SQAK: Doing More with Keywords

Sandeep Tata, Guy M. Lohman
IBM Almaden Research Center
San Jose, CA, U.S.A.
{stata,lohman}@us.ibm.com

ABSTRACT

Today's enterprise databases are large and complex, often relating hundreds of entities. Enabling ordinary users to query such databases and derive value from them has been of great interest in database research. Today, keyword search over relational databases allows users to find pieces of information without having to write complicated SQL queries. However, in order to compute even simple aggregates, a user is required to write a SQL statement and can no longer use simple keywords. This not only requires the ordinary user to learn SQL, but also to learn the schema of the complex database in detail in order to correctly construct the required query. This greatly limits the options of the user who wishes to examine a database in more depth.

As a solution to this problem, we propose a framework called SQAK[1] (SQL Aggregates using Keywords) that enables users to pose aggregate queries using simple keywords with little or no knowledge of the schema. SQAK provides a novel and exciting way to trade-off some of the expressive power of SQL in exchange for the ability to express a large class of aggregate queries using simple keywords. SQAK accomplishes this by taking advantage of the data in the database and the schema (tables, attributes, keys, and referential constraints). SQAK does not require any changes to the database engine and can be used with any existing database. We demonstrate using several experiments that SQAK is effective and can be an enormously powerful tool for ordinary users.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design, Languages

[1]pronounced "squawk"

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

Keywords

keyword queries, aggregates, query tools, relational database, SQL

1. INTRODUCTION

Designing querying mechanisms that allow ordinary users to query large, complex databases has been one of the goals of the database research community. Todays databases are extremely complex, and correctly posing a structured query in SQL (or XQuery) can be challenging and error-prone. Firstly, the user is required to be familiar with the schema of the database. Normalized relational databases can have hundreds of tables with dozens of attributes in each of them. Additionally, structured languages require the user to generate syntactically correct queries that list several join conditions to relate the entities in the query along with additional constraints from the query logic. Consider the simple schema in Figure 1 of a university database that tracks student registrations in various courses offered in different departments. Suppose that a user wished to determine the number of students registered for the course "Introduction to Databases" in the Fall semester in 2007. This simple query would require the user to 1) identify the name of table that stored course titles (*courses*) and the name of the attribute (*name*), 2) identify the name of the table that stored registration information (*enrollment*), 3) identify the table and the attribute that stored information on the courses being offered in each term (table *section* and attribute *term*), 4) identify the names of the appropriate join keys, and finally 5) construct a syntactically correct SQL statement such as:

```
SELECT courses.name, section.term, count(students.id)
 as count
FROM students, enrollment, section, courses
WHERE students.id = enrollment.id
 AND section.classid = enrollment.classid
AND courses.courseid = section.courseid AND
 lower(courses.name) LIKE '\%intro. to databases\%'
AND lower(section.term) = '\%fall 2007\%'
GROUP BY courses.name, section.term
```

While this may seem easy and obvious to a database expert who has examined the schema, it is indeed a difficult task for an ordinary user. In the case of a real world database with hundreds of tables, the task of locating the table that contains a particular attribute, such as a course title, can itself be daunting! Only after locating all the relevant schema elements can the user proceed to the next task of actually

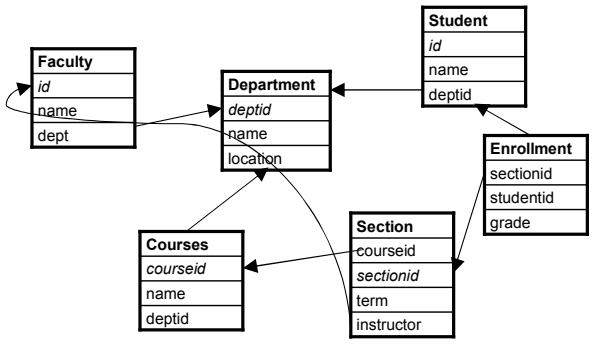

Figure 1: Sample University Schema

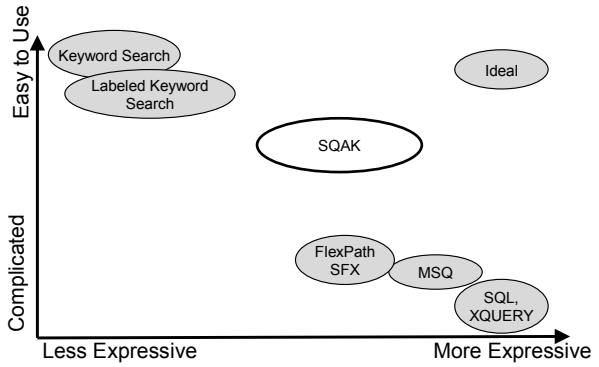

Figure 2: Comparison of Existing Approaches

writing the query. Ideally, the user should be able to pose this query using simple keywords such as *"Introduction to Databases" "Fall 2007" number students*. Such a querying interface will put sophisticated querying capabilities in the hands of non-expert users. Our system SQAK achieves exactly this! By empowering end users to pose more complex queries, enterprises will be able to easily derive significantly increased value from their databases without having to develop an application or enhance existing ones. Applications that previously only supported a predetermined set of queries can leverage such an interface to provide a more powerful experience to the users. While such an interface would be limited in expressive power, the ease of use would be vastly superior to full fledged SQL on complex databases.

1.1 Power vs. Ease of Use

The research community has proposed several different solutions aimed at making it easier for users to query databases. Each of these solutions offers a different tradeoff between the expressive power of the query mechanism and the ease of use. These approaches span from keyword based systems that require absolutely no knowledge of the database schema to SQL and XQuery which are structured languages that use detailed knowledge of the schema. Figure 2 pictorially describes the different trade-offs that several existing systems make. Along the x-axis, are increasingly more powerful querying mechanisms, and the y-axis are systems in increasing order of ease of use. While the expressive powers of the different systems are not strictly comparable (for instance, a keyword query system such as in [6] can produce results that might be extremely difficult to duplicate with an SQL or XQuery statement), in general, systems on the right end can precisely express many classes of queries that systems to their left cannot.

Keyword search systems like BANKS [6], DISCOVER [13], and DBXplorer [1] make it possible for a user to search a database using a set of terms. Each term in the query is mapped to a set of tuples that are subsequently joined together to form tuple trees. A tuple tree (usually) contains all the terms in the query and is considered a potential answer. The result of a query in these systems is a ranked list of tuple trees, much like a web search engine. Although these systems are easy to use and can produce interesting results, they do not accommodate even simple aggregate queries. The result of a query is always a list of tuple trees, and this limits the expressive power of this approach. Other approaches like [8] offer some increased precision over keyword queries, but still do not tackle aggregate queries.

