Cloud Service Placement via Subgraph Matching

Bo Zong[†], Ramya Raghavendra[*], Mudhakar Srivatsa[*], Xifeng Yan[†], Ambuj K. Singh[†], Kang-Won Lee[*]

[†]*Dept. of Computer Science, University of California at Santa Barbara, CA, USA*
{bzong, xyan, ambuj}@cs.ucsb.edu

[*]*IBM T.J. Watson Research Center, Yorktown Heights, NY, USA*
{rraghav, msrivats, kangwon}@us.ibm.com

***Abstract*—Fast service placement, finding a set of nodes with enough free capacity of computation, storage, and network connectivity, is a routine task in daily cloud administration. In this work, we formulate this as a subgraph matching problem. Different from the traditional setting, including approximate and probabilistic graphs, subgraph matching on data-center networks has two unique properties. (1) Node/edge labels representing vacant CPU cycles and network bandwidth change rapidly, while the network topology varies little. (2) There is a partial order on node/edge labels. Basically, one needs to place service in nodes with enough free capacity. Existing graph indexing techniques have not considered very frequent label updates, and none of them supports partial order on numeric labels. Therefore, we resort to a new graph index framework,** Gradin**, to address both challenges.** Gradin **encodes subgraphs into multi-dimensional vectors and organizes them with indices such that it can efficiently search the matches of a query's subgraphs and combine them to form a full match. In particular, we analyze how the index parameters affect update and search performance with theoretical results. Moreover, a revised pruning algorithm is introduced to reduce unnecessary search during the combination of partial matches. Using both real and synthetic datasets, we demonstrate that** Gradin **outperforms the baseline approaches up to 10 times.**

I. Introduction

Dynamic graphs have been applied to model frequently updated data-center networks [24]. Node/edge on these graphs contain numerical values describing network states, such as machines' CPU/memory usage and links' available bandwidth. These numerical values are frequently updated to reflect network dynamics [24].

Given a cloud residing in a data-center network, it is important to place services into the cloud so that users' requirements are satisfied [2], [14]. Cloud service placement can be naturally formulated as *dynamic subgraph matching* queries: Given a large dynamic graph G with numerical node/edge labels and a smaller query graph Q with user-specified numerical node/edge labels (*e.g.,* required computation and communication resources), the goal is to return a set of subgraphs of G, each of which is structurally isomorphic to Q, and whose node/edge labels are compatible with Q (*i.e.,* the corresponding nodes/edges can provide enough computation and network resources). Consider the following example.

Example 1: In Fig. 1, an accounting service is defined as a query graph. Numerical labels on nodes represent the amount of memory required for diverse types of servers, while labels on edges represent the amount of bandwidth required among servers. Given such a query graph, a cloud administrator is obliged to find a subgraph from a dynamic cloud graph to place the service. A qualifying subgraph should be structurally

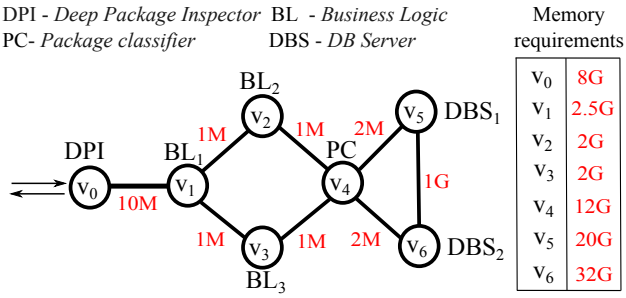


Fig. 1. A user-defined accounting service with diverse memory and bandwidth requirements on nodes and edges

isomorphic to the query graph, and its nodes/edges should have enough resources to satisfy the specified requirements.

The aforementioned subgraph matching problem not only brings a new, critical graph query application, but also new challenges to existing techniques. First, existing graph indexing techniques, *e.g.,* [9], [17], [32], [40], [45], [46], are focused more on graph structure and fixed node/edge labels, while data-center networks usually have quite stable structure, but more frequent label updates. Although some incremental graph indexing algorithms are available [40], they are not designed to accommodate frequent label updates (*e.g.,* $10-50\%$ node/edge labels are updated every 10 seconds [24]). Second, existing techniques supporting approximate or probabilistic graph matching can hardly handle partially ordered numerical labels in service placement. For example, a server with 32G free memory can accommodate a service requiring 1G memory, even when these two values are very different from each other. These challenges motivate us to develop a new graph indexing mechanism that is specific for data-center networks and service placement.

In this work, we propose a graph index framework Gradin (Graph index for dynamic graphs with numerical labels) to address the above challenges. Gradin encodes subgraphs into multi-dimensional vectors and organizes them such that it can efficiently search the matches of a query's subgraphs and combine them into full matches of the query graph. First, we propose a multi-dimensional index that supports vector search and is able to handle frequent updates. Different from existing indices that are efficient for index updates but suffer from low pruning power, we develop a search algorithm that preserves the pruning power. Moreover, we present a theoretical analysis of how index parameters affect update and search performance. Second, we propose pruning techniques to enable a fast combination of partial matches of a query graph. A naïve solution is costly, when the number of matches

for a query's subgraphs is large. Using a minimum cover of subgraphs in a query and subgraphs' *fingerprints*, our method is able to significantly improve query response time.

Our main contribution is the identification of a key application of graph query in cloud computing, and the formulation of a new graph index framework that accelerates subgraph matching on dynamic graphs of numerical labels. To the best of our knowledge, this is the first study on this topic. Using both real and synthetic datasets, we demonstrate that Gradin outperforms the baseline approaches up to 10 times.

II. Problem definition

For the sake of simplicity, we examine undirected graphs where only nodes have single labels, and assume that labels are normalized [2], [25]. Our work can be extended to general graphs where both nodes and edges have multiple labels.

Data graph. A *data graph* G is represented by a tuple (V, E, A), where (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges; and (3) $A : V \rightarrow [0, 1]$ is a function that assigns a numerical label to each node $u \in V$.

Query graph. A *query graph* is defined as $Q = (V', E', A', p)$, where (1) V' and E' are a node set and an edge set, respectively; (2) $A' : V' \rightarrow [0, 1]$ is a labeling function; and (3) p is a predicate function that assigns a predicate for each node $u' \in V'$. In other words, p specifies search conditions: $p(u', u)$ defines a predicate $A'(u')$ op $A(u)$, where (1) u' is a node in a query graph; (2) u is a matched node of u' in a data graph; and (3) op is a comparison operator drawn from the set $\{<, \leq, =, \neq, \geq, >\}$. Here, we focus on the predicate $A'(u') \leq A(u)$.

Compatibility. Given two graphs $H_1 = (V_1, E_1, A_1)$ and $H_2 = (V_2, E_2, A_2)$, H_2 is *compatible* with H_1, if (1) H_1 is structurally graph isomorphic to H_2 by a bijective function $f : V_1 \rightarrow V_2$; and (2) $\forall u \in V_1$, $A_1(u) \leq A_2(f(u))$.

Graph update in data centers. In data-center networks, the most frequent updates come from numerical values on nodes and edges. (1) Update frequency is close to, or even higher than query frequency, and (2) a large portion $(10 - 50\%)$ of nodes/edges in a data graph are frequently updated. In contrast, the physical connections of nodes are relatively stable. Thus, topological update is not the focus of this study.

Definition 1 (Dynamic subgraph matching): Given a data graph G with its node/edge labels frequently updated, a query graph Q, and an integer r, Dynamic subgraph matching for cloud service placement is to find up to r compatible subgraphs of Q from G.

The number of returned subgraphs r is decided by applications. To place a service into a cloud, we might need more than one compatible subgraphs in order to optimize the performance of the whole cloud [20]: (1) some compatible subgraphs might not be available due to the network dynamics and the query processing delay; and (2) cloud administrators might be interested in optimizing other performance metrics, such as network congestion [1] and transmission cost [20].

Dynamic subgraph matching is a hard problem. By a reduction from the well-known *subgraph isomorphism* problem [12], the problem can be shown to be NP-complete. On the other hand, for small query graphs and sparse data-center networks, it is possible to build indices to solve the matching problem in a practical manner. In the following sections, we investigate the feasibility and principles of building a graph index on networks with partially ordered numerical labels, and study how to speed up index update while preserving search speed. We will also discuss how to optimize query processing.

III. An overview of Gradin

Fragment. A **fragment** $h = (V_h, E_h, A_h)$ is a connected subgraph from a graph $H = (V, E, A)$, where (1) $V_h \subset V$ and $E_h \subset E$ are a node set and an edge set, respectively; and (2) $\forall u \in V_h$, $A_h(u) = A(u)$.

In particular, we use g to denote a fragment extracted from a data graph G, referred as a *graph fragment*, and use q to denote a fragment extracted from a query graph Q, referred as a *query fragment*. To facilitate index building and searching, we represent fragments by *fragment coordinates*.

