Compact Graph Representative Structure for Efficient Graph Querying in Chemical Compound Graph Databases

Aye Nwe Thaing
University of Computer Studies, Yangon
ayenwethaing@gmail.com

***Abstract*—Graph has become a powerful tool for representing and modeling objects and their relationships in various application domains such as protein interaction, social networks and chemical informatics. With the emergence of these applications, developments of graph databases are very useful to store graph data. However, performance query processing on graph database (GDB) is still inadequate due to the high complexity of processing graph data. As a result, effective graph structures to store graphs are the current research area to complete efficient query processing on the graph databases. In this paper, a new compact graph representative structure(CGRS) is proposed to perform efficient graph query processing. In CGRS, a graph is represented holistically via its edge code using edge dictionary. The idea of graph decomposition is used to specify all connected, induced subgraphs of a given graph without changing the structure of the original graphs. Using CGRS, two types of graph queries can be processed: graph isomorphism query and subgraph isomorphism query. Chemical compound dataset is applied to test the effectiveness of our proposed CGRS. The analysis of storage space, graph construction time and query response time offers a positive response to our newly proposed CGRS structure.**

Keywords—graph databases, graph query processing, graph isomorphism query,subgraph isomorphism query

I. Introduction

A graph describes relationships over a set of entities. With node and edge labels, a graph can depict the attributes of both the entity set and the relation. The graph database contains large amount of graphs and a graph handles billions of nodes and relationships. Determining the graph database members which constitute answer set of a graph query from a graph database is a key performance issue in all graph-based applications [7,1].

A graph database is a kind of No-SQL data store [4,3] that uses graph structures with nodes, edges and properties to represent and store information. No-SQL may not require fixed table schemas and avoids join operations. Graph databases are the most scalable, high performance way to query and store highly interconnected data.

Though studied previously, efficient graph query processing in the graph database is central research area[9,2]. Typical querying tasks for the graph databases include graph isomorphism query processing, subgraph isomorphism query processing and similarity query processing.

For graph isomorphism query, one looks for a specific graph structure in the graph database. Graph isomorphism test is one of the most commonly used matching conditions which determine that a query graph ***q*** matches a graph ***g*** if and only if ***q*** is isomorphic to ***g***. For subgraph isomorphism query, one looks for a specific pattern in the graph database. A matching condition is used to decide if a pattern subgraph occurs in a graph. Subgraph isomorphism test is one of the most commonly used matching conditions which determine that a pattern ***p*** matches a graph ***g*** if and only if ***p*** is a subgraph of ***g***. For similarity query, one looks for graphs in a graph database that are similar to a query graph. Graph edit distance is a common way to measure the similarity between two graphs.

A primary challenge in computing the answers of graph queries is that pair-wise comparisons of graphs are really hard problems. For graph queries, one faces the graph isomorphism problem, known to be NP-complete. Therefore, finding an efficient search technique is immensely important due to the costs of pair-wise comparisons and the increasing size of modern graph databases. The success of any graph database application is directly dependent on the efficiency of graph indexing and query processing mechanisms.

In this paper, a proficient compact graph representative structure is proposed to process graph isomorphism query and subgraph isomorphism query. In CGRS, edge code is used to represent the graph. Edge code of each graph incorporates the structural information of graph in the database. Edge code is based on edge-based representation of the undirected connected graph. Edge dictionary in CGRS contains the distinct edges in the graph database with unique identifiers. The graph decomposition work is the enumeration of all connected, induced subgraphs of a graph. Moreover, algorithms for CGRS structure, graph isomorphism query and subgraph isomorphism query are proposed to query graphs efficiently. The comprehensive study shows that CGRS decreases storage space and time complexity (i.e. graph construction time and query processing time) than OrientDB graph storage structure [10].

The rest of the paper is organized as follows. Section II discusses about the related work of graph index structures and query processing. Section III represents the formal definitions and notations used in the proposed work. Section IV discusses about proposed approach. Section V presents illustration of CGRS algorithm. Section VI discusses graph isomorphism query and subgraph isomorphism query. In section VII, we discuss the experimental result of our proposed system. Section VIII concludes our paper.

II. Related Work

In recent years, a number of efficient indexes have been proposed for graph query processing on graph databases.

Among existing techniques, GDIndex[6] proposed a method of indexing graph databases using structured graph decomposition. The index is made up of two parts: directed acyclic graph (DAG) and a hash table. DAG contains nodes which are distinctive, induced subgraphs of the database graphs. Hash table indexes each subgraph for fast isomorphic lookup. Canonical codes of graphs are used to form hash keys. No candidate verification is required using GDIndex. However, this index is not designed for databases that do not have a large number of distinct graphs.

An efficient Fast-Graph Automorphic Filter (F-GAF) algorithm [11] is proposed that used grid-code representation of graphs. Given a graph database, this algorithm checks the automorphism of graphs without generating huge number of permutation matrices used in canonical labeling.

GraphGrep[5] is a path-based technique to index graph databases. This technique enumerates paths up to a threshold length from each graph. An index table is constructed and each entry in the table is the number of occurrences of the path in the graph. However, the graph database contains huge amount of paths and can have an effect on the performance of the index.

To overcome the path enumeration overhead of GraphGreph, C-tree[8] was proposed that creates graph closures and builds an index over these closures. C-tree organizes graphs hierarchically where each internal node summarizes its descendants by a graph closure. Candidate verification is still needed for sub graph queries to match a query with every leaf node.

OrientDB[10] is the another way of storing graph structure. It is a high-performance open source graph database. It uses a new indexing algorithm called MVRB-Tree, derived from Red-Black Tree and B+ Tree. Therefore, it has benefits of having both fast insertions and ultra fast lookups. OrientDB holds the promise of a flexible schema. It has a strong security profiling system based on users and roles and supports SQL as a query language.

III. PRELIMINARIES

Definition 1. Labeled Graph

A labeled graph G is defined as 5-tuple, (V,E,L_V,L_E,l) where V is the non-empty finite vertex set called vertices, and E is the unordered pairs of vertices called edges. L_V and L_E are the set of labels of vertices and edges and l is a labeling function assigning a label to a vertex $l:V \rightarrow L_v$ or an edge $l:E \rightarrow L_E$.

Definition 2. Graph Isomorphism

Let $G=(V,E,L_V,L_E,l)$ and $G'=(V',E',L'_V,L'_E,l')$ be two graphs. A graph isomorphism from G to G' is an injective function $f: V \rightarrow L_V$ such that (1) $\forall u \in V$, $l(u)=l'(f(u))$, and (2) $\forall (u,v) \in E$,$l(u,v)=l'(f(u),f(v))$.

