THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA7/15

Answering Keyword Queries involving Aggregates and Group-Bys in Relational Databases

Zhong Zeng, Mong Li Lee and Tok Wang Ling

July 2015

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

David ROSENBLUM
Dean of School

Answering Keyword Queries involving Aggregates and Group-Bys in Relational Databases

Zhong Zeng
National University of Singapore
zengzh@comp.nus.edu.sg

Mong Li Lee
National University of Singapore
leeml@comp.nus.edu.sg

Tok Wang Ling
National University of Singapore
lingtw@comp.nus.edu.sg

ABSTRACT

Keyword search over relational databases has gained popularity as it provides a user-friendly way to explore structured data. Current research has focused on the computation of minimal units that contain all the query keywords, and largely ignores queries to retrieve statistical information from the database. The latter involves aggregate functions and group-bys, and are called *aggregate queries*. In this work, we propose a semantic approach to answer keyword queries containing aggregates and group-bys. Our approach utilizes the ORM schema graph to capture the semantics of objects and relationships in the database, and determines the various interpretations of a query. Based on each interpretation, we generate an SQL statement to apply aggregates and group-bys. Further, we detect duplications of objects and relationships arising from denormalized relations so that the aggregate functions will not compute the statistics for the same information repeatedly. Experimental results demonstrate that our approach is able to return correct answers to aggregate queries.

1. INTRODUCTION

As databases increases in size and complexity, the ability for users to issue structured queries in SQL has become a challenge. Keyword search over relational databases has gained traction as it enables users to query the database without knowing the database schema or having to write complicated SQL queries [7, 11, 8, 10, 1, 2]. Research on relational keyword search has focused on the efficient computation of the minimal set of tuples that contain all the query keywords [9, 6, 5, 4], and strategies to retrieve relevant answers to the query [6, 11, 14, 3]. However, these works do not handle aggregate queries.

Aggregate queries is a powerful mechanism that provides users with a summary of the data and statistical information. The work in [13] proposes a prototype system called SQAK that allows aggregate queries to be expressed using simple keywords. A keyword query in SQAK comprises of a set of terms, and at least one of the terms is an aggregate function such as count, number, sum, min, or max. The terms in the query may match the names of relations or attributes or tuple values.

Consider the sample university database in Figure 1. Suppose we want to know the total credits obtained by the student Green, we can issue the keyword query Q_1 = {Green SUM Credit}, where the term SUM indicates the aggregate function SUM on the course credits.

Student

Sid	Sname	Age
s1	George	22
s2	Green	24
s3	Green	21

Course

Code	Title	Credit
c1	Java	5.0
c2	Database	4.0
c3	Multimedia	3.0

Enrol

Sid	Code	Grade
s1	c1	A
s1	c2	B
s1	c3	B
s2	c1	A
s3	c1	A
s3	c3	B

Teach

Code	Lid	Bid
c1	l1	b1
c1	l1	b2
c1	l2	b1
c2	l1	b2
c2	l1	b3
c3	l2	b4

Textbook

Bid	Tname	Price
b1	Programming Language	10
b2	Discrete Mathematics	15
b3	Database Management	12
b4	Multimedia Technologies	20

Lecturer

Lid	Lname	Did
l1	Steven	d1
l2	George	d1

Department

Did	Dname	Fid
d1	CS	f1

Faculty

Fid	Fname
f1	Engineering

Figure 1: Example university database

In order to answer these queries, SQAK [13] models the database schema as a schema graph where each node represents a relation and each edge represents a foreign key-key constraint. Then SQAK identifies the matches of each term in a query. A relation is matched if a term matches its name, or the name of one of its attributes, or the value of some of its tuples. A set of minimal connected subgraphs of the schema graph that contain the matched relations are generated. These subgraphs are translated into SQL statements to retrieve answers from the database. Note that an aggregate function(s) is applied to the attribute that follows the aggregate term in the query. For example, SQAK will generate the SQL statement: SELECT SUM(Credit) FROM Student WHERE Sname='Green' GROUP BY Sname for the query Q_1={Green SUM Credit}.

We observe that incorrect answers may be returned by SQAK when a term in the query matches multiples tuples in a relation. We see that the term Green in Q_1 matches the names of two students $s2$ and $s3$ in Figure 1. This naturally implies that we should find the sum of the credits obtained by each of these students, that is, the total credits for $s2$ is 5 while the total credits for $s3$ is 8. However, SQAK does not distinguish between these two "different" name matches, and outputs a total credits of 13 for students called Green.

Similarly, SQAK may return incorrect answers when a query matches a relation that has more than 2 foreign keys. For instance, the *Teach* relation in Figure 1 contains 3 foreign keys that reference the *Course*, *Lecturer* and *Textbook* relations respectively. If we have a query Q_2 = {Java SUM Price}, the term Java matches a course title while the term Price matches an attribute of the *Textbook* relation. This implies that we should return the total price of the textbooks that are used in the Java course. Based on the *Teach* relation, there are 2 such textbooks $b1$ and $b2$ whose total price is 25. But SQAK will generate the following SQL statement:

```
SELECT SUM(Price) FROM Course C, Teach T, Textbook B
WHERE C.Title='Java' AND T.Code=C.Code AND T.Bid=B.Bid
GROUP BY C.Title
```

which returns 35 for total price because textbook $b1$ appears 2 times for the Java course (i.e., $c1$) in the *Teach* relation.

Finally, many applications that capture historical data and provide exploratory and in-depth data analysis often denormalize their databases to improve runtime performance. This denormalization affects the database schema graph with data duplication. As SQAK does not consider denormalized relations in the database, it may return incorrect answers for queries involving aggregate functions.

Figure 2 shows a denormalized university database where the *Lecturer* relation now has a foreign key that references the *Faculty* relation. Consider the query Q_3={Engineering COUNT Department}, where the term Engineering matches a faculty name while the term Department matches the name of the *Department* relation. SQAK will find the number of departments in Engineering faculty by joining the *Department*, *Lecturer* and *Faculty* relations, and outputs the incorrect answer 2. This is because the values of attributes *Did* and *Fid* in the *Lecturer* relation are duplicated.

Lecturer

Lid	Lname	Did	Fid
l1	Steven	d1	f1
l2	George	d1	f1

Department

Did	Dname
d1	CS

Faculty

Fid	Fname
f1	Engineering

Figure 2: A denormalized university database

In this paper, we present a generic solution to answer keyword queries involving aggregates and group-bys in relational database keyword search. This requires us to address two challenges. First, keyword queries are inherently ambiguous and thus can have multiple interpretations. Thus, we need to identify the various interpretations and apply aggregate functions on the appropriate attributes. Second, we need a mechanism to detect duplications arising from denormalized relations so that the aggregate functions will not repeatedly compute statistics for the same information. We utilize the extended keyword query language and the Object-Relationship-Mixed (ORM) semantics introduced in [15] to identify the context of keywords and interpret the queries. This enables us to process aggregates in keyword queries correctly.

