



Topic:
SQAK: Doing More with
Keywords

Speaker:
YINGJING YAN

Why SQAK?

- 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.

Why SQAK?

- However, in order to compute even simple aggregates, a user is required to write a SQL statement and can no longer use simple keywords.
- As a solution to this problem, we propose a framework called SQAK (SQL Aggregates using Keywords) that enables users to pose aggregate queries using simple keywords with little or no knowledge of the schema.

INTRODUCTION

- Consider the simple schema in Figure 1 of a university database that tracks student registrations in various courses offered in different departments:

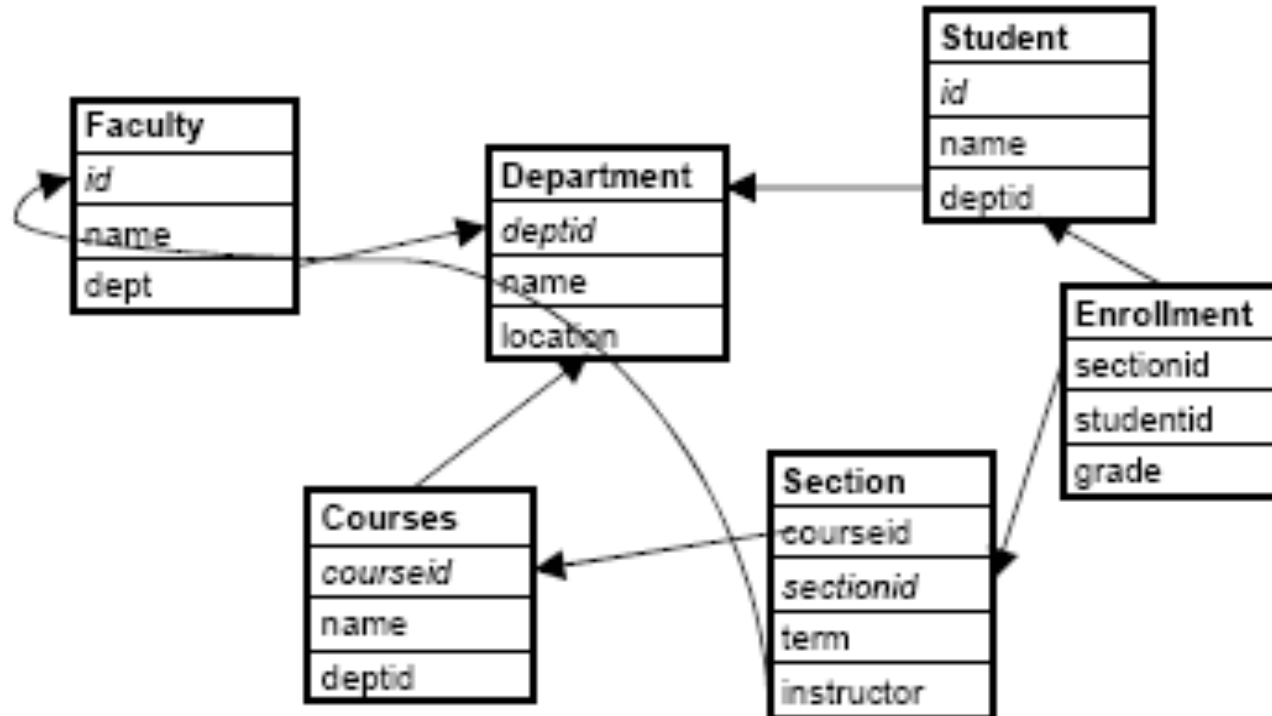


Figure 1: Sample University Schema

INTRODUCTION

- Suppose that a user wished to determine the number of students registered for the course “Introduction to Databases” in the Fall semester in 2007.
- The SQL statement would be written as follows:

INTRODUCTION

```
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
```

INTRODUCTION

- 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.
- Ideally, the user should be able to pose this query using simple keywords such as *“Introduction to Databases” “Fall 2007” number students.*
- SQAK system achieves exactly this by empowering end users to pose more complex queries.