Definition 3: Exact Graph Matching

A graph G* matches exactly with *graph Q* if there exists a bijective mapping *f: V(G) → V(Q) and two vertices* v₁ *and* v₂ *are adjacent in G, if and only if , their corresponding vertices* u₁ *and* u₂ *are adjacent in Q.

Definition 4. Exact Subgraph Matching

A subgraph $G_i(V_i,E_i)$ of a graph $G(V,E)$ has a vertex set $V_i(G_i) \subseteq V(G)$ and $E_i(G_i) \subseteq E(G)$. For labeled graphs, an additional condition is that the labels of corresponding vertices in the bijective mapping should also be identical.

IV. OUR PROPOSED APPROACH

The proposed CGRS contains mainly two phases: input graph phase and query graph phase. Input graph phase covers four components: preprocessing, subgraph decomposition engine, edge code generating engine and storage. Query graph phase also covers four components: preprocessing, edge code generating engine, edge code querying engine and storage. Therefore, edge code generating engine and storage are common used in both phases. The architecture of CGRS is shown in Figure 1.

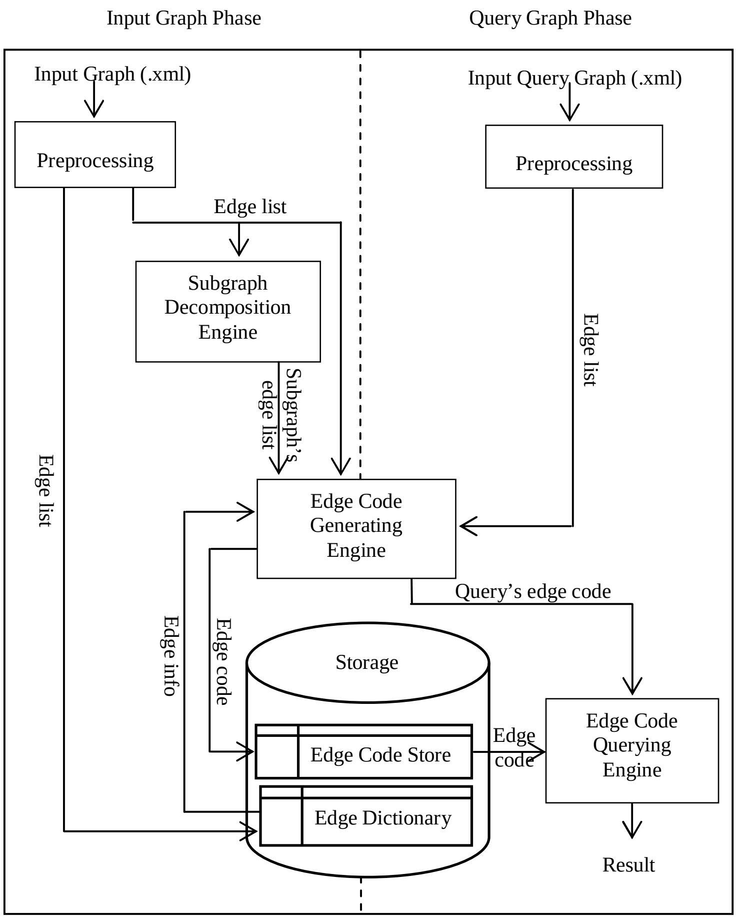


Figure 1. Architecture of proposed system

$$$$

A. *Preprocessing*

The chemical datasets are in the form of xml files and image files (such as .png file). In graph theory, the graphs can be represented in terms of xml files. These xml files are in the form of GraphML format. The input graph to the system is GraphML file format. GraphML is a comprehensive and easy-to-use file format for graphs.

In the preprocessing step, the vertex list of the input graph or query graph is generated from GraphML file. Every vertex in the graph is assigned with unique ID. Then the edge list of the graph is computed. The edge in the graph is defined as (S_{id},E,D_{id}) where S_{id} is the source vertex *id*, E is the edge label and D_{id} is the destination vertex *id*. Figure 2 shows the chemical compound halothane. Figure 3 depicts the corresponding GraphML file of halothane inputted to the system.

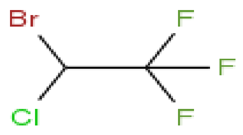


Figure 2 Halothane Compound

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns">
<key id="atom" for="node" attr.name="atom" attr.type="string"/>
<key id="label" for="edge" attr.name="edgeLabel" attr.type="string"/>

<graph id="G1" edgedefault="undirected" info="halothane">

<!-- nodes -->
<node id="1"><data key="atom">Br</data></node>
<node id="2"><data key="atom">C</data></node>
<node id="3"><data key="atom">Cl</data></node>
<node id="4"><data key="atom">C</data></node>
<node id="5"><data key="atom">F</data></node>
<node id="6"><data key="atom">F</data></node>
<node id="7"><data key="atom">F</data></node>

<!-- edges -->
<edge source="1" target="2"><data key="label">s</data></edge>
<edge source="2" target="3"><data key="label">s</data></edge>
<edge source="2" target="4"><data key="label">s</data></edge>
<edge source="4" target="5"><data key="label">s</data></edge>
<edge source="4" target="6"><data key="label">s</data></edge>
<edge source="4" target="7"><data key="label">s</data></edge>

</graph>
</graphml>
```

Figure 3 GraphML File of Halothane Compound

The input GraphML file of halothane is parsed using xml parser in the preprocessing step. The vertices list and edge list of halothane are as follows.

	1	2	3	4	5	6	7
Vertices list(halothane):	Br	C	Cl	C	F	F	F

Edge list(halothane): <Br,s,C>, <C,s,Cl>, <C,s,C>, <C,s,F>, <C,s,F>, <C,s,F>

B. *Storage*

The storage contains two parts: edge dictionary and edge code store. Edge dictionary stores the edges of the incoming graphs to the system. Edge code storage stores the edge code of incoming graphs and their subgraphs. In CGRS, the graphs are transformed to their edge codes and these edge codes are stored in the CGRS structure for the purpose of disk-based storage.

C. *Edge Dictionary*

The edge dictionary contains unique edges with global unique IDs to represent edges in the graph database. When a graph introduces to the database, the distinct edges(D_e) in the graph are searched and added to the edge dictionary if the dictionary does not contain these edges. In the dictionary, these edges are assigned with global unique IDs for further graph processing. Therefore, it is efficient to retrieve the equivalent edge came along in the graph. Moreover, most of the edges in chemical structures are the same and storing the edges of the chemical structures in the dictionary is more efficient.

Figure 4 shows the step by step procedure of processing edge dictionary. The input to the edge dictionary is the edge list of the incoming graph. The edge dictionary works according to the steps in the following procedure.

