# Efficient Algorithms based on Relational Queries to Mine Frequent Graphs

Walter Garcia
UH-Downtown
Dept. of Computer Science
Houston, TX 77002, USA

Carlos Ordonez
University of Houston
Dept. of Computer Science
Houston, TX 77204, USA

Kai Zhao
University of Houston
Dept. of Computer Science
Houston, TX 77204, USA

Ping Chen
UH-Downtown
Dept. of Computer Science
Houston, TX 77002, USA

## ABSTRACT

Frequent subgraph mining is an important problem in data mining with wide application in science. For instance, graphs can be used to represent structural relationships in problems related to network topology, chemical compound, protein structures, and so on. Searching for patterns from graph databases is difficult since graph-related operations generally have higher time complexity than equivalent operations on frequent itemsets. From a practical standpoint, databases keep growing with lots of opportunities and need to mine graphs. Even though there is a significant body of work on graph mining, most techniques work outside the database system. Programming frequent graph mining in SQL is more difficult than traditional approaches because the graph must be represented as a table and algorithmic steps must be written as relational queries. In our research, we study three fundamental problems under a database approach: graph storage and indexing, frequent subgraph search, and identifying subgraph isomorphism. We outline main research issues and our solution towards solving them. We also present preliminary experimental validation focusing on query optimizations and time complexity.

## Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

## General Terms

Algorithms, Experimentation, Performance

## Keywords

DBMS, SQL, Graph Mining

## 1. INTRODUCTION

Frequent subgraph mining is an active research topic in the data mining community. The earliest studies to find subgraph patterns characterized by some measures from massive graph data were SUBDUE [5] and GBI [15]. SUBDUE uses a computationally bounded inexact graph match that identifies similar, but not identical, instances of a substructure and finds an approximate measure of closeness of two substructures when under computational constraints. In addition to the minimum description length principle, other background knowledge can be used by SUBDUE to guide the search toward more appropriate substructures. GBI is capable of solving a variety of learning problems by mapping the different learning problems into colored digraphs. The generality and scope of this method can be attributed to the expressiveness of the colored digraph representation, which allows a number of different learning problems to be solved by a single algorithm. In 1998, WARMR [6], a general purpose inductive logic programming algorithm that addresses frequent query discovery was proposed by Dehaspe and Toivonen. This algorithm offers the flexibility required to experiment with standard and novel settings not supported by special purpose algorithms.

Recent subgraph mining algorithms can be roughly classified into two categories: depth-first search and level-wise search. Algorithms in the first category use a depth-first search for finding candidate frequent subgraphs. One example algorithm in this category is G-SPAN(graph-based Substructure pattern mining) [14] which is the first algorithm that explores depth-first search in frequent subgraph mining. This algorithm builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. G-SPAN discovers all the frequent subgraphs without candidate generation and false positives pruning. It combines the growing and checking of frequent subgraphs into one procedure, thus accelerating the mining process. Recently, a new mining framework, LEAP [13], was developed to exploit the correlation between structural similarity and significance similarity in a way that the most significant pattern could be identified quickly by searching dissimilar graph patterns. This mining method revealed that the widely adopted branch-and-bound search in data mining literature is indeed not the best.

In the second category the algorithm uses a level-wise search scheme like Apriori to enumerate the recurring subgraphs. Inokuchi et al. [8] proposed a computationally efficient algorithm, AGM, that can be used to find all frequent induced subgraphs in a graph database. This algorithm finds all frequent induced subgraphs using an Apriori-based approach similar to [1] which extends subgraphs by adding one vertex at each step. FSG [9] is another computationally efficient algorithm proposed by Kuramochi et al. This algorithm, on the other hand, finds all frequent connected subgraphs in a graph database. FSG uses a sparse graph representation which minimizes both storage and computation and it increases the size of frequent subgraphs by adding one edge at each step. In [3] the authors proposed the DB-SUBDUE algorithm which is the first attempt to mine substructures over graphs using the database approach. This algorithm uses standard SQL and stored procedures inside a relational DBMS and overcomes the performance and scalability problems that is inherent to main memory algorithms.