Fragment coordinate. Given the *canonical labeling* [39] of a fragment h of k nodes, the fragment coordinate, denoted by $x(h)$, is a k-dimensional vector, where the i-th dimension contains the information about the i-th visited node in its canonical labeling. In particular, the i-th dimension of a fragment coordinate could either be the *id* or the *label* of the i-th visited node in the canonical labeling.

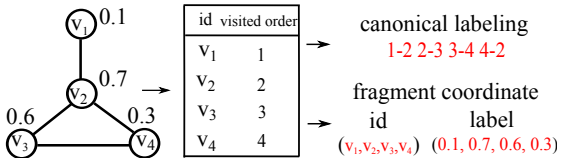


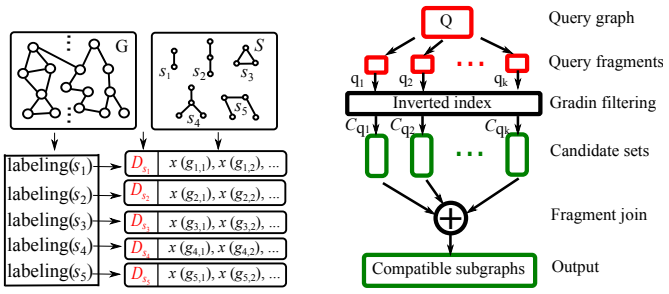
Fig. 2. A fragment with its canonical labeling (top right) and fragment coordinates (bottom right)

Fig. 2 shows an example of fragment coordinates. (1) The canonical labeling of the fragment is shown in the top-right corner, where i denotes the i-th visited node, and $i-j$ denotes a visit from the i-th visited node to the j-th visited node. (2) The coordinate (v_1, v_2, v_3, v_4) stores node id information, and the coordinate $(0.1, 0.7, 0.6, 0.3)$ stores node label information. Note that each graph has a unique canonical labeling, and the fragment coordinates specify how to assign ids and labels to nodes and edges.

In addition, we refer to the coordinate of node id information as an *id coordinate*, and the coordinate of label information as a *label coordinate*. When the context is clear, the term fragment coordinate is used without ambiguity.

Let Q be a query graph. A region in a data graph is worth searching, if for any query fragment q, the region contains graph fragments that are compatible with q; otherwise, we can safely exclude that region from search. Gradin implements this idea in two components: *offline index construction* and *online query processing*.

Offline index construction. Let G be a data graph and $\mathcal{S}$ be a graph structure set that is decided by existing structure selection algorithms [40]. For each structure $s \in \mathcal{S}$, we use subgraph mining technique [39] to search $\mathcal{D}_s$, which contains



(a) Offline index building (b) Online query processing

Fig. 3. Gradin consists of two parts: (1) offline index building and (2) online query processing

all the graph fragments in G of the structure s. Using the canonical labeling of structures as keywords, graph fragments are organized by inverted indices. To further optimize search and storage, graph fragments in the inverted indices are denoted by their fragment coordinates.

Fig. 3(a) illustrates an example of the offline index construction. G is a data graph at the top-left corner, and $\mathcal{S}$ at the top-right corner is a set of structures we aim to index. First, $\mathcal{D}_{s_i}$ – all graph fragments corresponding to structure s_i – are mined from the data graph. Using an inverted index, the canonical labeling of s_i points to $\mathcal{D}_{s_i}$. In particular, the inverted index stores fragment coordinates.

Online query processing. As shown in Fig. 3(b), given a query graph Q, Gradin searches the compatible subgraphs of Q in three steps. (1) **Decomposition**. Gradin decomposes Q into query fragments, whose structures have been indexed. (2) **Filter**. For each query fragment q, Gradin first finds the set of graph fragments sharing the same structure, and only returns those fragments that are compatible with q. The returned set of graph fragments is also referred to as *candidate set* $\mathcal{C}_q$. (3) **Join**. Gradin conducts fragment join among candidate sets. By stitching fragments in candidate sets, Gradin returns compatible subgraphs for Q.

To provide a good query processing performance, Gradin needs to enable fast search and join at the filter and the join phase. We need to address two challenges: (1) frequent updates and (2) the large search space for fragment join.

Frequent updates. It is preferable to build a search index for $\mathcal{D}_s$ (the graph fragments of structure s), especially when the size of $\mathcal{D}_s$ is very large (*e.g.,* the number of 3-star, a star structure with three branches, in a medium-size data-center network reaches 10M). However, when node labels are frequently updated, it is non-trivial to build the desired search index. On the one hand, a sophisticated search index (*e.g.,* R-tree [16]) offers strong pruning power; however, it is costly to perform updates [29]. On the other hand, a simple search index (*e.g.,* an inverted index) provides faster index update speed; however, the index's pruning power decreases and longer time is needed for filtering the remaining candidates. In Sect. IV, we discuss how we address this challenge.

The large search space for fragment join. This challenge includes two aspects. First, since the size of a candidate set can be very large, a naïve join algorithm will be extremely slow. Second, since a query graph might be decomposed into dozens of query fragments (*e.g.,* a 10-star query graph contains 110 query fragments of no more than 2 edges), it is preferable to select a subset of query fragments that covers the query graph and minimizes the amount of redundant intermediate results. In Sect. V, we propose a two-step algorithm that prunes the large search space for fragment join.

IV. Fragment Index

In this section, we present an index FracFilter that efficiently processes frequent updates, and preserves search speed.

A. Naïve solutions

Let $\mathcal{D}_s$ be the set of graph fragments of structure s. There are three basic options to build a search index for $\mathcal{D}_s$: (1) R-tree variant, (2) inverted index, or (3) grid index.

R-tree variant. One might build R-tree variants [5], [16] based on fragment label coordinates to offer good pruning power. However, when node labels are frequently updated, the search trees will process a massive number of update operations. Update operations on R-tree variants are costly [29]. Even though with sophisticated insertion strategies R-tree-like search structures can process around $16,000$ updates per second [29], they will spend a considerable amount of time in processing index updates. For example, in a data-center network of $3,000$ nodes, when 30% node labels are updated, in the case of graph fragments of the structure 3-star, more than 5M label coordinates need to be updated. Therefore, the state-of-the-art R-tree variant might take more than 5 minutes to update the index. As queries need to wait on index update, the throughput of the whole system will suffer.

Inverted index. One might consider fragment id coordinates, and build inverted indices on id coordinates with the canonical labeling as keywords. To prune unpromising graph fragments for a query fragment q_s, we have to verify all graph fragments in $\mathcal{D}_s$. Since updates on node labels never change id coordinates, these indices take little index update cost; however, the size of $\mathcal{D}_s$ is usually large (*e.g.,* tens of millions), so a thorough scan will slow down query processing.

Grid index. One might apply grid indices to allow affordable update operations [3], [28]. The general idea is as follows. (1) The multi-dimensional space of label coordinates are partitioned into grids. (2) The fragments in the same grid are managed with a light-weight data structure (*e.g.,* list). (3) If a fragment label coordinate is updated, update operations will be conducted only when the updated coordinate moves out of the original grid. (4) When a query fragment arrives, we issue a range query based on its label coordinate: (a) mark those fragments in the grids that are fully covered by the range query as candidates; and (b) verify those fragments in the grids that are partially covered by the range query. Note that the search speed depends on the amount of time taken by verification. If we apply a naïve method that compares the targeted graph fragments with the query fragments, it will take a considerable amount of time. Consider the following example.

Example 2: 10M graph fragments of the structure 3-star (*i.e.,* 4D coordinates) are managed by a grid index with 10,000

grids. Suppose graph fragments are uniformly distributed, each grid covers 1,000 fragments. A 10-star query has 720 query fragments of the structure 3-star. Suppose a query fragment is uniformly distributed, it will partially cover 505 grids in average. Therefore, a query fragment needs up to $4 * 505 * 1000 = 2.02$M comparisons to complete verification, and in total we need up to $720 * 2.02M = 1454.4$M comparisons for one indexed structure.

In Example 2, we consider the case of node labels and single label per node. In a general case of edge labels and multiple labels per node/edge, the number of comparisons will proportionally increase. Therefore, the verification phase will slow down by a large amount of comparisons.

In the following discussion, we introduce an index FracFilter that addresses both update and search issues. FracFilter is a grid-based index that inherits the merit of update efficiency; moreover, we propose a verification algorithm that accelerates search by avoiding redundant comparisons.

B. FracFilter construction

We start with the construction of FracFilter. For an indexed structure, a FracFilter is constructed by two steps: (1) it partitions the label coordinate space into grids, and (2) maps graph fragments into the corresponding grids.