```
ID:=0
    **For each** $e \in EL(h_k)$ **do**
    **If** $e_i = e_j$ **then**
        $D_e(h_k) := e_i$
        **If** $\nexists D_e(h_k) \in EDict$ **then**
            EDict.ID:= EDict.ID+1
            EDict.Edge:=EDict.Edge+$D_e(h_k)$
**Return** EDict
```

Figure 4 Procedure of Processing Edge Dictionary

Halothane compound shown in Figure 2 contains 6 edges: <Br,s,C>, <C,s,Cl>, <C,s,C>, <C,s,F>, <C,s,F> and <C,s,F>. These edges are inputted to the edge dictionary. In the edge dictionary, the distinct edges are searched. There are 4 distinct edges in halothane: <Br,s,C>,<C,s,Cl>,<C,s,C> and <C,s,F>. After that each of these distinct edges are matched with edges in the dictionary. If the edge is not defined in the dictionary, the edge is inserted into the dictionary with unique identifier. If the edge is already defined in the dictionary, the edge is filtered out. In this way, the dictionary can save the storage space. Assume that there is no edge in the edge dictionary and halothane is the first compound entering into the CGRS structure. Table 1 shows the edge dictionary containing the edges in halothane.

Table 1 Edge Dictionary Containing Edges in Halothane

ID	Edge
1	Br, s, C
2	C, s, Cl
3	C, s, C
4	C, s, F

D. *Subgraph Decomposition Engine*

Graph decomposition is the enumeration of all connected, induced subgraphs of a graph. All edges in graph G are partitioned into subsets so that each subset is the induced subgraph (g_i) of G. The smallest subgraph contains two nodes and one edge exactly. Subgraph decomposition engine contains two steps: (1) finding *c*-organizer between subgraphs and (2) forming a new subgraph by unifying two subgraphs.

In the first step, *c*-organizer is determined between two subgraphs. To compute c-organizer, the vertices of two subgraphs are the *ids* of corresponding vertices. The smallest subgraphs of the original graph are all edges in this graph. The number of subgraphs with their respective vertices is defined as ik_m where *i* is the number of vertices and *m* is the number of subgraphs with *i* vertices. To compute $3k_m$ subgraphs for the given graph, we need to examine whether $2k_2$ subgraphs have *c*-organizer to merge these two graphs. In that case, we have some restrictions: to merge $2k_2$ subgraphs, *c*-organizer must be '*1*' (i.e; the two subgraphs has *1* common organizer), and to combine $3k_2$ subgraphs, the organizer must be '*2*' etc. Thus, *c*-organizer needs *(i-1)* to combine two ik_m subgraphs. To find *c*-organizer between subgraphs, we define the formula as follows:

$$c\text{-}organizer = id_{\cap} = Graph_{[size]}-1$$

In the second step, ik_m subgraphs are merged to form new $(i+1)k_m$ subgraphs depending on the *c*-organizer in the previous step. The formula of unifying two subgraphs is defined as follows:

$$V(g_{new}) = id[V(g_i)] \cup id[V(g_j)]$$

The proposed subgraph decomposition engine works according to the step by step procedure decribed in Figure 5.

```
n=|G_k|, a=b=2, m=1
For each e ∈ EL(G_k) do
    g_i:=e
    SG.SubGraph:=SG.SubGraph ∪ (g_i)
    bk_m := bk_m+g_i
    m:=m+1
    While ((n-a)>1)
    For each g_i ∈ bk_m do
        For each g_j ∈ bk_m do
            If (g_i ≠ g_j) then
                If (|g_i ∩ g_j|) =| g_i|-1 then
                    g_ij := {g_i} ∪ {g_j}
                        SG.SubGraph:=SG.SubGraph ∪ (g_ij)
    a:=a+1
```

Figure 5 Procedure of Decomposing Subgraph

Halothane compound shown in Figure 2 has 56 subgraphs in total. However, these subgraphs contains duplicated subgraphs and our proposed system filters out the duplicated subgraphs. In that case, filtering the duplicated subgraphs cannot affect the original structure of input graph. If the subgraphs are identical, only one subgraph is stored in CGRS structure. Removing the duplicated subgraphs can also reduce the amount of storage space in our proposed system. After the duplicated graphs are filtered out, halothane compound contains 21 subgraphs and these subgraphs are stored in CGRS structure.

E. *Edge Code Generating Engine*

A graph is represented holistically into an edge code that captures the structural representation of the graph. The edge code generating engine computes the adjacent edge information of each edge appeared in the graph. Every edge in the graph is assigned with global unique identifier already defined in the edge dictionary. For each edge *e*, the adjacent edges of *e* are investigated in the graph where the identifiers of the adjacent edges are the global edge identifiers in the edge dictionary. Figure 6 describes the procedure for constructing edge codes of the graphs.

