Answering Keyword Queries involving Aggregates and GROUPBY on 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 in keyword search has largely ignored queries to retrieve statistical information from the database. The work in [13] extends keywords by supporting aggregate functions in their SQAK system. However, SQAK does not consider the semantics of objects and relationships in the database, and thus suffers from the problems of returning incorrect answers. In this work, we propose a semantic approach to answer keyword queries involving aggregates and GROUPBY. Our approach utilizes the ORM schema graph to capture the Object-Relationship-Attribute (ORA) semantics in the database, and determines the various interpretations of a query before generating the corresponding SQL statements. These semantics enable us to distinguish objects with the same attribute value and detect duplications of objects in relationships to compute the answers correctly. Our approach can also handle unnormalized relations in the database and GROUPBY in keyword queries which SQAK cannot. Experiments on the TPC-H and ACM Digital Library publication datasets demonstrate the advantages of the proposed semantic approach in retrieving correct statistical information for users.

1. INTRODUCTION

As databases increase in size and complexity, the ability for users to issue structured queries in SQLs 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, 10, 8, 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, 10, 14, 3]. However, these works do not handle keyword queries involving aggregate functions and GROUPBY. We call these aggregate queries.

Aggregate queries is a powerful mechanism that provides users with a summary of the data. The work in [13] developed a prototype system called SQAK that allows aggregate queries to be expressed using simple keywords. An aggregate query in SQAK comprises of a set of terms and one of the terms is an aggregate function such as $COUNT$, SUM, AVG, MIN, or MAX. SQAK models the database schema as a schema graph where each node represents a relation and each edge represents a foreign key-key reference. 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.

Figure 1 shows a sample university database. Suppose we want to know the total credits obtained by the student Green. We can issue a keyword query Q_1 = {Green SUM Credit}, where the term SUM indicates the aggregate function SUM on the course credits, and SQAK will generate the following SQL statement for the query:

```
SELECT S.Sname, SUM(C.Credit)
FROM Student S, Enrol E, Course C
WHERE E.Sid=S.Sid AND E.Code=C.Code
AND S.Sname='Green' GROUP BY Sname
```

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

We observe that SQAK may compute incorrect answers when a query term matches multiple tuples. 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,

i.e., 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, which is incorrect.

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 relations *Course*, *Lecturer* and *Textbook* respectively, and depicts that a course can be taught by more than one lecturer using different textbooks. Suppose we have a query Q_2 = {Java SUM Price}, where 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 C.Title, SUM(B.Price)
FROM Course C, Teach T, Textbook B
WHERE T.Bid=B.Bid AND T.Code=C.Code
AND C.Title='Java' 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. This answer is incorrect as a student does not need 2 copies of the same textbook for a course.

In addition, many applications often denormalize their databases to improve runtime performance. This denormalization leads to data duplication which affects the database schema graph. As SQAK does not consider unnormalized relations in the database, it will return incorrect answers for aggregate queries.

Figure 2 shows an unnormalized 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 relations *Department*, *Lecturer* and *Faculty*, and output an 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: An unnormalized university database

We advocate that a relational database is essentially a repository of objects that interact with each other via relationships that are embedded in foreign key-key references. Since SQAK does not consider the semantics of objects and relationships in the database, it will not be able to distinguish objects with the same attribute value (as in Q_1), and it will fail to detect the duplications of objects in relationships (as in Q_2). This leads to the incorrect computations of aggregate queries. Further, if relations are unnormalized with duplicate information of objects and relationships, SQAK may compute the same information repeatedly and return incorrect answers to aggregate queries (as in Q_3).

In this paper, we propose a semantic approach to answer keyword queries involving aggregates and GROUPBY on relational databases. Our approach utilizes the ORM schema graph introduced in [15] to capture the Object-Relationship-Attribute (ORA) semantics in the database. Given an aggregate query, we analyze the context of query keywords, interpret the various interpretations of the query and then apply aggregate functions and GROUPBY on the appropriate attributes of objects/relationships based on the ORM schema graph. Each query interpretation is denoted as a minimal connected graph called annotated query pattern. The top-k ranked annotated query patterns are translated into SQL statements to compute the answers to the aggregate query. By using the ORA semantics, we can distinguish the objects with the same attribute value as well as detect the duplications of objects in relationships in order to avoid incorrect computations of aggregate functions. Otherwise, it is impossible to answer aggregate queries correctly. We also develop a mechanism to detect duplicate information of objects/relationships arising from unnormalized relations so that the aggregate functions will not repeatedly compute statistics for the same information.

The contributions of our work are summarized as follows:

1. We examine how SQAK answers aggregate queries in relational keyword search, and identify its problems of returning incorrect answers due to its unawareness of the ORA semantics in the database.

2. We extend the keyword query language to incorporate aggregates and GROUPBY, and propose a semantic approach to process aggregate queries. We show that without the ORA semantics, it is impossible to process the aggregate functions correctly.