SQAK Overview

- 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, 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.

SQAK Overview

- 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.

Architecture of SQAK

- 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.

Architecture of SQAK

- The SQAK system consists of three major components – the Parser/Analyzer, the SQN-Builder, and the Scorer. 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.

Architecture of SQAK

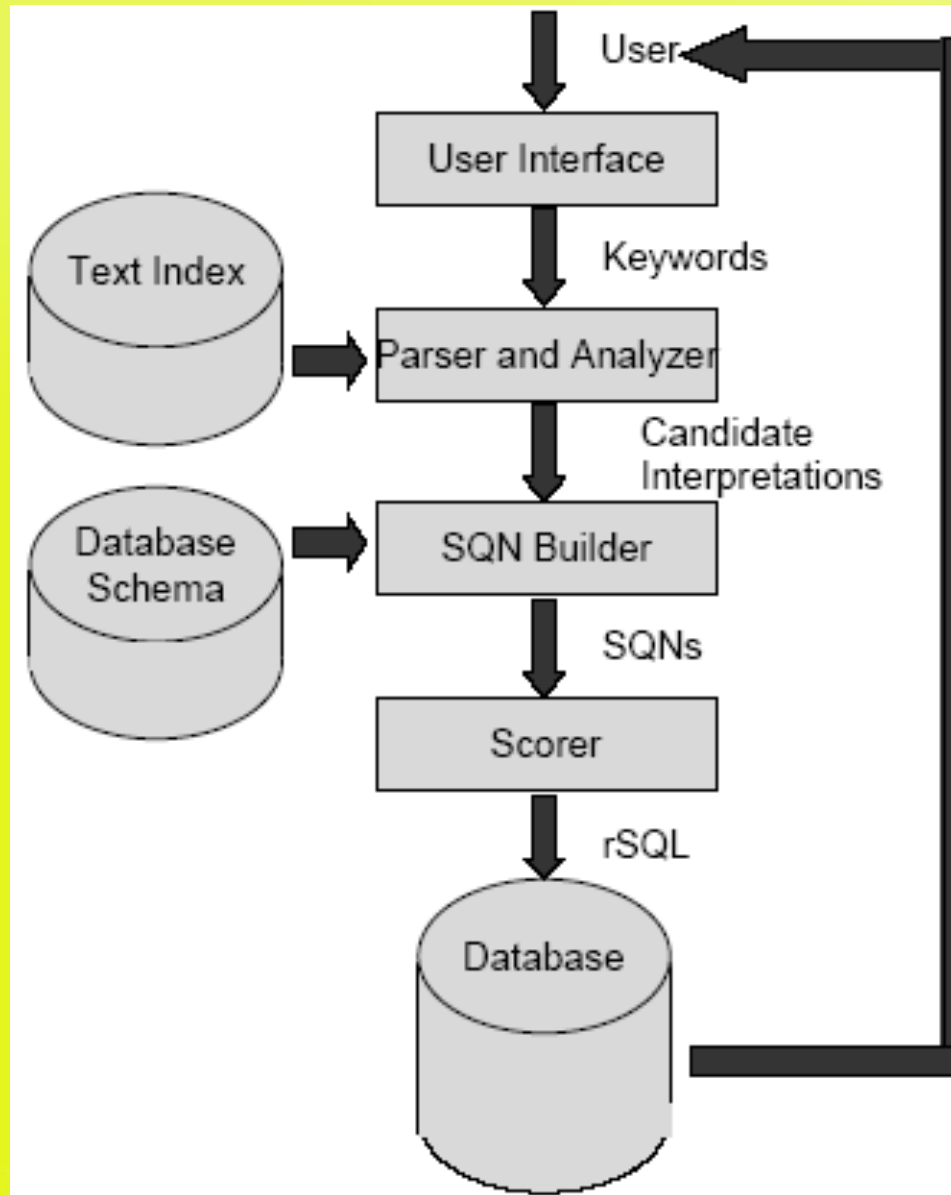


Figure 3: Architecture of SQAK

Candidate Interpretation (CI)

- A 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.