The contributions of our work are summarized as follows:

1. We identify the limitations of the previous work SQAK, as it does not consider the semantics of objects and relationships in the database.
2. We design the syntax for aggregate queries in relational database keyword search, and propose a semantic approach to process these queries.
3. We detect the duplications of objects and relationships arising from denormalized relations, and extend our approach to handle aggregate queries on denormalized databases.
4. We conduct extensive experiments to demonstrate the effectiveness of our approach in retrieving statistical information for users.

2. PRELIMINARIES

The work in [15] extends the keyword query language to include the keywords that match meta-data, i.e., the names of relations and attributes. These keywords provide the context of the subsequent keywords in the query and thus reduce the ambiguity of the query. Consider the query {Lecturer George} on the database in Figure 1. The keyword George can refer to a student name or a lecturer name. However, since the keyword Lecturer matches the name of the relation *Lecturer* and provides the context of the keyword George, indicating that the user is more likely to be interested in the lecturer named George rather than a student. Here, we further extend the query language to incorporate aggregates.

DEFINITION 1. *A keyword query Q is a sequence of terms $\{t_1\ t_2\ \cdots\ t_n\}$ where each term t_i either matches a relation name, an attribute name, a tuple value, GROUPBY or an aggregate function MIN, MAX, AVG, SUM or $COUNT$.*

In order to properly interpret a keyword query involving aggregate function and GROUPBY, we impose the following constraints on the terms in the query:

1. The last term t_n cannot match an aggregate function or GROUPBY.
2. For each term t_i, $i < n$ that matches the aggregate function MIN, MAX, AVG or SUM, the next term t_{i+1} should match an attribute name.
3. For each term t_i, $i < n$ that matches $COUNT$ or GROUPBY, the next term t_{i+1} should match either a relation name or an attribute name.

An example query that satisfies the last constraint is {COUNT Student GROUPBY Course}, and we generate an SQL statement to find the number of students in each course:

```
SELECT COUNT(S.Sid) As numSid
FROM Student S, Enrol E, Course C
WHERE S.Sid=E.Sid AND E.Code=C.Code
GROUPBY C.Code
```

Note that the terms Student and Course which match relation names are mapped to the primary key of these relations, namely *Sid* and *Code* respectively.

2.1 Query Patterns

A keyword query is inherently ambiguous as each keyword in the query can have multiple matches. [15] introduces the notion of query patterns to depict the various interpretations of a keyword query. These query patterns are generated from the Object-Relationship-Mixed (ORM) schema graph of the relational database.

The ORM schema graph is an undirected graph that captures the semantics of objects/relationships in the database. Each node in the ORM schema graph comprises of an object/relationship/mixed relation and its component relations,

and is associated with a type (object, relationship and mixed). An object (relationship) relation captures the information of objects (relationships), namely, the single-valued attributes of an object class (relationship type). Multivalued attributes are stored in the component relations. A mixed relation contains information of both objects and relationships, which occurs when we have a many-to-one relationship. Two nodes are connected if there exists a foreign key - key constraint between the relations in these two nodes.

In Figure 1, the relations *Student*, *Course*, *Faculty* and *Textbook* are object relations while *Enrol* and *Teach* are relationship relations. Relations *Lecturer* and *Departement* are mixed relations because of the many-to-one relationships between lecturers and departments, and the many-to-one relationships between departments and faculties respectively. Figure 3 shows the ORM schema graph of the database.

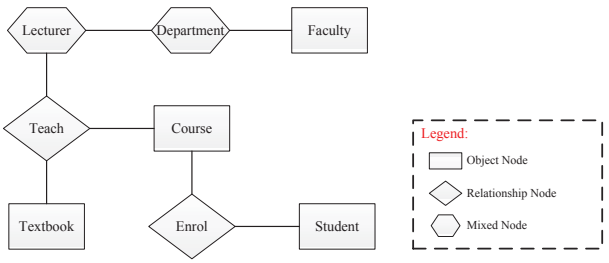


Figure 3: ORM schema graph of Figure 1

Figure 4 shows a query pattern for the keyword query `{Green George Code}`. This pattern depicts the query interpretation to find the common courses taken by students Green and George. To generate this query pattern, we first identify the matches of each term in the query. The term Code matches the name of an attribute in the *Course* relation, while both terms Green and George match some tuple value in the *Student* relation, specifically, the value of the *Sname* attribute. Based on these matches, we know that Green and George refer to two student objects and Code refers to a course object. We create 2 Student nodes and 1 Course node to represent these objects. From the ORM schema graph in Figure 3, the Student node and the Course node can be connected via an Enrol node. Hence, we create 2 Enrol nodes and obtain the query pattern in Figure 4.

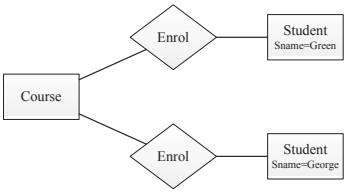


Figure 4: Query pattern of `{Green George Code}`

In this work, we want to utilize these query patterns to capture the interpretations of a keyword query. However, since we extend the keyword query to include GROUPBY and aggregate functions, we need to annotate the nodes that the GROUPBY and aggregate functions are applicable to. Annotating the appropriate nodes is important as it will facilitate the translation of the query pattern into SQL statements to retrieve the correct answers for the aggregate query. We will discuss how we achieve this in the next section.

3. AGGREGATE QUERIES ON NORMALIZED DATABASE

Given a keyword query $Q = \{t_1\ t_2\ \cdots\ t_n\}$, we first classify the terms in the query into *basic terms* and *operators*. A basic term matches a relation name, or an attribute name, or a tuple value in the database, while an operator matches GROUPBY or an aggregate function. We use the basic terms in a query to generate an initial query pattern P which contains a set of nodes that represent the objects or relationships referred to by the basic terms. The nodes are connected based on the ORM schema graph as described in [15]. A node is annotated with the condition $a = t$ if the basic term t refers to the value of the attribute a of the object/relationship.

Next, we consider the operators in the query. For each operator $t_i \in Q$, we examine the matches of its subsequent term t_{i+1} in Q and annotate the query pattern P as follows:

1. t_{i+1} matches the name of some object/mixed/relationship relation.

   This indicates that t_{i+1} refers to some object or relationship, and the operator t_i is applied on the identifier k of this object/relationship. We annotate the node that represents this object/relationship in P with $t_i(k)$, k is given by the primary key of the relation.

2. t_{i+1} matches the name of a component relation or an attribute name.

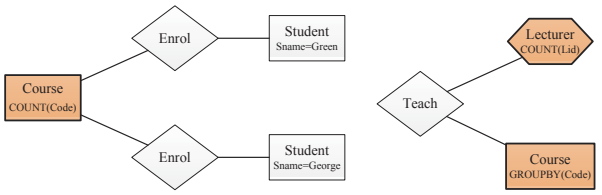
   This indicates that t_{i+1} refers to some attribute a of an object or relationship, and t_i is applied on this object/relationship attribute. We annotate the node that represents this object/relationship in P with $t_i(a)$.