3. By using the ORA semantics, we detect the duplications of objects and relationships arising from unnormalized relations, and extend our approach to handle aggregate queries on unnormalized databases correctly.

4. We conduct extensive experiments to demonstrate the correctness of our approach in retrieving statistical information for users.

2. PRELIMINARIES

The work in [15] extends the keyword query language to include keywords that match meta-data, i.e., the names of relations and attributes. These keywords reduce query ambiguity by providing the context of subsequent keywords in 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, we deduce that the user is more likely to be interested in a lecturer named George rather than a student. Here, we further extend the query language to incorporate aggregates and GROUPBY.

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 functions 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.

A query that satisfies the last constraint is {COUNT Student GROUPBY Course}. An SQL statement to find the number of students in each course is generated as follows:

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

Note that the terms Student and Course match the names of the *Student* and *Course* relations, and are mapped to *Sid* and *Code* respectively.

2.1 Query Patterns

The work in [16] classifies the relations in a database into object relations, relationship relations, mixed relations and component relations. An object (relationship resp.) relation captures the information of objects (relationships resp.), i.e., the single-valued attributes of an object class (relationship type). Multivalued attributes of an object class (relationship type) are captured in object/relationship component relations. A mixed relation contains information of both objects and relationships, which occurs when we have a many-to-one relationship. We call these semantics the Object-Relationship-Attribute (ORA) semantics.

A keyword query is inherently ambiguous as each keyword can have multiple matches. [15] introduces the notion of query patterns to depict the interpretations of a keyword query. These query patterns are generated from the Object-Relationship-Mixed (ORM) schema graph of the database. The ORM schema graph is an undirected graph that captures the ORA semantics 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). Two nodes are connected if there exists a foreign key - key reference 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.

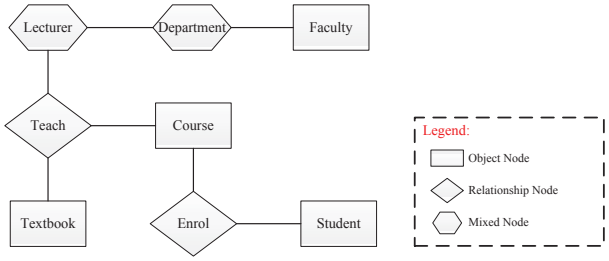


Figure 3: ORM schema graph of Figure 1

A query pattern for the keyword query {Green George Code} is shown in Figure 4. This pattern depicts the query interpretation to find the common courses taken by students Green and George. We generate this pattern by first identifying 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 *Sname* attribute. Based on these matches, we know that Green and George refer to two student objects and Code refers to a course object. Thus, we create 2 Student nodes and 1 Course node to represent these objects. From the ORM schema graph in Figure 3, we know that the Student node and the Course node are 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}

Note that the term George can refer to a lecturer object since it also matches some tuple value in the *Lecturer* relation. Hence, this keyword query has more than one query pattern. We rank these query patterns and translate the top-k ranked patterns into SQL statements.

In this work, we want to utilize these query patterns to capture the interpretations of an aggregate 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

3.1 Simple Aggregate Queries

Given a keyword query $Q = \{t_1\ t_2\ \cdots\ t_n\}$, we first classify the terms t_i into *basic terms* and *operators*. A basic term matches a relation name, or an attribute name, or a tuple value, while an operator matches GROUPBY or an aggregate function. Then we process Q as follows:

1. **Pattern generation and annotation.** We utilize the ORM schema graph of the database and the basic terms in the query to generate a set of query patterns, and annotate these patterns with the operators.
2. **Pattern disambiguation.** We disambiguate the query patterns by annotating the object/mixed nodes with GROUPBY. This is to distinguish objects with the same attribute value in the database.
3. **Pattern translation.** We translate the top-k ranked query patterns into SQL statements to compute the aggregate functions in the query.

We explain the details of these steps next.

3.1.1 Pattern Generation and Annotation

We use the basic terms in a query to generate a set of initial query patterns. Each pattern P 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.

For each operator $t_i \in Q$, we examine the matches of its subsequent term t_{i+1} in Q to annotate query pattern P. We have two cases:

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

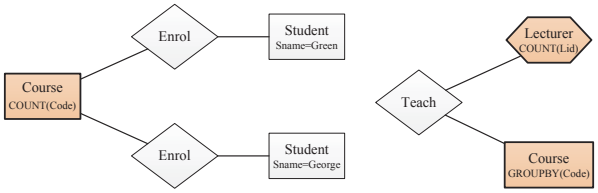
   This indicates that t_{i+1} refers to some object or relationship, and the operator t_i is applied on the identifier id of this object/relationship. We annotate the node that represents this object/relationship in P with $t_i(id)$, id is given by the primary key of the relation.

b. 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 $t_i(a)$.