On the other end of the spectrum, approaches like Schema-Free XQuery [16] and FlexPath [3] allow the user to specify structured queries in a loose fashion on XML data. FlexPath, for instance allows the user to specify a template query tree that is subsequently converted to less restrictive trees using a set of relaxation rules. While this is a powerful approach, the task of constructing an approximately correct query tree is still significantly more complicated than a keyword query. Furthermore, FlexPath focuses on retrieving the top-k best matches to the template query, and does not focus on supporting aggregates. The SFX approach in [16] proposes using schema free XQuery, an extension to XQuery that combines structure-free conditions with structured conditions. [16] uses the idea of meaningful lowest common ancestor (MLCAS) to relate together a set of hierarchically linked nodes in close proximity. SFX is error prone when the entities of interest are remotely related. Furthermore, it is not clear how the idea of an MLCAS can be adapted to the relational world. The Meaningful Summary Query [22] presents an elegant way to incorporate partial schema knowledge (by way of a schema summary) into an XQuery like language. The MSQ paradigm permits users to write complex queries with high accuracy using only a summary of the schema. MSQ also focuses on XML data, and it is not clear how well it can be adapted to the relational world. Moreover, the complexity of writing an MSQ query is comparable to XQuery and far from the ease of keyword queries.

Several systems have not been included in Figure 2 since they are not directly comparable to the other approaches discussed here. Query building tools, for instance have been available for a long time. These tools can help a user navigate the schema, locate the elements of interest, and visually construct the appropriate join clauses, thus making the task of writing complicated SQL statements somewhat easier. However, they are still much more complicated to use and focus on assisting the user exploit full power of SQL. Natural language interfaces to databases have had various degrees of success. These approaches provide a mechanism to parse queries in natural languages such as English to convert them into SQL queries. This is a notoriously difficult

problem to solve in the general case and has not gained widespread use.

We have already observed that existing solutions do not provide a way for non-expert users to easily express aggregate queries. In this paper, we describe a prototype system called SQAK (SQL Aggregates with Keywords) that allows users to compute a variety of aggregate queries on a relational database with little to no knowledge of the schema using a simple keyword based interface. As depicted in Figure 2, SQAK combines the luxury of having little to no knowledge of the schema (such as in [22, 16]) with the ease of use of a keyword driven interface while still allowing a significant amount of expressive power. SQAK achieves this powerful tradeoff by limiting the possible space of aggregate queries to a subset of SQL, thus making is possible to reason about the appropriate SQL statement that should be constructed for a given keyword query. SQAK takes advantage of the data in the database, metadata such as the names of tables and attributes, and referential constraints. SQAK also discovers and uses functional dependencies in each table along with the fact that the input query is requesting an aggregate to aggressively prune out ambiguous interpretations. As a result, SQAK is able to provide a powerful and easy to use querying interface that fulfills a need not addressed by any existing systems. We make the following contributions in this paper:

1. We argue that a querying mechanism that permits aggregate queries using simple keywords can be extremely useful for ordinary users in exploring complex databases. We describe the problems that need to be solved to achieve this, and present our prototype system – SQAK.
2. We describe a formalism called a Simple Query Network (SQN) to trade-off the expressive power of SQL for the ability to construct aggregate queries from keywords while minimizing ambiguity. We show that the problem of computing a minimal SQN is NP-complete and propose a greedy heuristic to solve it.
3. We describe a subset of SQL caller rSQL and show how SQNs can be uniquely translated to rSQL statements. We demonstrate using several experiments that SQAK is effective, efficient, and useful on a range of real databases for a variety of queries.

The rest of the paper is organized as follows – Section 2 defines the problem of supporting aggregate queries using keywords in more detail and provides an overview of the architecture of SQAK. Section 3 describes the algorithm used for finding minimal SQN's and translating them to SQL statements. Section 4 addresses several other real world challenges that must be solved in order to make SQAK work well. Section 5 presents the results of several experiments that explore various aspects of SQAK. Section 6 discusses the relationship of SQAK with previous research work, and finally Section 7 summarizes our findings and concludes the paper.

2. OVERVIEW

A keyword query in SQAK is simply a set of words (terms) with at least one of them being an aggregate function (such as count, number, sum, min, or max). Terms in the query may correspond to words in the schema (names of tables or columns) or to data elements in the database. The SQAK system consists of three major components – the Parser/Analyzer, the SQN-Builder, and the Scorer (see Figure 3). A query that enters the system is first parsed into tokens. The analyzer then produces a set of Candidate Interpretations (CI's) based on the tokens in the query. For each CI, the SQN Builder builds a tree (called an SQN) which uniquely corresponds to a structured query. The SQN's are scored and ranked. Finally, the highest ranking tree is converted to SQL and executed using a standard relational engine and the results are displayed to the user. Figure 3 provides an overview of SQAK's architecture.

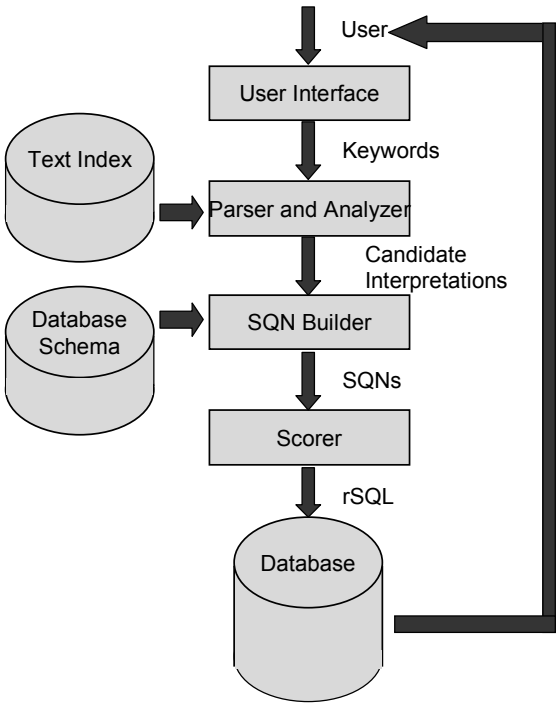

Figure 3: Architecture of SQAK

2.1 Definitions

Before we describe each of the components in more detail, we introduce the terms and concepts used in this paper.

2.1.1 Candidate Interpretation

A Candidate Interpretation (CI) can be thought of as an interpretation of the keyword query posed by the user in the context of the schema and the data in the database. A CI is simply a set of attributes from a database with (optionally) a predicate associated with each attribute. In addition, one of the elements of the CI is labeled with an aggregate function F. This aggregate function is inferred from one of the keywords – for instance, the "average" function from keyword query *"John average grade"* would be the aggregate function F in a CI generated from it. One of the elements of the CI may be optionally labeled as a "with" node (called a w-node). A w-node is used in certain keyword queries where an element with a maximum (or minimum) value for an ag-

gregate is the desired answer. For instance, in the query *" student with max average grade"*, the node for *student* is designated as the w-node. This is discussed in more detail in Section 2.1.3. Definition 1 formally introduces a Candidate Interpretation.

DEFINITION 1. *Given a database D containing a set of tables T, each with a set of columns $C(t_i)$, a Candidate Interpretation $S = (C, a, F, w)$ where*

- $C = \{(c_i^j, p) | c_i^j$ *is the* j_{th} *column of* $Table_i\}$, *and* p *is a predicate on* c_i^j
- $a \in C$,
- F *is an aggregate function.*
- $w \in CU\phi$, *is the optional w-node*

CI's which do not have a w-node are defined to be simple CI's. A CI which satisfies any one of the following tests is considered a trivial interpretation:

DEFINITION 2. *A Trivial CI $S = (C, a, F, w)$ is a CI that satisfies one of:*

- *There exists* $c \in C, c \neq a$ *such that* $c \rightarrow a$ *(c functionally determines a),*
- *There exist two columns* c_i *and* c_j *in* C *that refer to the same attribute,*
- *There exist two columns* c_i *and* c_j *in* C *such that* c_i *and* c_j *are related as primary key and foreign key.*

If a CI does not satisfy any of these conditions, it is considered a non-trivial CI.