```
    ∀EL(h_k) ∈ h_k
        Find all e_adj for each e ∈ EL(h_k)
        Substitute each e_adj and e with corresponding EDict.ID
        EC(h_k):= all e_adj of each e
    Return EC(h_k)
```

Figure 6 Procedure of Constructing Edge Code

According to the procedure in Figure 6, the edge code of halothane compound is defined as follows:

Edge code (halothane): 1{2,3}2{1,3}3{1,2,4,4,4}4{3,4,4}4{3,4,4}4{3,4,4}

Table 2 describes the subgraphs of halothane with their edge codes.

Table 2 Subgraphs and Edge Codes of Halothane Compound

No.	Subgraph	Edge Code
1	BrC	1
2	CCl	2
3	CC	3
4	CF	4
5	CClBr	1{2}2{1}
6	CCBr	1{3}3{1}
7	CCCl	2{3}3{2}
8	CFC	3{4}4{3}
9	CFF	4{4}4{4}
10	CClCBr	1{2,3}2{1,3}3{1,2}
11	CCBrF	1{3}3{1,4}4{3}
12	CCClF	2{3}3{2,4}4{3}
13	CFFC	3{4,4}4{3,4}4{3,4}
14	CFFF	4{4,4}4{4,4}4{4,4}
15	CClCBrF	1{2,3}2{1,3}3{1,2,4}4{3}
16	CCBrFF	1{3}3{1,4,4}4{3,4}4{3,4}
17	CCClFF	2{3}3{2,4,4}4{3,4}4{3,4}
18	CFFFC	3{4,4,4}4{3,4,4}4{3,4,4}4{3,4,4}
19	CClCBrFF	1{2,3}2{1,3}3{1,2,4,4}4{3,4}4{3,4}
20	CCBrFFF	1{3}3{1,4,4,4}4{3,4,4}4{3,4,4}4{3,4,4}
21	CCClFFF	2{3}3{2,4,4,4}4{3,4,4}4{3,4,4}4{3,4,4}

F. *Edge Code Querying Engine*

The edge code of the input query graph is matched with that of other graphs in the database to check whether the two graphs are isomorphic or not. Using the proposed edge code, the system finds the graphs that are exact match to the query graph. The input query graph can be graph isomorphism query or subgraph isomorphism query. If the query graph is isomorphic to the graph stored in the CGRS structure, the two graphs have the same edge code representation and the proposed system concludes that the graphs are isomorphic and then the original graph stored in CGRS is displayed. If query graph is subgraph isomorphic to the graph in CGRS, the system finds the original graph(s) that contain the query graph. Then the original graph(s) in CGRS is displayed. The detail description of graph query processing is discussed in section VI.

V. CGRS ALGORITHM

The CGRS algorithm works according to the steps described in section IV. For each input graph, the algorithm generates edge code for each graph and stores them in edge code store. For edge dictionary, if the edges in the input graph are not already defined in the dictionary, these edges are inserted into it with global unique identifiers. Table 3 explains notations used in CGRS algorithm.

Table 3 Notations Used in CGRS Algorithm

Notation	Definition
GDB	Graph database
G_i	Graph in GDB
$EL(G_i)$	Edge list in G_i
EDict	Edge dictionary
n	No. of vertices in G_i
$EC(G_i)$	Edge code of G_i
MG	Main graph array list
SG	Subgraph array list
e	Edge of G_i
e_{adj}	Adjacent edge of e
bk_m	No. of m subgraphs with b vertices
$D_e(G_i)$	Distinct edge in G_i

Alogrithm 1 CGRS

```
Input:  GDB ⟵{G_1,G_2,…,G_n}, EL(G_k), EDict
Output: cgrs, EDict
n=|G_k|, a=b=2, m=1
For each G_k ∈ GDB do
    EDict:=UpdateEdgeDictionary(EL(G_k),EDict)
    EC(G_k):=GenerateEdgeCode(EL(G_k),EDict)
    MG.Graph:=MG.Graph ∪ {G_k}
    MG.EdgeCode:=MG.EdgeCode ∪ EC(G_k)
    For each e ∈ EL(G_k) do
    g_i:=e
     EC(g_i):=GenerateEdgeCode(EL(g_i),EDict)
    SG.SubGraph:=SG.SubGraph ∪ (g_i)
    SG.EdgeCode:=SG.EdgeCode ∪ EC(g_i)
    bk_m := bk_m+g_i
    m:=m+1
    While ((n-a)>1)
    For each g_i ∈ bk_m do
        For each g_j ∈ bk_m do
            If (g_i ≠ g_j) then
                If (|g_i ∩ g_j|) =| g_i|-1 then
                    g_ij := {g_i} ∪ {g_j}
            EC(g_ij):=GenerateEdgeCode(EL(g_ij),EDict)
            SG.SubGraph:=SG.SubGraph ∪ (g_ij)
        SG.EdgeCode:=SG.EdgeCode ∪ EC(g_ij)
    a:=a+1
  Return cgrs
```

Procedure 2 GenerateEdgeCode(EL(G_k),EDict)

```
∀EL(h_k) ∈ h_k
    Find all e_adj for each e ∈ EL(h_k)
        Substitute each e_adj and e with corresponding EDict.ID
        EC(h_k):= all e_adj of each e
Return EC(h_k)
```

Procedure 2 UpdateEdgeDictionary(EL(h_k),EDict)

```
ID:=0
    For each e ∈ EL(h_k) do
    If e_i = e_j then
        D_e(h_k):= e_i
        If ∄D_e(h_k) ∈ EDict then
            EDict.ID:= EDict.ID+1
            EDict.Edge:=EDict.Edge+D_e(h_k)
Return EDict
```

VI. GRAPH QUERY PROCESSING

Using CGRS, two types of graph queried can be processed: graph isomorphism query and subgraph isomorphism query.

A. *Graph Isomorphism Query*

Graph isomorphism query retrieves the graph in the database that is equal to the given query graph. It can be responsively answered without candidate verification through the use of proposed CGRS structure. Candidate verification is to test whether each graph in the candidate set is indeed the result of the query. When the query graph enters, the query is transformed into edge code. The query's edge code is matched with all graphs' edge codes that are the same size of the query graph. Then, the database graph is returned that matches the graph query. The following algorithm describes the step-by-step process for graph isomorphism query.

Algorithm 4 GraphIsomorphismQuery

```
Input:  CGRS-structure, EDict, G_q
Output: D_q ⟵{G_i}
EC(G_q):=GenerateEdgeCode(EL(G_q),EDict)
∀ G_i ∈ CGRS-structure
If |G_q| =|G_i| then
    If EC(G_q) is matched with EC(G_i) then
        D_q := G_i
Return Dq
```

Suppose the edge code of the query graph is constructed using procedure ***GenerateEdgeCode***. We assume that a query graph has at least two vertices and one edge. We establish a necessary condition that forms the basis for processing exact graph isomorphism queries. Thus we state the following theorem.

Theorem 1 Given a query graph $\boldsymbol{Q}$, if $\boldsymbol{Q}$ is an exact graph of database graph $\boldsymbol{G}$, then EC($\boldsymbol{Q}$) = EC($\boldsymbol{G}$).

Proof. By definition, if $\boldsymbol{Q}$ and $\boldsymbol{G}$ *are same*, then every edge of $\boldsymbol{Q}$ appears in $\boldsymbol{G}$. while applying procedure ***GenerateEdgeCode*** on $\boldsymbol{Q}$, all adjacent edges for every edge in $\boldsymbol{Q}$ were computed during the edge code construction for $\boldsymbol{Q}$. Therefore, if parametric quantities of EC($\boldsymbol{Q}$) are identical to EC($\boldsymbol{G}$), then EC($\boldsymbol{Q}$) = EC($\boldsymbol{G}$).

The intuition is as follows. If a query is an exact graph of the database graph, then its edge list is equal to the edge list of the database graph. Therefore, the adjacent edges of each edge that appear in the edge code of the query will definitely appear in the edge code of the database graph.

B. *Subgraph Isomorphism Query*

A subgraph query retrieves all the graphs in the database that are supergraphs of a given query graph. Subgraph isomorphism query can be responsively answered without verification through the use of proposed CGRS. When the query graph enters, the query is transformed into an edge code. The query's edge code is matched with all graphs' edge codes that are the same size of the query graph. Then, the database graphs are returned that contain the subgraph query. The following algorithm describes the step-by-step process for subgraph isomorphism query.

Algorithm 5 SubgraphIsomorphismQuery