EXAMPLE 1. *Consider query Q_4 = {Green George COUNT Code}. Figure 4 shows a query pattern obtained using the basic terms Green, George and Code. For the operator COUNT, its subsequent term Code matches an attribute name 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.* □

EXAMPLE 2. *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, 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.* □



(a) P_1 (b) P_2

Figure 5: Annotated query patterns of Q_4 and Q_5

3.1.2 Pattern Disambiguation

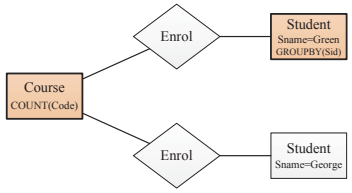
After annotating the query pattern with operators, we examine the object and mixed nodes in the pattern. An object/mixed node with the condition $a = t$ refers to an object such that its value of attribute a matches the basic term t. However, since this condition could be satisfied by more than one object in the database, there are two different interpretations of t in the context of the aggregate query:

1. Apply the aggregate function(s) for every *distinct* object satisfying the condition $a = t$.
2. Apply the aggregate function(s) for *all* the objects satisfying the condition $a = t$.

These two interpretations will lead to different results of the aggregate function(s) and we need to distinguish them in the annotated query pattern. Note that SQAK does not distinguish objects satisfying the same condition, and thus returns incorrect answers to the query.

Let P be an annotated query pattern for aggregate query Q and U be a set of object/mixed nodes in P. We generate a set of patterns S to indicate if objects with the same value will be distinguished for aggregates. Initially, S only contains the pattern P. For each node $u \in U$ that is annotated with the condition $a = t$, we check if more than one object satisfies this condition in the database. If so, we create a copy of each pattern in S to indicate if objects that satisfy the condition $a = t$ will be distinguished in these patterns. Let P_1 be a pattern in S and P_2 be a copy of P_1. We annotate u in $P2$ with GROUPBY(id), where id is the identifier of the object referred to by u. In particular, P_1 indicates that aggregate function(s) is applied for all the objects that satisfy $a = t$, while P_2 indicates that aggregate function(s) is applied for every distinct object with $a = t$.

EXAMPLE 3. *Consider the query pattern P_1 for the query Q_4 = {Green George COUNT Code} in Figure 5(a). This pattern contains three object/mixed nodes: one Course node and two Student nodes that are annotated with the conditions Sname = Green and Sname = George respectively. For the Student node imposed by the condition Sname = George, we do not need to create new copies of query patterns as there is only one student called George in Figure 1. However, for the Student node imposed by the condition Sname = Green, we know that there are two students called Green in Figure 1. Hence, we create a copy P_3 of the pattern P_1 and annotate this node with GROUPBY(Sid) in P_3. Figure 6 shows the query pattern P_3. It indicates that the aggregate function counts the number of courses enrolled by each student called Green. In contrast, pattern P_1 indicates that the count aggregate is applied for both these two students.* □



P_3

Figure 6: A query pattern for Q_4

Next, we rank the query patterns. [15] classifies nodes in a query pattern into target nodes and condition nodes. A target node specifies the search target of the query, and a condition node indicates the search conditions of the query. A query pattern is ranked based on its number of object/mixed

nodes and the average distance between the target and condition nodes. Patterns with fewer object/mixed nodes, and a shorter average distance are ranked higher.

Here, we extend the definitions of target nodes and condition nodes to rank the query patterns of an aggregate query. Let P be an annotated query pattern and u be a node in P. We say u is a target node if u is annotated with an aggregate function. Otherwise, u is a condition node if u is annotated with a condition or GROUPBY. We can now use the same method as [15] to rank the annotated query patterns.

3.1.3 Pattern Translation

Finally, we translate the top-k ranked query patterns into SQL statements. A straightforward way to translate an annotated query pattern is to join the relations of all the nodes in the pattern, select the tuples that 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 4. *Consider the query pattern P_2 for the query $Q_5 = \{$COUNT Lecturer GROUPBY Course$\}$ in Figure 5(b). If we simply translate P_2 into an SQL statement that joins the relations Teach, Lecturer and Course, and applies the count aggregate on the lecturer id Lid after grouping the tuples by the course 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 Bid 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 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 the query pattern contains a relationship node u, we should look at its corresponding node v in the ORM schema graph to determine if a projection is needed to remove duplicates. Note that SQAK does not detect the duplications of objects in relationships, and suffers from the problem of returning incorrect answers.

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

SELECT clause. If a node $u \in P$ is annotated with $t(a)$ and t matches an aggregate function, we include t in the SELECT clause. t is applied on attribute a. If u is annotated with $GROUPBY(a)$, we include a in the SELECT clause to facilitate user understanding of the aggregate function(s).

