# Efficiently Extracting Frequent Subgraphs using MapReduce

Wei Lu [#1] Gang Chen [$2] Anthony K.H.Tung [#1] Feng Zhao [#1]

[#]National University of Singapore [$]Zhejiang University

[1]{luwei1,atung,zhaofeng}@comp.nus.edu.sg [2]cg@cs.zju.edu.cn

***Abstract*—Frequent subgraph extraction from a large number of small graphs is a primitive operation for many data mining applications. To extract frequent subgraphs, existing techniques need to enumerate a large number of subgraphs which is superlinear with the cardinality of the dataset. Given the rapid growing volume of graph data, it is difficult to perform the frequent subgraph extraction on a centralized machine efficiently. In this paper, we investigate how to efficiently perform this extraction over very large datasets using MapReduce.**

**Parallelizing existing techniques directly using MapReduce does not yield good performance as it is difficult to balance the workload among the compute nodes. We therefore propose a framework that adopts the breadth first search strategy to iteratively extract frequent subgraphs, i.e., all frequent size-$(i+1)$ subgraphs are generated based on frequent size-$i$ subgraphs at the $i^{\text{th}}$ iteration using a single MapReduce job. To efficiently extract frequent subgraphs, we propose an isomorphism-testing-free approach by properly maintaining how frequent subgraphs are mapped within each graph. Extensive experiments conducted on our in-house clusters demonstrate the superiority of our proposed solution in comparison with the baseline approach.**

## I. Introduction

As a primitive operation, frequent subgraph extraction has a wide range of applications in data mining, such as extracting frequent patterns from chemical compounds, identifying the relationship between chemical compounds, and building graph indexes to facilitate subgraph containment queries [20], [21], [3], [2]. Given a dataset $D$ consisting of a set of graphs, a subgraph $s$ is defined to be *frequent* if the number of graphs in $D$ containing $s$ is not less than a frequent threshold.

The majority of existing work adopts a pattern-growth approach [19], [1], [6], [7], [13] that typically applies the depth first search strategy to extract frequent subgraphs. Figure 1 shows an overview of how a pattern-growth approach works. Initially, by extracting every distinct edge from the graphs, it is easy to identify all frequent size-1 subgraphs. Starting from a certain frequent size-1 subgraph $s$, $s$ is then recursively *extended* until all frequent subgraphs containing $s$ are generated. An *extension* over a frequent subgraph $s$ is to append an edge from some vertex of $s$, and the new generated subgraph is called a *child* of $s$. In Figure 1, the extension order for the offsprings of $s_{11}$ is $s_{21} > s_{31} > s_{32} > s_{22}$. Subsequently, the remainder of each frequent size-1 subgraph is extended one by one in the same manner. Note that a frequent size-$i$ subgraph ($i > 1$) might be generated by multiple size-$(i-1)$ subgraphs. To avoid redundant extensions, one typical solution

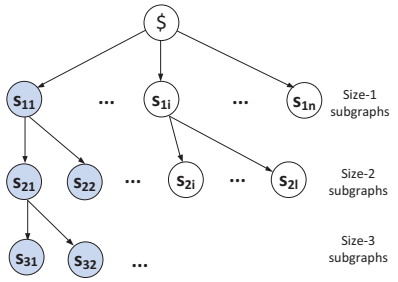



Fig. 1. An Overview of Pattern-growth Approaches

is to define a global order for the subgraphs and each size-$i$ subgraph is always generated by its smallest size-$(i-1)$ subgraph. For example, suppose $S_{11}$ is the smallest size-1 subgraph and hence all size-2 subgraphs that contains $s_{11}$ will be only extended from $s_{11}$. In this way, we can imagine that the cardinality of the offsprings rooted at different subgraphs with the same size is rather skewed and a smaller subgraph usually has more number of offsprings.

Regarding these approaches, it is necessary to maintain considerable mappings from frequent subgraphs to graphs that contain them while the number of mappings is superlinear with the cardinality of the dataset. Considering the AIDS Antiviral Screen[1] dataset consisting of only 43,905 chemical compounds, these pattern-growth approaches including gSpan [19], Mofa [1], FFSM [6], [7] and Gston [13] consume 300MB, 600MB, 1.2GB and 1.3GB memory usage, respectively, when the frequent threshold is set to 5% of the cardinality of the dataset [5]. Consider the forever increasing data size. For example, SCIFinder[2], which provides the world's largest collection of chemistry and related science information, reports that about 4,000 new compound structures are added each day. It is not appropriate to employ these in-memory pattern-growth approaches, and somehow difficult to perform frequent subgraph extraction on a centralized machine efficiently.

MapReduce [4] was recently proposed as a programming model for supporting data intensive applications in a distributed computing environment. It has gained wide acceptance and been applied in various application domains due to its simplicity, flexibility, fault tolerance and scalability. It therefore

[1]http://dtp.nci.nih.gov/docs/aids/aids_data.html

[2]http://www.cas.org

is a good platform for extracting frequent subgraphs over large graph datasets. In this paper, we focus on how to design and implement a solution that enables fast and scalable frequent subgraph extraction using MapReduce.

A *baseline* approach to achieving parallelism directly using MapReduce is to assign each compute node to process the same number of size-1 frequent subgraphs and generate the other frequent subgraphs based on them. However, it does not yield good performance as it is difficult to balance the workload among the compute nodes. We observe that the number of frequent subgraphs generated in different compute nodes is rather skewed and the majority of frequent subgraphs are generated based on very few size-1 subgraphs. Such unbalance is caused by the generation scheme in the pattern-growth approach for avoiding redundant extensions. To alleviate this problem, an alternative is to re-assign the children of these subgraphs to other compute nodes. However, such re-assignment is challenging since we do not know how many frequent subgraphs can be generated based on a given subgraph in advance.

Regarding the baseline approach, it is challenging to balance the workload by assigning the frequent subgraphs to compute nodes. To tackle this imbalance issue, we propose a framework, called MRFSE (MapReduce based frequent subgraph extraction), to iteratively extract frequent subgraphs using breath first search, and apply the *generation-and-verification* mechanism over the graphs at each iteration. The generation task takes each graph $g$ as the input, and generates size-$(i+1)$ subgraphs for $g$ based on all frequent size-$i$ subgraphs that $g$ contains. With the help of existing technique including the minimum DFS code in [19], and CAM in [6], we can obtain a unique key that is a sequence for all isomorphic subgraphs. Therefore, the verification collects each frequent size-$(i+1)$ subgraph from the generation task by counting the number of the same keys. In our case, the cost of generation task is obviously more expensive than that of the verification task. Therefore, we propose a strategy on how to assign graphs to compute nodes so that they could take similar generation cost. In this way, we can balance the workload over each compute node.