```
Input  :  CGRS-structure, EDict, $G_q$
Output:  $D_q \longleftarrow \{G_1,G_2,\dots,G_i\}$
$EC(G_q)$:=GenerateEdgeCode($EL(G_q)$,EDict)
$\forall$ $G_i \in$ CGRS-structure
If $|G_q| = |G_i|$ then
    If $EC(G_q)=EC(G_i)$ then
    If $G_i$ is DB graph then
        $D_q := D_q + G_i$
    Else if $G_i$ is the subgraph then
        find supergraph $G_s$ of $G_i$
        $D_q := D_q + G_s$
Return $D_q$
```

Assume that we construct the edge code of the query graph using procedure ***GenerateEdgeCode***. We establish a necessary condition that forms the basis for processing subgraph isomorphism queries. Thus we state the following theorem.

Theorem 2 Given a query graph ***Q***, if ***Q*** is a subgraph of database graph ***G***, then EC(***Q***) ⊆ EC(***G***).

Proof. By definition, if ***Q*** is a subgraph of ***G***, then every edge of ***Q*** appears in ***G***. while applying procedure 2 on ***Q***, all adjacent edges for every edge in ***Q*** were computed during the edge code construction for ***Q***. Therefore, if parametric quantities of EC(***Q***) are contained in EC(***G***), then EC(***Q***) ⊆ EC(***G***).

The intuition is as follows. If a query is a subgraph of a database graph, then its edge list is a subset of the edge list of the database graph. Therefore, the adjacent edges of each edge that appear in the edge code of the query will definitely appear in the edge code of the database graph.

VII. EXPERIMENTAL RESULTS

A performance study for CGRS is presented in this section which is focused on chemical compound graph dataset. Chemical graph datasets are in the forms of image files such as (.png) files and XML files. Chemical datasets are downloaded from http://pubchem.ncbi.nlm.nih.gov/. CGRS applied OrientDB to store information about registered graphs in our proposed system. Therefore, we compare CGRS to OrientDB to describe our comparison results. Moreover, OrientDB's Apache 2.0 license is extremely liberal. The performance evaluation judges the efficiency of the proposed work.

A. *Performance Analysis of Storage Space*

A comprehensive study on chemical compound dataset to compare storage space of CGRS and other techniques was conducted. The storage space for various types of graphs such as complete, dense and sparse graphs was studied. A complete graph is a graph that contains the maximum number of edges i.e. n(n-1) edges in the graph. A dense graph is a graph in which the number of edges is close to the maximal number of edges. A graph with only a few edges is called a sparse graph. Figure 6 shows the analysis of storage space to allocate different graph structures. Various numbers of graphs between 500 and 3000 graphs are tested. The maximum graph size is 195 (i.e., no. of edges in the graph) and the minimum graph size is 3. Storage space of four techniques such as image file, xml file, OrientDB and CGRS are analyzed.

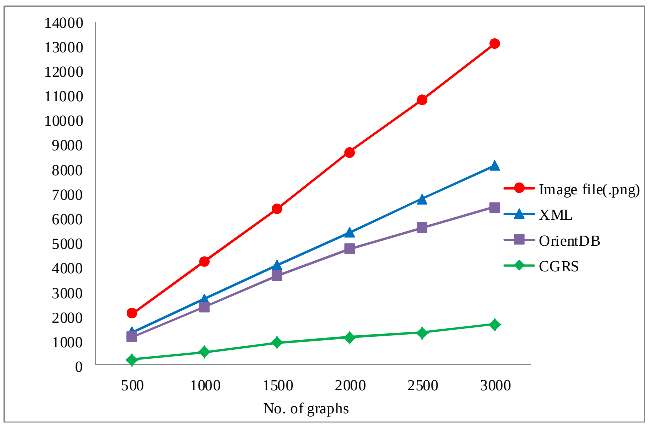


Figure 6 Analysis of storage space for various no. of graphs between different graph structures

From our empirical analysis, the storage space of CGRS using edge code is compact when compared to image file(.png), xml file and OrientDB. It can reduce about 8 times storage space than image file(.png)and about 4.9 times than xml file. It can also reduce the storage space about 4.31 times than OrientDB.