Candidate Interpretation (CI)

- 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.

Candidate Interpretation (CI)

- 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 aggregate 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 later.

CI Definition

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

- $C = \{(c_i^j, p) \mid c_i^j \text{ is the } j_{\text{th}} \text{ column of Table } i\}$, 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 Definition

- 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.

CI Definition

- 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: ($\{\text{Student.name} [= \text{John}], \text{Courses.courseid}\}$, Courses.courseid , count , $w = \emptyset$).
- Depending on the query, there may be several other CI’s generated for a given query.

CI Definition

- 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 will be described later.

CI Definition

- 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.*

CI Definition

- 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.

CI Definition

- 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.

CI Definition

- And finally, the first condition will always produce groups with identical values. Again, we discard such interpretations in favor of more interesting interpretations.
- 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.

A Simple Query Network

- A Simple Query Network is a connected subgraph of the schema graph.
- A schema graph is a graph with the nodes representing tables, and edge 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 $S = (C, a, F, w)$ if it satisfies the following conditions:

A Simple Query Network

Minimality (a) Deleting a node from Q will violate 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,

A Simple Query Network

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

A Simple Query Network

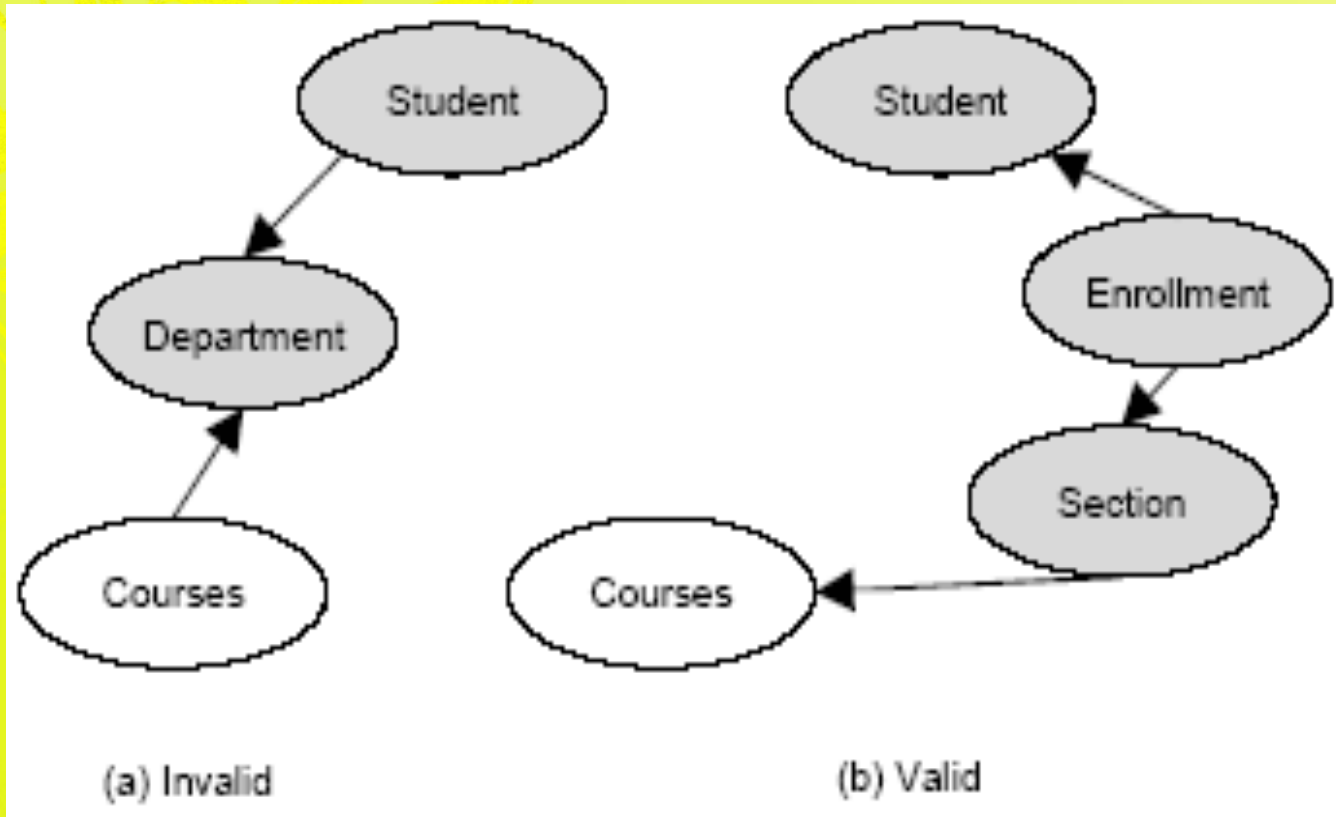


Figure 4: Example SQNs

A Simple Query Network

- We 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 will be described later.

A Simple Query Network

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

A Simple Query Network

- 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.
- 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.

A Simple Query Network

- 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.

A Simple Query Network

- 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.
- 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.

rSQL

- 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.

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 SQAQ.

rSQL

- 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.

rSQL

- 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”.
- We observe that a top1 query is really a combination of a group-by and a min or max aggregate query in SQL.

System Architecture-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.

Parser/Analyzer

- 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.

Parser/Analyzer

- 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.

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.

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.

Scorer

- 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 $[1, \infty)$ 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.

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.
- 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 .

SIMPLE QUERY NETWORKS

- 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.

SIMPLE QUERY NETWORKS

- The Exact 3-Cover problem (X3C) can be stated as follows: Given a set S with $|S| = 3k$, and $C = \{C_1, C_2, \dots, C_n\}$ 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.