Simply applying the generation-and-verification mechanism may involve two MapReduce jobs, one for the generation task and the other for the verification task. However, too many MapReduce jobs could incur considerable overhead of writing the intermediate results. In our case, we use only one MapReduce job. Specifically, the generation task involves the *map* phase only. In the *map* function, it takes the ID and all frequent size-$i$ subgraphs of a graph $g$ as the input, and generate a superset of all frequent size-$(i+1)$ subgraphs of $g$. The verification task involves the *reduce* phase only. In each *reduce* function, it collects all isomorphic size-$(i+1)$ subgraphs by the same key. Although the number of MapReduce jobs is reduced, it is more difficult to balance the workload using a single MapReduce. In our case, as a graph is always processed by the same compute node during the entire iterations, we need to guarantee that each mapper processes similar generation computation at each iteration while the graphs are assigned to the compute nodes before the first iteration. We demonstrate that our proposed strategy on how to assign graphs to compute nodes is effectiveness to balance the workload in this case.

To efficiently produce size-$(i+1)$ subgraphs in the generation task, we propose an isomorphism-testing-free approach by properly maintaining how frequent subgraphs are mapped within each graph. In our design, at the $i^{\text{th}}$ iteration, for each graph $g$, we maintain mappings from frequent size-$i$ subgraphs that $g$ contains to $g$. These mappings for each graph are generated at the $(i-1)^{\text{th}}$ iteration. Based on the mappings, we propose an extension based approach that is similar to pattern-growth approaches, but without any isomorphism testing operations to generate size-$(i+1)$ subgraphs for each graph.

In summary, we make the following contributions:

- We propose framework MRFSE to iteratively extract frequent subgraphs. All frequent size-$(i+1)$ subgraphs are extracted at the $i^{\text{th}}$ iteration using a single MapReduce job. We also propose a strategy on how to balance the workload in term of the generation cost.
- We propose an isomorphism-testing-free approach for the generation task by properly maintaining how frequent subgraphs are mapped within each graph.
- We conduct extensive experiments on our in-house clusters to demonstrate the superiority of our proposed solution in comparison with the baseline approach.

The remainder of the paper is organized as follows. Section II discusses related work. Section III introduces the necessary background knowledge and provides the problem definition. Section IV describes an overview of the framework. Section V presents the extension of existing techniques to generate new subgraphs in our framework. Section VI reports the experimental results and Section VII concludes the paper.

## II. Related Work

Frequent subgraph extraction is a well studied research area in the centralized systems. Existing work that extracts frequent subgraphs exactly can be broadly classified into two categories: Apriori-based approaches [8], [9] and pattern-growth approaches [19], [1], [6], [7], [13]. Apriori-based approaches iteratively extract the frequent subgraphs using breath first search. At the $i^{\text{th}}$ iteration ($i \geq 1$), all frequent size-$(i+1)$ subgraphs are generated by joining two frequent size-$i$ subgraphs. Since a new subgraph can be generated by multiple pairs, a *graph isomorphism testing* [15] is required to remove duplicates. Finally, each distinct subgraph is verified whether it is frequent by counting the number of occurrences in the graphs of the dataset using *subgraph isomorphism testing* [15]. Although Apriori-based approaches also use breath first search, the difference between our framework and them is two-fold. First, we take each graph as the process unit while the latter take each frequent subgraph as the process unit. Second, our framework employ a isomorphism-testing-free approach to generate new subgraphs. Apriori-based approaches have considerable overhead for join and isomorphism testing operations. Therefore, most of existing work applies

pattern-growth approaches to extract frequent subgraphs. The most widely used pattern-growth approaches is gSpan [19], in which the authors introduce a novel coding technique, to restrict the extensions for each frequent subgraph from partial vertexes. Besides, it also provides a novel mechanism to avoid redundant extensions to generate the same frequent subgraphs. gSpan has been widely used to extract frequent subgraphs in graph index construction techniques [20], [3], [2]. Other pattern-growth algorithms, such as MOFA [1], FFSM [6], SPIN [7] and GASTON [13], adopt various coding techniques and optimization strategies to reduce the enumeration space. There are also quite a few studies on extracting frequent subgraphs using disk-based algorithms. In [17], Wang et al. propose an index structure, called the ADI, to support a disk-based extension for existing techniques. However, for processing large scale graph data, the cost of deserializing graphs from external storage to main memory still contribute to a bottleneck according to our experiment.

Recently, there is a rapidly growing trend to design and implement graph algorithms in the distributed graph-parallel systems, including Pregel [12], GraphLab [10], etc. These systems regard each vertex in the graph as the process unit. The vertex can receive/send messages from/to the neighbors, and modify its own status based on the received messages. In particular, these systems are mainly designed for the applications involving implicit or explicit dependencies among the process units, such as PageRank and shortest path computation. However, in our application scenario, we extract the frequent subgraphs from a large number of small graphs. Each graph is taken as the process unit and there does not exist any dependency among different graphs. Therefore, the reason of not using Pregel-like system is two-fold. First, our problem is embarrassingly parallel, which can be efficiently supported by MapReduce. Second, MapReduce shows better performance than Pregel and GraphLab in our scenario. As the mappers and reducers do not need to communicate with each other in the process, MapReduce reduces the cost of synchronizations. Besides, there exists extensive amount of work that aims to implement the typical database operators via MapReduce framework, including theta-join [14], multi-way join [18], and similarity join [16], [11]. All of these work is orthogonal to our work.

## III. Preliminaries

In this section, we first formally describe the problem definition and then present a brief review of gSpan [19]. Table I summarizes the notations used throughout this paper.

### A. Frequent Subgraph Extraction

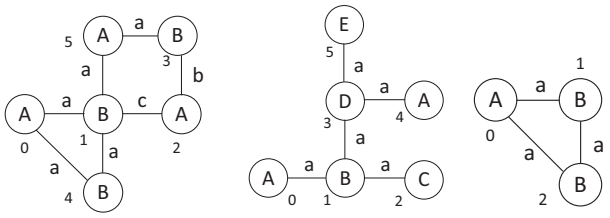
For ease of presentation, we model each complex structure as an undirected, labeled graph. Typically, a labeled graph $g$ is able to be represented as a quadruple $(V_g, E_g, L_g, l_g)$, in which $V_g$ is the set of vertices, $E_g$ is the set of undirected edges, $L_g$ is a set of labels, and $l_g$ is a labeling function that maps every vertex and edge to a single label in $L_g$. Given a graph $g$, we use $l_g[v]$ to denote the label of vertex $v$ and $l_g[u, v]$ to denote the label of edge $(u, v)$ connecting vertices $u$ and $v$ in $g$. For sake of brevity, let $g.id$ represent the identifier of graph $g$.