An intuitive way of understanding a CI is to think of it as supplying just the SELECT clause of the SQL statement. In translating from the CI to the final query, SQAK "figures out" the rest of the SQL. Consider the sample schema showed in Figure 1. An edge from one table to another simply means that a column in the source table refers to a column in the destination table. Now consider the aggregate keyword query "John num courses" posed by a user trying to compute the number of courses John has taken. One of the possible CI's that might be generated for this is: ({Student.name[=John], Courses.courseid}, Courses.courseid, count, w=ϕ). Depending on the query, there may be several other CI's generated for a given query. Now suppose that a user wishes to find the course that is offered most often in each department, a reasonable keyword query would be *department course with max num sections*. A possible CI for this query is: ({Department.id, Courses.courseid, Section.sectionid}, Section.sectionid, max count, Courses.courseid). The task of generating a set of CI's from a given keyword query is the responsibility of the Parser/Analyzer and is described in Section 2.2.1. Trivial CI's are detected and eliminated in the enumeration stage – they are not scored or used to produce the final SQL query. This is because Trivial CI's produce "uninteresting results", and we assume that the user is seeking to locate some interesting result, and therefore weight a more interesting interpretation higher. For instance, if it is unlikely that the user mentioned two keywords each separately referring to the same attribute of the same table (Condition 2). The intuition behind third condition is similar – the user is unlikely to mention both the foreign key and the primary key columns by keywords in the query. And finally, the first condition will always produce groups with identical values. Again, we discard such interpretations in favor of more interesting interpretataions. Pruning trivial CI's is one of the important ways in which SQAK reduces ambiguity while translating keyword queries to an aggregate query in SQL.

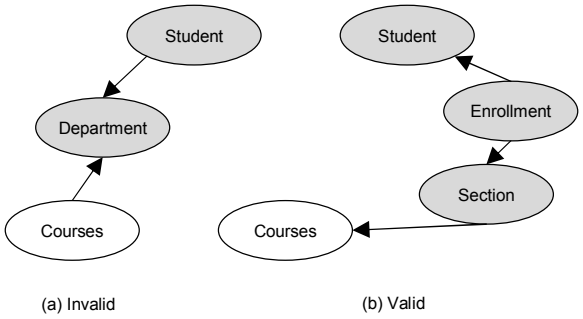

Figure 4: Example SQNs

2.1.2 Simple Query Network

A Simple Query Network is a connected subgraph of the schema graph. Recall that a schema graph is a graph with the nodes representing tables, and edges representing relationships between the tables such as foreign key - primary key constraints. A Simple Query Network is said to be valid with respect to a CI if it satisfies several conditions:

DEFINITION 3. *A Simple Query Network Q is a connected subgraph of the schema graph D. A Simple Query Network Q is said to be valid with respect to a CI $S = (C, a, F, w)$ if it also satisfies the following conditions:*

Minimality *(a) Deleting a node from Q will violate of one of the remaining conditions, (b) Q is a tree,*

Completeness $Tables(C) \subset Nodes(Q)$,

Aggregate Equivalence *One of the nodes of* $g \in Nodes(Q)$ *– is marked as an aggregate node and* $g = Table(a)$,

Node Clarity *Q does not contain any node with multiple incoming edges,*

Figure 4 shows two SQNs (a) and (b). SQN (a) is invalid with respect to the CI from the previous example: ({Student.id, Courses.courseid}, Courses.courseid, count, ϕ). This is because it violates node clarity – the node "Department" has two incoming edges.

One can think of an SQN as a completely specified query corresponding to CI. A valid SQN statement completely specifies the query by supplying the FROM, WHERE, and GROUP BY clauses to the SELECT clause supplied by the CI. The task of converting a CI to valid SQN is the task of the SQN Builder.A valid SQN can be uniquely translated into an rSQL (reduced SQL, described below) query. These algorithms are described in Section 3.

The principle that guides SQAK is one of choosing the simplest model that fits the problem. The same principle

of making the fewest assumptions possible is the basis of systems such as BANKS [6], DISCOVER [13], and DBXplorer [1]. Minimality condition requires us to use as few nodes in a CI as possible in order to satisfy (a). Clearly, a minimal graph that connects all the nodes in the CI will be a tree. This is why SQAK requires the SQN to be a tree. The completeness requirement ensures that none of the terms in the keyword query or CI are ignored. SQAK requires the resulting statement to display every column in the CI. This is an important requirement because keyword querying systems such as [17] allow tuple trees that do not contain *all* the keywords, much like todays web search engines. Aggregate Equivalence is a simple condition that requires that the aggregate implied in the CI is the same as the one in the corresponding SQN.

Since SQAK queries are aggregate queries, they are likely to contain a group-by clause. Node Clarity is one of the principal mechanisms by which SQAK trades off power of expression for ease of use. It is a way of requiring the nodes in the graph to be related strongly. The strongest relationship one can require between a pair of nodes is that the path connecting them in the SQN be a directed path. Ensuring this for every pair would lead to the strongest connection between the nodes specified in the query. However, one often finds nodes in a schema with multiple outgoing edges. Such nodes are a way of implementing many to many relationships, and we wish to allow queries on such data. A node with multiple incoming edges implies that a pair of nodes in that graph are connected more weakly through an "incidental" relationship. We find that the constraint of node clarity effectively narrows down the space of possible queries that correspond to a CI while still allowing SQNs to represent a large class of queries.

Consider the following example that illustrates the use of node clarity. Assume that a user wishes to find for each course, the number of students who have registered for it. She types the query *courses count students*. A CI corresponding to this would be S = ({Courses.courseid, Students.id},Students.id, count, ϕ). Now, Figure 4(a) is the smallest subgraph that covers the tables in S. On the other hand, Figure 4(b) is larger than the graph in Figure 4(a) (by one node and one edge). However, the tables (nodes) in S in this graph are connected by a directed path. Even though (a) is smaller than (b), (b) is exactly the query the user had in mind. Using node clarity, SQAK correctly deduces that (b) implies a stronger relationship. Figure 4(a) corresponds to the query "For each course, list the number of students that are in the same department that offers the course". It is important to note here that in SQAK such queries with weaker relationships cannot be expressed. The expectation is that the non-expert user is more likely to pose queries with stronger relationships than with such weak relationships. This is one of the central trade-offs that allows SQAK its ease of use.

2.1.3 rSQL

A key idea in this paper is that we identify a subset of SQL that can express a wide range of queries. By carefully choosing this subset, SQAK achieves a judicious tradeoff that allows keyword queries to be translated to aggregate queries in this subset while controlling the amount of ambiguity in the system. We call this subset of SQL "reduced SQL" or simply rSQL.

Queries in rSQL are essentially of two types – *simple* aggregate queries and *top1*-queries. Aggregate queries are simply queries that compute some aggregate on one measure. A query such as "Find the number of courses each student has taken" is an example of an aggregate query. The keyword query *students num courses* would solve this problem in SQAK. A *top1* query computes either a max or a min aggregate and also produces the entity corresponding to that value. For instance, consider the query "Find the department with the maximum number of students". This is an example of a top1 query. The keyword query "department WITH max num students" would solve this. A more complex *top1* query is: "In each department, find the student with the highest average grade". This too can be solved in SQAK with the query "department student WITH max avg grade". The careful reader will observe that a *top1* query is really a combination of a group-by and a min or max aggregate query in SQL.