FROM clause. This clause includes the relations of all the nodes in P. For each relationship node $u \in P$, we check its corresponding node v in the ORM schema graph. Let $N_u = \{u_1, u_2, \cdots, u_x\}$ be a set of object/mixed nodes that are directly connected to u in P, and $N_v = \{v_1, v_2, \cdots, v_y\}$ be the set of object/mixed nodes that are directly connected to v in the ORM schema graph. If $x < y$, then this indicates that P contains a subset of the participating objects of the relationship v, and 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 N_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 references. For each node $u \in P$ that is annotated with a condition $a = t$, we include the condition "$R_u.a$ contains t" where R_u is the relation corresponding to u.

GROUPBY clause. If a node u is annotated with $t(a)$ and t matches GROUPBY, then we include the attribute a in the GROUPBY clause.

EXAMPLE 5. *The query pattern P_3 in Figure 6 for query $Q_4 = \{$Green George COUNT Code$\}$ depicts the total number of courses enrolled by the student George and each of the students called Green. The Course node is annotated with COUNT(Code), thus we include $COUNT(Code)$ in the SELECT clause. The FROM clause contains the relations corresponding to each of the nodes in P_3. Next, 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 the Student node imposed by the condition Sname = Green is also annotated with GROUPBY to distinguish different students called Green, we include the id of its relation in the GROUPBY clause, and obtain the SQL statement:*

```
SELECT S1.Sid, 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 S1.Sid=E1.Sid AND S1.Sname contains 'Green'
    AND S2.Sid=E2.Sid AND S2.Sname contains 'George'
GROUP BY S1.Sid
```

By applying GROUPBY on student ids, we distinguish students $s2$ and $s3$ who have the same name Green, and the aggregate function COUNT is computed for the courses of each student. □

EXAMPLE 6. *The query pattern P_2 in Figure 5(b) for query $Q_5 = \{$COUNT Lecturer GROUPBY Course$\}$ depicts the number of lecturers for each 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. 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. The subquery has a DISTINCT keyword and thus eliminates any duplicates of $\langle Lid, Code\rangle$ for different Bid. We use the result of this subquery to join the other relations in the FROM clause. The SQL statement generated is:*

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

3.2 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.

EXAMPLE 7. *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 7 shows the annotated query pattern. We translate the query pattern by first generating an inner SQL query similar to that in Example 6. Then we put it in the FROM clause of the outer SQL query to compute the aggregate function AVG as follows:*

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

□

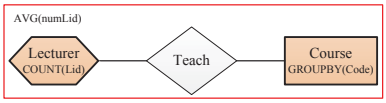


Figure 7: Query pattern in Example 7

4. AGGREGATE QUERIES ON UNNORMALIZED DATABASE

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

Recall that in Figure 2, the *Lecturer* relation is denormalized by adding a foreign key *Fid* 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 $Q_3 = \{$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 unnormalized relations.

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

Consider the unnormalized relation *Enrolment* in Figure 8 that is obtained by joining the *Student*, *Enrol* and *Course* relations in Figure 1. The following functional dependencies hold on the *Enrolment* relation:

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

We deduce that $\{Sid, Code\}$ is the key of the *Enrolment* relation, and it violates the second normal form (2NF) definition as *Sname* and *Age* only depend on *Sid*.

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 8: An unnormalized relation

A naive approach to handle a keyword query involving aggregate functions on the unnormalized 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 unnormalized, 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 an unnormalized database, then we can continue to process keyword queries involving aggregates and GROUPBY 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 unnormalized 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 ORA semantics in the unnormalized database.

Let $D = \{R_1, R_2, \cdots, R_j\}$ be the set of relations in the original unnormalized 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 8. *Let us generate the normalized view of the unnormalized database in Figure 8. The database consists of a single relation Enrolment and has the schema D below:*

$$Enrolment(\underline{Sid, Code}, Sname, Age, Title, Credit, Grade)$$

Since the Enrolment relation is not in 3NF, we decompose it into 3NF relations Student′, Enrol′ and Course′. Thus, the normalized view D′ will have the relations:

$$Student'(\underline{Sid}, Sname, Age)$$
$$Enrol'(\underline{Sid, Code}, Grade)$$
$$Course'(\underline{Code}, Title, Credit)$$

Based on D′, we construct the ORM schema graph of the unnormalized database as shown in Figure 9. Table 1 shows the mappings of the relations in D and D′. □

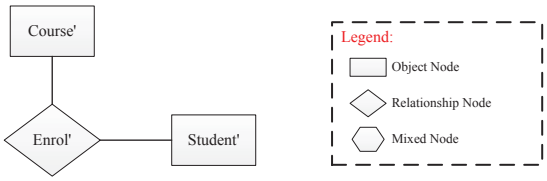


Figure 9: ORM schema graph of Figure 8

Table 1: Mappings of relations in Example 8

$Student' = \Pi_{Sid,Sname,Age}(Enrolment)$
$Enrol' = \Pi_{Sid,Code,Grade}(Enrolment)$
$Course' = \Pi_{Code,Title,Credit}(Enrolment)$

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

$Enrolment = Student' \bowtie Enrol' \bowtie Course'$

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

After obtaining the normalized view D' and the ORM schema graph G of the unnormalized database with schema D, we can proceed to evaluate an aggregate query Q on D correctly as follows:

First, we identify the matches of each basic term in the unnormalized 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'.