Figures 7, 8 and 9 show the storage space to allocate sparse, dense and complete graphs for different graph structures. Various numbers of graphs between 200 and 1000 graphs are tested for different types of graphs such as sparse, dense and complete graphs. CGRS saves storage space about 23.83 times and 10.79 times for sparse graphs when compared to image file(.png) and xml file. It can save the storage space about 7.49 times than OrientDB for sparse graphs. It saves about 10.77 times, 6.35 times and 4.53 times storage space for dense graphs when compared to image file(.png), xml file and OrientDB. It also reduces about 4.56 times and 3.54 times storage space for complete graphs when compared to image file(.png) and xml file. It can save the storage space about 2.94 times than OrientDB for complete graphs.

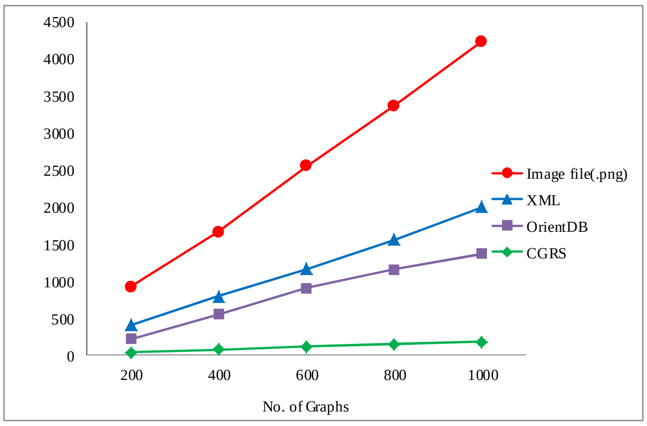


Figure 7 Analysis of storage space for sparse graph between different graph structures

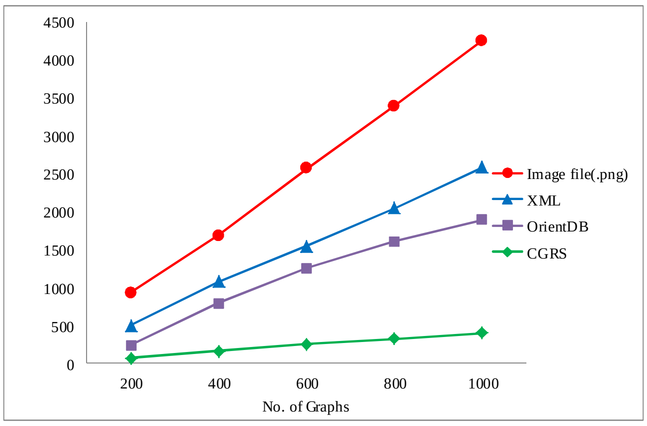


Figure 8 Analysis of storage space for dense graph between different graph structures

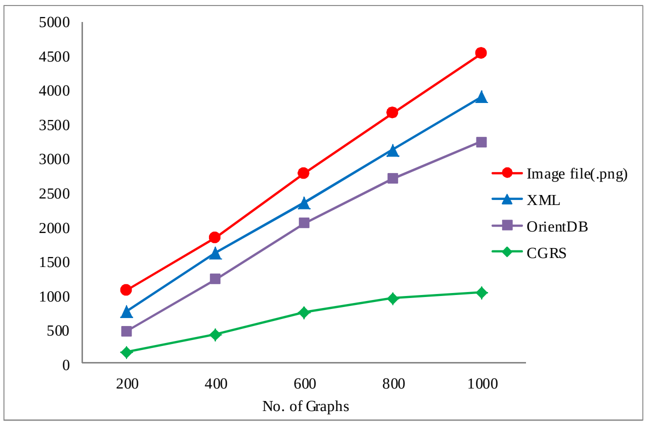


Figure 9 Analysis of storage space for complete graph between different graph structures

B. *Performance Analysis of Computational Time Complexity*

The computational time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input to the problem. The time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm where an elementary operation takes a fixed amount of time to perform.

In CGRS algorithm, generating edge code takes ***O(3m)*** times as follows. CGRS algorithm takes ***m*** comparisons to generate distinct edge list of each graph where ***m*** is the number of edges in the graph. The same number of ***m*** comparisons is required to check the edge dictionary whether the edges in the input graph already exist in it or not. Then, ***m*** comparisons are needed to get the adjacent edge information of edges in the graph. Therefore, the total number of comparisons needed for constructing edge code is ***(m+m+m)*** = ***3m***.

Table 4 describes computational time complexity between four techniques for generating graph representative structures such as canonical code, OrientDB, F-GAF and edge code. Table 5 shows the number of comparisons between four techniques for 100 graphs. CGRS can reduce at least 105 times computational time complexity when compared is to canonical labeling. It can reduce about 5 times computational time complexity than OrientDB. It can lessen at least 1.33 times computational time complexity than F-GAF.

Table 4 Computational Time Complexity between Four Techniques

Technique	**Computational Time Complexity**
Canonical Code	*O(n!)*
OrientDB	*O(nm)*
F-GAF	*O(4m)*
Edge Code	*O(3m)*

where,
n= number of vertices in the graph
m= number of edges in the graph

Table 5 Comparison between Four Techniques for (100) Graphs

No. of graphs in GDB=100					
No. of vertices	**No. of Edges**	**Techniques**			
		Canonical Code	**OrientDB**	**F-GAF**	**Edge Code**
Sparse graphs					
10	12	3.62×10^8	1.2×10^4	4.8×10^3	3.6×10^3
15	18	2.4×10^{20}	4.6×10^4	9.2×10^3	6.9×10^3
20	24	2.65×10^{34}	1.05×10^5	1.4×10^4	1.05×10^4
Dense graphs					
10	21	3.62×10^8	2.5×10^4	1×10^4	7.5×10^3
15	28	2.4×10^{20}	8.4×10^4	1.6×10^4	1.26×10^4
20	35	2.65×10^{34}	1.53×10^4	2×10^4	1.53×10^4
Complete graphs					
10	30	3.62×10^8	4×10^4	1.6×10^4	1.2×10^4
15	40	2.4×10^{20}	1.16×10^5	2.3×10^4	1.74×10^4
20	48	2.65×10^{34}	1.8×10^5	2.4×10^4	1.8×10^4

C. *Performance Analysis of CGRS Construction Time*

CGRS is implemented in java and uses OrientDB graph database to store CGRS structure. All experiments were made using Pentium(R)Dual Core CPU with 2 GB memory and Window 8. From the empirical analysis, execution time varies for sparse, dense and complete graphs. This is because our proposed work is based on edge based representation. Figure 10 shows the construction time for various no. of graphs between CGRS and OrientDB. In our analysis, the minimum number of vertices in a graph is 3 and maximum number of vertices is 90.