TABLE I
Symbols and Definitions

| Symbols | Definitions |
|---|---|
| $D$ | a collection of graphs |
| $g$ | a graph in $D$ |
| $V_g$ | the set of vertices in $g$ |
| $E_g$ | the set of edges in $g$ |
| $g.ID$ | the graph identifier of $g$ in $D$ |
| $l_g[v]$ | the label of vertex $v$ in $g$ |
| $l_g[u, v]$ | the label of edge $(u, v)$ connecting $u$ and $v$ in $g$ |
| $s$ | a subgraph |
| $s \subseteq g$ | $s$ is a subgraph of $g$ (or $g$ is a supergraph of $q$) |
| $g_e^s$ | a subgraph in $g$ built by embedding $e$ and subgraph $s$ |
| $s.D$ | an ID list of graphs in $D$ containing $s$ |
| $T$ | the frequent threshold |
| $F_i$ | all frequent size-$i$ subgraphs in $D$ |
| $F_i^g$ | subgraphs in $F_i$ that a graph $g$ contains. |

Given two graphs $g$ and $s$, we can verify whether $s$ is a subgraph of $g$ (i.e., $g$ is a supergraph of $s$ or $g$ contains $s$) by performing *subgraph isomorphism*, which is defined as follows:

**Definition 1. (Subgraph Isomorphism $\subseteq$)** *Given a graph $s = (V_s, E_s, L_s, l_s)$ and a graph $g = (V_g, E_g, L_g, l_g)$, $s$ is said to be subgraph isomorphic to $g$ (denoted as $s \subseteq g$) if and only if there exists an injective function $f : V_s \to V_g$ such that (1) $\forall v \in V_s$, we can have $f(v) \in V_g$, and $l_s(v) = l_g(f(v))$; (2) $\forall (u, v) \in E_s$, we can have $(f(u), f(v)) \in V_g$, and $f_s[u, v] = f_g[f(u), f(v)]$.*




(a) $g_1$ (b) $g_2$ (c) $g_3$

Fig. 2. An Example of Subgraph Isomorphism

**Example 1. (Subgraph Isomorphism)** *Figure 2 shows an example of three undirected, labeled graphs. According to Definition 1, we can find that $g_3 \subseteq g_1$ since there exists an injective function $f$: $\{0 \to 0, 1 \to 1, 2 \to 4\}$, while $g_3 \nsubseteq g_2$.*

Particularly, given two graphs $s$ and $g$, if $s \subseteq g$ and $|V_s| = |V_g|$, then we say $s$ is *graph isomorphic* to $g$. That is, graph isomorphism is a special case of subgraph isomorphism.

**Definition 2. (Embedding)** *Given a subgraph $s$ and a graph $g$, suppose $s \subseteq g$ and $f : V_s \to V_g$ is an injective function that satisfies the requirements in Definition 1. An* **embedding** *$e$ is a sequence of vertices in $V_g$ that are mapped to the vertices of $V_s$ using $f$, i.e., $e = [f(v_1), \ldots, f(v_i), \ldots, f(v_{|V_s|})]$, where*

$v_i \in V_s$ *and* $f(v_i) \in V_g$.

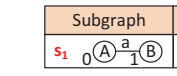
| Subgraph | Embeddings | | | Subgraph | Embeddings | |
|---|---|---|---|---|---|---|
| $s_1$  | [0 1] | [0 4] | [5 1] [5 3] | $s_2$  | [1 4] | [4 1] |

Fig. 3. An Example of Embeddings

Figure 3 shows two subgraphs $s_1$, $s_2$ and the embeddings in $g_1$ (shown in Figure 2(a)) for them, respectively. In practice, an embedding acts like an injective function. Given a subgraph $s$ and a graph $g$, we can build a unique subgraph $g_e^s$ in $g$ based on an embedding $e$ such that $g_e^s$ is graph isomorphic to $s$. It enables us to fast identify all isomorphic subgraphs to $s$ in $g$ by maintaining all these possible embeddings.

A subgraph $s$ is called *connected* if for any two vertices $u, v \in V_s$, there exists a path from $u$ to $v$ in $s$. Given a graph collection $D$ and a subgraph $s$, the *posting list* of $s$ is defined as an ID list of graphs in $D$ containing $s$, i.e., $s.D = \{g.id | g \in D, s \subseteq g\}$. Given a subgraph $s$, $s$ is said to be *frequent* if $|s.D| \geq T$, where $T$ is called *frequency threshold* which is a user-defined number. In many applications, users are more concerned with the frequent recurring components of graphs [17]. Hence, in this paper, we focus on identifying the frequent subgraphs that are connected.

**Problem Definition:** Given a graph collection $D$ and a frequency threshold $T$, the problem of *frequent subgraph extraction* is to identify all connected subgraphs which are contained in at least $T$ graphs of $D$.

In the remainder of this paper, a subgraph that we mention is connected. Additionally, we $F_i$ to denote all frequent size-$i$ subgraphs in $D$ and $F_i^g$ to denote the frequent size-$i$ subgraphs that a graph $g$ contains. Trivially, $F_i^g \subset F_i$.Given a size-$i$ subgraph $s$, we say $s'$ is a *child* of $s$ when $s \subseteq s'$ and $|E_{s'}| = i+1$; similarly, we say $s'$ is a *parent* of $s$ when $s' \subseteq s$ and $|E_{s'}| = i-1$. An *enumeration* operation of a size-$i$ subgraph $s$ is to *enumerate* all children of $s$ by appending one edge from every vertex in $V_s$ if possible.