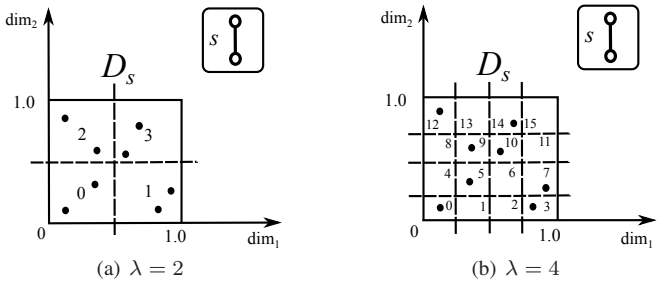


Fig. 4. FracFilters of density 2 (left) and 4 (right): s in the top right corner is the structure of $\mathcal{D}_s$, points are label coordinates, and the integer in each grid is the grid id.

Partition the space into grids. Let λ be a positive integer, called *grid density*, n_s be the number of fragments with structure s, and d be the number of dimensions for fragments' label coordinates. One partition strategy is to uniformly slice label coordinate space into λ parts for each dimension; however, this strategy might result in unacceptable index searching and updating performance when label coordinate distribution is skew. Therefore, we consider to use the empirical distribution of label coordinates in each dimension to partition the space: (1) slice each dimension into λ parts, and each part contains $\frac{n_s}{\lambda}$ fragments; and (2) independently repeat this procedure in each dimension. In this way, we obtain λ^d d-dimensional grids in total. Moreover, each grid is associated with a grid id represented by a base-λ integer. Suppose the i-th dimension of a grid falls into the j-th partition, then the i-th bit of the grid id is j. The advantages of using the above way to assign a grid id include (1) the ease of designing fragment mapping functions, and (2) the ease of avoiding redundant comparisons (discussed in Sect. IV-C).

Fig. 4 demonstrates two FracFilters of density 2 and 4, respectively, on a 2-dimensional space: (1) grids are disjoint, (2) a point (fragment) is covered by one and only one grid, and (3) the whole space is covered.

Map fragments into grids. Let g be a graph fragment, $x(g) = (x_1, x_2, ..., x_k)$ be g's fragment coordinate, and gid be the id of the grid to which g should be mapped. The mapping function is designed as follows. (1) Starting with x_1, if x_1 falls into the j_1-th partition, we set the first bit of gid to be j_1. (2) At the i-th dimension, if x_i falls into the j_i-th partition, we set the i-th bit of gid to be j_i. (3) Repeat this process for all dimensions. we use lists to manage fragments in grids.

The pseudo code of the construction algorithm is shown in Fig. 5 for reference. The above construction algorithm shows that a FracFilter can be constructed in linear time, and the following result indicates the computation complexity.

Input: (1) grid density λ;
(2) the number of dimensions d;
(3) a list of label coordinates of $\mathcal{D}_s$, $frag$;
Output: a FracFilter, $filter$.

1. $grid$.resize(λ^d)
2. $locX$.resize(len($frag$)), $locY$.resize(len($frag$))
3. **for** i in range(0, len(frag))
4. gid = gridID($frag[i]$)
5. $grid[gid]$.append($frag$[i])
6. $locX[i] = gid$
7. $locY[i]$ = len($grid[gid]$) − 1
8. **return** $filter$ ($grid$, $locX$, $locY$, d, λ)

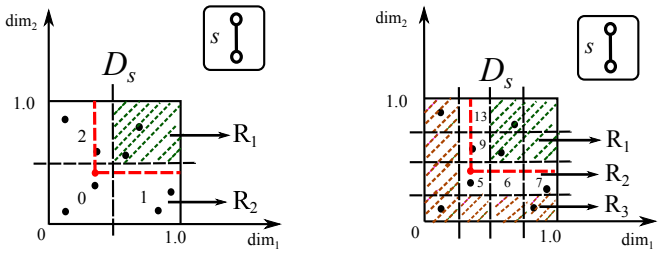
Fig. 5. The Algorithm sketch for constructing a FracFilter

Proposition 1: Let λ be the grid density, n_s be the number of fragments, and d be the number of dimensions of a label coordinate space. We can construct a FracFilter in $O(\max\ (\lambda^d, dn_s))$.

Remark. (1) The construction algorithm will be executed once for each indexed structure. In other words, if we index 5 structures and obtain 5 sets of graph fragments, the construction algorithm will be executed for 5 times, and each run builds a FracFilter for the corresponding structure. (2) We define the grid density λ and divide each dimension into λ partitions for the ease of discussion. Indeed, with little modification, the construction algorithm along with its theoretical results works in the cases where each dimension is divided into a variable number of partitions.

C. Searching in FracFilter

In this section, we present how a FracFilter avoids unnecessary comparisons and accelerates search for a query fragment. When a query fragment arrives, we formulate a range query that is a multi-dimensional box with the query fragment's label coordinate as the bottom corner and $(1.0, 1.0, ..., 1.0)$ as the top corner. The range query divides the label coordinate space into three regions R_1, R_2, and R_3: (1) R_1 contains the grids that are fully covered by the range query; (2) R_2 contains the grids that are partially covered by the range query; and (3) R_3 contains rest of the grids. Fig. 6 gives examples of R_1, R_2, and R_3. In particular, we mark fragments in R_1 as candidates, discard fragments in R_3, and verify fragments in R_2.



(a) $\lambda = 2$ (b) $\lambda = 4$

Fig. 6. The same query fragment (the red dot) requests fragment searching on two FracFilters of density 2 (left) and 4 (right).

For graph fragments in R_2, a naïve verification algorithm blindly makes comparisons in every dimension; however, for some dimensions, comparisons are unnecessary. Consider an example in Fig. 6(b). A query fragment falls into grid 5 (11_4 in base-4 form), and grid $5(11_4)$, $6(12_4)$, $7(13_4)$, $9(21_4)$, and $13(31_4)$ are in R_2. For fragments in grid $5(11_4)$, we have to take all dimensions into consideration for verification. However, for fragments in grid $6(12_4)$ and $7(13_4)$, we only need to consider dim_2 since their label values in dim_1 is surely greater; and similarly, for fragments in grid $9(21_4)$ and $13(31_4)$, we only need to consider dim_1.

Let c_q be the grid where a query fragment falls, and c be a grid in R_2. We obtain the following pruning rule.

Lemma 1: Comparisons at the i-th dimension are necessary, only if the i-th bit of c_q's id equals the i-th bit of c's id.

A natural question is how many comparisons we can avoid from this rule. Suppose that (1) label coordinates of graph fragments are uniformly distributed in grids and (2) a query fragment's label coordinate is uniformly distributed in grids as well, the following result shows the expected number of comparisons a FracFilter makes for verification.

Theorem 1: Given λ, grid density, d, the number of dimensions, and n_s, the number of graph fragments, the expected number of extra comparisons for an arbitrary query fragment is $\frac{dn_s}{\lambda^d}\left(\frac{\lambda+1}{2}\right)^{d-1}$.

Proof: We show the expected number of comparisons a FracFilter requires for an arbitrary query fragment. (1) The probability that a query fragment falls into grid c_q with a base-λ id $a_d a_{d-1} \cdots a_1$ is $\frac{1}{\lambda^d}$. (2) The number of grids in R_2 that need d more comparisons for each graph fragment is 1 (c_q itself), the number of grids that need $d-1$ more comparisons is $\sum_{j=1}^{d}(\lambda - a_j - 1)$, and in general, the number of grids that need $d-k$ more comparisons for each graph fragment is $\sum_{(j_1,j_2,\cdots,j_k)}\prod_{i=1}^{k}(\lambda - a_{j_i} - 1)$ where $(j_1, j_2, \cdots, j_k)$ enumerates all k-combinations of $(1, 2, \cdots, d)$. Taking summation over all possible c_q, the number of grids that need $d-k$ more comparisons is $\binom{d}{k}\lambda^{d-k}\left[\frac{\lambda(\lambda-1)}{2}\right]^k$. Thus, the expected number of comparisons is derived by

$$\begin{aligned} EX &= \frac{n_s}{\lambda^d}\frac{1}{\lambda^d}\sum_{k=1}^{d} k \cdot \binom{d}{k}\lambda^k \left[\frac{\lambda(\lambda-1)}{2}\right]^{d-k} \\ &= -\frac{n_s}{\lambda^d}\frac{(\lambda-1)^{d+1}}{2^d}\left[\left(\frac{2}{\lambda-1}+1\right)^d\right]' \\ &= \frac{dn_s}{\lambda^d}\left(\frac{\lambda+1}{2}\right)^{d-1}. \end{aligned}$$

Therefore, Theorem 1 is proved. ■

Consider an inverted index that scans the whole set of graph fragments taking dn_s comparisons, and a naïve verification algorithm on a grid index that scans $\frac{dn_s}{\lambda^d}\left[\left(\frac{\lambda+1}{2}\right)^d - \left(\frac{\lambda-1}{2}\right)^d\right]$ in average (the derivation is similar to the proof of Theorem 1). The ratio from the number of comparisons made by a FracFilter to the number by the inverted index is $\frac{d}{\lambda^d}\left(\frac{\lambda+1}{2}\right)^{d-1}$; similarly, the ratio from a FracFilter to the naïve verification algorithm is $\frac{2(\lambda+1)^{d-1}}{(\lambda+1)^d-(\lambda-1)^d}$. In other words, when $d = 7$ (a 3-star fragment with single node and edge labels), and $\lambda = 25$, this ratio is lower than 0.005 for the inverted index, and lower than 0.18 for the naïve verification algorithm.