In this paper, we propose an efficient algorithm to generate frequent subgraphs inside a relational DBMS. We implement all the graph mining steps with SQL. The representation of graphs using indices could be applied to most graphs. We also implement the canonical ordering technique to identify similar subgraphs with different order of vertex and edge labels. Our algorithm achieves good performance and is scalable to large graph datasets.

Here is an outline of the structure of this paper:

1. Introduction
2. Definitions
    - General Definitions
    - Subgraph Isomorphism
    - Frequent Subgraphs
3. Efficient Algorithms in SQL to Mine Graph Databases
    - Graph Storage and Indexing
    - Frequent Subgraph Search
    - Subgraph Isomorphism
    - Algorithm
4. Preliminary Experiments
5. Related Work
6. Conclusion and Future Work

## 2. DEFINITIONS

### 2.1 General Definitions

*Definition 1.* A labeled graph G, as shown in Figure 1, is a five element tuple $G = \{V, E, \Sigma_V, \Sigma_E, l\}$ where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. $\Sigma_V$ and $\Sigma_E$ are the sets of vertices and edge labels respectively. The labeling function $l$ defines the mapping $V \rightarrow \Sigma_V$ and $E \rightarrow \Sigma_E$.

*Definition 2.* Graph dataset GD is a set of labeled graphs, GD=$\{G_1, G_2, \ldots, G_n\}$, and $|M|$ denotes the number of graphs in a graph dataset, that is the size of the graph dataset. $|E|$ denotes the number of edges of the graph dataset.

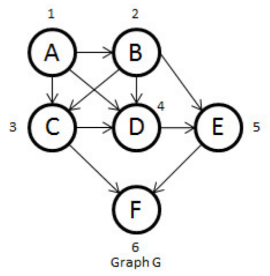


**Figure 1: Example of a graph.**

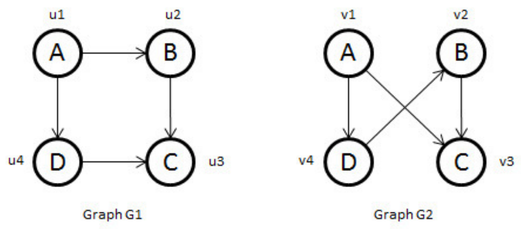


**Figure 2: Example of graph isomorphism.**

*Definition 3.* The size of a graph G is the number of edges in E(G).

*Definition 4.* A subgraph of a graph $G1 = \{V, E\}$ is a graph $G2 = \{W, F\}$, where $W \subseteq V$ and $F \subseteq E$. A subgraph $G2$ of $G1$ is a proper subgraph of $G1$ if $G2 \neq G1$.

Given two labeled graphs $G1 = \{V, E, \Sigma_V, \Sigma_E, l\}$ and $G2 = \{V^{'}, E^{'}, \Sigma_V^{'}, \Sigma_E^{'}, l^{'}\}$, $G2$ is a subgraph of $G1$ iff

1. $V^{'} \subseteq V$
2. $\forall u \in V^{'}, (l(u) = l^{'}(u))$
3. $E^{'} \in E$
4. $\forall (u, v) \in E^{'}, (l(u, v) = l^{'}(u, v))$

### 2.2 Subgraph Isomorphism

*Definition 5.* A labeled graph $G1 = \{V, E, \Sigma_V, \Sigma_E, l\}$, as shown in Figure 2, is isomorphic to graph $G2 = \{V^{'}, E^{'}, \Sigma_V^{'}, \Sigma_E^{'}, l^{'}\}$, iff there exists a bijection $f : V \rightarrow V^{'}$ such that

1. $\forall u \in V, (l(u) = l^{'}(f(u)))$
2. $\forall u, v \in V, (u, v) \in E \Leftrightarrow (f(u), f(v) \in E^{'})$
3. $\forall (u, v) \in E, (l(u, v) = l^{'}(f(u), f(v)))$

*Definition 6.* A labeled graph $G2$ is subgraph isomorphic to labeled graph $G1$, denoted by $G2 \subseteq G1$, iff there exists a subgraph $G2^{'}$ of $G1$ such that $G2$ is isomorphic to $G2^{'}$.