### *B. gSpan and the Minimum DFS Code*

gSpan [19] is the most widely used approach to extract frequent subgraph in the centralized systems. As a pattern-growth approach, it was proposed with two common objectives: (i) avoiding redundant enumeration for the same subgraph and (ii) reducing the enumeration spaces during the extension of frequent subgraphs by appending all possible one edges.

| Order | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Sequence | (0,1,A,a,B) | (1,2,B,a,C) | (1,3,B,a,D) | (3,4,D,a,A) | (3,5,D,a,E) |

Fig. 4. An Example of DFS Code

To tackle both problems, gSpan proposes DFS coding technique to translate each subgraph into a sequence of DFS codes. Each DFS code is an edge sequence, generated by performing a depth first search (DFS) on the edges of the subgraph, i.e., edges in the DFS code are ranked by the discovery order in the DFS search. Accordingly, each vertex in the subgraph is also marked by its discovery order. In the DFS coding technique, an edge in the DFS code is represented by a 5-tuple, $(i, j, l_s[i], l_s[i,j], l_s[j])$, where $l_s[i], l_s[i,j], l_s[j]$ are the labels of vertex $i$, edge $(i,j)$, and vertex $j$ in the subgraph $s$, respectively. We show a DFS code consisting of five ordered edges in Figure 4 for the graph $g_2$ shown in Figure 2(b). Since there exist multiple depth first search trees for a given subgraph $s$, $s$ can have many DFS codes. Therefore, a lexicographic order is used so that every two DFS codes can be compared with each other [19].

**Lemma 1. (Minimum DFS Code)***[19] Given a size-$i$ subgraph $s$, let parent$(s).min$ be its parent in $F_{i-1}$ with the minimum DFS code. Then, $s$ generated by parent$(s).min$ takes the minimum DFS code.*

Initially, each frequent size-1 subgraph is generated with its minimum DFS code. This generation is able to be implemented by scanning the dataset once. The minimum DFS codes of other frequent subgraphs are able to be generated based on Lemma 1. By introducing the minimum DFS code technique, gSpan is able to avoid redundant enumeration and restrict the extension of a frequent subgraph in a proper way.

**Lemma 2. (Redundant Enumeration Elimination)***[19] Given a frequent subgraph $s$, children of $s$ will be enumerated if and only if it takes the minimum DFS code.*

Lemma 2 eliminates redundant enumeration for the other isomorphic subgraphs in order to avoid many unnecessary computations. Consider the graph $g_2$ shown in Figure 2(b) as an example. Subgraph $(0,3,5)$ of $g_2$ can be extended from either $(0,3)$ or $(3,5)$. Since subgraph $(0,3,5)$ that has been extended from $(3,5)$ does not take the minimum DFS code, no enumeration is required in this case. On the other hand, the same subgraph extended from $(0,3)$ takes the minimum DFS code, and hence, enumeration of all its children will be performed by extending one edge. In this manner, we can ensure that enumeration for all isomorphic subgraphs is executed only once.

**Lemma 3. (Enumeration Restriction)***[19] Given a frequent subgraph $s$ associated with the minimum DFS code, suppose $L$ and $R$ are the last and first discovery vertices in the depth first search. Let $P_{R\to L}$ be the path from $R$ to $L$. We enumerate children of $s$ by appending one edge in one of the following two cases: (1)* backward extension: *from $L$ to a vertex in $P_{R\to L}$; (2)* forward extension: *from a vertex in $P_{R\to L}$ to a new vertex.*

Lemma 3 restricts the enumeration of a frequent subgraph by extending one edge from its partial vertices instead of the complete vertices. For example, enumerating children of $g_2$ shown in Figure 2 constrains the extension of edges from vertices 0, 3, 5. We use $Enu(s)$ to denote the children of $s$ by extending one edge using Lemma 3. After generating all distinct size-1 subgraphs, the details of extracting all frequent subgraphs and their posting lists based on each size-1 subgraph

**Algorithm 1: gSpan**$(D, F, s)$

```
if s.code is not the minimum DFS code of s then
    return;
F ← F ⋃ {s} // F is the set of frequent subgraphs ;
compute Enu(s);
foreach s' ∈ Enu(s) do
    if |s'.D| ≥ T then
        gSpan(D, F, s');
```

are shown in Algorithm 1.

A *baseline* approach to parallelizing gSpan[3] using MapReduce is to partition all frequent size-1 subgraphs to reducers evenly and run Algorithm 1 in each reducer. Although the baseline approach is easy to implement, it has two drawbacks. On the one hand, it causes the load skew problem among the reducers. The extension space for deriving the frequent subgraphs is not evenly distributed. Typically, the majority of frequent subgraphs are generated based on very few frequent size-1 subgraphs with the smallest minimum DFS codes. This is because, for a new generated frequent subgraph $s$ based on the other frequent size-1 subgraphs, we often do not need to enumerate $s$ according to Lemma 2. On the other hand, when the dataset cannot be held in main memory, each enumeration of a frequent size-$i$ subgraph to size-$(i+1)$ subgraphs incurs a large number of deserializing operations by loading the graphs into main memory, resulting in an I/O bottleneck.

## IV. Overview of the Framework

In order to overcome the limitations of simply parallelizing gSpan-like approaches, we propose a framework, called MRFSE (MapReduce based frequent subgraph extraction), to iteratively extract frequent subgraphs using breadth first search, i.e., at the $i^{\text{th}}$ iteration ($i \geq 1$), we identify $F_{i+1}$ and their posting lists. Figure 5 shows the overview of the framework at the $i^{\text{th}}$ iteration. Typically, our framework employs a generation-and-verification mechanism at each iteration using a single MapReduce job.

The generation, involving the map phase only, aims to generate a superset of $F_{i+1}$, which is shown on top of Figure 5. In the map function, the input key is a graph ID, and the input value is a superset of frequent size-$i$ subgraphs that this graph contains. In our design, each size-$i$ subgraph $s$ is associated with a set of embeddings from a graph $g$, which are used to facilitate generation of new size-$(i+1)$ subgraphs from $s$. *In what follows, we assume that a subgraph is associated with a set of embeddings whenever there is no ambiguity in our discussion.* We generate size-$(i+1)$ subgraphs contained in $g$ based on these size-$i$ subgraphs. Details of how to generate size-$(i+1)$ subgraphs for each graph is presented in the next section. Since a frequent size-$(i+1)$ subgraph cannot be generated based on a non-frequent size-$i$ subgraph, before proceeding the generation, we filter all non-frequent size-$i$ subgraphs of $g$ based on $F_i$, which is shown on bottom of Figure 5 and highlighted in the rectangle with a dashed line. In our case, $F_i$ is generated at the $(i-1)^{\text{th}}$ iteration and pre-loaded in the map setup phase at the $i^{\text{th}}$ iteration. The output key and output value in the map function are the graph ID and the newly generated size-$(i+1)$ subgraphs. To enable the iterations, the output of the map phase at the $i^{\text{th}}$ iteration is taken as the input of that at the $(i+1)^{\text{th}}$ iteration. Obviously, union of size-$(i+1)$ subgraphs generated in all graphs constitute a superset of $F_{i+1}$.