Next, we utilize the relations in D' to generate the query patterns based on G, and annotate these patterns with the operators in the query as described in Section 3. Note that the generated query patterns are based on the normalized view D', since G is constructed from D'.

Finally, we translate the annotated query patterns into SQL statements to be executed over the original unnormalized database. This requires us to map the relations in D' 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 SQL.

EXAMPLE 9. *Consider query Q_4 = {Green George COUNT Code} on the unnormalized database in Figure 8. The terms Green and George match the Sname attribute values of some tuples in the Enrolment relation, while the term Code matches the name of an attribute in Enrolment.*

Based on Table 1(a), these matches correspond to 2 Student′ relations and 1 Course′ relation in the normalized view of the database respectively, indicating that Green and George refer to two student objects, while Code refers to a course object. Based on the ORM schema graph in Figure 9, we generate a query pattern that connects 2 Student′ nodes and 1 Course′ node via 2 Enrol′ nodes, and annotate it with operators COUNT and GROUPBY.

Figure 10 shows the query pattern obtained. It depicts the query interpretation to find the total number of courses taken by the student George and each student called Green. We use the mappings in Table 1(b) to translate the query pattern and obtain an SQL statement with 5 subqueries, namely, 2 subqueries for Student′ relation, 2 subqueries for Enrol′ relation, and 1 subquery for Course′ relation in the normalized view of the database.

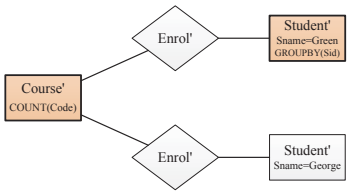


Figure 10: Query pattern in Example 9

```
SELECT S1'.Sid, 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 C'.Code=E2'.Code
    AND S1'.Sid=E1'.Sid AND S1'.Sname contains 'Green'
    AND S2'.Sid=E2'.Sid AND S2'.Sname contains 'George'
GROUP BY S1'.Sid
```
□

4.1 Query Rewriting

The generated SQL statement may contain a lot of subqueries since the mapping from a relation $R' \in D'$ to a relation $R \in D$ for a node in P often involves a subset of the attributes of R. Joining relations obtained from subqueries is time consuming due to the lack of indexes. Hence, it is crucial to rewrite the SQL to improve query performance.

We observe that some attributes in the SELECT clause of subqueries are never used, and can be removed. In Example 9, we can rewrite the subquery "SELECT DISTINCT Code, Title, Credit FROM Enrolment" to "SELECT DISTINCT Code FROM Enrolment", since Title and Credit are not used.

Further, some select conditions in the SQL statement can be moved to the WHERE clause of subqueries so that tuples can be filtered out before the join, 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 unnormalized to reduce the number of joins. We can try to use the unnormalized relation to replace the joining of relations obtained from subqueries. 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.

Based on the above observations, We derive the following heuristics to rewrite an SQL statement *sql* for the unnormalized database:

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 10. *Consider the SQL statement in Example 9. 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 replace* $C' \bowtie E1' \bowtie S1'$ *by the Enrolment relation. Further, 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 replace* $E2' \bowtie S2'$ *by the Enrolment relation, and obtain:*

```
SELECT R1.Sid, 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
```
□

5. ALGORITHMS

Algorithm 1 generates a normalized view D' of the database schema D, if D is not normalized. For each relation R in D, if R is in 3NF, we add it into D' directly, otherwise we normalize R into a set of 3NF relations and add them into D'. After checking the relations in D, we enumerate each pair of relations R'_1 and R'_2 in D'. If R'_1 and R'_2 have the same key, then we merge them into a single relation R'.

After generating the database schema D and the normalized view D', we obtain the mappings between D and D'. We call Algorithm 2 to process a keyword query Q and generate SQL statements. If the schema D is normalized, we construct the ORM schema graph G based on D. For each basic term t in Q, we find its matches in the database and create a tag for each of the matches to capture the interpretation of t. We insert these tags into a tag set *taglist* (Lines 4-7). Based on *taglist* and G, we generate a list of query patterns *ptnlist* as described in [15], and annotate these patterns with the operators in Q (Lines 8-9). Then we translate each pattern P in *ptnlist* into an SQL statement *sql* according to D, and insert it into *sqllist* (Lines 10-12).