The following example illustrates the various annotations.

EXAMPLE 1. *Consider the keyword query* $Q_4 = \{$*Green George COUNT Code*$\}$. *Figure 4 shows the query pattern obtained using the basic terms* Green, George *and* Code. *For the operator* COUNT, *its subsequent term* Code *matches the name of an attribute in the Course relation. Hence, we annotate the* Course *node with* COUNT(Code). *Figure 5(a) shows the annotated query pattern* P_1 *that depicts the query interpretation to find the total number of courses taken by students* Green *and* George.

On the other hand, the query $Q_5 = \{$*COUNT Lecturer GROUPBY Course*$\}$ *has two basic terms* Lecturer *and* Course. *We generate a query pattern that contains a* Teach *relationship node between the objects* Lecturer *and* Course. *For the operator* GROUPBY, *since its subsequent term* Course *matches the name of the Course relation, and refers to a course object, we obtain the identifier of the course object and annotate the corresponding* Course *node in the query pattern with* GROUPBY(Code). *Similarly, the operator* COUNT *has a subsequent term* Lecturer *that matches the name of the Lecturer relation. We annotate the* Lecturer *node with* COUNT(Lid) *and obtain the annotated query pattern* P_2 *in Figure 5(b). This query pattern indicates that the user is interested in the number of lecturers for each course.* □

After obtaining the annotated query patterns, we will translate them into SQL statements. The straightforward way to translate an annotated query pattern is to join the relations of all the nodes in the pattern, select the tuples that



(a) P_1 (b) P_2

Figure 5: Annotated query patterns of Q_4 and Q_5

satisfy the conditions imposed by basic terms from the join result, and then apply GROUPBY and aggregate function(s) on the selected tuples. However, this may generate an SQL statement that gives an incorrect answer to the query.

EXAMPLE 2. *In Figure 5(a), the query pattern P_1 contains an object node* Student *that is annotated with the condition* Sname = Green*. However, there are two students called* Green *in the database in Figure 1. If we only impose the condition* Sname = Green *on the Student relation, the SQL statement generated will count the total number of courses taken by these two students together, which is not correct.*

Similarly, if we simply translate the query pattern P_2 in Figure 5(b) into an SQL statement that joins the relations Teach, Lecturer and Course, and apply the count aggregate on the lecturer id Lid after grouping the tuples by the course code Code, we may obtain wrong answers as the same lecturer may be counted multiple times. This is because the Teach *node in P_2 is in fact a ternary relationship involving the objects course, lecturer and textbook (see the ORM schema graph in Figure 3). Since different Bids may have the same Lid and Code, we should project the Teach relation on the foreign keys $\langle Lid, Code\rangle$ to remove duplicates before joining with the other relations Lecturer and Course.* □

The above example demonstrates the need to examine the type of nodes in a query pattern if we want to generate the SQL statement correctly. In particular, if we have an object/mixed node v that is annotated with a condition $a = t$, then we should apply the GROUPBY on the primary key of the object/mixed relation of v in order to distinguish the objects that have the same value t for attribute a. Further, if the query pattern contains a relationship node u, we should look at its corresponding node w in the ORM schema graph to determine if a projection is needed to remove duplicates.

Given a query pattern P, we generate the various clauses in an SQL statement as follows:

SELECT clause. If a node $v \in P$ is annotated with $t(a)$ and t matches an aggregate function, we include t in the SELECT clause. t is applied on the attribute a.

FROM clause. The FROM clause includes the relations of all the nodes in P. However, for each relationship node $u \in P$, we check its corresponding node w in the ORM schema graph. Let $S_u = \{u_1, u_2, \cdots, u_x\}$ be a set of object/mixed nodes that are directly connected to u in the query pattern P, and $S_w = \{w_1, w_2, \cdots, w_y\}$ be the set of object/mixed nodes that are directly connected to w in the ORM schema graph. If $x < y$, then this indicates that P contains a subset of the participating objects of the relationship w. In this case, we project the foreign keys $k_1, k_2, \cdots, k_x$ in the relation of u such that k_i references the relation of u_i in S_u, $i \in [1, x]$. This projection eliminates duplicates and we replace the relation of u in the FROM clause with the relation obtained by this projection.

WHERE clause. The WHERE clause joins all the relations in the FROM clause based on foreign key - key constraints. Moreover, for each node $v \in P$ that is annotated with a condition $a = t$, we include the condition "$R_v.a$ contains t" where R_v is the relation corresponding to v.

GROUPBY clause. If a node v is annotated with $t(a)$ and t matches GROUPBY, then we include the attribute a in the GROUPBY clause. Further, for each object/mixed node v that is annotated with $a = t$, we include the primary key of the object/mixed relation corresponding to v in the GROUPBY clause.

EXAMPLE 3. *Consider the query pattern P_1 in Figure 5(a) for the query Q_4 = {*Green George COUNT Code*}. The* Course *node is annotated with* COUNT(Code)*, so we include the aggregate function $COUNT(Code)$ in the SELECT clause. The FROM clause contains the relations corresponding to each of the nodes in P_1. Then we add the conditions to join these relations in the WHERE clause, as well as the conditions in the two annotated Student object nodes. Since P_1 contains two object nodes that are annotated with conditions* Sname = Green *and* Sname = George *respectively, we include the ids of their relations in the GROUPBY clause, and obtain the following SQL statement:*

```
SELECT COUNT(C.Code) AS numCode
FROM Course C, Enrol E1, Student S1, Enrol E2, Student S2
WHERE C.Code=E1.Code AND C.Code=E2.Code
    AND E1.Sid=S1.Sid AND S1.Sname contains 'Green'
    AND E2.Sid=S2.Sid AND S2.Sname contains 'George'
GROUP BY S1.Sid, S2.Sid
```

By applying GROUPBY on the two student ids, we can distinguish the students $s2$ and $s3$ so that the aggregate function COUNT *is computed for each of their courses.* □

EXAMPLE 4. *Next, let us translate the query pattern P_2 in Figure 5(b) for the query Q_5 = {*COUNT Lecturer GROUPBY Course*}. The* Lecturer *node is annotated with* COUNT(Lid) *while the* Course *node is annotated with* GROUPBY(Code)*. Hence, we include the aggregate function $COUNT(Lid)$ in the SELECT clause, and the attribute Code in the GROUPBY clause. In addition, the* Teach *node in P_2 is connected to two object/mixed nodes, while the corresponding* Teach *node in the ORM schema graph in Figure 3 is connected to three object/mixed nodes. We generate a subquery "*SELECT DISTINCT Lid, Code FROM Teach*" to project the attributes Lid and Code in the Teach relation and eliminate any duplicates of $\langle Lid, Code\rangle$. The result of this subquery is used to join the other relations in the FROM clause. The SQL statement generated is as follows:*

```
SELECT count(L.Lid) AS numLid
FROM Lecturer L, Course C,
    (SELECT DISTINCT Lid, Code FROM Teach) T
WHERE L.Lid=T.Lid AND T.Code=C.Code
GROUP BY C.Code
```

□

3.1 Nested Aggregate Queries

So far, we have described how to handle keyword queries involving simple aggregate functions and GROUPBY. In order to maximize the power of aggregate queries, we also want to support queries with nested aggregate functions.