[3]Although we focus on gSpan algorithm, the baseline approach can easily be extended to any other pattern-growth approach.

**Lemma 4.** *[19] Given subgraphs $s_1$, $s_2$, if the minimum DFS codes of $s_1, s_2$ are equal, then $s_1$,$s_2$ are graph isomorphic.*

Based on Lemma 4, all isomorphic subgraphs take the same minimum DFS code that is a sequence of vertex and edge labels in essence. Therefore, a straightforward approach verifies size-$(i+1)$ subgraphs generated in the map phase as follows. By shuffling each pair (minimum DFS code, <size-$(i+1)$ subgraph, graph ID>) to the reducers, in each reduce function, we collect each frequent size-$(i+1)$ subgraph by counting the number of graph IDs with the same minimum DFS code, and build the posting list of graph IDs for this subgraph. However, in this way, it incurs unnecessary I/O and communication overhead in that we have to shuffle a large amount of embeddings associated with subgraphs while these embeddings are useless for the verification. To tackle this problem, we decompose the output in the map function into two parts. The first part consists a pair (graph ID, size-$(i+1)$ subgraph) which is then taken as the input of the map phase at the $(i+1)^{\text{th}}$ iteration. The second part, highlighted in the rectangle with a dashed line, consists a pair (minimum DFS code, graph ID) which is then taken as the input of the reduce phase. Clearly, in this way, we can save both I/O and communication cost by omitting the embeddings.

The above iteration goes on until no further frequent subgraphs are generated. In order to utilize the MapReduce framework to support parallel computation, we split the dataset into partitions at the beginning and each partition will be held and processed by a separate mapper during all the iterations. Regarding the generation-and-verification mechanism used in our framework, an important observation is that that the cost of generation task is obviously more expensive than that of the verification task (it only counts the graph IDs in the reduce function). Therefore, we balance the workload in term of the generation task. A straightforward approach to partitioning the graph dataset is to apply random partitioning, i.e., randomly assigning graphs to different partitions. In this way, each partition could take the approximately equal number of graphs. However, this approach may incur load imbalance among mappers. Imagine an extreme case that size of all graphs in one partition are very large while that of graphs in the other partitions are small. Clearly, the mapper that involves in this partition will take expensive generation cost and contributes to a bottleneck. As described in the next section, our generation scheme is based on the extension of edges. Therefore, we

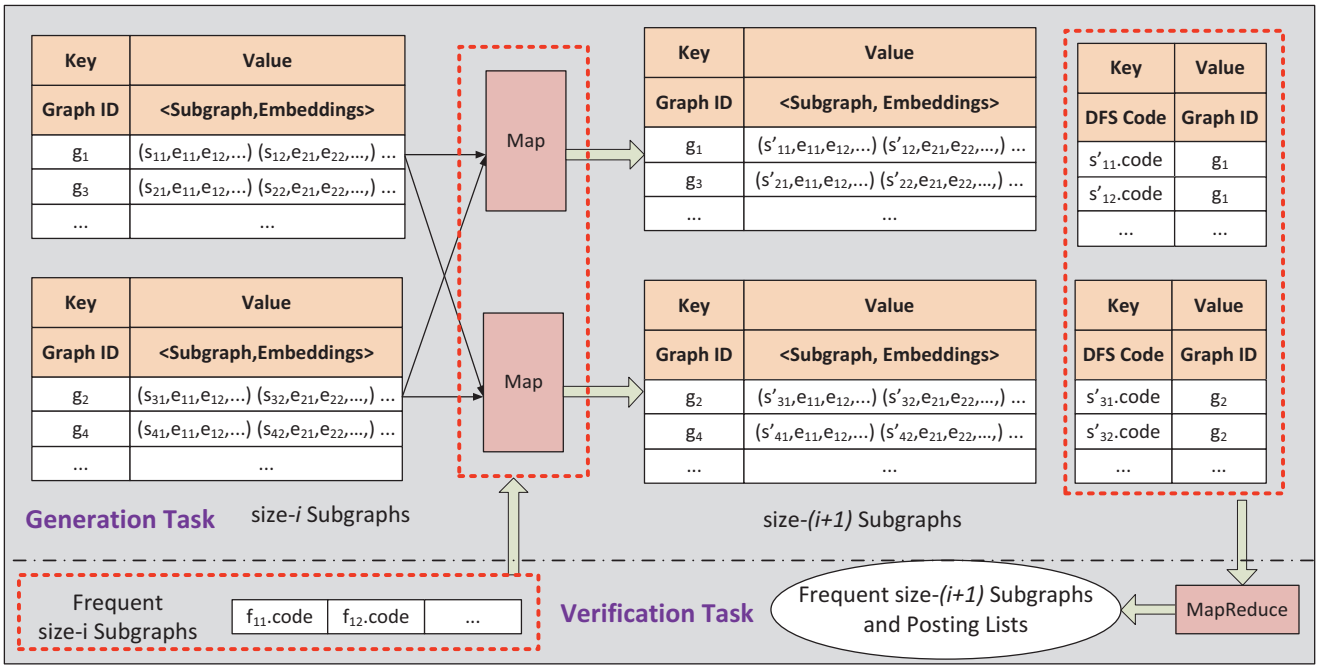



Fig. 5. Overview of the Framework to Extract Frequent Subgraphs

propose the equal size partitioning to split the graph dataset, i.e., assign each graph to the partition with the minimum number of edges so that each partition takes the approximately equal number of edges. Later in the experimental section, we will show that equal size partitioning outperforms random partitioning significantly.

Subsequently, we will focus on the details of generating a superset of $F_{i+1}^g$ based on $F_i^g$ for each graph in our framework.

## V. Generation of New Subgraphs

In general, we can employ two strategies: (1) *join-based enumeration* and (2) *extension-based enumeration* that follow the ideas of Apriori-based approaches and pattern-growth approaches, respectively to perform the generation.