Remark. (1) Theorem 1 demonstrates that a FracFilter of a larger grid density has a faster pruning speed in average. In particular, when $\lambda^d \leq \frac{n_s}{2}$, the first derivatives of the above ratios will be negative so that the *efficiency* will increase if λ increases; moreover, when $\lambda^d \leq \frac{n_s}{2}$, the second derivatives of the above ratios will be positive so that the *efficiency gain* will diminish if λ increases. (2) Although in practice graph fragments might not strictly uniformly distributed in grids, our experimental results show that FracFilter performs well with both real and synthetic fragment distributions in Sect. VI.

D. Index update in FracFilter

In this section, we discuss the update operations in FracFilter. When a graph fragment is updated, it triggers one of the two events in FracFilter: *bounded* or *migration*. (1) If an update triggers *bounded*, the fragment stays in the same grid. (2) If an update triggers *migration*, the fragment moves out of the old grid, and moves into another one.

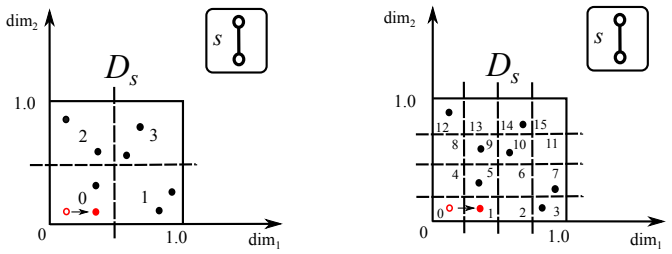
Given a fragment update, FracFilter is updated in two steps: (1) find which grid should accommodate the updated fragment; and (2) decide which event this update triggers and take the corresponding action to update FracFilter. In the second step, if the update triggers event *bounded*, it takes no update operation; if the update triggers event *migration*, it takes two operations: (a) delete the fragment from the old grid's fragment list, and (b) insert the updated fragment into the right grid.

Suppose the label coordinate of an updated fragment is uniformly distributed in the space, we obtain the following complexity result for index update in FracFilter.

Theorem 2: Given d is the number of dimensions and λ is the grid density, FracFilter takes $2(1 - \frac{1}{\lambda^d})$ operations per update in average.

Proof: For an arbitrary update, the probability of staying in the original grid is $\frac{1}{\lambda^d}$. Thus, the expected number of operations an update takes is $0 \cdot \frac{1}{\lambda^d} + 2 \cdot (1 - \frac{1}{\lambda^d})$. ■

Remark. Theorem 2 suggests that a FracFilter of smaller grid density λ is more likely to take fewer update operations. In Fig. 7, we make the same update to the fragment in the bottom-left corner: (1) the update in the FracFilter of $\lambda = 2$ triggers a *bounded* event, and requires no operation (Fig. 7(a)); however, (2) the update in the FracFilter of $\lambda = 4$ triggers a *migration* event requiring a deletion and an insertion on lists (Fig. 7(b)). Indeed, if an update triggers a *migration* event in a FracFilter of smaller grid density, it must triggers a *migration* event in a FracFilter of larger grid density; on the other hand, if an update triggers a *migration* event in a FracFilter of larger grid density, it might only trigger a *bounded* event in a FracFilter of smaller



(a) $\lambda = 2$ (b) $\lambda = 4$

Fig. 7. An update on FracFilter of density 2 (left) and 4 (right), respectively. When a fragment in bottom left corner is updated, it triggers a bounded event on the left, but a migration event on the right.

grid density. Note that a larger grid density λ brings better search performance at the cost of higher memory usage and slower update performance. In practice, one can try different λ, and use the one that best fit the application.

V. Optimize Query Processing

In this section, we discuss how to accelerate subgraph matching at the fragment join phase. Various heuristics have been proposed to join paths or two-level trees in a selected order such that the amount of redundant intermediate results is reduced [31], [46]. Different from these studies, we deal with general subgraphs in this work. In particular, we need to address two critical issues. First, we need to define the join selectivity of general subgraphs, and propose an algorithm to find a set of query fragments that minimizes redundancy. Second, in a considerable number of cases, no matter which join order we apply, a naïve join will take a long time. Consider the following example.

Example 3: C_{q_1}, C_{q_2}, and C_{q_3} are the candidate sets of query fragment q_1, q_2, and q_3, and the size of these candidate sets is uniformly 10^6. In the data graph, there is no matched subgraph, but we do not know it when we conduct the join. In this case, no matter how we place the join order, a naïve join for the first two candidate sets will take 10^{12} comparisons.

In this paper, we propose a two-step method to address the above issue: (1) we use *minimum fragment cover* to find a set of query fragments whose candidate sets potentially involve the minimum amount of redundant intermediate results; and (2) *fingerprint based pruning* is applied to prune redundant comparisons between a pair of candidate sets.

A. Minimum fragment cover

Minimum fragment cover finds a small set of selective query fragments with small candidate sets to reduce computation cost at the join phase, with the constraint that the result of fragment join over this small set of query fragments is equivalent to that over the whole set of query fragments. There are two intuitions behind minimum fragment cover. (1) We only need a subset of query fragments that jointly cover all nodes and edges in the query graph, and we refer to such a subset as a *fragment cover*. (2) As there are multiple ways to select fragment covers, one might prefer to take the one of a smaller search space for fragment join. In the following, we define an optimization problem that implements the above intuition.

Given a fragment cover $\{q_1, q_2, ..., q_k\}$ and their corresponding candidate sets $\{C_{q_1}, C_{q_2}, ..., C_{q_k}\}$, the joint search space size is bounded by $\exp(J)$,

$$J = \log(\prod_{i=1}^{k} |C_{q_i}|) = \sum_{i=1}^{k} \log(|C_{q_i}|).$$

Indeed, one may prefer a fragment cover that optimizes the upper bound J. Therefore, with J as the objective function, we define the minimum fragment cover problem as follows.

Definition 2 (Minimum fragment cover): Given a query graph Q with its whole set of query fragments $\{q_1, q_2, ..., q_l\}$ and their candidate sets $\{C_{q_1}, C_{q_2}, ..., C_{q_l}\}$, a *minimum fragment cover* is a subset of query fragments $\{q_{i_1}, q_{i_2}, ..., q_{i_k}\}$ such that (1) this subset is a fragment cover, and (2) its corresponding J is minimum.

However, the following result indicates that it is difficult to find the *minimum fragment cover* in polynomial time. Instead, one can obtain an approximated solution in polynomial time with approximation guarantee.

Theorem 3: The *minimum fragment cover* problem is NP-complete; however, there exist greedy algorithms with an approximation ratio $O(\ln n)$, where n is the sum of the number of nodes and the number of edges.

Proof: To prove the NP-completeness of the minimum fragment cover problem, one could reduce an arbitrary set cover instance [12] to a minimum fragment cover instance by (1) constructing a depth-1 tree where each leaf maps to an element in the ground set and (2) constructing a smaller depth-1 tree for each set where a leaf maps to an element contained by the set. One can verify that this transformation runs in polynomial time, and a solution to the transformed instance is a solution to the original set cover instance. Since the minimum fragment cover problem is NP, the NP-completeness of the problem follows. Moreover, an arbitrary minimum fragment cover instance can be easily transformed into a set cover instance. Therefore, the approximation guarantee $O(\ln n)$ for set cover [12] can be applied to minimum fragment cover. In sum, the correctness of Theorem 3 is proved. ■

B. Fingerprint based pruning

The intuition of fingerprint based pruning includes two aspects. (1) Given a fragment cover, it is preferable to join fragments in an order such that it results in connected subgraphs at every intermediate step. Indeed, disconnected subgraphs at intermediate step will lead to an explosion of the search space. To obtain a connected intermediate subgraph, two query fragments at any intermediate step have to share a set of nodes. In other words, these overlapping and non-overlapping nodes form the join conditions for graph fragments. (2) Suppose join operations are conducted between two candidate sets C_{q_1} and C_{q_2} where q_1 and q_2 share several common nodes. Let g_i be a graph fragment from C_{q_1}. It is very likely that only a small portion of graph fragments in C_{q_2} share the required common nodes with g_i; meanwhile, only this small portion of graph fragments are worth checking. Therefore, instead of linearly scanning C_{q_2}, it is preferable for g_i to only check those graph fragments of the required common nodes. With this spirit, we propose *fingerprint based pruning* that (1) extracts the required common nodes for fragment join, (2) makes *fingerprints* based on these common nodes, and (3) prunes redundant search if two fragments have different fingerprints.