*Definition 7.* Given a set of graphs GD(graph database) and the support of a graph $G1$, denoted by $sup_{G1}$ is defined as the fraction of graphs in GD to which $G1$ is subgraph isomorphic.

$sup_{G1} = \frac{|\{G2 \in GD | G1 \subseteq G2\}|}{|M|}$

## 2.3 Frequent Subgraphs

*Definition 8.* G is frequent iff $sup_G \geq \sigma$. The frequent subgraph mining problem is given a threshold $\sigma$ where $0 \leq \sigma \leq 1$ and a graph database $GD$, to find all frequent subgraphs in $GD$.

An example of labeled graphs and subgraph isomorphism is presented in Figure 3. All three graphs in Figure 3 are labeled graphs, and graph 1 is subgraph isomorphic to graph 2.

## 3. EFFICIENT ALGORITHMS IN SQL TO MINE GRAPH DATABASES

We have identified the following problems as being fundamental in graph mining. Solving them with SQL [4] is likely to have a broad impact. (1) Graph storage and indexing: since performing a sequential scan on the whole graph dataset is inefficient, it is necessary to build a graph index in order to help with the processing of the graph queries. (2) Frequent subgraph search: the possible number of subgraphs increases with the size of graph dataset, so it is important to find an efficient candidate generation approach which is scalable to large datasets. (3) Subgraph isomorphism: subgraph isomorphism, which has been proven to be NP-complete [7], is used to determine whether a given subgraph occurs in the input graph or not.

In this section we will describe the technical details of how our algorithm overcomes the three problems mentioned above inside a relational DBMS.

## 3.1 Graph Storage and Indexing

In order to accommodate mining over a set of graphs we extend the graph representation of DB-SUBDUE [3, 2]. Since data is stored in relational DBMS as tables we need to present graphs as rows in a table. Normally a graph is a set of edges and vertices, so we represent graphs using two tables: one is $Vertex(V\_No, VL, GID)$ and another is $Edge(E\_No, V1, V2, EL, GID)$. Figure 3 gives an example of a graph dataset. The vertices of the graphs are stored in the $Vertex$ table as shown in table 1 and the edges are stored in the $Edge$ table as shown in table 2.

The $Vertex$ table contains three attributes: $V\_No$ is the number of the vertex, $VL$ is the label of vertex and $GID$ is the id of the graph that the vertex belongs to. The $Edge$ table contains five attributes: $E\_No$ is the number of the edge, $V1$ is the number of the vertex where the edge originates, $V2$ is the number of the vertex where the edge terminates, $EL$ is the label of the edge and $GID$ is the id of graphs that the edge belongs to. The GID(Graph Id) attribute helps to identify the vertices and edges belonging to the same graph. In other words, each graph is given a unique identification number that is also stored as part of the representation. The value of the $E\_No$(number of edge) attribute is unique to an edge. By imposing the constraint that the new edge number should not be equivalent to any edge that is already

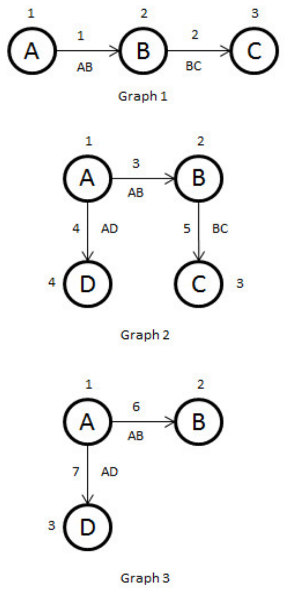


**Figure 3: Example of graph dataset.**

| V_No | VL | GID |
|---|---|---|
| 1 | A | 1 |
| 2 | B | 1 |
| 3 | C | 1 |
| 1 | A | 2 |
| 2 | B | 2 |
| 3 | C | 2 |
| 4 | D | 2 |
| 1 | A | 3 |
| 2 | B | 3 |
| 3 | D | 3 |