SIMPLE QUERY NETWORKS

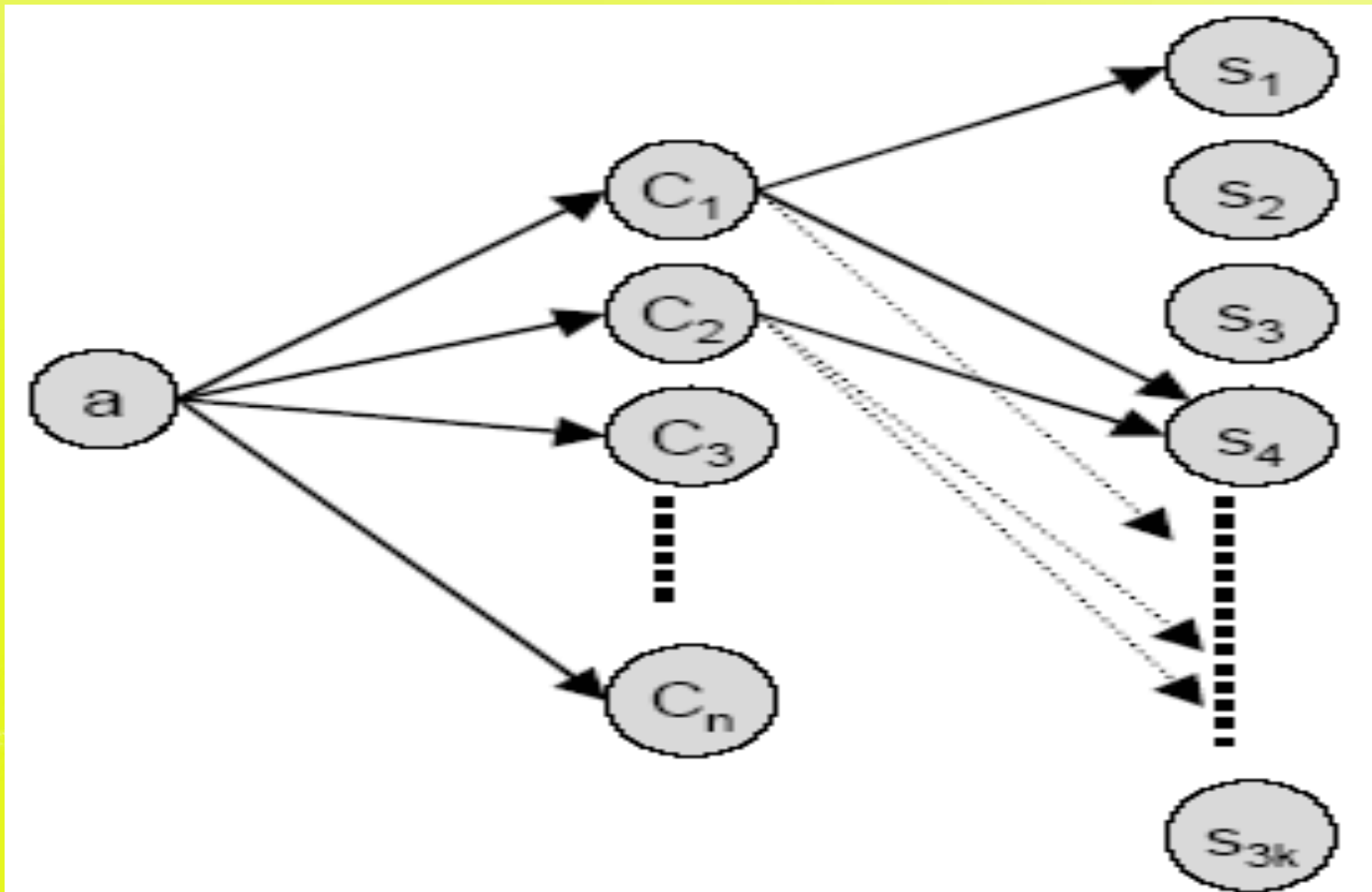


Figure 5: Reducing X3C to minimal SQN

SIMPLE QUERY NETWORKS

- 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 $W = \{a, s_1, s_2, \dots, s_n\}$.
- 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.

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.

An Approximate Algorithm

- 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.

An Approximate Algorithm

- 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.

An Approximate Algorithm

- 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 with the current solution and terminate at a node not in the current solution.

An Approximate Algorithm

- 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.

Algorithm 1 Algorithm for Finding SQN

```
FindSQN(aggNode, otherNodes, curGraph, schema-
Graph)
if otherNodes is empty then
    return curGraph
end if
Let expanded = false, temp = curGraph
while true do
    expanded = expandAllByOneEdge(temp, schema-
Graph, 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
```

Algorithm 2 Procedure ExpandAllByOneEdge

```
procedure EXPANDALLBYONEEDGE(graph, schema-  
Graph, 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
```

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.

Discussion

- 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.

Discussion

- 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 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 and we are therefore guaranteed a solution.

Discussion

- 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.