Simple queries in rSQL are subject to the following limitations:

- Joins in rSQL may only be equi-joins of primary key and foreign key attributes.
- Any join path in the query may not consist of an attribute in a table that is the primary key for multiple key – foreign-key joins.
- The select clause is required to specify exactly one aggregate computation
- A query may not contain self joins
- A query may not use nested subqueries with correlation

top1 queries in rSQL are also subject to the same limitations, however, they may use self joins *indirectly* through the use of temporary views.

2.2 System Architecture

2.2.1 Parser/Analyzer

The Parser/Analyzer in SQAK parses the query and transforms it into into a set of Candidate Interpretations. For each token produced by the parser, the analyzer generates a list of candidate matches to schema elements (table names and column names). It does this by searching through the names of the schema elements and matching the token to the element name using an approximate string matching algorithm. If the match score between the schema element and the keyword is higher than a threshold, it is considered a possible match. Each possibility is given a score based on the quality of the approximate match. Additionally, SQAK also uses an inverted index built on all the text columns of the database to match keywords that might refer to a data value. Instead of returning a document identifier, this inverted index returns the table name and column in which the keyword occurs. The analyzer also locates terms that match aggregate functions (sum, count, avg) or their synonyms and associates the aggregate function with the next term in the query. The term preceding the reserved word "with" is labeled the w-term.

Once a list of candidate matches is generated for each term, the list of CI's is generated by computing the cross

product of each term's list. The analyzer is also responsible for identifying trivial interpretations using known functional dependencies in the database and eliminating them before invoking the SQN-Builder.

2.2.2 SQN Builder

The SQN Builder takes a CI as input and computes the smallest valid SQN with respect to the CI. The intuition behind this approach is that the CI must contain all the data elements that the user is interested in. The smallest valid SQN is the "simplest" way to connect these elements together. This idea of using the simplest model that solves the problem has been used in several previous works [6, 13, 1]. This is the focus of Section 3.

2.2.3 Scorer

The SQN Builder produces the best SQN for each CI. Since each keyword query might have multiple CI's, the set of all SQNs for a query are sent to the Scorer which ranks them. The score for an SQN is the sum of the weights of its nodes and edges. The SQN with the smallest weight is chosen as the best completion of the CI.

The weights of the nodes are determined using the match scores from the parser/analyzer. The same match score for each node is determined by the Analyzer – a value in [1,∞) where 1 implies a perfect match, and ∞ implies no match. All edges have unit weight. Additional nodes not in the CI that may be included in the SQN are all given unit weights.

3. SIMPLE QUERY NETWORKS

We now formally state the problem of computing a valid SQN given a CI and show that this problem is NP-Complete. We then describe our heuristic algorithm to solve it and discuss the merits of this algorithm. We show through experiments in Section 5 that this algorithm works well on many real schemas.

Formally, the problem of finding a minimal SQN can be stated as a graph problem : Given a directed graph $G(V, E)$ and a set of nodes C, we are required to find the smallest subtree $T(B, F)$ such that $C \subset B$ and no node in B has multiple incoming edges from F. Readers might notice the similarity to the Steiner tree problem. In fact, if the node clarity condition is relaxed, this problem reduces exactly to the problem of finding a Steiner tree.

The Steiner tree problem is known to be an NP-Complete problem [10]. We know that for a given graph and a set of nodes, a Steiner tree always exists. The same is not true of the minimal SQN problem. For instance, if the schema graph contains a cut vertex that only has incoming nodes, and the CI has two nodes each on one side of the vertex, then any solution would include this cut vertex and therefore violate node clarity.

The addition of the node clarity condition does not make the minimal SQN problem any easier than the Steiner cover problem. In fact, the minimal SQN problem is NP-Complete. We provide a brief sketch of the proof:

The basic idea of this proof is by reduction from the Exact 3-Cover problem [10]. The Exact 3-Cover problem (X3C) can be stated as follows: Given a set S with $|S| = 3k$, and $C = \{C_1, C_2, ..., C_n\}$ where $|C_i| = 3$ and $C_i \subset S$. Is there a cover of S in C of size k? The decision problem corresponding to finding the minimal SQN is: Given a graph $G = (V, E)$, $W \subset V$, and $a \in W$ is there an SQN H with at most r edges? It is easy to see that given H, we can verify that it is an SQN with at most r edges in polynomial time. Now, we transform an instance of the Exact 3-Cover problem to an instance of the minimal SQN problem as shown in Figure 5.

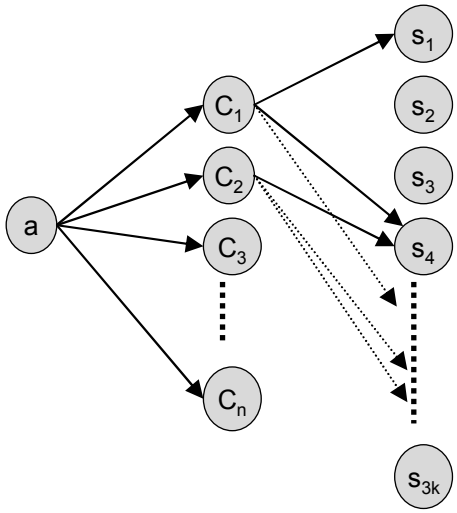

Figure 5: Reducing X3C to minimal SQN

We construct a vertex for each element s_i of S, and each element C_i of C. If $s_i \in C_i$, we add an edge from C_i to s_i. We add a new node a and add C edges, from a to each C_i. We set the nodes to be covered as $W = \{a, s_1, s_2, ..., s_n\}$. It is easy to show that an exact 3 cover of size k exists if and only if there exists an SQN covering S with at most $r = 4k$ edges.

3.1 An Approximate Algorithm

Having shown that finding the minimal SQN for a given CI is NP complete, we outline a greedy backtracking heuristic to solve it (Algorithm 1). The basic idea of the FindSQN algorithm is to start with a copy of a partial solution (called *temp*) initialized to contain only the aggregate node. We then find a node in the CI whose distance to *temp* in the schema graph is shortest. The path between *temp* and this CI node is added to *temp* if node clarity is not violated. The algorithm iteratively adds nodes from the CI nodes to *temp* in order of their distance from the *temp* graph. If at any point the algorithm is unable to proceed without violating node clarity, the algorithm backtracks – the last node added to the current solution is discarded (along with the path to that node), and the algorithm tries to continue to add the node at the next shortest distance. When all the CI nodes are covered, the algorithm terminates. If the algorithm is unable to backtrack any further and has not yet found a solution, it terminates and reports a failure.