**Table 1: Vertex: Structure of Vertex table.**

| E_No | V1 | V2 | EL | GID |
|---|---|---|---|---|
| 1 | 1 | 2 | AB | 1 |
| 2 | 2 | 3 | BC | 1 |
| 3 | 1 | 2 | AB | 2 |
| 4 | 1 | 4 | AD | 2 |
| 5 | 2 | 3 | BC | 2 |
| 6 | 1 | 2 | AB | 3 |
| 7 | 1 | 3 | AD | 3 |

**Table 2: Edge: Structure of Edge table.**

present in the instance, we avoid the expansion of subgraphs on edges that are already present in the instance.

The *Edge* table cannot represent one-edge subgraphs since this table does not contain information about vertex labels. Therefore, we create a new table called *Subgraph_1* by joining the $Vertex$ table and the *Edge* table. The *Subgraph_1* table will contain all the subgraphs of size one as shown in table 6. For one-edge subgraphs, the edge direction is always from the first to the second vertex. This is why there are no attributes in the *Subgraph_1* table that specify the direction. For higher edge subgraphs, we introduce the connectivity attributes to denote the direction of edges between vertices. Since the edges having less frequency than the minimum support will not expand to subgraphs that satisfy the minimum support, we remove those edges from the *Subgraph_1* table.

Since the base table *Subgraph_1* is created by a join operation on attribute $V\_No$ of the $Vertex$ table and the $V1$, $V2$ attributes of the *Edge* table, so we create index($V\_No$) for $Vertex$ table and index($V1$, $V2$) for the *Edge* table. The *Subgraph_1* table is the base table for the frequent subgraph search process and we create index($E\_No$, $VL1$, $VL2$) for this base table. With the index, the speed of the join operation on these tables will be improved significantly.

## 3.2 Frequent Subgraph Search

In order to count the isomorphic subgraphs across graphs, we need to systematically generate subgraphs of increasing size in all of the input graphs and count them. In our algorithm, the subgraphs are allowed to expand on any matching vertex(belong to the same graph) along the edges that do not exist in the subgraphs that are being expanded. This unconstrained expansion can generate all the possible subgraphs. For example, to expand a one-edge subgraph into a two-edge subgraph we join the *Subgraph_1* table with itself on matching vertices. To make sure that the expansion is done within the same graph we use the constraint that of GID of both one-edge subgraphs should be the same. In general, subgraphs of size i are generated by joining the *Subgraph_*$(i-1)$ table with the base table *Subgraph_1*. Since the join operation is performed on the vertex label and edge label attributes, we also need to set vertex label attributes $VL1$,...,$VL(i+1)$ and edge label attributes $EL1$,...,$ELi$ to be the primary key of the *Subgraph_i* table.

The edge label attribute does not give any information about the vertex it is originating or terminating to. The directions and the vertices associated with the edge can be obtained from the connectivity attributes $Fi$ and $Ti$, which tell us that the edge i originates from $Fi$ and terminates at $Ti$.

A graph may have many subgraphs of the same substructure. If we count the frequency of the subgraph from the candidate table, it will be counted many times. Hence, in order to obtain the correct frequency of a subgraph in the graph dataset, we need to include one instance per subgraph within one graph. For this purpose, we project distinct vertex labels, edge labels, connectivity map, and GID and store them in the *Distinct_i* table. Then we perform a GROUP BY operation on the vertex labels, edge labels, and connectivity map in *Distinct_i* to obtain the correct frequency of each subgraph. Since the subgraphs with less frequency than provided support value will not contribute to future generation, we prune the *Subgraph_i* table by removing those subgraphs.