If the schema D is unnormalized, we construct the ORM schema graph G based on D'. For each basic term t, we create tags for t based on the matches in D and the mappings in D'. We generate a list of query patterns *ptnlist* based on the tags and the ORM schema graph, and annotate each pattern P in *ptnlist* with the operators. Then we translate each pattern P into an SQL statement *sql* based on D', and map the relations of D' back to the relations of D in *sql*. Finally, we rewrite *sql* to *sql'* to reduce the number of subqueries and insert *sql'* into *sqllist* (Lines 14-26).

Algorithm 3 annotates the query patterns. For each pattern P in *ptnlist*, we annotate P with operators. For each operator t in Q, let t' be the next term of t in Q. If t' is a basic term, we check its matches in D. Let u be a node in P and R be the relation of u. If t' matches the name of R, then we annotate u with $t(R.key)$; otherwise if t' matches an attribute a of R, we annotate u with $t(R.a)$. If t' is also an operator, then we annotate pattern P with $t(t')$ to indicate that t is a nested aggregate function (Lines 3-12). Next, we check the annotated nodes in the patterns. For each pattern P in *ptnlist*, we create a set S and add P into S. For each object/mixed node u in P, if u is annotated with the condition $a = t$, we find a set of tuples T that satisfy this condition in the relation of u. If T contains more than one tuple, we generate new copies of patterns in S to distinguish the objects that satisfy the same condition. For each pattern P in S, we create a copy P' of P, annotate node u in P' with GROUPBY($R.key$), and add P' into S. Finally, we add the patterns in S into the list *aptnlist* (Lines 13-24).

Algorithm 1: NormalizeDB

```
   Input: database schema D
   Output: normalized view D'
1  D' ← ∅;
2  foreach relation R in D do
3      if R is in 3NF then
4          Add R into D';
5      else
6          Normalize R into a set of 3NF relations F;
7          foreach relation R' in F do
8              Add R' into D';
9  foreach pair of relations R'_1 and R'_2 in D' do
10     if R'_1.key = R'_2.key then
11         Merge R'_1 and R'_2 into R';
12 return D';
```

Algorithm 2: Keyword Search

```
   Input: Query Q, database schema D, normalized view D'
   Output: a list of SQL statements sqllist
1  sqllist ← ∅; ptnlist ← ∅; taglist ← ∅;
2  if D is normalized then
3      G = createORMGraph(D);
4      foreach basic term t in Q do
5          matches = findMatch(t, D);
6          tagset = createTag(matches, G);
7          Insert tagset into taglist;
8      ptnlist = createPattern(taglist, G);
9      ptnlist = annotatePattern(Q, ptnlist);
10     foreach Pattern P in ptnlist do
11         sql = translate(P, D);
12         Insert sql into sqllist;
13 else
14     G = createORMGraph(D');
15     foreach basic term t in Q do
16         matches = findMatch(t, D);
17         Map matches of D into matches' of D';
18         tagset = createTag(matches', G);
19         Insert tagset into taglist;
20     ptnlist = createPattern(taglist, G);
21     ptnlist = annotatePattern(Q, ptnlist);
22     foreach Pattern P in ptnlist do
23         sql = translate(P, D');
24         Map the relations of D' to the relations of D in sql;
25         sql' = rewrite(sql);
26         Insert sql' into sqllist;
27 return sqllist;
```

6. PERFORMANCE STUDY

In this section, we evaluate the performance of our approach to process keyword queries involving aggregates and GROUPBY. We implement the algorithms in Java and carry out experiments on a 3.40 GHz CPU with 8 GB RAM. We use the relational databases TPC-H (TPCH) and ACM Digital Library publication (ACMDL). Table 2 shows the schemas of these databases. Tables 3 and 4 show the queries we constructed for each database and the corresponding descriptions (or search intentions).

Algorithm 3: Annotate Pattern

Input: query Q and a list of patterns *ptnlist*
Output: a list of annotated patterns *aptnlist*

```
aptnlistist ← ∅;
foreach Pattern P in ptnlist do
    foreach operator t in Q do
        Let t′ be the next term of t in Q;
        if t′ is a basic term then
            Let u be a node in P and R be the relation of u;
            if t′ matches the name of R then
                Annotate u with t(R.key);
            else if t′ matches an attribute a in R then
                Annotate u with t(R.a);
        else if t′ is an operator then
            Annotate P with t(t′);
foreach Pattern P in ptnlist do
    S = {P};
    foreach Object/Mixed node u in P do
        if u is annotated with condition a = t then
            Let R be the relation of u and T be the tuples
            satisfying a = t in R;
            if |T| > 1 then
                foreach Pattern P in S do
                    Create a copy P′ of P;
                    Annotate u in P′ with
                    GROUPBY(R.key);
                    Add P′ into S;
    Add the patterns in S into aptnlistist;
return aptnlistist;
```

6.1 Effectiveness Experiments

Our approach utilizes the ORM schema graph to capture the ORA semantics in the database, and generates a list of annotated query patterns from the ORM schema graph to represent the various interpretations of a keyword query. Based on these patterns, we distinguish the objects with the same value and detect duplicate objects in relationships in order to compute the answers correctly.