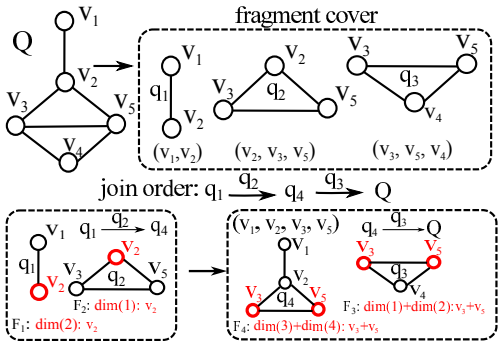


Fig. 8. An example of fingerprint based pruning

To illustrate how we perform fingerprint based pruning, an example is presented in Fig. 8.

First, for a query graph Q, three query fragments q_1, q_2, and q_3 form a fragment cover, and their id coordinates are (v_1, v_2), (v_2, v_3, v_5), and (v_3, v_5, v_4), respectively.

Second, the order of fragment join is as follows. (a) Join the candidate sets of q_1 and q_2; and (b) join the intermediate subgraphs from (a) with the candidate set of q_3.

Third, starting with q_1 and q_2, they share v_2 that is the *second* dimension of q_1's id coordinate, and the *first* dimension of q_2's id coordinate. Thus, for each graph fragment g_i in the candidate set of q_2, the fingerprint of g_i is constructed by the node id at the *first* dimension of its id coordinate. Using fingerprints as keys, graph fragments from q_2's candidate set are organized by an inverted index. Given a graph fragment g_j from C_{q_1}, we first extract its fingerprint by the node id at the *second* dimension of its id coordinate. With its fingerprint, we search q_2's inverted index, and only check those graph fragments sharing g_j's fingerprint.

Fourth, at intermediate steps, a similar procedure is conducted. As shown in Fig. 8, the intermediate query graph q_4 is obtained by joining q_1 and q_2, and we next join q_4 with q_3. The common nodes of q_4 and q_3 are node v_3 and node v_5. v_3 and v_5 are the nodes at the *third* and the *fourth* dimension of q_4's id coordinate; meanwhile, they are at the *first* and *second* dimension of q_3's id coordinate. Similarly, we obtain the fingerprints of q_3's candidate fragments, and organize them by an inverted index. For a graph fragment g_i of q_4, we first extract g_i's fingerprint, locate those graph fragments sharing its fingerprint by q_3's inverted index, and only check those promising graph fragments.

In addition, given a fragment cover, the join order is determined by a heuristic algorithm. We start with the query fragment of the smallest candidate set in the fragment cover. At each step, we select the query fragment that shares common nodes with the query fragments that have been joined. If there exist multiple such query fragments, we select the one with the smallest candidate set. We repeat this process until all the query fragments in a fragment cover are joined.

VI. Experiments

Using real and synthetic data, we conducted three sets of experiments to evaluate the performance Gradin. Both real and synthetic data are used to evaluate Gradin's performance on query processing and index construction/update. The synthetic data is used to study its scalability.

Gradin is implemented in C++. All experiments were executed on a machine powered by an Intel Core i7-2620M 2.7GHz CPU and 8GB of RAM, using Ubuntu 12.10 with GCC 4.7.2. Each experiment was run 10 times. In particular, for query processing, each run includes 100 query graphs. For all experiments, their average results are presented.

A. Experiment setup

Data graphs. We used the following network topologies as data graphs. (1) BCUBE is a network architecture for data centers [15]. We generated BCUBE networks as follows. (a) The number of nodes in the networks ranges from $3,000$ to $15,000$ with step $2,000$; and (b) the average degree of a network is between 18 and 20. In particular, a BCUBE network of $3,000$ nodes was used to compare Gradin with its baselines, and the rest networks were used to evaluate Gradin's scalability. (2) CAIDA[1] dataset contains 122 Autonomous System (AS) graphs, from January 2004 to November 2007. In particular, we used the largest AS graph of $26,475$ nodes and $106,762$ edges to compare query processing and indexing performance between Gradin and its baselines.

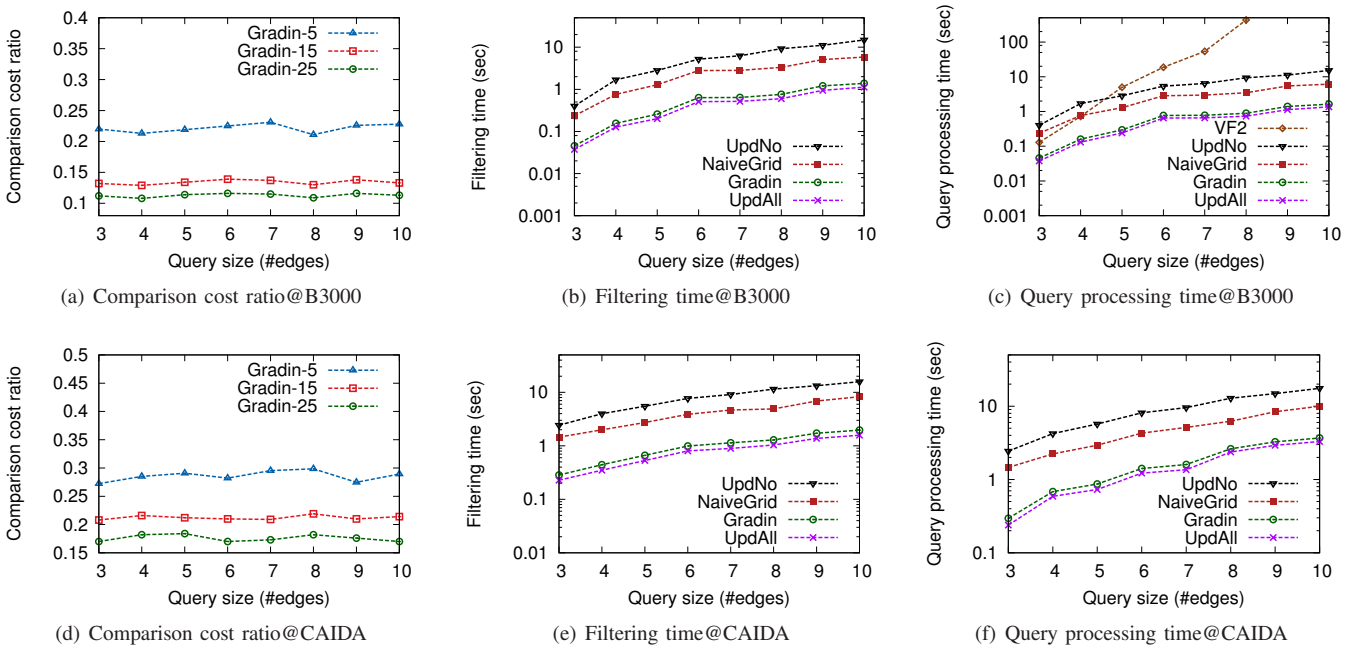
Numerical labels and updates. We obtained the numerical labels and their updates from the ClusterData[2]. It contains the trace data from about 11k machines over about a month-long period in May 2011 [25]. For each machine, we extracted its CPU and memory usage traces, and each trace is represented as a sequence of normalized numerical values between 0 and 1. Moreover, we randomly mapped a cluster machine to a node in a data graph, and obtained numerical labels on nodes along with their updates. In particular, we applied the node labels/updates from ClusterData to the evaluation for *query processing* (Sect. VI-B) and *indexing performance* (Sect. VI-C).

In order to explore how our technique performs on different numerical label distributions, we generated labels and updates from statistical distributions of estimated parameters (using ClusterData as sample data). The generated labels are applied to the evaluation for *scalability* in Sect. VI-D.

Query graphs. As query graphs are usually small [36], [46], we consider all possible connected graphs of 3 to 10 edges as possible queries, and the numerical labels on nodes are randomly drawn from 0 to 1.

Baselines. Five baselines are considered: VF2, UpdAll, NaiveGrid, NaiveJoin, and UpdNo. (1) VF2 [8] is a state-of-the-art subgraph search algorithm without any index. (2) UpdAll indexes fragment label coordinates with multi-dimensional search trees. This index enables fast candidate search; however, update operations are costly on the search tree. In particular, we implemented two versions of search trees: (a) a multi-dimensional binary tree of better update processing performance is used to compare indexing performance in Sect. VI-C; and (b) an R-tree [16] of faster query processing performance is used to compare query processing performance in Sections VI-B and VI-D. (3) NaiveGrid is a grid index with

[1] http://snap.stanford.edu/data/as-caida.html
[2] http://code.google.com/p/googleclusterdata/wiki/ClusterData2011_1



(a) Comparison cost ratio@B3000 (b) Filtering time@B3000 (c) Query processing time@B3000

(d) Comparison cost ratio@CAIDA (e) Filtering time@CAIDA (f) Query processing time@CAIDA

Fig. 9. Query processing performance on B3000 and CAIDA with 100 compatible subgraphs returned

a naïve verification algorithm. (4) NaiveJoin uses FracFilter at the filtering phase, but a naïve join is applied for fragment join. (5) UpdNo is an inverted index where each entry points to a list of fragments of the same structure. It never updates index; however, it requires a large amount of comparisons for searching candidate fragments.