| ID | V | VL | Pos |
|---|---|---|---|
| 2 | 1 | A | 1 |
| 2 | 4 | D | 2 |
| 2 | 3 | C | 3 |
| 3 | 3 | C | 1 |
| 3 | 4 | D | 2 |
| 3 | 1 | A | 3 |

**Table 4: Unsorted_Vertex.**

## 3.3 Subgraph Isomorphism

To obtain the frequency of a subgraph $G2$, we should count the number of graphs to which the given subgraph $G2$ is subgraph isomorphic. According to the definition, if $G2$ is subgraph isomorphic to $G1$, there is a subgraph of $G1$ which is isomorphic to $G2$. So in order to obtain the frequency of a subgraph after the candidate generation step, we just need to count the number of graphs that contain this subgraph. During the frequent subgraph search step, the expansion of subgraphs may cause similar subgraphs to grow in different ways. This will make the frequency counting incorrect. Our algorithm introduced the canonical ordering technique to identify the similar subgraphs.

Table 3 represents an example of some similar subgraphs of size 2. Here we introduce an additional attribute called $ID$ in unordered table 3. Each row in the table should have a unique ID. In order to obtain the correct frequency of the subgraphs, we need to identify similar subgraphs regardless of their growing order. This problem is addressed by a canonical ordering step. Canonical ordering is an expensive process that involves maintaining six intermediate tables, sorting of two intermediate tables, one 3-way join, and one 2n+2 way join where n is the size of subgraph.

In order to identify two similar subgraphs, vertex labels and the connectivity attributes, which denote the direction of the edges, need to be used. If two subgraphs have the same vertex labels and edge directions, then they can be identified as similar subgraphs. Since a DBMS does not allow rearrangement of columns to obtain canonical ordering, we have to transpose the rows of each subgraph into columns, sort them, and then reconstruct them to get the order. We show the canonical ordering of table 3 as an example.

First, we project the vertex number($Vi$) and vertex labels ($VLi$) from the original table and insert them into a table called $Unsorted\_Vertex$ as shown in table 4. We also include the position in which the vertex occurs in the original table. Next we sort table 4 on $ID$ and vertex label($VL$) and insert them into the table called $Sorted\_Vertex$ as shown in table 5. The $New\_Pos$ attribute denotes the new position of the vertex in the Sorted table and the *Old_Pos* attribute denotes the old position of vertex in the Unsorted table.

Similarly we also transpose the connectivity attributes into a table called *Old_Conn* as shown in table 7. Since the sorting on vertex numbers has changed its position we need to update the connectivity attributes to reflect this change. We perform a join of the Sorted_Vertex table and the *Old_Conn* table on the *Old_Pos* attributes to get the updated connectivity attributes and store them in the table

| ID | V1 | V2 | V3 | VL1 | VL2 | VL3 | F1 | T1 | F2 | T2 | EL1 | EL2 | GID |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 4 | 3 | C | D | C | 1 | 2 | 2 | 3 | AD | DC | 2 |
| 2 | 3 | 4 | 1 | D | D | A | 2 | 1 | 3 | 1 | DC | AD | 3 |

**Table 3: Similar Subgraph Instances.**

| ID | V | VL | Old_Pos | New_Pos |
|---|---|---|---|---|
| 2 | 1 | A | 1 | 1 |
| 2 | 3 | C | 3 | 2 |
| 2 | 4 | D | 2 | 3 |
| 3 | 1 | A | 3 | 1 |
| 3 | 3 | C | 1 | 2 |
| 3 | 4 | D | 2 | 3 |

**Table 5: Sorted_Vertex.**

| E_No | V1 | V2 | VL1 | VL2 | EL1 | GID |
|---|---|---|---|---|---|---|
| 1 | 1 | 2 | A | B | AB | 1 |
| 2 | 2 | 3 | B | C | BC | 1 |
| 3 | 1 | 2 | A | B | AB | 2 |
| 4 | 1 | 4 | A | D | AD | 2 |
| 5 | 2 | 3 | B | C | BC | 2 |
| 6 | 1 | 2 | A | B | AB | 3 |
| 7 | 1 | 3 | A | D | AD | 3 |

**Table 6: Subgraph_1: the base table for expansion.**

called *New_Conn* as shown in table 8. Then we sort the *New_Conn* table on the $ID$,$F$, and $T$ attributes and store them in the *Sorted_Conn* table as shown in table 9.