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 SE-
    LECT 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
```

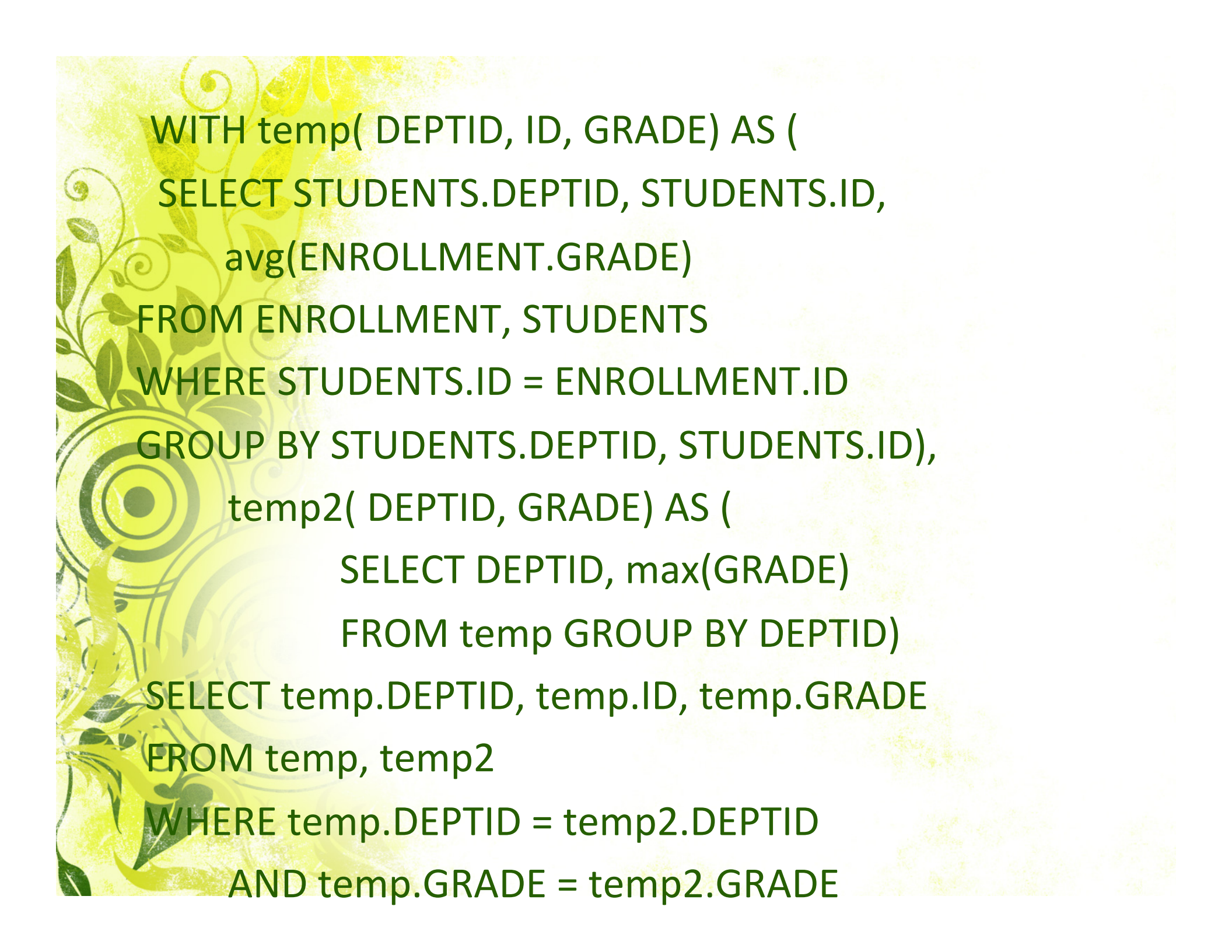
Discussion

- 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
```

Discussion

- 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
```

Approximate Matching

- The first hurdle when using a system like SQAK is 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”).

Approximate Matching

- 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:

Approximate Matching

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

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 ∞ 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.

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.

Tied or Close Plans

- 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.

Expressiveness

- If the SQAK system encounters a keyword query that does not contain an aggregate term, we may simply interpret it as a regular keyword query 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.

Effectiveness

- Query “the average grade William obtained in the courses offered by the EECS department”, (William EECS avg grade) gets translated as:

Effectiveness

```
SELECT STUDENTS.NAME, DEPARTMENT.NAME,  
       avg(ENROLLMENT.GRADE)  
FROM ENROLLMENT, STUDENTS, DEPARTMENT  
WHERE STUDENTS.ID = ENROLLMENT.ID AND  
       DEPARTMENT.DEPTID = STUDENTS.DEPTID AND  
       lower(STUDENTS.NAME) LIKE '\%william\%' AND  
       lower(DEPARTMENT.NAME) LIKE '\%eecs\%'  
GROUP BY STUDENTS.NAME , DEPARTMENT.NAME
```

Effectiveness

- 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 is the case with any querying mechanism that takes as input an incompletely specified Query, SQAK too must deal with some inherent possibility of interpreting the query incorrectly.

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 '%eeecs%'
```

```
GROUP BY DEPARTMENT.NAME
```

Savings

- 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 $5 - 1 = 4$.

Savings

- 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.

Parameters

- SQAK has relatively few parameters. In fact, the only aspect one needs to tune is the “looseness” of the approximate match between a schema element name (or its synonyms) and the keyword.
- As described before, 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 .