Given a keyword query $Q = \{t_1\ t_2\ \cdots\ t_n\}$, we relax the constraints on the terms so that if the term t_i, $i < n$ matches an aggregate function, the next term t_{i+1} can also match an aggregate function. In this case, the aggregate function t_i is applied on the result of the aggregate function t_{i+1}.

Let P be a query pattern obtained from basic terms in the query. We annotate P with $t_i(f)$, where f is the attribute name assigned to the result of aggregate function t_{i+1}. Then we generate a nested SQL statement for P. The inner query computes the aggregate function t_{i+1}, while the outer query includes the inner query in the FROM clause and computes the aggregate function t_i.

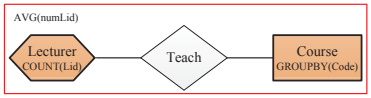


Figure 6: Query pattern in Example 5

EXAMPLE 5. *Suppose the user issues a query {AVG COUNT Lecturer GROUPBY Course} to find the average number of lecturers that teach a course. Both the terms AVG and COUNT match some aggregate function. We obtain the query pattern and annotate the operators COUNT and GROUPBY. For the AVG operator, we annotate the pattern with AVG(numLid), where numLid is the attribute name given to the result of the aggregate function COUNT. Figure 6 shows the annotated query pattern. To translate the query pattern, we first generate the inner SQL query similar to that in Example 4. Then we include it in the FROM clause of the outer SQL query to compute the aggregate function AVG. The SQL statement generated is as follows:*

```
SELECT AVG(R.numLid) AS avgnumLid
FROM ( SELECT COUNT(L.Lid) AS numLid
       FROM Lecturer L, Course C,
            (SELECT DISTINCT Lid, Code FROM Teach) T
       WHERE L.Lid=T.Lid AND T.Code=C.Code
       GROUP BY C.Code) R
```

□

4. AGGREGATE QUERIES ON DENORMALIZED DATABASE

Relations in a relational database are often denormalized to reduce the number of joins and improve query processing performance. A relational database that contains denormalized relations is called a denormalized database. The denormalization process will duplicate information in the database and we may obtain incorrect results for keyword queries involving aggregates.

Recall that in Figure 2, the *Lecturer* relation is denormalized by adding a foreign key that references the *Faculty* relation. This allows queries that are frequently issued on lecturers and their faculties to be answered quickly without the need to join the *Department* relation. Given a query $Q_3 = \{\text{Engineering COUNT Department}\}$, SQAK will join the relations *Lecturer*, *Department* and *Faculty* and return incorrect number of departments in the Engineering faculty as it does not handle denormalized relations.

In order to generate SQL statements correctly for keyword queries involving aggregates, we need to determine if relations are denormalized. This can be done by examining the functional dependencies that hold on the relations.

Consider the denormalized relation *Enrolment* in Figure 7 that is obtained by joining the *Student*, *Enrol* and *Course* relations in Figure 1. The following functional dependencies hold in the *Enrolment* relation:

- $Sid \rightarrow Sname, Age$
- $Code \rightarrow Title, Credit$
- $Sid, Code \rightarrow Grade$

We deduce that $\{Sid, Code\}$ is the key of *Enrolment* relation, which violates second normal form (2NF) definition.

Enrolment

Sid	Sname	Age	Code	Title	Credit	Grade
s1	George	22	c1	Java	5.0	A
s1	George	22	c2	Database	4.0	B
s1	George	22	c3	Multimedia	3.0	B
s2	Green	24	c1	Java	5.0	A
s3	Green	21	c1	Java	5.0	A
s3	Green	21	c3	Multimedia	3.0	B

Figure 7: A denormalized relation

A naive approach to handle a keyword query involving aggregate functions on the denormalized database is to generate a copy of the database where every relation is normalized and then process the query as described in Section 3. However, this approach is expensive and not feasible in practice.

We observe that although the relations are denormalized, the information of objects and relationships in the database remain the same. Hence, if we can keep track of the objects and relationships information in a denormalized database, then we can continue to process keyword queries involving aggregates correctly.

Recall that the ORM schema graph captures the information of objects and relationships in the database by classifying the relations into different types. These relations are assumed to be in 3NF. Thus, we generate a *normalized view* of the denormalized database comprising of a minimal set of relations in 3NF. Then we classify the relations in this normalized view and construct the ORM schema graph to represent the semantics of objects and relationships in the denormalized database.

Let $D = \{R_1, R_2, \cdots, R_j\}$ be the set of relations in the original database schema, and D' be the set of relations in the normalized view. For each $R_i \in D$, $1 \leq i \leq j$, if R_i is in 3NF, then we add it to D'. Otherwise, we normalize R_i into a set of relations in 3NF and add them to D'. Finally, relations in D' with the same key are merged. We use relational algebra operators to express the mappings of the relations from D to D', and vice versa.

EXAMPLE 6. *Let us generate the normalized view of the denormalized university database schema D below:*

$Teach(\underline{Code, Lid, Bid})$
$Textbook(\underline{Bid}, Tname, Price)$
$Faculty(\underline{Fid}, Fname)$
$Department(\underline{Did}, Dname)$
$Enrolment(\underline{Sid, Code}, Sname, Age, Title, Credit, Grade)$
$Lecturer(\underline{Lid}, Lname, Did, Fid)$

Since the relations Teach, Textbook, Department and Faculty are all in 3NF, we add them to the normalized view D' directly. To avoid confusion, we name these relations in D' as $Teach'$, $Textbook'$, $Department'$ and $Faculty'$ respectively. For the Enrolment relation, we decompose it

into the 3NF relations $Student'$, $Enrol'$ and $Course'$, and add them to D'. Similarly, we decompose the Lecturer relation into two relations $Lecturer'(Lid, Lname, Did)$ and $DF'(Did, Fid)$. Since both the $Department'(Did, Dname)$ and the $DF'(Did, Fid)$ relations have the same key $\{Did\}$, we merge them into one relation $Department'(Did, Dname, Fid)$. Thus, the normalized view D' will have the relations:

$Teach'(\underline{Code, Lid, Bid})$
$Textbook'(\underline{Bid}, Tname, Price)$
$Faculty'(\underline{Fid}, Fname)$
$Department'(\underline{Did}, Dname, Fid)$
$Student'(\underline{Sid}, Sname, Age)$
$Enrol'(\underline{Sid, Code}, Grade)$
$Course'(\underline{Code}, Title, Credit)$
$Lecturer'(\underline{Lid}, Lname, Did)$