Join-based enumeration generates a size-$(i+1)$ subgraph by joining two size-$i$ subgraphs once they share common size-$(i-1)$ subgraph. Regarding the Apriori-based approaches, it is expensive to identify the common size-$(i-1)$ subgraph for a join pair. However, in our case, as we maintain the embeddings within each graph $g$ for all frequent size-$i$ subgraphs that $g$ contains, a join pair is easily identified based on their embeddings. Specifically, given two size-$i$ subgraphs $s$ with its embedding $e$ and $s'$ with its embedding $e'$, we join $g_e^s$ and $g_{e'}^{s'}$ if they share common $i-1$ edges ($i \geq 2$) or a vertex ($i = 1$). The new size-$(i+1)$ subgraph with its embedding can be generated by joining $g_e^s$ and $g_{e'}^{s'}$. To save both computation and storage cost, for any new generated size-$(i+1)$ subgraph $s$ with their embeddings, it is necessary to eliminate duplicate embeddings once $g_e^s = g_{e'}^s$. Taking embedding [4 1] associated with subgraph $s_2$ in Figure 3 for example, we remove it as it is redundant with embedding [1, 4]. Due to the space limitation, we do not elaborate the details of employing join-based enumeration in that the cost of performing join is more expensive than that of performing extension. Therefore, in this section, we mainly focus on how to employ *extension-based enumeration*, to generate $(i+1)$-subgraphs.

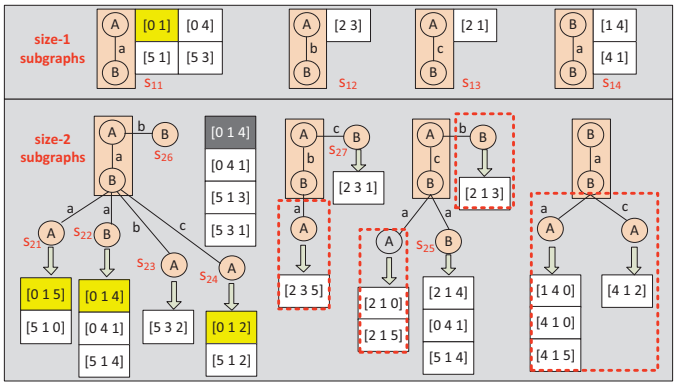



Fig. 6. An Example of Extension-based Enumeration

*For ease of presentation, given a subgraph $s$ from $F_i^g$, let $s.E$ denote the embeddings in $g$ that are associated with $s$.* As extension-based enumeration follows the ideas of pattern-growth approaches, we employ gSpan to generate the new subgraphs and their minimum DFS codes. In comparison with gSpan-like approaches, both edges and size-$i$ subgraphs in extensions must be contained in $g$. This could help us eliminate duplicates without any isomorphism-testing operations.

First, it is easy to identify $F_1^g$ and the embeddings for each frequent size-1 subgraph by scanning the dataset once using MapReduce. For a better understanding, the entire size-

1 subgraphs and their embeddings of $g_1$ in Figure 2 are shown on top of Figure 6. Next, given a subgraph $s$ and all its embeddings in $g$, we shall discuss how we can enumerate all children of $s$ by appending one edge in $g$ efficiently. Consider an example by taking $s_{11}$ shown in Figure 6 as $s$ and $g_1$ in Figure 2 as $g$. According to Lemma 3, we first identify the vertices in path $P_{R\to L}$ for $s$, which are vertex 0 labeled with $A$ and vertex 1 labeled with $B$. Since there exist only two vertices, we do not need to perform the backward extension. We begin from the vertex 1 of $s$ to conduct the forward extension by appending a new edge. Since we have already identified all embeddings in $g$ for $s$, we sequentially check the vertex $v$ in each embedding $e$ where $v$ in $g_e^s$ is mapped to vertex 1 of $s$. Specifically, we first collect all frequent edges which start from $v$ but not contained in $g_e^s$, and append the edge one by one. For example, for embedding [0 1] (emphasized in yellow of embbedings of $s_{11}$ in Figure 6), starting from vertex 1, we can append edges $(1,5),(1,4),(1,2)$, based on which three new size-2 subgraphs are generated and the corresponding embeddings are highlighted in yellow in the figure. Similarly, we can perform the forward extension for the other embeddings associated with the same subgraph. After extending one edge from vertex 1, we perform with the forward extension from the vertex 0 of $s$. For embedding [0 1], we append one edge $(0,4)$ that starts from 0. A new size-2 subgraph is generated and the corresponding embedding is highlighted in black, and the complete extension for $s$ in $g$ is shown at the bottom of Figure 6.

**Lemma 5. (Isomorphism-free Verification)** *Given a subgraph $s$ with an embedding $e$, and another subgraph $s'$ with an embedding $e'$, if $|E_{g_e^s}| = |E_{g_{e'}^{s'}}|$, and $\forall(i,j) \in E_{g_e^s}$, $(i,j) \in E_{g_{e'}^{s'}}$, then $g_e^s$ and $g_{e'}^{s'}$ are the identical subgraphs. Therefore, $s$ is graph isomorphic to $s'$.*

To avoid unnecessary enumeration of the same subgraphs, we need to remove redundant subgraphs with non-minimum DFS code. Take the subgraph $s_{12}$ shown in Figure 6 as an example. By appending edge $(3,5)$ for embedding [2 3], we can generate a subgraph $s$ with embedding [2 3 5]. Although we can perform either a graph isomorphism testing of $s$ on the generated subgraphs, or a minimum DFS code verification of $s$, these two operations are expensive. In practice, we can verify whether $s$ is a redundant subgraph simply based on Lemma 5. For example, by constructing the subgraph from embedding [2 3 5] of $s$, we find that it is identical to the subgraph constructed from embedding [5 3 2] of subgraph $s_{23}$ shown in Figure 6. Hence $s$ is subgraph isomorphic to $s_{23}$, and we do not need to enumerate $s$.

According to Lemma 1, for each subgraph $s$, $s$ generated by $parent(s).min$ takes the minimum DFS code. Hence, we rank the subgraphs in $\mathbb{F}_i^g$ in the ascending order of their minimum DFS codes. When a subgraph is verified as a duplicate based on Lemma 5, we can safely discard this subgraph and its embeddings since the associated DFS code is not minimum. After ranking and enumerating all 1-subgraphs, we can detect the other redundant subgraphs and their embeddings surrounded by dashed red lines shown in Figure 6.