FindSQN is called with 4 parameters: the aggregated node, the list of other nodes in the CI, the current graph – a partial solution initialized to a one node graph consisting of just the aggregated node (*temp* starts by making a copy of this), and the schema graph. The procedure *ExpandAllByOneEdge* (Algorithm 2) is used iteratively to locate the CI node that is closest to the current solution. *ExpandAllByOneEdge* finds edges in the schema graph that are incident

Algorithm 1 Algorithm for Finding SQN

```
FindSQN(aggNode, otherNodes, curGraph, schemaGraph)
if otherNodes is empty then
    return curGraph
end if
Let expanded = false, temp = curGraph
while true do
    expanded = expandAllByOneEdge(temp, schemaGraph, aggNode)
    if expanded = false then return null
    matchnodes = findNodesInGraph(othernodes,temp)
    if matchnodes is empty then continue
    atLeastOnePathAdded = false
    for each match in matchnodes do
        new_curGraph = curGraph
        new_otherNodes = otherNodes
        if match.path satisfies node clarity then
            atLeastOnePathAdded = true
            new_curGraph.addPath(match.path)
            new_otherNodes.remove(match)
        end if
    end for
    if atLeastOnePathAdded = true then
        res = findSQN(aggNode, new_otherNodes, new_curGraph, schemaGraph)
        if res!= null then return true;
    end if
    else continue
end while
```

with the current solution and terminate at a node not in the current solution. After each invocation to this procedure, the algorithm checks to see if the expanding *temp* has encountered any of the nodes in *othernodes* using the *findNodesInGraph* call. If it has, these nodes and the paths are added to the *curGraph*, and are removed from *othernodes*. The findSQN algorithm continues recursively until *othernodes* is empty.

3.2 Discussion

Algorithm FindSQN is a heuristic. If it does not encounter any backtracking, a loose upper bound for the running time is $O(q^2E^2)$, where q is the number of nodes in the CI, and E is the number of edges in the schema graph. (*ExpandAllByOneEdge* runs in $O(q^2E)$ times and with no backtracking, it can be called at most E times). In the worst case, the running time of findSQN is exponential. Since this algorithm runs as part of the response to a keyword query, it is important to ensure a reasonable response time. For this reason, SQAK terminates the algorithm after a fixed amount of time and returns no solution. For schemas with relatively low complexity such as star and snowflake schemas, findSQN is unlikely to encounter any backtracking. In fact, backtracking usually happens only when entities in the schema can be connected in multiple ways, often of comparable strength leading to significant ambiguity. We show using multiple databases in Section 5 that findSQN completes in a reasonable amount of time for a variety of queries.

The FindSQN algorithm may fail to return a solution for a query. This may happen either because no valid SQN exists for the input, or because our heuristic could not locate

Algorithm 2 Procedure ExpandAllByOneEdge

```
procedure expandAllByOneEdge(graph, schemaGraph, aggNode)
    exp = false
    for each node n in graph do
        for each edge e in schemaGraph do
            if e is incident with n and e.destination is not in graph then
                t = n; t.path = t.path + e
                add t and e to graph
                Set exp = true
            end if
        end for
    end for
    return exp
end procedure
```

the solution in the given amount of time. When this happens, instead of simply returning to the user with an empty response, SQAK re-runs the algorithm by relaxing the node clarity constraint. This is equivalent to solving the Steiner tree problem using a greedy heuristic and we are therefore guaranteed a solution. When SQAK returns such a solution, it alerts the user by displaying a message that says that the solution might not be accurate.

Having found the SQN using the above algorithm, translating it to the corresponding rSQL query is the next task. This is outlined in Algorithm 3. The case of simple CI's without a w-term is straightforward. On the other hand, *top1* queries require more involved processing to produce the corresponding SQL statement.

Consider the keyword query *department with max num courses* which tries to find the department that offers the most number of courses. The corresponding rSQL query that is produced is:

```
WITH temp(DEPTID, COURSEID) AS (
SELECT DEPARTMENT.DEPTID, count(COURSES.COURSEID)
FROM COURSES, DEPARTMENT
WHERE DEPARTMENT.DEPTID = COURSES.DEPTID
GROUP BY DEPARTMENT.DEPTID),
temp2( COURSEID) AS (SELECT max(COURSEID) FROM temp )
SELECT  temp.DEPTID, temp.COURSEID
FROM temp, temp2
WHERE  temp.COURSEID = temp2.COURSEID
```

As an example for a double level aggregate query, consider the example *department student max avg grade* which tries to find the student with the highest average grade in each department. The rSQL query produced by SQAK is:

```
WITH temp( DEPTID, ID, GRADE) AS (
SELECT  STUDENTS.DEPTID, STUDENTS.ID,
 avg(ENROLLMENT.GRADE)
FROM ENROLLMENT, STUDENTS
WHERE STUDENTS.ID = ENROLLMENT.ID
GROUP BY STUDENTS.DEPTID , STUDENTS.ID),
temp2( DEPTID, GRADE) AS (SELECT DEPTID, max(GRADE)
FROM temp GROUP BY DEPTID)
SELECT  temp.DEPTID, temp.ID, temp.GRADE
FROM temp, temp2
WHERE  temp.DEPTID = temp2.DEPTID
AND temp.GRADE = temp2.GRADE
```

Algorithm 3 Algorithm for Translating to rSQL

translateSQN(CI,SQN)
if SQN does not have a w-node **then**
Return makeSimpleStatement(CI,SQN)
end if
if SQN has a w-node and a single level aggregate **then**
Produce view u = makeSimpleStatement(CI,SQN)
Remove w-node from u's SELECT clause and GROUP BY clause
r = makeSimpleStatement(CI,SQN)
Add u to r's FROM clause
Add join conditions joining all the columns in u to the corresponding ones in r
return r
end if
if SQN has a w-node and a double level aggregate **then**
Produce view u = makeSimpleStatement (CI,SQN)
Produce view v = aggregate of u from the second level aggregate term in the CI excluding the w-node in the SELECT and GROUP BY clauses
Produce r = Join u and v, equijon on all the common columns
Return r
end if
procedure MAKESIMPLESTATEMENT(CI,SQN)
Make SELECT clause from elements CI
Make FROM clause from nodes in SQN
Make WHERE clause from edges in SQN
Make GROUP BY clause from elements of CI except aggregated node
Add predicates in CI to the WHERE clause
Return statement
end procedure

4. OTHER CHALLENGES

While making a system like SQAK work in the context of real world databases several challenges must be overcome. This section describes them and the approach we take in SQAK to overcome them.

4.1 Approximate Matching

The first hurdle when using a system like SQAK is that that the user often does not know the exact names of the entities (table, attributes) she wants to query. She may either misspell, choose an abbreviation of the word, or use a synonymous term. The problem of tolerating alternate terms can be addressed by listing synonyms for each schema element (such as "instructor" for "faculty"). This process may be performed by the DBA who wants to make a database available for querying SQAK. Alternately, synonymous terms in the query may be automatically matched using ontology-based normalization [15]. However, this still leaves us the problem of misspellings and abbreviations. SQAK solves this problem by matching a term in the keyword query to a schema element using an edit distance based measure. If this distance is less than a threshold, then the schema element is considered a potential match for that term. This measure is computed as follows:

$d = e^{f \times \frac{edit\_distance(x,y)}{(|x|+|y|)/2}}$, if $\frac{edit\_distance(x,y)}{(|x|+|y|)/2} < \gamma$, ∞ otherwise.

If the edit distance (expressed as a fraction of the average length of the strings) is less than a threshold γ, then the distance measure is e raised to f times this fraction, and 0 otherwise. If this score is nonzero, then the pair of strings is considered a potential match. A larger value of the measure implies a weaker match. The best possible score is 1, when the edit distance is 0. A larger value of f imposes a higher penalty for differences between the two strings. A larger value of γ allows more distant matches to be considered for processing. Experiments in section 5.3 examine the impact of these parameters on the performance of the system in terms of accuracy and efficiency.