Table 1 shows the mappings of the relations in D and D'. □

Table 1: Mappings of relations in Example 6

$Teach = Teach'$
$Textbook' = Textbook$
$Faculty = Faculty'$
$Department = \Pi_{Did,Dname}(Department')$
$Enrolment = Student' \bowtie Enrol' \bowtie Course'$
$Lecturer = Lecturer' \bowtie \Pi_{Did,Fid}(Department')$

(a) From original schema D to normalized view D'

$Teach' = Teach$
$Textbook' = Textbook$
$Faculty' = Faculty$
$Department' = Department \bowtie \Pi_{Did,Fid}(Lecturer)$
$Student' = \Pi_{Sid,Sname,Age}(Enrolment)$
$Enrol' = \Pi_{Sid,Code,Grade}(Enrolment)$
$Course' = \Pi_{Code,Title,Credit}(Enrolment)$
$Lecturer' = \Pi_{Lid,Lname,Did}(Lecturer)$

(b) From normalized view D' to original schema D

Next, we construct the ORM schema graph G of the database based on the normalized view D'. Given a query Q on the denormalized database with schema D, We first identify the matches of each basic term in the denormalized database.

Let R be the relation in D such that a basic term t matches the relation name of R, or the name of an attribute in R, or the value of some tuples in R. We obtain the corresponding relations of R in D' based on the mappings from D to D'. From these relations in D', we generate the query patterns based on G, and annotate these patterns with the operators in the query as described in Section 3.

Finally, we translate the annotated query pattern into an SQL statement to be executed over the original denormalized database. This requires us to map the relations in the query pattern back to their corresponding relations in D. Depending on the mappings, a relation R' that corresponds to a node in the query pattern may become a subquery in the SQL statement as we illustrate in the following example.

EXAMPLE 7. *Suppose we denormalize the database in Figure 1 based on the schema in Example 6. Consider the query Q_3 = {Engineering COUNT Department}. The term Engineering matches some tuple value in the Faculty relation, while Department matches the name of the Department relation. Based on Table 1(a), these two relations correspond to relations $Faculty'$ and $Department'$ respectively. This indicates that the term Engineering refers to a faculty object, while the term Department refers to a department object. Based on the ORM schema graph in Figure 3, we generate a query pattern that connects a Faculty node and a Department node, and annotate it with the operator COUNT. Figure 8 shows the query pattern obtained. From the mappings in Table 1(b), we generate the SQL statement:*

```
SELECT COUNT(D'.Did) AS numDid
FROM Faculty F',
    (SELECT DISTINCT D.Did, D.Dname, L.Fid
    FROM Department D, Lecturer L WHERE D.Did=L.Did) D'
WHERE D'.Fid=F'.Fid AND F'.Fname contains 'Engineering'
GROUP BY F'.Fid
```

Note that we have a subquery for the $Department'$ relation based on the mapping $Department' = Department \bowtie \Pi_{Did,Fid}(Lecturer)$. □

Figure 8: Query pattern in Example 7

4.1 Query Rewriting

Given a query pattern P, the mapping from the relation $R' \in D'$ to the relation $R \in D$ for a node in P may often involve part of attributes of R. The SQL statement obtained may contain a lot of subqueries. Joining relations obtained from subqueries is time consuming due to the lack of indexes.

EXAMPLE 8. *Consider the query pattern in Figure 5(a) for the query Q_4 = {Green George COUNT Code}. The SQL statement generated has 5 subqueries for the denormalized university database:*

```
SELECT COUNT(C.Code) AS numCode FROM
  (SELECT DISTINCT Code, Title, Credit FROM Enrolment) C',
  (SELECT Sid, Code, Grade FROM Enrolment) E1',
  (SELECT DISTINCT Sid, Sname, Age FROM Enrolment) S1',
  (SELECT Sid, Code, Grade FROM Enrolment) E2',
  (SELECT DISTINCT Sid, Sname, Age FROM Enrolment) S2'
WHERE C'.Code=E1'.Code AND E1'.Sid=S1'.Sid AND
      C'.Code=E2'.Code AND E2'.Sid=S2'.Sid AND
      S1'.Sname contains 'Green' AND
      S2'.Sname contains 'George'
GROUP BY S1'.Sid, S2'.Sid
```
□

Hence, it is crucial to rewrite the generated SQL statement to improve query performance. We observe that some attribute in SELECT clause of the subqueries are never used, and thus can be removed. In Example 8, we can rewrite the subquery "SELECT DISTINCT Code, Title, Credit FROM Enrolment" to "SELECT DISTINCT Code FROM Enrolment", as the attributes Title and Credit are not used.

Further, some select conditions of the SQL statement can be moved to the WHERE clause of the subqueries so that unsatisfied tuples can be filtered out before joining, e.g., we can rewrite the subquery "SELECT DISTINCT Sid, Sname, Age FROM Enrolment" to "SELECT DISTINCT Sid, Sname, Age FROM Enrolment WHERE Sname contains 'Green'" to filter out the students whose names are not Green.

Finally, relations are denormalized to reduce the number of joins. We can try to use the denormalized relation to replace the joining of relations obtained from sub-

queries. For example, the *Enrolment* relation is equivalent to the joins of relations obtained from the subqueries "SELECT DISTINCT Code, Title, Credit FROM Enrolment", "SELECT Sid, Code, Grade FROM Enrolment", and "SELECT DISTINCT Sid, Sname, Age FROM Enrolment". Hence, we can use the *Enrolment* relation to replace these subqueries. We derive three heuristics to rewrite an SQL statement *sql*:

Rule 1: If a subquery projects an attribute that does not appear in the SELECT and WHERE clause of *sql*, then remove this attribute.

Rule 2: If a subquery projects an attribute a that appears in the condition "a contains t" of *sql*, then put this condition in the WHERE clause of the subquery.

Rule 3: Let $s_1, s_2, \cdots, s_m$ be a set of subqueries in *sql*. if there exists a relation R such that $s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m = \Pi_L(R)$, where L is a superkey of R, then replace $s_1 \bowtie s_2 \bowtie \cdots \bowtie s_m$ with R.

EXAMPLE 9. *Consider the SQL statement in Example 8. Since the joins of the subqueries "SELECT DISTINCT Code, Title, Credit FROM Enrolment", "SELECT Sid, Code, Grade FROM Enrolment", and "SELECT DISTINCT Sid, Sname, Age FROM Enrolment" is equivalent to the Enrolment relation, we use the Enrolment relation to replace $C' \bowtie E1' \bowtie S1'$. Besides, we see that the joins of the subqueries "SELECT Sid, Code, Grade FROM Enrolment" and "SELECT DISTINCT Sid, Sname, Age FROM Enrolment" is equivalent to a relation obtained by projecting a super key (Sid, Code, Title, Credit, and Grade) of the Enrolment relation. Hence, we can also use the Enrolment relation to replace $E2' \bowtie S2'$. As a result, we rewrite the SQL and obtain the following statement:*

```
SELECT COUNT(R1.Code) AS numCode
FROM Enrolment R1, Enrolment R2
WHERE R1.Code=R2.Code AND R1.Sname contains 'Green'
      AND R2.Sname contains 'George'
GROUP BY R1.Sid, R2.Sid
```
□

5. ALGORITHMS

Given the schema D of a relational database, we first check if every relation in D is in 3NF. If not, we generate a normalized view D' of the database, and obtain the mappings between D and D'. Then, we use Algorithm 1 to process a keyword query Q and generate the SQL statements.