Since we also need the GID attribute to generate new subgraphs, we transpose the GID attribute to a table called *Sorted_GID* as shown in table 10. Now we have the order vertex table *Sorted_Vertex* as well as the connectivity table *Sorted_Conn*. We can perform a join on the n+1 *Sorted_Vertex*, *Sorted_Conn*, and *Sorted_GID* tables to reconstruct the original subgraphs in canonical order.

Subgraph isomorphism is the most time consuming step in our algorithm. In the future we plan to create a User Defined Function to read each subgraph into a adjacency matrix and then transpose the adjacency matrix into canonical form which is the same for all the similar subgraphs. This method will greately reduce the number of join operations in the DBMS.

## 3.4 Algorithm

Our algorithm to find the frequent subgraphs takes a similar apporach to the Apriori algorithm [12]. The main difference is that instead of using set containment we use subgraph ismorphism. As shown in Figure 4, the first step in our algorithm is to generate all the possible subgraphs of

| ID | EL | F | T |
|---|---|---|---|
| 2 | AD | 1 | 2 |
| 2 | DC | 2 | 3 |
| 3 | DC | 2 | 1 |
| 3 | AD | 3 | 1 |

**Table 7: Old_Conn.**

| ID | EL | F | T |
|---|---|---|---|
| 2 | AD | 1 | 3 |
| 2 | DC | 3 | 2 |
| 3 | DC | 3 | 2 |
| 3 | AD | 1 | 3 |

**Table 8: New_Conn.**

| ID | EL | F | T |
|---|---|---|---|
| 2 | AD | 1 | 3 |
| 2 | DC | 3 | 2 |
| 3 | AD | 1 | 3 |
| 3 | DC | 3 | 2 |

**Table 9: Sorted_Conn.**

increasing size and count each one. The next step is to join the found subgraphs with the base subgraph and prune this table to find the candidate subgraphs. We then scan the graph dataset to get the support of each candidate subgraph and compare it to the specified minimum threshold support value. The subgraphs that meet the minimum threshold support value are added to the table of frequent subgraphs. In the next step, we check the candidate subgraph table to see if it is emtpy. If it is we terminate the algorithm, otherwise, we repeat step number two and cyle until the candidate subgraph table is empty.

## 4. PRELIMINARY EXPERIMENTS

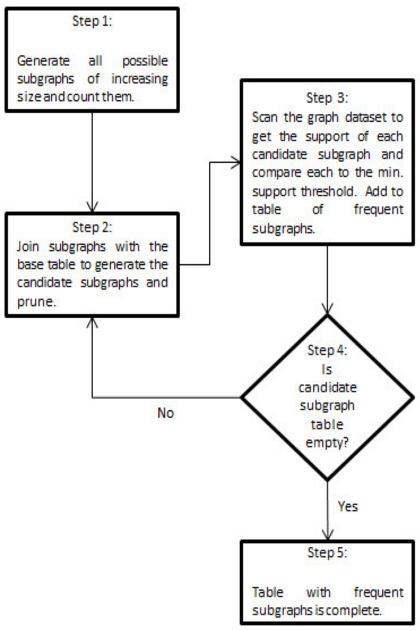
Our experiments focus on evaluating the performance of our algorithm on graph datasets with varying size and threshold. The basic idea behind the data generator of synthetic datasets is described in [9] and the parameters are shown in table 11. All the experiments were performed with a DBMS server with 3.2GHz CPU, 4GB of RAM, and 600GB of storage space.