Parameters

- Consider queries that use 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 . (a higher value of f imposes a greater penalty for a mismatch.)
- The results are shown in Figure 7.

Parameters

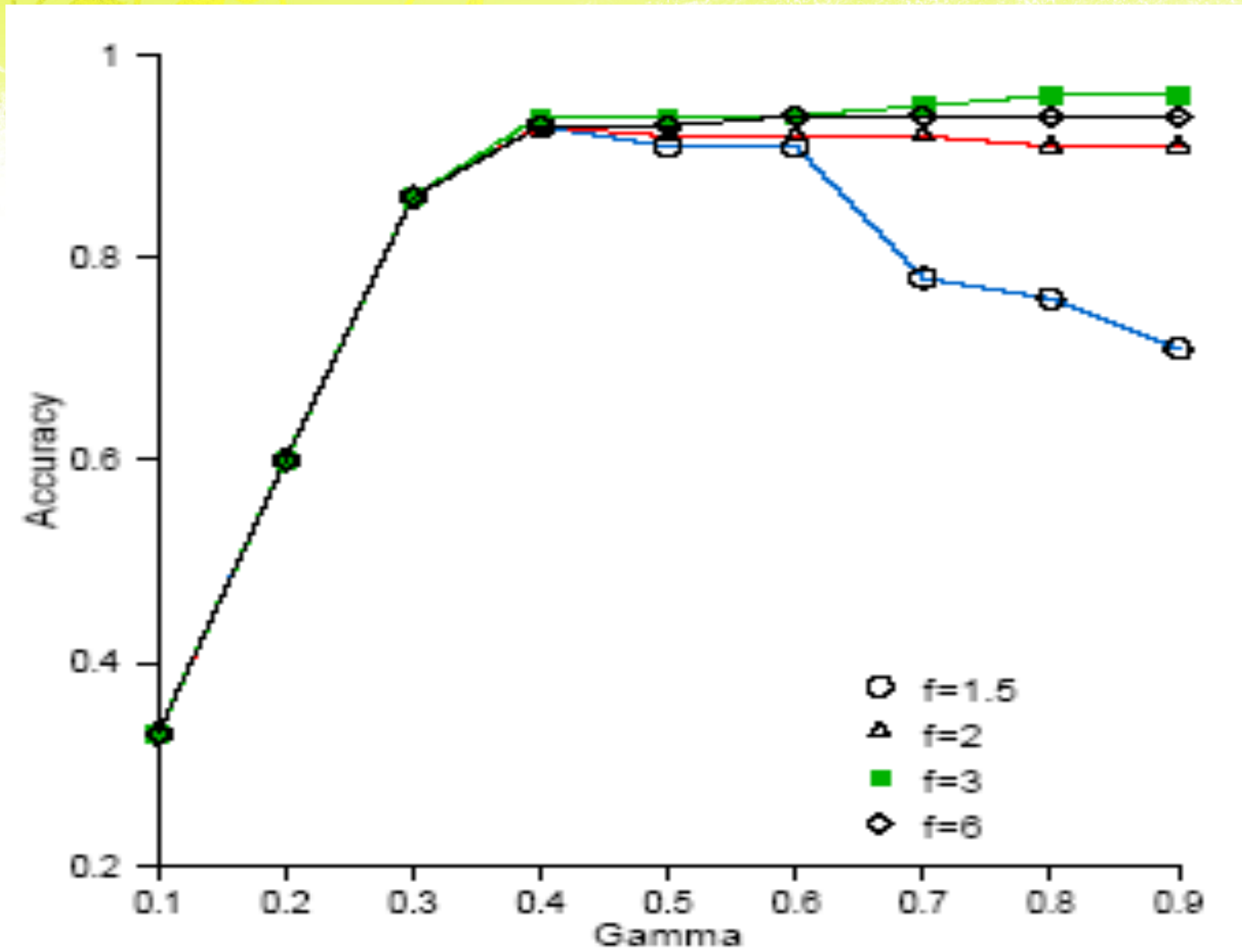


Figure 7: Mismatch Sensitivity

Parameters

- 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.