If the database schema D is normalized, we construct the ORM schema graph G based on D. For each term t_i in Q, if t_i is a basic term, we create a set of tags for t_i to capture its interpretations, and insert this tag set into *Tlist*; otherwise, t_i is an operator and we inert it into *Olist* (Lines 4-9). Based on *Tlist* and G, we generate a list of query patterns *Plist*. For each pattern P in *Plist*, we annotate P with the operators in *Olist*. For each operator t in *Olist*, let t' be the next term of t in Q. If t' is a basic term, we check its matches in D. Let v be a node in P and R be the relation of v. If t' matches the name of R, then we annotate v with $t(R.key)$. Otherwise, if t' matches the name of an attribute a of R, we annotate v with $t(R.a)$. If t' is also an operator, then we annotate the pattern P with $t(f)$ to indicate that t is a nested aggregate function (Lines 11-21). After annotating P, we translate P into an SQL statement *sql* according to D, and insert it into *SQLlist* (Lines 22-23).

On the other hand, if the database schema D is denormalized, we construct the ORM schema graph G based on D'. For each basic term t_i, we create the tags for t_i based on the matches in D and their mappings in D'. Similarly, we generate a list of query patterns *Plist* based on the tags and the ORM schema graph, and annotate each pattern P in *Plist* with the operators. Then we translate each pattern P into an SQL statement *sql* based on D' and D. Finally, we rewrite *sql* to *sql'* to reduce the number of subqueries and insert *sql'* into *SQLlist* (Lines 25-46).

Algorithm 1: Keyword Search

```
   Input: Q = {t_1 ··· t_n}, database schema D and normalized
          view D'
   Output: a list of SQL statements SQLlist
 1 SQLlist ← ∅; Plist ← ∅; Tlist ← ∅; Olist ← ∅;
 2 if D is normalized then
 3     G = createORMGraph(D);
 4     for i = 1 to n do
 5         if t_i is a basic term then
 6             Tset_i = createTags(t_i,D, G);
 7             Insert Tset_i into Tlist;
 8         else
 9             Insert t_i into Olist;
10     Plist = createPatterns(Tlist, G);
11     foreach Pattern P in Plist do
12         foreach Operator t in Olist do
13             Let t' be the next term of t in Q;
14             if t' is a basic term then
15                 Let v be a node in P and R be the relation of v;
16                 if t' matches R then
17                     Annotate v with t(R.key);
18                 else if t' matches an attribute a in R then
19                     Annotate v with t(R.a);
20             else
21                 Annotate P with t(f);
22         sql = translate(P, D);
23         Insert sql into SQLlist;
24 else
25     G = createORMGraph(D');
26     for i = 1 to n do
27         if t_i is a basic term then
28             Tset_i = createTags(t_i, D', D, G);
29             Insert Tset_i into Tlist;
30         else
31             Insert t_i into Olist;
32     Plist = createPatterns(Tlist, G);
33     foreach Pattern P in Plist do
34         foreach Operator t in Olist do
35             Let t' be the next term of t in Q;
36             if t' is a basic term then
37                 Let v be a node in P and R be the relation of v;
38                 if t' matches R then
39                     Annotate v with t(R.key);
40                 else if t' matches an attribute a in R then
41                     Annotate v with t(R.a);
42             else
43                 Annotate P with t(f);
44         sql = translate(P, D', D);
45         sql' = rewrite(sql, D', D);
46         Insert sql' into SQLlist;
47 return SQLlist;
```

6. PERFORMANCE STUDY

In this section, we evaluate the performance of our approach to process keyword queries involving aggregrates and group-bys. We implement the algorithms in Java and carry out experiments on a 3.40 GHz CPU with 8 GB RAM. We use two relational databases in our experiments: the TPC-H database (TPCH) and the ACM Digital Library publication (ACMDL). Table 2 shows the schemas of these databases. We construct queries involving aggregates and group-bys for each database. Table 3 shows the queries.

Table 2: Database schemas

TPCH
Part(partkey, pname, type, size, retailprice)
Supplier(suppkey, sname, nationkey, acctbal)
Lineitem(partkey, suppkey, orderkey)
Order(orderkey, custkey, totalprice, date, priority)
Customer(custkey, cname, nationkey, mktsegment)
Nation(nationkey, nname, regionkey)
Region(regionkey, rname)

ACMDL
Paper(paperid, procid, date, ptitle)
Author(authorid, fname, lname)
Editor(editorid, fname, lname)
Proceeding(procid, acronym, title, date, pages, publisherid)
Publisher(publisherid, code, name)
Write(authorid, paperid)
Edit(editorid, procid)

Table 3: Queries used in experiments

TPCH	
$T1$	AVG totalprice
$T2$	MAX COUNT order GROUPBY nation
$T3$	COUNT order "royal olive"
$T4$	MAX acctbal "yellow tomato"
$T5$	COUNT supplier "Indian black chocolate"
$T6$	COUNT part GROUPBY supplier
$T7$	COUNT order SUM totalprice GROUPBY mktsegment
$T8$	COUNT supplier "pink rose" "white rose"

ACMDL	
$A1$	AVG pages
$A2$	COUNT paper GROUPBY proceeding SIGMOD
$A3$	COUNT proceeding Smith
$A4$	MAX date Gill
$A5$	COUNT author "database tuning"
$A6$	COUNT paper MAX date IEEE
$A7$	COUNT paper author John Mary
$A8$	COUNT editor SIGIR CIKM

6.1 Effectiveness Experiments

Our approach has the ability to identify various interpretations of a keyword query in order to compute answers correctly. This is achieved by examining the semantics of objects and relationships in the database. We compare our approach with SQAK [13], the state-of-the-art relational keyword search engine that processes aggregate queries without considering the semantics of objects and relationships.

SQAK takes an aggregate query and finds a set of relations that are matched by query terms. A relation is matched if a term matches the name of the relation, or the name of one of its attributes, or the relation tuples. Based on these relations, it generates a set of minimal connected graphs called simple query networks (SQN). The SQNs are used to generate the SQL statements to return the answers.

6.1.1 Results for TPCH Database

Table 4 shows the answers of queries returned by our approach and SQAK, as well as explanations for these answers. Although both our approach and SQAK give the same answer for queries $T1$ and $T2$, they differ greatly for the rest.