The first set of experiments were conducted to analyze the performance of our algorithm on graph datasets with varying size. The size of the datasets $|M|$ varied from 100K to 300K graphs. The other parameters were set as follows: $|T| = 20$, $|I| = 5$, $|L| = 200$ and $N = 30$. The minimum threshold $\sigma$ was set to 1% . Table 12 and Fig 5 show the processing time of our algorithm on the dataset.

The second set of experiments were conducted to analyze the performance of the algorithm with varying minimum threshold. The graphs contained in the datasets are also

| ID | GID |
|---|---|
| 1 | 2 |
| 3 | 3 |

**Table 10: Sorted_GID.**



Step 1:
Generate all possible subgraphs of increasing size and count them.
Step 2:
Join subgraphs with the base table to generate the candidate subgraphs and prune.
Step 3:
Scan the graph dataset to get the support of each candidate subgraph and compare each to the min. support threshold. Add to table of frequent subgraphs.
Step 4:
Is candidate subgraph table empty?
No
Yes
Step 5:
Table with frequent subgraphs is complete.

**Figure 4: Flowchart of algorithm.**

| Notation | Parameter |
|---|---|
| $\lvert M\rvert$ | The total number of graphs |
| $\lvert T\rvert$ | The average size of graphs |
| $\lvert I\rvert$ | The average size of potentially frequent subgraphs |
| $\lvert L\rvert$ | The number of potentially frequent subgraphs |
| $N$ | The number of edge and vertex labels |

**Table 11: Synthetic dataset parameters.**

| $\lvert GD\rvert$ | $\lvert E\rvert$ | Time(s) |
|---|---|---|
| 100K | 4M | 1063 |
| 150K | 6M | 1874 |
| 200K | 8M | 3027 |
| 250K | 10M | 3916 |
| 300K | 12M | 4755 |

**Table 12: Processing time with varying size of dataset.**

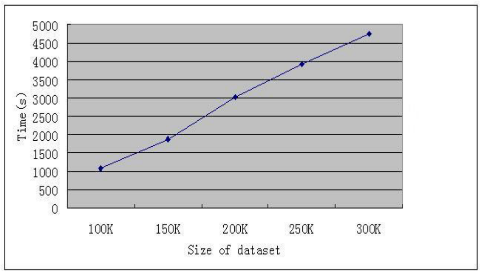


**Figure 5: Processing time with varying size of dataset.**

| $\sigma$ | $\lvert GD\rvert$ | Time(s) |
|---|---|---|
| 5% | 100K | 659 |
| 4% | 100K | 744 |
| 3% | 100K | 872 |
| 2% | 100K | 937 |
| 1% | 100K | 1063 |

**Table 13: Processing time with varying support threshold.**

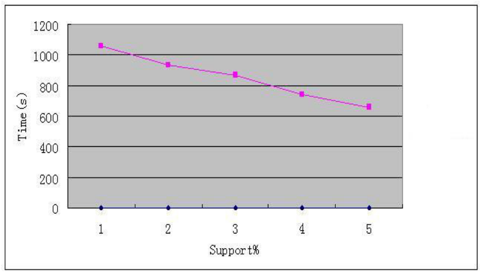


**Figure 6: Processing time with varying support value.**

without cycles or multiple edges. All the parameters for the synthetic dataset were set as follows: $|M| = 100K$, $|T| = 20$, $|I| = 5$, $|L| = 200$ and $N = 30$.. We varied the minimum threshold $\sigma$ from 1% to 5% while keeping the max size of subgrphs as 5. Table 13 and Fig 6 show the experiment results on varying support value.

## 5. RELATED WORK

In recent years, a number of efficient and scalable algorithms have been developed for frequent subgraph mining. The most famous are AGM [8], FSG [9] and G-SPAN [14]. In this section we will briefly overview these three algorithms.