We compare our approach with SQAK [13], the state-of-the-art relational keyword search engine that processes aggregate queries without considering the ORA semantics.

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

We use the generated SQL statements that best match the query descriptions in Table 3 to compute the query answers. Table 5 shows the results returned by SQAK and our approach, as well as explanations for these answers. Although both SQAK and our approach 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". Note that our approach can also generate query patterns to compute aggregates without distinguishing objects with the same value.

Table 2: Database schemas

TPCH
Part(partkey, pname, type, size, retailprice)
Supplier(suppkey, sname, nationkey, acctbal)
Lineitem(partkey, suppkey, orderkey, quantity)
Order(orderkey, custkey, amount, 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(paperid, authorid)
Edit(editorid, procid)

Queries $T5$ and $T6$ show that by examining the relationships and their participating objects, our approach is able to detect duplicate objects in relationships and 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 orders and returns 22, a value that is way above the actual number. 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 6 shows the answers and explanations 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 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 the most recent date of papers respectively.

Similarly, for query $A5$, our approach returns 6 answers while SQAK only returns 4 answers, as it mixes some papers with the same title.

Query $A6$ 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.

Table 3: Queries for TPCH database

#	Query	Description
$T1$	order AVG amount	Find the average amount of orders
$T2$	MAX COUNT order GROUPBY nation	Find the maximum number of orders among nations
$T3$	COUNT order "royal olive"	Find the number of orders that contains the "royal olive"
$T4$	supplier MAX acctbal "yellow tomato"	Find the maximum balance of suppliers that supply the "yellow tomato"
$T5$	COUNT supplier "Indian black chocolate"	Find the number of suppliers for "Indian black chocolate"
$T6$	COUNT part GROUPBY supplier	Find the number of parts supplied by each supplier
$T7$	COUNT order SUM amount GROUPBY mktsegment	Find the number of orders and their total amount for each market segment
$T8$	COUNT supplier "pink rose" "white rose"	Find the number of suppliers for "pink rose" and "white rose"

Table 4: Queries for ACMDL database

#	Query	Description
$A1$	proceeding AVG pages	Find the average pages of proceedings
$A2$	COUNT paper GROUPBY proceeding SIGMOD	Find the number of papers in each 'SIGMOD' proceeding
$A3$	COUNT proceeding editor Smith	Find the number of proceedings edited by 'Smith'
$A4$	paper MAX date Gill	Find the date of the latest papers written by 'Gill'
$A5$	COUNT author "database tuning"	Find the number of authors for each "database tuning" paper
$A6$	COUNT paper MAX date IEEE	Find the number of papers published by 'IEEE' and most recent date
$A7$	COUNT paper author John Mary	Find the number of papers co-authored by 'John' and 'Mary'
$A8$	COUNT editor SIGIR CIKM	Find the number of editors that edit proceedings 'SIGIR' and 'CIKM'

6.1.3 Queries on Unnormalized Databases

Next, we denormalize the ACMDL and TPCH databases, and obtain the unnormalized database schemas in Table 7. We use the queries in Tables 3 and 4 on the unnormalized databases and compare the results returned by SQAK and our approach.

Tables 8 and 9 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 unnormalized relation *Ordering*. Similarly, SQAK returns the answer 637 for $A1$, and 2000, 408, 14858, etc. (totally 36 answers) for $A2$, both of which are incorrect as the information of proceedings and papers are duplicated in the unnormalized relations *EditorProceeding* and *PaperAuthor*. Note that these queries are answered correctly by SQAK when the database is normalized.

For queries $T3$ to $T6$ and queries $A3$ to $A5$, SQAK returns the incorrect answers for the same reason as discussed in Section 6.1.1 and Section 6.1.2.

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

6.2 Efficiency Experiments

Finally, we compare the time taken by our approach and SQAK to generate SQL statements. Figure 11 shows the results for TPCH and ACMDL queries in Tables 3 and 4.

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. It also does not distinguish objects with the same attribute value or detect the duplicate objects in relationships. Besides, it does not consider the duplications arising from unnormalized relations.

Take query $A7$ for example. We first parse this query into basic terms (paper, author, John, Mary) and operators (COUNT). Then, we generate a query pattern with one Paper node, two Write nodes and two Author nodes. We annotate the Paper node with the Count operator, and distinguish the authors called John and the authors called Mary respectively. Finally, we detect if information of paper and author objects are duplicated in write relationships, and translate the patterns into SQL statements. In contrast, SQAK does not handle the query because both the terms John and Mary match the values of some tuples in the *Author* relation.