**Algorithm 2: E-Enumeration($F_i^g$)**

```
1  F^g_{i+1} ← ∅; genG ← ∅;
2  sort subgraphs of F^g_i in the ascending order of minimum DFS codes;
3  E ← collect distinct edges from F^g_i;
4  foreach s ∈ F^g_i do
5      foreach extension do
6          foreach e ∈ s.E do
7              S ← ext(e, E, extension);
8              foreach s' ∈ S do
9                  let e' be an embedding of s'.E;
10                 if !contain(genG, g^{s'}_{e'}) then
11                     genG ← genG ∪ {g^{s'}_{e'}};
12                     if !contain(F^g_{i+1}, s'.code) then
13                         F^g_{i+1} ← F^g_{i+1} ∪ {s'};
14                     else
15                         s̄ ← get(F^g_{i+1}, s');
16                         s̄.E ← s̄.E ∪ s'.E;
17 return F^g_{i+1};
```

The extension-based enumeration algorithm is outlined in Algorithm 2. At first, we initialize $\mathbb{F}_{i+1}^g$, and a hash set $genG$ which maintains all distinct $g_e^s$ for each new subgraph $s \in \mathbb{F}_{i+1}^g$ associated with an embedding $e \in s.E$ (line 1). We then sort the subgraphs in $F_i^g$ in the ascending order of their minimum DFS codes, and collect the distinct edges from their minimum DFS codes (line 2–3). For each subgraph $s$ in $F_i^g$, we enumerate all children of $s$ by appending one edge from every possible vertex, which is described in Lemma 3 (line 4–16). For each possible vertex $v$, we sequentially check the embeddings of $s$ and extend one edge from the mapping vertex $v'$ of each embedding to $v$ by probing the edges starting from $v'$ in $\mathbb{E}$ (line 7). For each newly generated subgraph $s'$, according to Lemma 5, if $g_{e'}^{s'}$ is contained in $genG$, we can verify that the associated DFS code of $s'$ is not the minimum. We note that $s'$ might have already been contained in $F_i^g$, since the same $s'$ is generated by $s$ but with another embedding. In this case, we merge their embeddings together (line 14–16). Finally, we return $\mathbb{F}_{i+1}^g$.

## VI. Experimental Evaluation

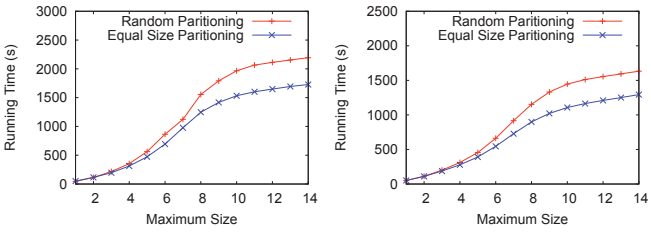
We evaluate the performance of the proposed algorithms on a 72-node cluster[4]. Each node in the cluster has one Intel Xeon X3430 2.4 GHz Quad core Processor, 8GB of RAM, two 500GB SATA hard disks and gigabit ethernet. Each node is equipped with the CentOS 5.5 operating system, Java 1.6.0 with a 64-bit server VM, and Hadoop 0.20.2. We configure

[4]http://awan.ddns.comp.nus.edu.sg/ganglia/

one node to act as the name node and job tracker and the remaining nodes as the data nodes and task nodes.

We compare three approaches to extract frequent subgraphs: (i) *Baseline* approach runs with a cache with 1GB size to buffer the graph data, (ii) *MRFSE-J* and (iii) *MRFSE-E* extract frequent subgraphs using MRFSE framework and generate new subgraphs using join-based and the extension-based enumeration respectively. We conduct the experiments over a real dataset, **PubChem**[5]. PubChem contains one million chemical structures. Each graph has 23.98 vertices, 25.76 edges, 3.5 distinct vertex labels, 2.0 distinct edge labels on average, and the total number of distinct vertex labels and distinct edge labels is 81 and 3, respectively. The size of PubChem dataset is 434 MB.

We evaluate the performance of the proposed approaches in terms of running time and I/O cost. Since the input at each iteration is the output at the previous iteration, for simplicity, we only collect the output at each iteration as well as the original data size. To compare the efficiency and the effectiveness of our proposed framework, we adjust the maximum size ($maxL$) of the frequent subgraphs in $\{2, 4, 6, 8, \mathbf{10}, 12, 14, 16\}$, the frequency threshold ($T$) in $\{4\%, 6\%, 8\%, \mathbf{10}\%, 12\%\}$ and the number of compute nodes in $\{10, \mathbf{20}, 30, 40\}$. By default, we set $T$ to 10% and the number of compute nodes to 20. In the default settings, the largest size of frequent subgraphs is 16.



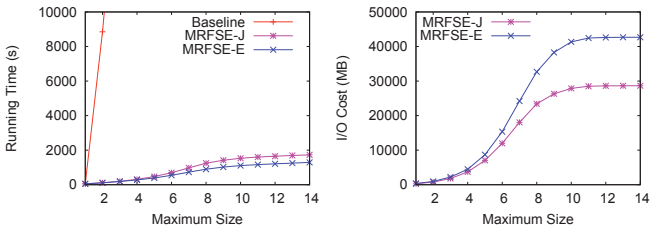

(a) MRFSE-J (b) MRFSE-E

Fig. 7. Data Partitioning

We first study the effect of partitioning strategies on MRFSE-J and MRFSE-E: (1) randomly partitioning and (2) equal size partitioning. The results are shown in Figure 7. We can observe that performance of using equal size partitioning strategy is better than that of using random partitioning in both MRFSE-J and MRFSE-E. This is because in both extended-based enumeration and join-based enumeration, we generate new subgraphs in term of edges. Therefore, the workload can be more balanced among the compute nodes if we partition the data based on edges. In the remainder of experiments, we employ equal size partitioning strategy to partition the data.

We then compare the performance of proposed algorithms and show the results in Figure 8. We only show partial results of the *Baseline* approach for sake of clarity since the running time is 57,713s even when the maximum size is just 5. We observe that MRFSE-J and MRFSE-E are significantly faster than the *Baseline* by two orders of magnitude because of

[5]http://pubchem.ncbi.nlm.nih.gov

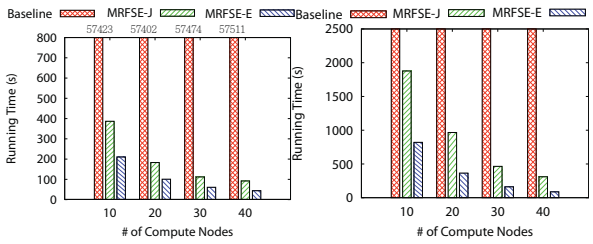



(a) Running Time (b) I/O Cost

Fig. 8. Comparison of the Proposed Approaches

two major reasons. First, *Baseline* has the severe load skew problem. We observe that the majority of frequent subgraphs are generated based on very few frequent size-1 subgraphs with many recursions while the others stop much earlier with less number frequent subgraphs. Moreover, although the cache is able to alleviate the repeated reading the graphs from the disks, deserializing process of a graph is time consuming and is executed millions of times.