The AGM [8] algorithm was initially developed to find frequently induced subgraphs. This paper introduced the mathematical graph representation of *adjacency matrix*. This normal form representation is general but not unique for a graph. To perform the support counting efficiently, canonical form is defined for normal forms of adjacency matrices representing an identical induced subgraph. FSG is the first work on finding frequent connected subgraphs from a set of labeled graphs. It uses a breadth-first approach to discover the lattice of frequent subgraphs. The size of these subgraphs is grown by adding one edge at a time, and the frequent pattern lattice is used to prune non-downward-closed candidate subgraphs. FSG [9] employs a number of techniques to achieve high computational performance including efficient canonical labeling and various optimization during frequency counting. The G-SPAN [14] algorithm finds the frequent subgraphs following a depth-first approach. Two techniques, *DFS lexicographic order* and *minimum DFS code* are introduced in this paper. To ensure that the subgraph comparisons are done efficiently, they use the canonical labeling scheme based on depth-first traversals.

## 6. CONCLUSION AND FUTURE WORK

In this paper we proposed an efficient algorithm to generate frequent subgraphs with SQL. We extend the graph representation of DB-SUBDUE [3, 2] by adding the *GID* attribute to accommodate the mining algorithm to a set of graphs. We also create some necessary indices in the *Edge*, *Vertex* and *Subgraph_i* tables to improve the speed of the join operation which is necessary in the frequent subgraph search process. In the frequent subgraph search step we implement unconstrained expansion to generate subgraphs of increasing size. This expansion method, which expands subgraphs on any matching vertex along the edges that do not exist in the subgraphs, allows our algorithm to generate all the possible subgraphs. Although unconstrained expansion is efficient at generating all subgraphs, it allows the similar subgraphs to grow in different order, making the frequency counting incorrect. In order to solve this problem, we introduced the canonical ordering technique in the subgraph isomorphism step. With the canonical ordering technique, we reorder all the subgraphs based on their vertex labels, edge labels, and connectivity attributes. After the reordering process, the similar subgraphs growing in different way will have the same order of vertex labels, edge labels, and connectivity attributes which help to identify the subgraph isomorphism.

There are some issues to be considered for future work. Our current algorithm uses vertex labels and edge labels to determine the frequent subgraphs. This means that frequent subgraphs will have the same vertex labels and edge labels. This greatly reduces the amount of true frequent subgraphs that are found since they must contain the same labels. The next step in our research is to use our algorithm but in a modified form that uses the actual form of the graph to determine the frequent subgraphs. This would yield a greater number of frequent subgraphs since labels would not be taken into consideration for subgraph isomorphism. One way to attempt to solve this problem is to use approaches similar to the ones presented in [10] and [11], which can aid in building models for the input graphs.

## 7. ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCF 0937562 and IIS 0914861.

## 8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB Conference*, pages 487–499, 1994.

[2] R. Balachandran, S. Padmanabhan, and S. Chakravarthy. Enhanced db-subdue: Supporting subtle aspects of graph mining using a relational approach. In *PAKDD*, pages 673–678, 2006.

[3] S. Chakravarthy, R. Beera, and R. Balachandran. Db-subdue: Database approach to graph mining. In *PAKDD*, pages 341–350, 2004.

[4] Z. Chen, C. Ordonez, and C. Garcia-Alvarado. Fast and dynamic OLAP exploration using UDFs. In *SIGMOD*, pages 1087–1090, 2009.

[5] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res. (JAIR)*, 1:231–255, 1994.

[6] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36, 1999.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[8] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.

[9] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.

[10] C. Ordonez. Models for association rules based on clustering and correlation. *Intelligent Data Analysis*, 13(2):337–358, 2009.

[11] C. Ordonez and Z. Chen. Evaluating statistical tests on OLAP cubes to compare degree of disease. *IEEE Transactions on Information Technology in Biomedicine (TITB)*, 13(5):756–765, 2009.

[12] C. Ordonez, K. Zhao, and Z. Chen. Bounding and estimating association rule support from clusters on binary data. In *IEEE FDM*, 2008.

[13] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD Conference*, pages 433–444, 2008.

[14] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[15] K. Yoshida, H. Motoda, and N. Indurkhya. Graph-based induction as a unified learning framework. *Appl. Intell.*, 4(3):297–316, 1994.