As the SQL execution time dominates the overall processing time (in seconds), we see that 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.

Table 7: Unnormalized database schemas

TPCH'
Ordering(partkey, suppkey, orderkey, pname, type, size, retailprice, sname, nationkey, regionkey, acctbal, custkey, amount, date, priority, quantity)
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)

7. RELATED WORK

Existing works on keyword search in relational databases can be classified into data graph approach and schema graph approach. In 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] uses bidirectional

Table 5: Answers of queries for normalized TPCH database

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

Table 6: Answers of queries for normalized ACMDL database

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

Table 8: Query answers on unnormalized TPCH (Our approach returns the same answer as Table 5)

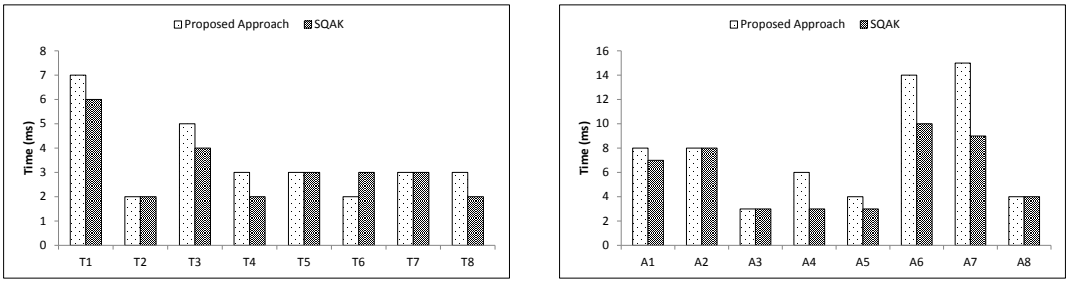
#	SQAK	Explanation
$T1$	AVG amount: 1.78×10^5	incorrect answer: count duplicate orders
$T2$	MAX COUNT order: 26485	incorrect answer: count duplicate orders
$T3$	1 answer: 229	incorrect answers: The same reason as Table 5
$T4$	1 answer: 9844.00	
$T5$	COUNT supplier: 22	
$T6$	1000 answers: 593, 571, ...	
$T7$	N.A.	
$T8$	N.A.	

Table 9: Query answers on unnormalized ACMDL (Our approach returns the same answer as Table 6)

#	SQAK	Explanation
$A1$	AVG ages: 637	incorrect answers: count duplicate proceedings
$A2$	36 answers: 2000, 408, 14858, ...	incorrect answers: count duplicate papers
$A3$	1 answer: 62	incorrect answers: The same reason as Table 6
$A4$	1 answer: 2011-06-13	
$A5$	4 answers: 2, 4, 6, 4	
$A6$	N.A.	
$A7$	N.A.	
$A8$	N.A.	

expansion to reduce the search space. [4] employs dynamic programming to identify the top-k minimal group Steiner trees. [12] finds a subgraph that contains all the keywords within a given distance to be a query answer, and captures more information than a Steiner tree.

In schema graph approach, the database schema is modeled as a graph where each node represents a relation and each edge 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) TPCH (b) ACMDL

Figure 11: Comparison of the time taken by our approach and SQAK to generate SQL statements

a minimal number of relations and outputs tuples that contain all the keywords. [6] and [10] 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 examine tuples that contain query keywords and try to link them by foreign key-key references. [17] 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. [11] classifies query keywords into dimensional and general keywords, and computes subgraphs that contain all dimensional keywords and some general keywords. These subgraphs are grouped based on dimensional keywords to compute the statistical information of the subgraphs. However, none of these works can answer queries involving aggregate functions and GROUPBY.

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

8. CONCLUSION

In this paper, we have studied the problem of answering keyword queries involving aggregates and GROUPBY on relational databases. Existing work does not consider the ORA semantics, and thus fails to distinguish objects with the same attribute value and detect duplications of objects in relationships. This leads to incorrect computation of aggregate queries. To avoid these problems, we utilize the ORM schema graph to capture the ORA semantics, and propose a semantic approach to answer aggregate queries. Given an aggregate query, we generate a set of annotated query patterns to represent various interpretations of the query. Based on these patterns, we distinguish objects with the same attribute value and detect duplications of objects in relationships. The top-k ranked patterns are translated into SQL statements which apply aggregate functions to compute the statistical information correctly. Further, we develop a mechanism to detect duplications arising from unnormalized relations, and extend our approach to handle aggregate queries on unnormalized databases. Experimental results demonstrate the our approach returns correct answers to aggregate queries both on normalized and unnormalized 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] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: top-k keyword query in relational databases. In *SIGMOD*, 2007.

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

[12] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, 2009.

[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] Z. Zeng, Z. Bao, M. L. Lee, and T. W. Ling. A semantic approach to keyword search over relational databases. In *ER*, 2013.

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