B. Query processing

In the first set of experiments, we investigated the query processing performance of Gradin on the BCUBE graph of 3,000 nodes (B3000) and the largest CAIDA graph (CAIDA).

Since a service placement task might require multiple compatible subgraphs to be returned [1], [20], in the experiments, the number of returned compatible subgraphs r is set to be 5, 10, or 100. Formally, given a graph G, taking a query graph Q as input, we find up to r compatible subgraphs.

Pruning power. We evaluate the pruning power of Gradin, UpdAll, NaiveGrid, and UpdNo, by measuring *comparison cost ratio* and *filtering time*. (1) *Comparison cost ratio* defines the ratio from the amount of label comparisons executed by Gradin to the amount by NaiveGrid. In particular, the *comparison cost ratio* is 100% for NaiveGrid. (2) *Filtering time* defines the total amount of time spent in searching candidate fragments. All the evaluations are conducted by varying query graph size (the number of edges).

On B3000, Fig. 9(a) and Fig. 9(b) present the pruning power of UpdAll, UpdNo, NaiveGrid, and Gradin: Fig. 9(a) presents the comparison cost ratio of Gradin with grid density 5, 15, and 25 (referred to as Gradin-5, Gradin-15, and Gradin-25), respectively, while Fig. 9(b) only presents the filtering time of the Gradin with grid density 25 (it always outperforms the other two variants). On the one hand, when query size increases, the Gradin variant of a greater grid density receives better pruning power; however, the power gain is diminishing when grid density grows. On the other hand, the filtering time of all algorithms increases, when graph query size increases. In particular, (1) as query size increases from 3 to 10, the *filtering time* of Gradin increases from 0.046 to 1.387 seconds, which is close to the performance of UpdAll (later, we will show that UpdAll is 4-10 times slower in index construction and update.); and (2) in terms of filtering time, Gradin is up to 10 times faster than UpdNo, and is around 5 times faster than NaiveGrid.

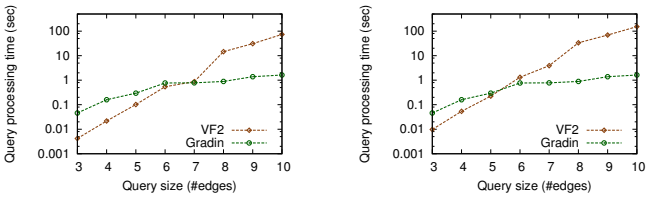
On CAIDA, Fig. 9(d) and Fig. 9(e) present consistent results: (1) Gradin with grid density 25 has the best pruning power, but the power gain is diminishing when grid density grows; and (2) in terms of filtering time, Gradin is close to UpdAll, is 10 times faster than UpdNo, and is up to 5 times faster than NaiveGrid.

In sum, the above results verify our theoretical analysis in Theorem 1.

Query processing time. We evaluate the total *query processing time* of Gradin, UpdAll, UpdNo, NaiveGrid, and VF2. The total *query processing time* includes query decomposition time, filtering time, and fragment join time. In addition, all the evaluations are conducted by varying query graph size (the number of edges).

First, we studied the total query processing time of different techniques, and set the number of returned compatible subgraphs to be 100. Fig. 9(c) and 9(f) present the total query processing time on B3000 and CAIDA, respectively. Since Gradin with grid density 25 always outperforms Gradin with density 5 or 15, we only present the query processing time of Gradin with grid density 25. On both data graphs, we observe two common trends: (1) Gradin processes queries as fast as UpdAll does; (2) Gradin outperforms UpdNo up to 10 times; and (3) Gradin is up to 5 times faster than NaiveGrid. In particular, on B3000, as the query size grows from 3 to 10,

the query processing time of Gradin increases from 0.046 to 1.62 seconds, while UpdNo's query processing time increases from 0.40 to 15.2 seconds, and NaiveGrid's query processing time increases from 0.27 to 6.12 seconds, which achieves up to 10 times and 5 times speedup, respectively; meanwhile, on CAIDA, as the query size grows, the query processing time of Gradin increases from 0.29 to 1.98 seconds, while UpdNo's query processing time increases from 2.44 to 17.5 seconds, and NaiveGrid's query processing time increases from 1.46 to 9.98 seconds, which is around 8 times and 5 times speedup, respectively. Moreover, VF2 cannot scale on both B3000 and CAIDA: (1) on B3000, when the query size is more than 8, VF2 cannot process 100 queries within 12 hours; and (2) on CAIDA, even when the query size is 3, VF2 cannot process 100 queries within 6 hours. In the case of NaiveJoin, within 6 hours, it cannot process 100 queries of size 4 on B3000, and cannot process 100 queries of size 3 on CAIDA (not shown).



(a) 5 subgraphs returned (b) 10 subgraphs returned

Fig. 10. Query processing on B3000 returning 5 or 10 matches

Second, we investigated how VF2, NaiveJoin, and Gradin perform when the number of returned subgraphs is smaller by setting the number of returned subgraphs to be 5 or 10. The query processing time on B3000 is presented in Fig. 10(a) and Fig. 10(b). Two observations are made: (1) VF2 is faster when the query size is small; and (2) Gradin performs better when the query size is relatively large (*e.g.,* ≥ 8 edges). When a query graph is smaller, the number of matched subgraphs in a data graph is larger in general, and the amount of redundant search for VF2 is smaller as well. Since VF2 does not conduct a global pruning as Gradin does, VF2 earns a better performance when the query size is small. However, when a query graph is relatively large, the number of matched subgraphs decreases, and VF2 cannot successfully skip the redundant search. Meanwhile, the pruning power of Gradin is pronounced, especially when the query size gets larger. In particular, when the query size is 10, Gradin performs around 50 times better than VF2 if 5 subgraphs are returned, and around 100 times better if the number of returned subgraphs is 10. On CAIDA, even when the number of returned subgraphs is 5, VF2 cannot process 100 queries of size 3 within 6 hours (not shown); however, Gradin is able to process 100 queries of size 10 within 6 minutes (*i.e.,* 3.6 seconds per query). In the case of NaiveJoin, even when the number of returned subgraphs is 5, within 6 hours, it cannot process 100 queries of size 4 on B3000, and cannot process 100 queries of size 3 on CAIDA (not shown).

Remarks. (1) When a query graph's structure is indexed, Gradin can directly answer the query without fragment join. For example, we can answer query graph of size 3 on B3000 after the filtering phase. (2) We indexed fragments of no more than 3 edges on B3000, while the indexed fragments on CAIDA have no more than 2 edges. On B3000, as the query size increases from 4 to 10, the fragment join time increases from 0.004 to 0.20; meanwhile, on CAIDA, the fragment join time increases from 0.011 to 1.72, when the query size grows from 3 to 10. This shows that our fragment join algorithms effectively prune redundant search. The experiments also show that as less sophisticated structures are indexed, it usually takes more time on fragment join, since we need more query fragments to cover a query graph, which results in a larger number of join operations. (3) As the fragment join time is largely reduced by our optimization techniques, the importance of reducing *filtering time* is highlighted. Using FracFilter, Gradin successfully reduces filtering time.

TABLE I
INDEX CONSTRUCTION TIME (SEC)

Data graph	UpdAll	Gradin-5	Gradin-15	Gradin-25
B3000	85.3	18.5	19.6	21.1
CAIDA	61.6	17.1	17.4	17.5

TABLE II
INDEX SIZE (MB)

Data graph	UpdAll	Gradin-5	Gradin-15	Gradin-25
B3000	1644	607	610	615
CAIDA	1486	583	585	589

C. Indexing performance

In the second set of experiments, we investigated the indexing performance of Gradin on the BCUBE of 3000 nodes (B3000) and the largest CAIDA graph (CAIDA). In particular, we built Gradin variants of grid density 5, 15, and 25, referred to as Gradin-5, Gradin-15, and Gradin-25.

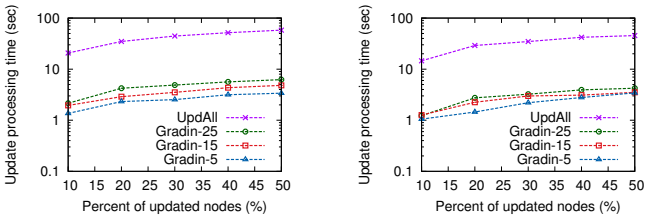
Index construction. We evaluate *index construction time* and *index size* of UpdAll and Gradin variants. *Index construction time* measures the amount of time spent in building index after graph fragments are mined, and we separately report the amount of time for mining graph fragments.