Queries $T3$ and $T4$ show that our approach is able to distinguish the various interpretations of query terms that match objects with the same value. For query $T3$, our approach returns the number of orders for each "royal olive" part, while SQAK returns the number of orders for all the "royal olive" parts. This is because we differentiate parts with the same name by their object identifiers partkey. Similarly, for $T4$, our approach returns the maximum account balance of suppliers for each "yellow tomato" part, whereas SQAK returns the maximum account balance among all the suppliers that supply a "yellow tomato".

Queries $T5$ and $T6$ show that by examining the relationships and their participating objects, our approach is able to generate SQL statements that compute the aggregates correctly. For query $T5$, our approach returns 4 for the number of suppliers that supply "Indian black chocolate". SQAK counts the same suppliers multiple times for different the orders and returns 22, a value that is way above the actual number of suppliers. Similarly for $T6$, our approach detects the duplicates of suppliers for different orders, and returns the correct number of parts supplied by each supplier, while SQAK returns incorrect answers.

Queries $T7$ and $T8$ demonstrate that our approach can answer aggregate queries that SQAK does not handle. Query $T7$ requires an SQL statement that contains 2 aggregate functions in the SELECT clause. However, SQAK restricts that the SELECT clause of a generated SQL statement specifies exactly one aggregate function. Query $T8$ requires an SQL statement to join 2 *Part* relations, but SQAK does not generate SQL statements that contain self joins of relations.

6.1.2 Results for ACMDL Database

Table 5 shows the answers for the queries on the ACMDL database. Query $A1$ is relatively straightforward, and both our approach and SQAK return the correct answer. For $A2$, SQAK also gives the correct answer because the term SIGMOD matches a proceeding acronym and there is no other proceedings with the same acronym. However, for queries $A3$ and $A4$, there are 61 editors with name Smith and 36 authors with name Gill in the database. Since SQAK does not distinguish the editors and authors with the same name, it returns incorrect number of proceedings and most recent date of papers respectively. Similarly, for $A6$, our approach returns 6 answers while SQAK only returns 4 answers, as it mixes some papers with the same title.

Query $A5$ involves 2 aggregate functions. Queries $A6$ and $A7$ require self joins of two *Author* relations and two *Editor* relations respectively. SQAK is unable to process these queries, while our approach returns the correct answers.

6.1.3 Queries on Denormalized Databases

Next, we denormalized the ACMDL and TPCH databases, and obtain the schemas in Table 6. We use the queries in Table 3 on the denormalized databases and compare the results returned by our approach and SQAK.

Table 6: Denormalized database schemas

TPCH'
Ordering(partkey, suppkey, orderkey, pname, type, size, retailprice, sname, nationkey, regionkey, acctbal, custkey, totalprice, date, priority)
Customer(custkey, cname, nationkey, regionkey, mktsegment)
Nation(nationkey, nname)
Region(regionkey, rname)

ACMDL'
PaperAuthor(paperid, authorid, procid, date, title, fname, lname)
EditorProceeding(editorid, procid, fname, lname, acronym, title, date, pages, publisherid)
Publisher(publisherid, code, name)

Table 4: Answers of queries for TPCH database

#	Proposed Approach		SQAK	
	Answer	Explanation	Answer	Explanation
$T1$	AVG totalprice: 1.42×10^5	average totalprice of orders	AVG totalprice: 1.42×10^5	average totalprice of orders
$T2$	MAX COUNT order: 6568	maximum number of orders in a nation	MAX COUNT order: 6568	maximum number of orders in a nation
$T3$	8 answers: 23, 22, 29, 27, 33, 35, 33, 27	number of orders for each "royal olive" part	1 answers: 229	mix all "royal olive" parts
$T4$	13 answers: 6361.20, 9538.15, ..., 7916.56	maximum account balance of suppliers for each "yellow tomato" part	1 answers: 9844.00	mix all "yellow tomato" parts
$T5$	COUNT supplier: 4	number of suppliers that supply "Indian black chocolate"	COUNT supplier: 22	same suppliers are counted multiple times for various orders
$T6$	1000 answers: 80, 80, 79, 80, ...	number of parts supplied for each supplier	1000 answers: 593, 571, 595, 606, ...	same parts are counted multiple times for various orders
$T7$	5 answers: $\langle 2.99 \times 10^4, 4.26 \times 10^9 \rangle$, ..., $\langle 3.03 \times 10^4, 4.33 \times 10^9 \rangle$	one answer for each market segment	N.A.	do not handle more than one aggregate
$T8$	3 answers: 1, 1, 1	number of suppliers that supply a particular "pink rose" and a particular "white rose"	N.A.	do not handle self joins of relations

Table 5: Answers of queries for ACMDL database

#	Proposed Approach		SQAK	
	Answer	Explanation	Answer	Explanation
$A1$	AVG ages: 297	average pages of proceedings	AVG ages: 297	average pages of proceedings
$A2$	36 answers: 84, 84, 82, ...	number of papers for each 'SIGMOD' proceeding	36 answers: 84, 84, 82, ...	number of papers for each 'SIGMOD' proceeding
$A3$	61 answers: 1, 1, 2, ...	number of proceedings edited by each editor named 'Smith'	1 answers: 62	mix all editors named 'Smith'
$A4$	36 answers: 1994-05-01, 1998-08-01, ...	most recent date of papers written by each author named 'Gill'	1 answer: 2011-06-13	mix all authors named 'Gill'
$A5$	6 answers: 2, 2, 2, 6, 2, 2	number of authors for each "database tuning" paper	4 answers: 2, 4, 6, 4	mix papers with the same title
$A6$	4 answers: $\langle$4011, 2011-01-25$\rangle$, ...	number of papers published by 'IEEE' and their most recent date	N.A.	do not handle more than one aggregate
$A7$	46 answers: 1, 32, 8, 1, ...	number of papers written by a particular author 'John' and a particular author 'Mary'	N.A.	do not handle self joins of relations
$A8$	2 answers: 1, 1	number of editors that edit a 'SIGIR' and a 'CIKM' proceeding	N.A.	do not handle self joins of relations

Table 7: Answers of queries on denormalized TPCH

#	Proposed Approach	SQAK
$T1$	AVG totalprice: 1.42×10^5	AVG ages: 1.78×10^5
$T2$	MAX COUNT order: 6568	MAX COUNT order: 26485
$T3$	8 answers: 23, 22, 29, 27, 33, 35, 33, 27	1 answers: 229
$T4$	13 answers: 6361.20, 9538.15, ..., 7916.56	1 answer: 9844.00
$T5$	COUNT supplier: 4	COUNT supplier: 22
$T6$	1000 answers: 80, 80, 79, ...	1000 answers: 593, 571, ...
$T7$	5 answers: $\langle 2.99 \times 10^4, 4.26 \times 10^9 \rangle$, ...	N.A.
$T8$	3 answers: 1, 1, 1	N.A.