From the empirical analysis, the construction times vary depending on the numbers of graphs in the storage structure. Number of graphs between 200 and 1000 graphs are tested and takes the average graph construction time to compare CGRS and OrientDB. In average, CGRS reduces about 5.8 times construction time than OrientDB for sparse graphs. For dense graphs, CGRS reduces about 4.5 times construction times than OrientDB. It lessens about 4.2 times construction time for complete graphs when compared to OrientDB.

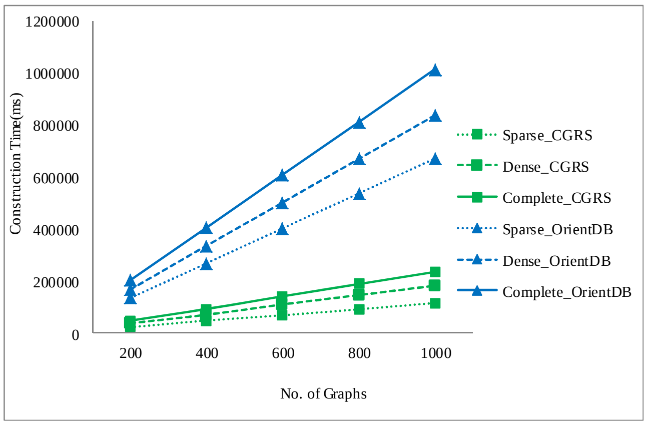


Figure 10 Analysis of construction time for various No. of graphs between CGRS and OrientDB

Figure 11 shows the analysis of construction time between CGRS and OrientDB using various vertices sizes. For CGRS, construction time can save about 7.58 times than OrientDB for sparse graphs. The construction time can save about 6 times in CGRS for dense graphs when compared to OrientDB. For complete graphs, CGRS reduces about 5.33 times construction times than OrientDB.

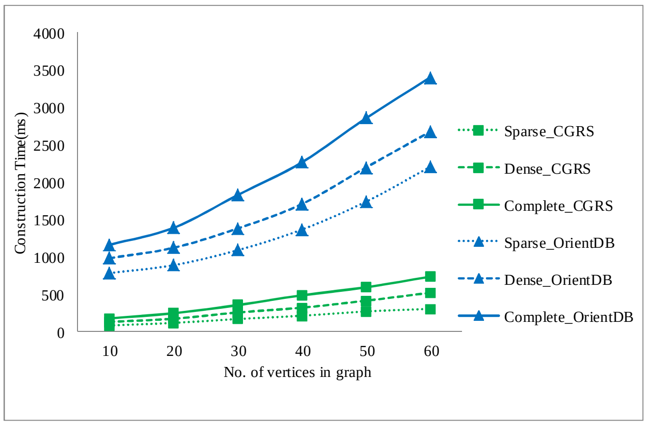


Figure11 Analysis of construction time for various no. of vertices in graph between CGRS and OrientDB

Figure 12 shows the analysis of CGRS construction time over sparse, dense and complete graphs using subgraph decomposition. From our analysis result, it takes more time for complete graphs when compared to sparse and dense graphs. The construction time of sparse graphs can reduce about 2.4 times than complete graphs. For dense graphs, construction time can save about 1.5 times when compared to complete graphs.

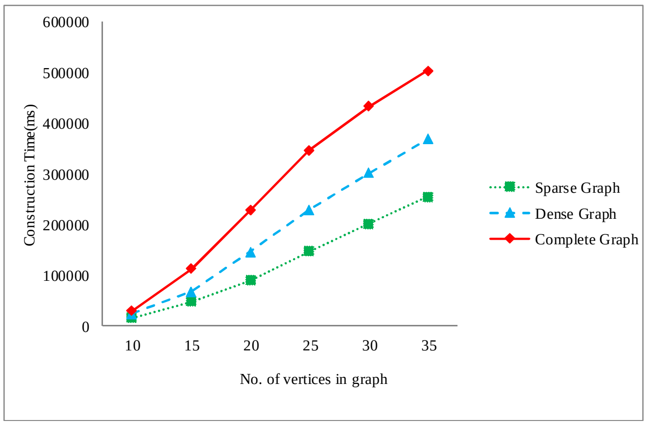


Figure12 Analysis of CGRS construction time for different types of graph using subgraph decomposition

D. *Performance Analysis of Query Response Time*

Figure 13 shows the analysis of the query response time between CGRS and OrientDB for graph isomporphism query. Average query size is 60 and we apply chemical databases with various sizes containing 200 to 1000 graphs. Assume that the query size is the no. of edges in the graph because our proposed work is based on edge-based representation. From our analysis result, the query response time of CGRS is reduced about 6.25 times for sparse graphs over OrientDB. It can decrease about 6.2 times for dense graphs when compared to OrientDB. The querying time of CGRS lessens about 6.09 times than OrientDB for complete graphs. Therefore, the querying time of CGRS decreases about 6 times than OrientDB in average.

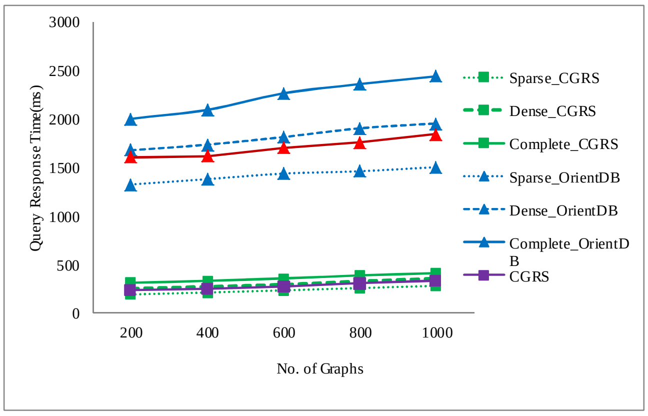


Figure13 Analysis of query response time for various no. of graphs between CGRS and OrientDB

Figure 14 shows the analysis of query response time for various query sizes between CGRS and OrientDB. From our empirical result, the query response time of CGRS reduces about 6.45 times than OrientDB for query size (20). It can reduce about 6.08 times than OrientDB for query size (30). For query size (40), the querying time of CGRS decreases about 5.91 times than OrientDB.

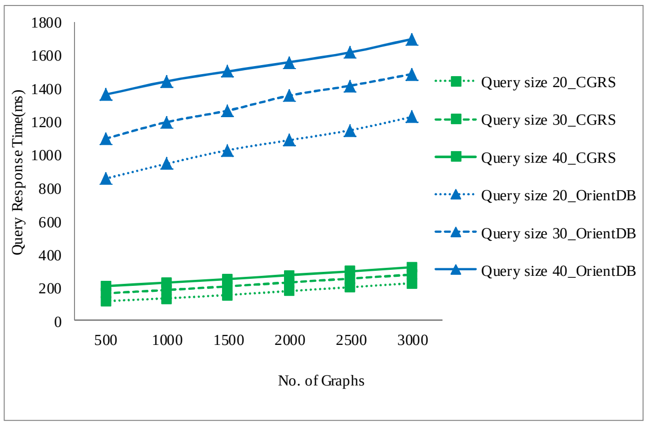
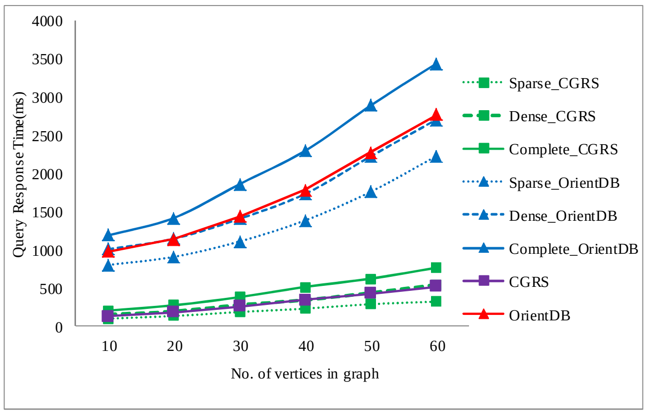


Figure14 Analysis of query response time for various query sizes between CGRS and OrientDB

Figure 15 shows the analysis of the query response time over 1000 database graphs for various numbers of vertices in graph between CGRS and OrientDB. From our analysis result, the query response time of CGRS reduces about 6.76 times than OrientDB for sparse graphs. It can lessen about 5.5 times than OrientDB for dense graphs. The querying time CGRS also reduces 4.95 times for complete graphs when compared to

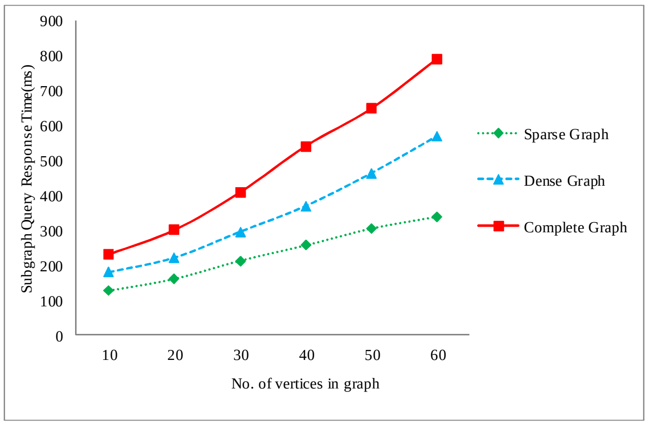


OrientDB. Therefore, the query response time of CGRS decreases about 5.91 times than OrientDB in general.