On B3000, we built Gradin variants and UpdAll based on fragments of no more than 3 edges. It took us 370 seconds to mine fragments from B3000, obtaining fragments of 5 different structures. Index construction time and index size of UpdAll and Gradin variants are shown in Tables I and II, respectively. All Gradin variants take less time and space to build indices on B3000.

On CAIDA, we built Gradin variants and UpdAll based on fragments of no more than 2 edges, and mined fragments of 2 different structures including one-edge and two-edge paths in 268 seconds. The performance results are reported in Tables I and II, respectively. All Gradin variants take less time and space to build indices on CAIDA as well.

Index update. We evaluate index update performance of Gradin and UpdAll by measuring their *update processing time* while varying the *percent of updated nodes*. Given a data graph, the *percent of updated nodes* is the percentage of nodes whose labels are updated. Since a node might appear in multiple fragments, if the label of a node is updated, multiple fragments might be simultaneously updated as well. Given a set of updated nodes, we first find the corresponding updated fragments, and then apply those fragment updates to a graph

index. The time spent in updating a graph index is referred to as *update processing time*.



(a) Index update@B3000 (b) Index update@CAIDA

Fig. 11. Update processing time on B3000 and CAIDA

Fig. 11(a) presents the update processing time of UpdAll, Gradin-5, Gradin-10, and Gradin-25 on B3000: (1) Gradin variants outperform UpdAll, and (2) when the percent of updated nodes increases, the update processing time of all algorithms increases. In sum, Gradin variants are about 10-20 times faster than UpdAll on B3000.

Fig. 11(b) shows update processing time on CAIDA. The trends are consistent with those shown in Fig. 11(a), and Gradin variants are 13-20 times faster than UpdAll on CAIDA.

Remarks. (1) As we consider more complex structures to build graph indices, it usually takes more time on index update. For both UpdAll and Gradin, a more complex structure increases the number of dimensions of the corresponding search index, which in turn increases update complexity. (2) When Gradin-5 processes updates, about 40% of updates trigger *bounded* events, which requires no update operations on Gradin. However, this ratio drops from 40% to 15%, as we apply Gradin-25. Even though Gradin-25 takes more time to process updates that trigger *migration* events, it is still efficient since update operations are merely insertions and deletions on lists. In contrary, UpdAll processes all updates with costly insertions and deletions on multi-dimensional search trees. It is Gradin's partial update strategy and inherent low-cost update operations that make Gradin more efficient on index update.

D. Scalability

In the third set of experiments, we investigated the scalability of Gradin by varying the size of a BCUBE graph.

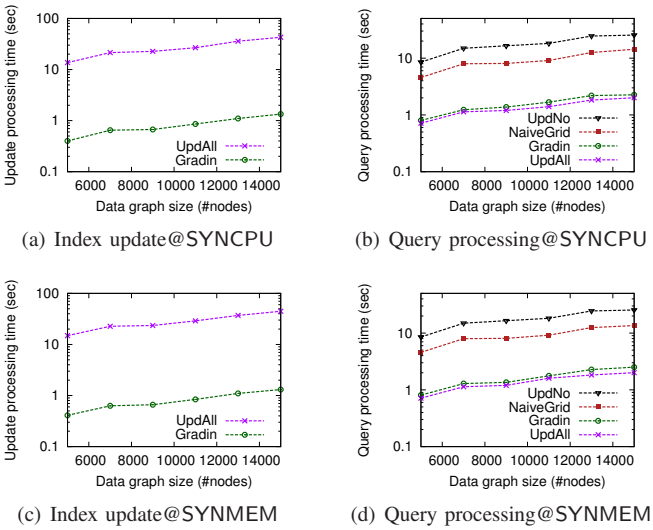
Constrained by security policies and communication latency, a service placement usually considers a few racks in a data center[3] [14], [20]. Suppose there are 40 machines in each rack[4], we ranged the number of racks in a data center from 125 to 375[4], and generated six BCUBE graphs of $5,000$, $7,000$, $9,000$, $11,000$, $13,000$, and $15,000$ nodes.

Node labels and updates were generated from beta distributions $B(a, b)$ of estimated parameters. The reasons why we employed beta distributions include (1) per machine attributes (*i.e.,* available CPU time/memory) in ClusterData are normalized between 0 and 1, while beta distribution is one of the widely-used distributions that could generate random numbers in such an interval; and (2) there exist efficient algorithms to estimate parameters a and b from sample data[5]. In particular,

[3]http://msdn.microsoft.com/en-us/library/windowsazure/jj717232.aspx
[4]http://news.cnet.com/8301-10784_3-9955184-7.html
[5]http://en.wikipedia.org/wiki/Beta_distribution

TABLE III
CONSTRUCTION TIME AND INDEX SIZE OF Gradin

Graph size	5K	7K	9K	11K	13K	15K
T_{mine} (sec)	49	89	142	227	357	513
T_{index} (sec)	2.8	4.0	4.3	6.2	7.0	7.8
Size (MB)	65	100	111	176	202	229



(a) Index update@SYNCPU (b) Query processing@SYNCPU

(c) Index update@SYNMEM (d) Query processing@SYNMEM

Fig. 12. Scalability on BCUBE graphs of 5K - 15K nodes

the estimated parameters based on machine CPU time are $a \approx 7.91$ and $b \approx 7.03$, and the estimated parameters based on machine memory are $a \approx 2.42$ and $b \approx 2.73$. Moreover, the corresponding distributions are referred to as SYNCPU and SYNMEM, respectively.

In addition, the grid density of Gradin is set to 25, and the indexed structures have no more than 2 edges.

Index construction. We evaluated index construction time and index size of Gradin by varying data graph size. As shown in Table III, When the graph size increases, for a data graph of $15,000$ nodes with more than $150,000$ edges, Gradin is constructed in less than 10 minutes (including fragment mining), taking 229MB memory.

Index update. We evaluated the index update performance of Gradin by varying data graph size. In particular, the percent of updated nodes is fixed to 30%. As seen in Fig. 12(a) and 12(c), Gradin takes less than 2 seconds to process more than 9M updates, outperforms UpdAll in all cases, and the resulting speedup is up to 20 times. Indeed, when data graph size gets larger, Gradin can efficiently process updates.

Query processing. We evaluated the query processing performance of Gradin by varying data graph size. In particular, the query size (the number of edges) is fixed to 7, and Gradin returns the first 100 distinct compatible subgraphs. Since VF2 and NaiveJoin cannot scale on the query graphs of size 7 (they cannot process 100 query graphs within 6 hours), Fig. 12(b) and 12(d) only reports the query processing time of Gradin, UpdAll, NaiveGrid, and UpdNo. When a data graph has 15K nodes with more than 150K edges, Gradin can process a query graph of size 7 within 3 seconds in average. In terms of query processing time, Gradin is close to UpdAll, and outperforms

UpdNo and NaiveGrid in all cases, with up to 8 times speedup.

E. Summary

We summarize the experimental results as follows. (1) Gradin is efficient for index updates. Gradin outperforms the baseline algorithm UpdAll, and the speedup is up to 10 times on our datasets. (2) The search algorithm and fragment join algorithms in Gradin accelerate query processing. Gradin outperforms the baseline algorithms VF2, NaiveGrid, NaiveJoin, and UpdNo. While VF2 cannot scale on larger query graphs, Gradin processes all query graphs in 4 seconds, matches the query processing speed of UpdAll, and is up to 10 times and 5 times faster than UpdNo and NaiveGrid, respectively.

VII. Related Work

Subgraph matching. Subgraph matching is one of the most critical primitives in many graph applications, such as pattern search in protein-protein interaction networks [42], [47], chemical compounds [17], [40], program invocation graphs [45], [46], and communication networks [14], [36]. In these applications, subgraph matching queries have been defined in their own ways, and a variety of techniques have been proposed to resolve the corresponding challenges.

Subgraph matching queries are usually defined by the NP-hard *subgraph isomorphism* [12]. Although branch-and-bound based algorithms, such as Ullmann's algorithm [34] and VF2 [8], were proposed to improve the search efficiency, these algorithms still cannot scale on large graphs [9], [40].

To accelerate subgraph-isomorphism based subgraph matching, a variety of graph indices have been proposed. Among them, substructure based indexing is the most widely adopted framework. Although the indexed substructures could be diverse, such as paths [13], [46], trees [26], [42], or general subgraphs [6], [18], [31], [38], [40], they follow a common spirit: a small data graph, or a small region in a large data graph, is worth searching, if it contains a query graph's substructures. In addition, He et al. [17] proposed *closure-tree*, an R-tree-like search index, which only checks the data graphs that are similar to a query graph and prunes unpromising search for dissimilar data graphs.