As we can see from Figure 8, MRFSE-J incurs less I/O cost. This is because given a graph $g$ and a subgraph $s$, MRFSE-J maintains only one embedding of each subgraph $sg$ in $g$ that is graph isomorphic to $s$ while MRFSE-E maintains all possible embeddings of $sg$. However, MRFSE-E performs better than MRFSE-J in that the cost of join operation is more expensive than that of extension operation.

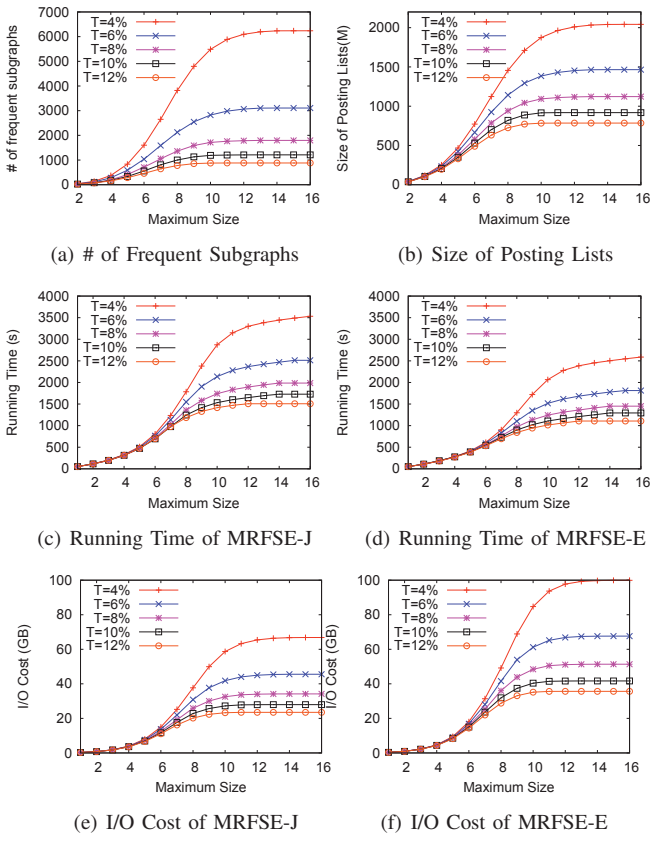



(a) $maxL$=5 (b) $maxL$=16

Fig. 9. Speedup

Figure 9(a) plots the results by varying the number of compute nodes when the maximum size is 5. To analyze the speedup more accurately, we omit the cost introduced by the MapReduce framework itself, i.e., the time of running MapReduce without any computation. We observe that the performance of *Baseline* remains almost the same in that the reducer generating the maximum number of the frequent subgraphs becomes the bottleneck. The running time of MRFSE-E and MRFSE-J decreases when the reducer number varies. Nevertheless, there is an obvious trend of diminishing returns. We show the results for extracting all frequent subgraphs in Figure 9(b), which follows the same trends.

We study the performance of MRFSE by varying the frequency threshold. Figure 10 plots the number of frequent subgraphs, the size of their posting lists, and the cost of MRFSE-J and MRFSE-E. Curves with different symbols demonstrate the trend for different frequency thresholds. From 10(a), we




(a) # of Frequent Subgraphs (b) Size of Posting Lists

(c) Running Time of MRFSE-J (d) Running Time of MRFSE-E

(e) I/O Cost of MRFSE-J (f) I/O Cost of MRFSE-E

Fig. 10. Effect of Frequency Threshold

observe that the number of frequent subgraphs first increases exponentially and then increases smoothly after an inflection point emerges. This is because frequent subgraphs with smaller size tend to have many extensions that are then able to be verified as frequent subgraphs as well. Nevertheless, frequent subgraphs with larger size may not have many valid extensions with respect to the frequency threshold. The position of the inflection point relies on the frequency threshold. Generally, a small frequency threshold leads to a larger inflection point. Another interesting finding is that the number of frequent subgraphs drops superlinearly when we enlarge the frequency threshold. Not surprisingly, the size of their posting lists shown in Figure 10(b) follows the same trends to that of the number of frequent subgraphs. The running time and the I/O cost for both MRFSE-J and MRFSE-E follow the same trends. When reducing the frequency threshold, we can find that the running time in MRFSE-J increases more significantly than that in MRFSE-E while the I/O cost is the exact opposite.

## VII. Conclusion

In this paper, we studied the problem of efficiently supporting frequent subgraphs extraction in a distributed environment, particularly, using the MapReduce framework. We presented a framework, called MRFSE, to iteratively extract frequent subgraphs on a MapReduce like platform such as Hadoop. We introduced a equal size partitioning strategy in order to balance the workload. We proposed an isomorphism-testing-free approach for generating new subgraphs by properly maintaining how frequent subgraphs are mapped within each graph. Extensive experiments conducted on our in-house clusters demonstrate that the techniques built using proposed framework are scalable and efficient.

## Acknowledgments

The work in this paper was supported by the Singapore Ministry of Education Grant No. MOE2010-T2-2-104 under the project name of epiC. We would also like to thank the anonymous reviewers for their insightful comments.

## References

[1] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 51–58, 2002.
[2] J. Cheng, Y. Ke, and W. Ng. Efficient query processing on graph databases. *ACM Trans. Database Syst.*, 34(1), 2009.
[3] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD Conference*, pages 857–872, 2007.
[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[5] J. Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
[6] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.
[7] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, pages 581–586, 2004.
[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] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.
[11] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *PVLDB*, 5(10):1016–1027, 2012.
[12] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.
[13] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.
[14] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD Conference*, pages 949–960, 2011.
[15] K. H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 5th edition, 2002.
[16] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD Conference*, pages 495–506, 2010.
[17] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *KDD*, pages 316–325, 2004.
[18] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *SoCC*, page 12, 2011.
[19] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
[20] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD Conference*, pages 335–346, 2004.
[21] X. Yan, P. S. Yu, and J. Han. Graph indexing based on discriminative frequent structure analysis. *ACM Trans. Database Syst.*, 30(4):960–993, 2005.