Figure 15 Analysis of query response time over (1000) graphs for various no. of vertices in graph between CGRS and OrientDB

Figure 16 shows the analysis of subgraph query response time over (1000) database graphs for various no. of vertices (10-60) in the graph. The maximum graph size in the database is 195 and minimum graph size is 5. The average size of query graph is 9. From our analysis result, it takes more time for complete graphs when compared to sparse and dense graphs. The subgraph query response time of sparse graphs can reduce about 2.08 times than complete graphs. For dense graphs, the response time can reduce about 1.39 times when compared to complete graphs.

Figure 17 shows the analysis of subgraph query response time for various subgraph query sizes (7-12) in the graph. From our analysis result, it takes more time when the query graph size is larger. The query response time also depends on the number of database graphs which are supergraphs of given query graph. For query size 7, the subgraph query response time decreases about 1.6 times than query size 12. For query



size 9, the query response time decreases about 1.3 times when compared to query size 12.

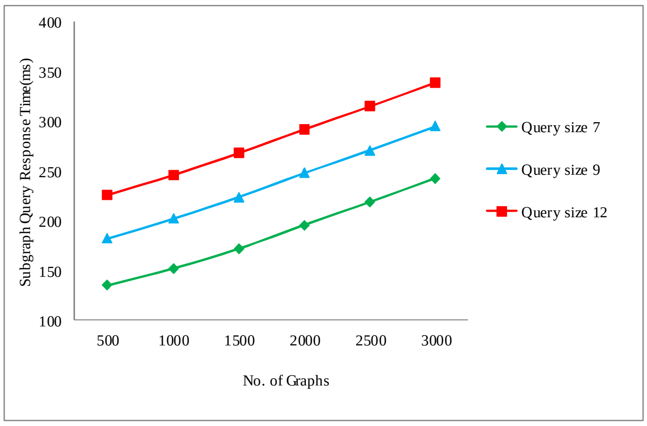


Figure 16 Analysis of subgraph query response time over (1000) graphs for various no. of vertices in graph

Figure 17 Analysis of subgraph query response time for various subgraph query sizes

VIII. CONCLUSION

Our goal is to process graph isomorphism query and subgraph isomporphism query in CGRS efficiently. Thus, a new way of representing a graph holistically via it edge code is proposed. Graphs can be stored in CGRS and queried holistically. CGRS avoids verification for answering graph query using edge code. CGRS is effective in storing graphs in terms of storage space. From our empirical analysis, the storage space of CGRS can save about 8 times than image file (.png), about 4.9 times than XML file and 4.31 times than OrientDB. The construction time of CGRS can reduce at least 3 times when compared to OrientDB. The querying time of CGRS reduces about 6 times than OrientDB. Performance analysis offers a positive response to our newly proposed CGRS structure for efficient graph query processing.

REFERENCES

[1] A. Golovin, K. Henrick, "Chemical Substructure Search in SQL", Journal of Chemical Information and Modeling, 2009.

[2] B.T. Messmer, H. Bunke, “Efficient Subgraph Isomorphism Detection: A Decomposition Approach”, IEEE Transactions on Knowledge and Data Engineering, 2000, 12(2) pp. 307-323.

[3] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, D. Wilkins, “A Comparison of a Graph Database and a Relational Database”, ACMSE’ 10, Oxford, MS, USA, April 15-17, 2010.

[4] D. Funaro, “NoSQL”, Torino, 11 luglio 2011.

[5] D. Shasha, J.T. L.bWang, R. Giugno, “Algorithmic and Applications if Tree and Graph Searching”, 2002.

[6] D.W. Willims, J. Huan, W. Wang, “Graph Database Indexing Using Structured Graph Decomposition”, In Proceedings of the 23th ICDE Conference, Istanbul, 2007, pp. 976-985.

[7] Huan, Wang, W. Banyopadhyay, D. Snoeyink, J. Prins, J. Tropsha, “Mining Protein Family Specific Residue Packing Patterns from Protein Structure Graphs”, In RECOMB, 2004, 308-315.

[8] H. He, A.K. Singh, “Closure-tree: An Indexing Structure for Graph Queries”, In Proceedings of the 22th ICDE Conference, Atlanta, 2006, pp. 38-49.

[9] J.R. Ullmann, “An Algorithm for Subgraph Isomorphism”, Journal of ACM,1976,23(1) pp.31-42.

[10] L. Garulli, “A NoSQL Database for the Internet age”, www.OrienTechnologies.com .

[11] R. Vijayalakshmi, R. Nadarajan, P. Nirmala and M. Thilaga, “ A Novel Approach for Detection and Elimination of Automorphic Graphs in Graph Databases”, Int. J. Open Problems , Vol. 3, No. 1, March 2010.