4.2 Missing Referential Integrity Constraints

As discussed in section 3, SQAK exploits referential integrity constraints to construct the appropriate SQL query from the Candidate Interpretation. However, in some real world databases these constraints are not explicitly declared! Some DBA's choose to save on the overheads associated with enforcing these constraints by not declaring these as part of the schema. Unsurprisingly, missing constraints can lead to errors in how SQAK processes queries. In severe cases, the schema graph might be disconnected, and keyword queries that need to connect entities in different components of the graph will no longer be admissible.

SQAK solves this problem by using a sampling based algorithm that is a simplified version of the algorithm in [7] to discover referential constraints. The process of discovering keys exploits the system catalog as well as data samples. This discovery process needs to be run only one time before SQAK can start processing queries. Heuristics are used to minimize the cost of discovery.

4.3 Tied or Close Plans

Sometimes, a given keyword query may have multiple corresponding rSQL queries with identical scores. This often happens if the schema contains many entities that are related in multiple ways and the query is inherently ambiguous. In such cases, SQAK arbitrarily breaks the tie and presents the results from one of the queries. At the same time, SQAK also presents the alternate queries that had the same score to alert the user that it could not uniquely translate the query. In some cases multiple plans may score very highly, but not equally. If SQAK detects alternate plans with a score very close to the best score, it alerts the user and presents an option to examine these plans. For instance, this may happen if there are several semantically different schema elements that have a similar names and the query is not sufficiently precise to distinguish between them.

Currently, SQAK simply lists the SQL and score from the alternate interpretations. The user can select any of them to see the results corresponding to them. Providing appropriate visual representations or keyword based summaries of this information to make it easier for the user to understand the choices available is an area of future research.

4.4 Expressiveness

Although simple keywords are a powerful way to quickly pose aggregate queries, users soon begin to want pose queries with non-equality constraints. For instance, a constraint such as $age > 18$. SQAK supports this by simply adding the appropriate where clause to the query it generates. This simple addition greatly enhances the expressive power of the keyword queries in the system. If the SQAK system

encounters a keyword query that does not contain an aggregate term, we may simply interpret is as a regular keyword query in the style of BANKS [6] and DISCOVER [13] and use a similar system to execute the query. Currently, SQAK does not admit any queries that do not contain at least one aggregate keyword.

Queries containing phrases need to be handled carefully. Although SQAK allows users to enter quote-delimited phrases, it also checks to see if a set of consecutive terms in the query might be combined into a phrase. If consecutive terms in the keyword query correspond to the same table and column in the database, then a CI is produced combining the elements into one that references that column. This CI is converted to an SQN and scored as normal.

5. EXPERIMENTS

Evaluating a novel system such as SQAK is a challenging task in itself. Traditional IR metrics like precision and recall are not entirely relevant. The objective in SQAK is to allow users to represent some SQL queries using keywords. Unlike the IR world where the objective is to return relevant documents, once translated, an rSQL query has precise semantics and the precision and recall are always 100%.

In this section, we explore many aspects of the SQAK system. Using many queries on multiple datasets, we show that SQAK is very effective in accurately interpreting keyword queries. We also quantify the savings SQAK provides in cost of query construction using the metric followed in [22]. We examine the impact of the parameters of approximate matching on the cost and accuracy of translating misspelled and abbreviated queries and show that SQAK requires virtually no tuning and performs well on even large databases.

Setup: The experiments designed in this section were performed on a system with a dual-core 2.16GHz Intel processor with 2GB of memory running Windows XP. SQAK algorithms were implemented in Java and were run using JVM 1.6.0_02. The inverted index was constructed using Lucene [18]. A popular commercial database was used to store the relational data and execute the rSQL queries.

5.1 Effectiveness

We first present a sample workload of keyword queries on two different databases. We examine the query produced by SQAK in each case manually to check if the rSQL produced accurately reflects the keyword query. For comparison, we present the results from using a simple Steiner tree. We use two databases, one with the university schema from Figure 1, and second is the TPCH database with the schema showed in Figure 6. The workloads for these schemas are shown in Tables 1, and 2. In each table, the query is listed in plain English along with the corresponding keyword. Mostly the keywords are just picked from the English query, and occasionally abbreviated. The table also lists if SQAK and Steiner were accurate on that query. As can be seen from able 1, SQAK correctly interprets 14 out of 15 queries. Query 8, "the average grade William obtained in the courses offered by the EECS department", (*William EECS avg grade*) gets translated as:

```
SELECT STUDENTS.NAME, DEPARTMENT.NAME,
 avg(ENROLLMENT.GRADE)
FROM ENROLLMENT, STUDENTS, DEPARTMENT
WHERE STUDENTS.ID = ENROLLMENT.ID AND
```

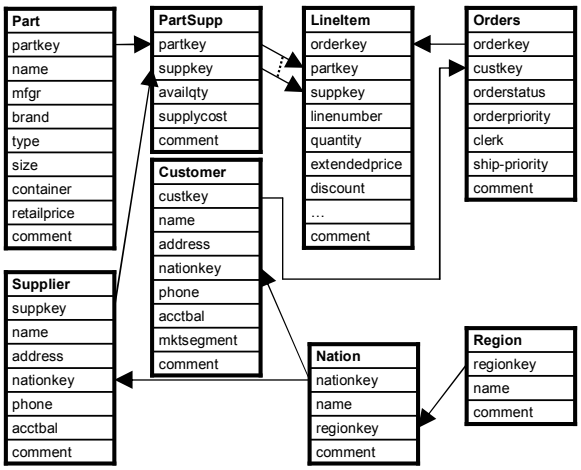

Figure 6: TPCH Schema

```
 DEPARTMENT.DEPTID = STUDENTS.DEPTID AND
 lower(STUDENTS.NAME) LIKE '\%william\%' AND
 lower(DEPARTMENT.NAME) LIKE '\%eecs\%'
GROUP BY STUDENTS.NAME , DEPARTMENT.NAME
```

This really is the result for the query "average grade obtained by the student William registered in the EECS department". Although this is a reasonable interpretation, the original query the user had in mind was different. Note that the Steiner algorithm also gets this query wrong. In fact, SQAK is 93% accurate while Steiner is only 60% accurate. SQAK correctly interprets the attributes corresponding to each field. However, the required query is not the minimal SQN, but a larger SQN that connects the Department table to Courses directly instead of the Students table.

As Section 4 points out, whenever SQAK finds multiple plans with scores close to the highest score, the user is alerted to the possibility that what she was looking for might be an alternate plan. As is the case with any querying mechanism that takes as input an incompletely specified query (e.g. [16, 22]), SQAK too must deal with some inherent possibility of interpreting the query incorrectly.