In addition to exact matches, an inexact subgraph matching query is a more general and flexible graph primitive. Given a query graph, it aims to find similar subgraphs in data graphs, where similarity is usually defined by *graph edit distance* [41]. To speed up query processing, substructure based indices are widely adopted, such as Grafil [41], SAPPER [45], and many others [19], [22], [32], [37].

Note that all the graph indices discussed above are designed for static data graphs with discrete node/edge labels; however, in our problem setting, node/edge labels are dynamically updated, and these labels are numerical values. The above indices cannot efficiently process subgraph matching queries on dynamic graphs with numerical labels: (1) frequent graph updates result in serious index maintenance issues; and (2) numerical node/edge labels cannot be naturally supported by most of those indices. In this paper, we propose a graph index addressing these issues.

Meanwhile, variants of subgraph matching queries have been studied. Tong et al. [33] proposed a proximity-based score function, and the top-k subgraphs of the highest score are returned as output. Cheng et al. [7] relaxed matching conditions: instead of an adjacent node pair, a pair of reachable nodes in a data graph is also eligible to match an edge on a query graph. Similarly, in [47], a pair of nodes that satisfy a pre-defined distance constraint is eligible to match an edge on a query graph as well. Moreover, Fan et al. [10], [11] defined subgraph matching by *graph simulation*, and Yuan et al. [44] studied the subgraph matching problem on uncertain graphs with categorical labels. Different from them, our work focuses subgraph-isomorphism based queries where data graphs have dynamically changing numerical labels.

Closer to our work are the studies from Wang et al. [4], Mondal et al. [21] and Fan et al. [9]. In [9], incremental algorithms are proposed to answer a fixed set of queries on dynamic graphs. In this paper, we consider a different setting and aim to serve arbitrary subgraph matching queries on dynamic graphs. Wang et al. [4] also considered the dynamic nature of data graphs; however, our work is different from theirs as follows: (1) in [4], the proposed technique aims to answer approximate matching, while we propose a solution for exact matching; and (2) although node/edge uncertainty represented by numerical values are also considered, the proposed indexing technique is still based on categorical node/edge labels [35]; thus it cannot solve the challenge in our problem setting. Mondal et al. [21] studied how to make node replication decisions based on dynamically changing workload to optimize the performance for queries such as reading neighbors' data. Instead, our work focuses on how to improve the performance for more sophisticated subgraph matching queries on dynamic graphs with numerical labels.

Multi-dimensional index. Multi-dimensional indices have been studied for applications that monitor moving objects. Early works [5] used variants of R-tree to index moving objects and accelerate range/kNN queries; however, the update operations in these techniques are usually costly, while data points are frequently updated in the applications [28], [30]. To address the issue from frequent data updates, grid based indices are proposed [3], [23], [27], [28], [43]. In particular, Chakka et al. [3] used a grid based index to manage a large number of trajectories, Mouratidis et al. [23] and Yu et al. [43] discussed how to use grid based indices to accelerate kNN queries, Šidlauskas et al. [28] conducted an experimental study to compare grid based indices with variants of R-tree in update-intensive applications, and the possibility of incorporating parallelism into grid based indices is investigated in [27]. Our grid based index shares this spirit. It differs in the following aspects. (1) We show *theoretical* results on a more *general* setting: instead of two-dimensional space widely discussed in the above works, we focus on indexing points in a general multi-dimensional space, and derive theoretical bounds for update and search performance. (2) We discuss how index parameters affect update and query performance with both theoretical and experimental results. (3) We investigate the feasibility and principles of applying multi-dimensional indices to prune redundant search for dynamic subgraph matching queries.

VIII. Conclusions

In this paper, we identified a new important application of graph query in cloud computing and defined a general subgraph matching problem on dynamic graphs of numerical labels. We introduced a new method called Gradin to index graphs of numerical labels. Gradin can efficiently process frequent index updates and prune unpromising matches. In particular, FracFilter, one important component of Gradin, was proposed to keep the cost of update operations low and enable fast search. Minimum fragment cover and fingerprint based pruning were proposed for fast query processing. Our experimental results have shown that Gradin has better index update and query processing performance in comparison to baseline algorithms.

Acknowledgment

This research was sponsored in part by the Defense Advanced Research Project Agency (DARPA) agreement number W911NF-12-C-0028, NSF grants IIS-0917228, IIS-0954125, IIS-1219254, and the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053 (NS-CTA). The first author, Bo Zong, spent the Summer of 2012 at IBM T.J. Watson Research Center, where he conducted this research. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

References

[1] N. Bansal, K. Lee, V. Nagarajan, and M. Zafer. Minimum congestion mapping in a cloud. In *PODC*, 2011.

[2] T. Benson, A. Akella, A. Shaikh, and S. Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *SoCC*, 2011.

[3] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003.

[4] L. Chen and C. Wang. Continuous subgraph pattern search over certain and uncertain graph streams. *TKDE*, pages 1093–1109, 2010.

[5] S. Chen, C. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. In *VLDB*, 2008.

[6] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.

[7] J. Cheng, J. Yu, B. Ding, P. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.

[8] L. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[9] W. Fan, J. Li, J. Luo, Z. Tan, X. Wang, and Y. Wu. Incremental graph pattern matching. In *SIGMOD*, 2011.

[10] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu. Graph pattern matching: From intractable to polynomial time. In *VLDB*, 2010.

[11] W. Fan, J. Li, X. Wang, and Y. Wu. Query preserving graph compression. In *SIGMOD*, 2012.

[12] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, 1979.

[13] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR*, 2002.

[14] I. Giurgiu, C. Castillo, A. Tantawi, and M. Steinder. Enabling placement of virtual infrastructures in the cloud. In *Middleware*, 2012.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *SIGCOMM Review*, pages 63–74, 2009.

[16] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[17] H. He and A. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.

[18] H. Jiang, H. Wang, P. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[19] A. Khan, N. Li, X. Yan, Z. Guan, S. Chakraborty, and S. Tao. Neighborhood based fast graph search in large networks. In *SIGMOD*, 2011.

[20] X. Meng, V. Pappas, and L. Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM*, 2010.

[21] J. Mondal and A. Deshpande. Managing large dynamic graphs efficiently. In *SIGMOD*, 2012.

[22] M. Mongiovì, R. Di Natale, R. Giugno, A. Pulvirenti, A. Ferro, and R. Sharan. Sigma: a set-cover-based inexact graph matching algorithm. *Journal of bioinformatics and computational biology*, pages 199–218, 2010.

[23] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[24] R. Raghavendra, J. Lobo, and K. Lee. Dynamic graph query primitives for sdn-based cloudnetwork management. In *HotSDN*, 2012.

[25] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*, 2012.

[26] H. Shang, Y. Zhang, X. Lin, and J. Yu. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. In *VLDB*, 2008.

[27] D. Šidlauskas, K. Ross, C. Jensen, and S. Šaltenis. Thread-level parallel indexing of update intensive moving-object workloads. *Advances in Spatial and Temporal Databases*, pages 186–204, 2011.

[28] D. Šidlauskas, S. Šaltenis, C. Christiansen, J. Johansen, and D. Šaulys. Trees or grids? indexing moving objects in main memory. In *SIGSPATIAL GIS*, 2009.

[29] M. Song, H. Choo, and W. Kim. Spatial indexing for massively update intensive applications. *Information Sciences*, pages 1 – 23, 2012.

[30] M. Song and H. Kitagawa. Managing frequent updates in r-trees for update-intensive applications. *IEEE Transactions on Knowledge and Data Engineering*, 21(11):1573–1589, 2009.

[31] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. In *VLDB*, 2012.

[32] Y. Tian and J. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.

[33] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *SIGKDD*, 2007.

[34] J. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[35] C. Wang and L. Chen. Continuous subgraph pattern search over graph streams. In *ICDE*, 2009.

[36] T. Wang, M. Srivatsa, D. Agrawal, and L. Liu. Learning, indexing, and diagnosing network faults. In *SIGKDD*, 2009.

[37] X. Wang, A. Smalter, J. Huan, and G. Lushington. G-hash: towards fast kernel-based similarity search in large graph databases. In *EDBT*, 2009.

[38] Y. Xie and P. Yu. CP-index: on the efficient indexing of large graphs. In *CIKM*, 2011.

[39] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.

[40] X. Yan, P. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, 2004.

[41] X. Yan, P. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, 2005.

[42] J. Yang, S. Zhang, and W. Jin. Delta: indexing and querying multi-labeled graphs. In *CIKM*, 2011.

[43] X. Yu, K. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.

[44] Y. Yuan, G. Wang, H. Wang, and L. Chen. Efficient subgraph search over large uncertain graphs. In *VLDB*, 2011.

[45] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. In *VLDB*, 2010.

[46] P. Zhao and J. Han. On graph query optimization in large networks. In *VLDB*, 2010.

[47] L. Zou, L. Chen, and M. Özsu. Distance-join: Pattern match query in a large graph database. In *VLDB*, 2009.