returns the forms most relevant to the keywords. Qunits [13] represents the database as a collection of independent basic conceptual documents (Qunits), each of which represents the desired results for some queries. Qunits adopts IR-based methods to find the most relevant Qunits as query results.

Visual query builders (e.g. [1, 2, 3]) are widely used to facilitate the building of SQL queries. We view the carefully designed GUI features and our approach as complimentary to each other - while GUI can help to fix the relaxations in Schema-free SQL interactively with the user, Schema-free SQL can provide more powerful functions for metadata exploration and join path auto-completion. Take join path auto-completion as an example. A visual query builder supports creating the join path by dragging and dropping all the relevant relations to a new window. However, this process is not often easy since the meaning of some relations in the join path may be implicit. By merging with our join path generation approach, a user can create a join path by dragging and dropping only the end relations (relations at the end of the join path whose meanings are always explicit for selection or projection), and our system will add the internal ones automatically. A system that performs this merger is left to future work: our experimental evaluation considered the two complementary approaches independently.

SchemaSQL [10] is a principled extension to SQL, which allows its variables to range over relations and attributes in multi-database systems. This paper elegantly supports multi-database heterogeneity, and develops precise semantics to handle concepts that are modeled as attributes in one database but as relations in another and as tuples in a third. In its implementation, all the schema elements are stored in a relation called Federation System Table (FST). By executing SQL on the FST, it resolves the open-ended schema elements of the query into specific ones. In this resolution of schema element variables, Schema-free SQL bears some superficial resemblance to SchemaSQL. However, the problem being addressed is fundamentally different. SchemaSQL gives users the tools to manage semantic heterogeneity, but does not relieve them of the burden of knowing the target schema, and of writing a precise query. Given a SchemaSQL query, there is only one correct interpretation of the query against any given schema (at most). In contrast, our focus is not directly on heterogeneity, but rather on the user's lack of schema knowledge, to which heterogeneity is likely to be an important contributing factor. Schema-free SQL queries are under-specified, and do not have unique interpretations. Our task is to infer user intent.

To facilitate users in querying XML database, Schema-free XQuery [11] integrates keyword search functionality into XQuery as built-in functions. By doing this, it enables users to query XML documents based on whatever partial knowledge they have. Our work is directly inspired by this.

9. CONCLUSION AND FUTURE WORK

In this paper, we proposed Schema-free SQL, which enables its users to compose complex queries over relational databases without requiring full knowledge of the database schema. To support Schema-free SQL, we provide a modular architecture and specific techniques in each module: representing the partial schema information in relation trees, mapping relation trees to relations in the database based on the similarity evaluation, generating "good" minimal total join networks of relations on the view graph, and finally translating the Schema-free SQL into full SQL. Our experiments suggest that Schema-free SQL can greatly decrease users' burden, particularly for complex queries, with hardly any loss in effectiveness compared to standard SQL.

Substantial scope for further work remains. First, to take the most advantage of view graph, we will develop techniques to manage the views: mining frequently appearing query patterns in the query log and setting a proper weight for each view. Second, we will merge Schema-free SQL with visual query builders to both resolve relaxations in Schema-free SQL interactively and provide more powerful functions for visual query builder in metadata exploration and join path auto-completion. Third, we want to take a step further in usability by developing a natural language query system over Schema-free SQL, which can enable naive users to compose complex queries over relational databases.

10. REFERENCES

[1] Active query builder: www.activequerybuilder.com.

[2] Flyspeed sql query: www.activedbsoft.com.

[3] Sqleo visual query builder: sqleo.sourceforge.net.

[4] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[5] B. Bercovitz, F. Kaliszan, G. Koutrika, H. Liou, Z. M. Zadeh, and H. Garcia-Molina. Courserank: a social system for course planning. In *SIGMOD Conference*, pages 1107–1110, 2009.

[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[7] E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining keyword search and forms for ad hoc querying of databases. In *SIGMOD Conference*, pages 349–360, 2009.

[8] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[9] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *SIGMOD Conference*, pages 13–24, 2007.

[10] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. Schemasql: An extension to sql for multidatabase interoperability. *ACM Trans. Database Syst.*, 26(4):476–519, 2001.

[11] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[12] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD Conference*, pages 605–616, 2007.

[13] A. Nandi and H. V. Jagadish. Qunits: queried units in database search. In *CIDR*, 2009.

[14] S. Tata and G. M. Lohman. Sqak: doing more with keywords. In *SIGMOD Conference*, pages 889–902, 2008.

[15] J. D. Ullman and J. Widom. *A first course in database systems (2. ed.).* Prentice Hall, 2002.

[16] C. Yu and H. V. Jagadish. Querying complex structured databases. In *VLDB*, pages 1010–1021, 2007.

[17] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.