5.2 Savings

Clearly, constructing a query such as *EECS num students* is easier and quicker than writing the corresponding SQL query:

```
SELECT DEPARTMENT.NAME, count(STUDENTS.ID)
FROM STUDENTS, DEPARTMENT
WHERE DEPARTMENT.DEPTID = STUDENTS.DEPTID AND
lower(DEPARTMENT.NAME) LIKE '%eecs%'
GROUP BY DEPARTMENT.NAME
```

One of the ways to quantify the benefits of using a system such as SQAK is to measure the savings in the cost of query construction.[22] proposes quantifying the savings in the human cost of query construction by counting the number of schema elements that a human trying to construct the query need not use/visit when using a keyword query. Schema elements in the relational world refer to tables and attributes. However, this does not take into account the

Number	Query	Keywords	SQAK	Steiner
1	number of students in EECS	EECS num students	✓	✓
2	number of courses in the biology department	Biology num courses	✓	✓
3	number of courses taken by each student	student num courses	✓	NO
4	avg grade earned by Michael	Michael avg grade	✓	✓
5	number of courses offered in the Fall 2007 term in EECS	EECS "Fall 2007" num courses	✓	✓
6	number of courses taught by Prof Shakespeare	Shakespeare num courses	✓	✓
7	average grade awarded by each faculty member	faculty avg grade	✓	✓
8	average grade William got in the courses offered by the eecs department	William EECS avg grade	NO	NO
9	number of courses Shaun has taken with Prof. Jack Sparrow	"Jack Sparrow" Shaun num courses	✓	NO
10	number of students that have taken databases	databases num students	✓	NO
11	highest grade awarded in databases	databases max grade	✓	✓
12	department that has the most number of courses	dept with max num courses	✓	✓
13	student with the highest grade in the database course	databases student with max grade	✓	NO
14	the most popular course	course with max num students	✓	NO
15	the professor who awards the highest grades in his courses	faculty with max avg grade	✓	✓

Table 1: Queries for the School Database

Number	Query	Keywords	SQAK	Steiner
1	number of orders placed by each customer	customer num orders	✓	✓
2	number of blue parts	num blue	✓	✓
3	number of suppliers in asia	num supplier asia	✓	✓
4	total price of orders from each customer	customer sum totalprice	✓	✓
5	number of customers in each market segment	marketsegment num customer	✓	NO
6	number of orders on each date	orderdate num orders	✓	✓
7	number of suppliers for each part	part num supplier	✓	✓
8	number of parts supplied by each supplier	supplier num part	✓	✓
9	number of parts in each order	order num parts	✓	✓
10	number of orders from america	america num orders	✓	✓
11	total sales in each region	region sum totalprice	✓	NO
12	number of suppliers of brushed tin in asia	"Brushed tin" asia num suppliers	✓	✓
13	region with most suppliers	region with max num suppliers	✓	✓
14	supplier who supplies for the most number of orders	supplier with max num orders	✓	✓
15	market segment with maximum total sales	marketsegment max sum totalprice	✓	✓

Table 2: Queries for the TPCH Database

cost of writing down the query logic. In order to account for some of the cost of query construction, we augment this measure and also count the number of join conditions that the user does not have to write. That is, we assume that the cost of a structured query is the sum of the number of schema elements mentioned in the query and the number of join conditions. The cost of a keyword query is simply the number of schema elements in the keyword query. For instance, keyword query *EECS num students* mentions only one schema element – the students table. Therefore its cost is 1. The corresponding SQL query mentions the tables Department and Students. It also mentions the attributes Department.Name and Students.ID. Additionally, it uses one join. The total cost of this query is 5. The total savings that SQAK provides for this query is 5 – 1 = 4. Note that this estimate is essentially a lower bound on the true savings in the human cost of query construction. SQL requires additional effort to be syntactically correct. Furthermore, complex queries in rSQL such as *top1* queries with w-terms have more involved query logic, and this measure should be interpreted as a lower bound on the amount of savings for that query.

We averaged the savings in the cost of query construction for the queries in Tables 1 and 2. This is summarized in Table 3. While the actual values varied between as little as 1 and as much as 11, the average savings in each case was 6.0 and 4.7 units. To the best of our knowledge the kinds of keyword queries supported by SQAK are not supported by any other system, therefore, we are unable to compare the savings provided by SQAK with an alternate approach.

Schema	Construction Savings
University	6.0
TPCH	4.7

Table 3: Query Construction Cost Savings

5.3 Parameters

SQAK has relatively few parameters. In fact, the only aspect one needs to tune is the "looseness" of the approximate

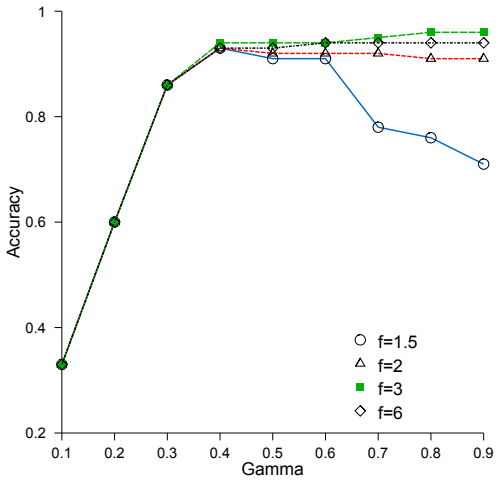

Figure 7: Mismatch Sensitivity

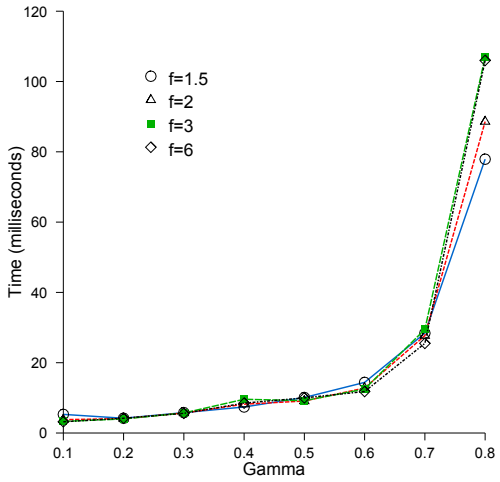

Figure 8: Query Translation Time

match between a schema element name (or its synonyms) and the keyword. As described in Section 4.1, we use an edit distance based method to match the keywords with the schema elements. The two parameters we explore here are a) the mismatch tolerance threshold γ and b)the mismatch penalty f.

For each of the keyword queries from the Tables 1 and 2, we generated 3 other queries that used mis-spelled or shortened versions of the keywords. For instance, "department" might be shortened to "dept". "Students" might be spelled as "Stuents". These were generated manually. In some of the cases, the spelling error caused the algorithm to map the term to a different schema element, and therefore the resulting query was different. We varied γ from 0.2 to 0.8. We repeat the experiment for f = 1.5, 2, 3, and 6. (Recall that a higher value of f imposes a greater penalty for a mismatch.) The results are shown in Figure 7.

As is evident from the figure, the accuracy of SQAK is not highly sensitive to γ. In fact, the accuracy is nearly stable between the values of 0.4 and 0.8. This simply means that for simple spelling errors and shortenings, the distance measure is robust enough that tolerating a small amount of mismatch is enough to ensure that the right schema element is picked for that keyword. Interestingly, we see that when f = 1.5, the accuracy is lower than then f=2.0. That is, imposing a low penalty might make SQAK pick the wrong query. Further, for the case of f=3.0, the accuracy improves even more. This tops off at f=6.0 here, and no further improvements are observed. We expect that f = 2 or 3 and γ between 0.4 and 0.8 are good choices in general.