Table 8: Answers of queries on denormalize ACMDL

#	Proposed Approach	SQAK
$A1$	AVG ages: 297	AVG ages: 637
$A2$	36 answers: 84, 84, 82, ...	36 answers: 2000, 408, 14858, ...
$A3$	61 answers: 1, 1, 2, ...	1 answers: 62
$A4$	36 answers: 1994-05-01, 1998-08-01, ...	1 answer: 2011-06-13
$A5$	6 answers: 2, 2, 2, 6, 2, 2	4 answers: 2, 4, 6, 4
$A6$	4 answers: $\langle$4011, 2011-01-25$\rangle$, ...	N.A.
$A7$	46 answers: 1, 32, 8, 1, ...	N.A.
$A8$	2 answers: 1, 1	N.A.

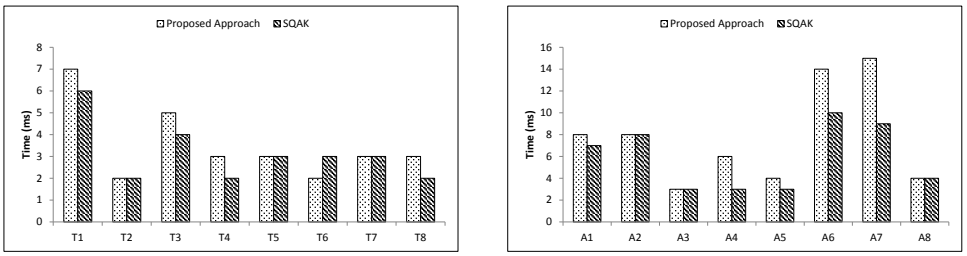
Tables 7 and 8 show that our approach continues to return correct answers to the queries. In contrast, SQAK either returns incorrect answers or does not handle the queries. For queries $T1$ and $T2$, SQAK returns the values 1.78×10^5 and 26485 respectively because the information of orders are duplicated in the denormalized relation *Ordering*. Similarly, SQAK returns the answer 637 for $A1$, and 2000, 408, 14858, etc. (36 answers) for $A2$, both of which are incorrect as the information of proceedings and papers are duplicated in the *EditorProceeding* and *PaperAuthor* relations. Note that these queries are answered correctly by SQAK when the database is normalized.

This set of experiments clearly demonstrate that the semantics of objects and relationships are important to distinguish the various interpretations of keyword queries so that the generated SQL statements will compute statistical information from the database correctly.

6.2 Efficiency Experiments

Finally, we compare the time taken by our approach and SQAK to generate SQL statements. Figure 9 shows the results for both TPCH and ACMDL queries in Table 3.

We observe that our approach is slightly slower than SQAK for most of the queries. This is because SQAK does not analyze the interpretations of keyword queries but only finds SQNs containing all the query terms. Besides, it also does not detect the duplications arising from denormalized re-



(a) TPCH (b) ACMDL

Figure 9: Comparison of the time taken by our approach and SQAK to generate SQL statements

lations. As the SQL execution time dominates the overall processing time (in seconds), the extra time (in ms) required by our approach to interpret the keyword queries and detect the duplicates is a good tradeoff and important to retrieve correct answers from the databases.

7. RELATED WORK

Existing works on keyword search in relational databases can be broadly classified into data graph approach and schema graph approach. In the data graph approach, the relational database is modeled as a graph where each node represents a tuple and each edge represents a foreign key - key reference. BANKS [8] defines an answer to a keyword query as a Steiner tree that contains all the keywords, and proposes a backward expansion search to find the Steiner trees. [9] use a bidirectional expansion technique to reduce the search space. [4] employs a dynamic programming technique to identify the top-k minimal group Steiner trees.

In the schema graph approach, the database schema is modeled as a graph where each node represents a relation and each edges represents a foreign key - key constraint. DISCOVER [7] proposes a breadth-first traverse on the schema graph to generate a set of SQL statements. Each SQL joins a minimal number of relations and ouputs tuples that contain all the keywords. [6] and [11] relax the requirement that output tuples should contain all the query keywords, and develop top-k keyword query techniques to improve efficiency of [7]. [1] exploits the relative positions of keywords in a query and auxiliary external knowledge to generate SQL statements that satisfy users' search intention.

The above works only consider individual tuples that contain query keywords and try to link them by foreign key - key references, and largely ignore statistical information in these tuples. [16] studies the problem of aggregate keyword search on a universal relation. Given a keyword query, it finds a set of tuples that are grouped by a minimal number of attributes and contain all the keywords. [12] classifies the query keywords into dimensional keywords and general keywords, and computes the subgraphs that contain all the dimensional keywords and some general keywords. Then these subgraphs are grouped based on the dimensional keywords to compute the statistical information of the subgraphs. However, none of these works can answer our aggregate queries.

SQAK [13] generates a set of SQL statments from a keyword query containing reserved keywords to indicate the aggregate functions in the SQL statements. But, it does not consider the semantics of objects and relationships in the database, and thus returns incorrect answers as we have highlighted. Moreover, SQAK cannot handle queries when the relations in the database are denormalized.

8. CONCLUSION

In this paper, we have studied the problem of answering keyword queries involving aggregates and group-bys in relational databases. This is achieved by capturing the semantics of objects and relationships in the database with the ORM schema graph. Given a query involving aggregates and group-bys, we utilize the ORM schema graph to determine the various interpretations of the queries. Based on these interpretations, we generate SQL statements which apply aggregate functions to compute the statistical information. We further detect duplications of objects and relationships arising from denormalized relations, so that the aggregate functions will not repeatedly compute statistics for the same information. Experimental results demonstrate the our approach returns correct answers to aggregate queries both on normalized and denormalized databases.

9. REFERENCES

[1] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo Lado, and Y. Velegrakis. Keyword search over relational databases: a metadata approach. In *SIGMOD*, 2011.

[2] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *CIKM*, 2010.

[3] J. Coffman and A. C. Weaver. Learning to rank results in relational keyword search. In *CIKM*, 2011.

[4] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[5] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.

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

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

[8] A. Hulgeri and C. Nakhe. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[9] V. Kacholia, S. Pandit, and S. Chakrabarti. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[10] G. Li, B. C. Ooi, and J. Feng. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.

[11] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: top-k keyword query in relational databases. In *SIGMOD*, 2007.

[12] L. Qin, J. X. Yu, and L. Chang. Computing structural statistics by keywords in databases. In *ICDE*, 2011.

[13] S. Tata and G. M. Lohman. SQAK: Doing more with keywords. In *SIGMOD*, 2008.

[14] X. Yu and H. Shi. CI-Rank: Ranking keyword search results based on collective importance. In *ICDE*, 2012.

[15] Z. Zeng, Z. Bao, T. N. Le, M. L. Lee, and W. T. Ling. Expressq: Identifying keyword context and search target in relational keyword queries. In *CIKM*, 2014.

[16] B. Zhou and J. Pei. Answering aggregate keyword queries on relational databases using minimal group-bys. In *EDBT*, 2009.