Parameters

- 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.

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 n as above.
- In each of the cases, SQAK was allowed to run to completion.
- The resulting times are plotted in Figure 8.

Cost

- 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.

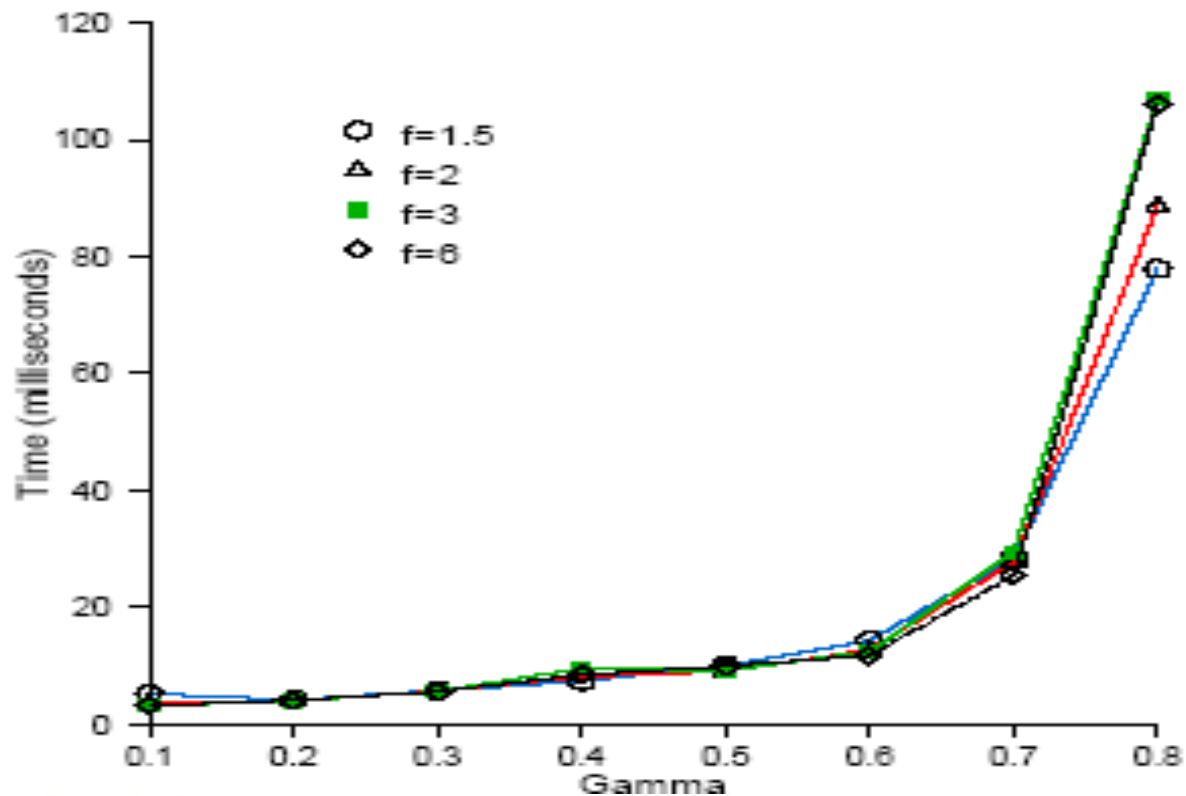


Figure 8: Query Translation Time

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.

DISCUSSION AND RELATED WORK

- The problem of keyword search in relational databases has received much attention in recent years. Early systems like BANKS, DISCOVER, and DBXplorer designed efficient ways to retrieve tuple trees that contained all the keywords in the query.
- A recent work that has addressed the problem of designing keyword based interfaces that can compute powerful aggregates is KDAP.

DISCUSSION AND RELATED WORK

- A recent effort 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 question of whether the techniques may be used to extend SQAK to work over multiple data sources is a topic of future investigation.

CONCLUSIONS

- We argued that current mechanisms do not allow ordinary users to pose aggregate queries easily on complex 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.

CONCLUSIONS

- 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.

CONCLUSIONS

- 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.