5.4 Cost

In a system like SQAK where the user poses keyword queries, response time is important. The overhead of translating the keyword query should be small compared to the cost of actually executing the SQL query. We measured the time taken by SQAK to perform this computation for the same sets of values of f and γ as above. In each of the cases, SQAK was allowed to run to completion. The resulting times are plotted in Figure 8. As is evident choosing a value of *gamma* between 0.4 and 0.6 along with an appropriate f should provide good accuracy for very little overhead.

5.5 Other Schemas

We also performed some preliminary tests of the SQAK system on a large database that stores all the configuration information about all the IT assets of a large enterprise. This includes computer systems and their hardware configuration, the operating system, the various database servers, application servers, applications, and services running on this system. This database contains over 600 tables, each with several columns. Sample queries on this database perform with an accuracy comparable to that reported earlier in the section. While the performance of the queries is variable, the translation step always took less than a second even for complex queries. We also performed tests on a data warehouse of retail sales data with a star schema containing 14 tables. A regular schema such as a star schema tends to be an easy case for SQAK since queries have little ambiguity and they perform with close to 100% accuracy. Errors usually happen only when the keywords are heavily misspelt. In the interest of space, we do not present the results from these studies here.

6. DISCUSSION AND RELATED WORK

The SQAK system leverages and combines several existing ideas from database research to achieve the balance of expressive power and ease of use. SQAK's algorithm for discovering functional dependencies and join key constraints is based on work like TANE [14], BHUNT [7], and [4].

The problem of keyword search in relational databases has received much attention in recent years. Early systems like BANKS [6], DISCOVER [13], and DBXplorer [1] designed efficient ways to retrieve tuple trees that contained all the keywords in the query. [12] and [17] proposed using techniques from the IR world to rank these tuple trees better. While these systems provide a way for users to employ keyword based search over structured data, they do not allow

the users to take advantage of the structure to compute aggregates. SQAK attempts to build on this work by providing a system where powerful aggregates can be computed with simple keywords.

Regular keyword search has been extended to provide some basic aggregates that are either predefined or dynamically determined. Such a capability, called faceted keyword search [9, 20] has been described in literature and can be seen on many commercial shopping websites. While faceted keyword search does not really permit the user to pose queries like in SQAK, these systems allow the user to get some aggregate counts in the context of their keyword query.

A recent work that has addressed the problem of designing keyword based interfaces that can compute powerful aggregates is KDAP [19]. The KDAP system provides a keyword based approach to OLAP where the user types in a set of keywords and the system dynamically determines multiple interesting facets and presents aggregates on a predetermined measure. KDAP is aimed at data warehouses with a star or snowflake schema. While KDAP focuses on automatically generating a set of interesting facets and presenting an aggregate result based on that, SQAK focuses on using the keywords to determine attributes to group the result by. Furthermore, in KDAP, the measure is predetermined, while SQAK allows the users to dynamically determine the aggregates and the measure using simple keywords. SQAK is more general in the sense that it is not limited to star or snowflake schemas. At a more basic level, SQAK focuses on being a querying mechanism while KDAP focuses on being a Business Intelligence tool for warehouses with large amounts of text.

XML full-text search has been an area of great recent interest to the research community. Several systems [11, 8, 5, 2] have been proposed to solve the problem of full-text search in XML databases. Many of these systems focus on combining structured querying with keyword search primitives such as boolean connectives, phrase matching, and document scoring and ranking. Efforts such as [3] propose an algebra to allow complex full-text predicates while querying XML documents. These are powerful systems that allow much expressive power, but their focus is not on computing aggregate queries using keywords, but on being able to support structured queries that can incorporate sophisticated full-text predicates.

Querying a structured database with limited or no schema knowledge is a difficult proposition. [16] proposes a solution in the context of XML databases where the user does not have any knowledge of the schema using the idea of a meaningful lowest common ancestor. [22] proposes a way to allow users to write XQuery-like queries using only a summary of the schema information. These approaches allow sophisticated users with limited schema knowledge great power in posing complex queries. However, the focus of our work is in enabling the ordinary user to leverage keyword queries to compute aggregates on relational data.

A recent effort [21] describes a technique for selecting a database from a set of different databases based on the terms in a keyword query and how they are related in the context of the database. The data source selection problem is somewhat orthogonal to our problem since SQAK assumes that the user is interested in querying a given database. The question of whether the techniques in [21] may be used to extend SQAK to work over multiple data sources is a topic of future investigation. Another promising area of investigation is the characterization of rSQL and the kinds of queries it can and cannot express. Extending SQAK to deal with multiple aggregates in a single query is fairly straightforward.

7. CONCLUSIONS

In this paper, we argued that current mechanisms do not allow ordinary users to pose aggregate queries easily on complex structured databases. Such a mechanism can be an extremely powerful in enabling non-expert users of large structured databases. We formally describe the problem of constructing a structured query in SQL from a keyword query entered by a user who has little knowledge of the schema. We describe algorithms to solve the problems involved and present a system based on this called SQAK.

We demonstrate through experiments on multiple schemas that intuitively posed keyword queries in SQAK are translated into the correct structured query significantly more often than with a naive approach like Steiner. We show that the algorithms in SQAK work on different databases, scale well, and can tolerate real world problems like approximate matches and missing schema information. We also show that SQAK requires virtually no tuning and can be used with any database engine.

In summary, we conclude that SQAK is a novel approach that allows ordinary users to perform sophisticated queries on complex databases that would not have been possible earlier without detailed knowledge of the schema and SQL skills. We expect that SQAK will bring vastly enhanced querying abilities to non-experts.

8. ACKNOWLEDGEMENTS

The authors are thankful to Prof. Jignesh M. Patel for initial discussions and to the anonymous reviewers for comments on improving the paper.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, pages 5–16, 2002.

[2] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. TeXQuery: A Full-text Search Extension to XQuery. In *WWW*, pages 583–594. ACM, 2004.

[3] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient XML search with complex full-text predicates. In *SIGMOD*, pages 575–586, 2006.

[4] P. Andritsos, R. J. Miller, and P. Tsaparas. Information-Theoretic Tools for Mining Database Structure from Large Data Sets. In *SIGMOD*, pages 731–742, 2004.

[5] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, 2004.

[6] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.

[7] P. G. Brown and P. J. Haas. BHUNT: Automatic Discovery of Fuzzy Algebraic Constraints in Relational Data. In *VLDB*, 2003.

[8] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, 2003.

[9] W. Dakka, R. Dayal, and P. G. Ipeirotis.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[13] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, 2002.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[15] Y. Li, H. Yang, and H. Jagadish. Constructing a Generic Natural Language Interface for an XML Database. In *EDBT*, 2006.

[16] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[17] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective Keyword Search in Relational Databases. In *SIGMOD*, pages 563–574. ACM Press, 2006.

[18] Lucene. http://lucene.apache.org.

[19] Ping Wu and Yannis Sismanis and Berthold Reinwald. Towards Keyword-driven Analytical Processing. In *SIGMOD*, pages 617–628, 2007.

[20] D. Tunkelang. Dynamic Category Sets: An Approach for Faceted Search. In *SIGIR Faceted Search Workshop*, 2006.

[21] B. Yu, G. Li, K. Sollins, and A. K. H. Tung. Effective Keyword-based Selection of Relational Databases. In *SIGMOD*, pages 139–150. ACM, 2007.

[22] C. Yu and H. V. Jagadish. Querying Complex Structured Databases. In *VLDB*, pages 1010–1021, 2007.