Scalable Pattern Matching over Compressed Graphs via Dedensification

Antonio Maccioni
Roma Tre University
maccioni@dia.uniroma3.it

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

One of the most common operations on graph databases is graph pattern matching (*e.g.*, graph isomorphism and more general types of "subgraph pattern matching"). In fact, in some graph query languages every single query is expressed as a graph matching operation. Consequently, there has been a significant amount of research effort in optimizing graph matching operations in graph database systems. As graph databases have scaled in recent years, so too has recent work on scaling graph matching operations. However, the performance of recent proposals for scaling graph pattern matching is limited by the presence of high-degree nodes. These high-degree nodes result in an explosion of intermediate result sizes during query execution, and therefore significant performance bottlenecks. In this paper we present a dedensification technique that losslessly compresses the neighborhood around high-degree nodes. Furthermore, we introduce a query processing technique that enables direct operation of graph query processing operations over the compressed data, without ever having to decompress the data. For pattern matching operations, we show how this technique can be implemented as a layer above existing graph database systems, so that the end-user can benefit from this technique without requiring modifications to the core graph database engine code. Our technique reduces the size of the intermediate result sets during query processing, and thereby improves query performance.

1. INTRODUCTION

Efficiently querying real-world graphs continues to be an important challenge for modern database systems — even those systems designed specifically for graph data. As the size of the graph increases, so too does the challenge of querying it at high performance. The problem of scaling queries over graphs is fundamentally harder than scaling queries over relational data. Relational data is set-oriented, and therefore many query operations can be parallelized in a straightforward way to work over partitions of the set of records rather than the entire set at once. Different cores, or even different servers, can therefore process the query over a partition of the data independently from each other, allowing for linear scaling of query operations as more cores/servers are added.

KDD '16, August 13 - 17, 2016, San Francisco, CA, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4232-2/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2939672.2939856

In contrast, graph query operations are fundamentally less partitionable. A big reason for this is that most real-world graphs follow a *power-law* [1, 2] or a *heavy-tailed* law [3], which yield a handful of nodes in the graph that are connected (via an edge) to an extraordinarily large number of other nodes. An example of this phenomenon occurs on Twitter, where 10% of Twitter accounts follow the same five users[1]. The graph is thus very "dense" in areas around these "high degree" nodes, and query processing operations involving high-degree nodes are extremely skewed, taking far more time than equivalent operations on "low degree" nodes. This makes parallelization of query operators over different partitions of nodes extremely complicated, and in some cases impossible. The problem only gets worse as the graph grows: the degree of such nodes increases linearly or super-linearly with respect to the increase in graph size, and new nodes become high-degree. This is a consequence of the *scale invariance* property of real graphs [1, 2].

Pattern matching is a common (perhaps the most common) graph query operation. For example, original versions of SPARQL exclusively supported pattern matching queries. For a graph, G, such operations retrieve all subgraphs $g_i \in G$ conforming to a graph pattern expressed via a query q_i.

Although there exist an increasingly large variety of techniques for storing graphs and processing graph matching queries (see Section 5), a large fraction of them store graph edges in two or three-column tables in relational database systems, and process pattern matching queries via self-joins of this edge table. In particular, they perform one self-join operation per edge intersection in q_i, successively finding larger and larger matching subgraphs by joining the set of all edges in G with intermediate subgraphs produced by previous steps in query processing. High-degree nodes are particularly problematic for these types of joins because they produce an elevated number of intermediate results when computing the join sets. Unfortunately, pattern matching queries encounter high-degree nodes very frequently because these nodes are: (i) often explicit in the query due to their centrality, and (ii) often implicit in the variables since the low-degree nodes are usually connected to many of them.

Many graph database systems rely heavily on indexing techniques to facilitate join operations [4, 5]. Unfortunately, indexing is not effective enough to counteract the power-

[1] http://twittercounter.com/pages/100

law, as it only helps when a highly selective predicate can be applied to the edges of a graph to reduce the effective degree of high-degree nodes. When edges are unlabeled, or effectively uniform (*e.g.*, the *follow* edges on Twitter), indexes are unable to reduce the intermediate result sizes of patterns involving high-degree nodes.

One solution to improving the performance of particular types of graph operations, as proposed by both the data mining and theoretical computer science communities, is to reduce the size of the original graph G by turning it into a smaller graph $\tilde{G}$ [6–10]. These approaches propose a reduction in the number of (less useful) edges while maintaining certain structural properties of the graph. This improves the performance of various graph operations, such as the bounded approximation of the laplacian matrix [9] or the bounded approximation of graph isomorphisms [7]. However, removing edges in this way is a *lossy* compression that is unsuitable for graph database systems that are designed for users who demand exact answers to their graph queries.

In this paper, we present a lossless compression technique, called *dedensification*, that changes the structure of the (directed or undirected) graph in order to reduce the number of adjacencies of high-degree nodes. The main intuition behind our work is that there is a large amount of information redundancy surrounding high-degree nodes that can be synthesized and eliminated. In particular, clusters of low-degree nodes can be found in the data set that are connected to the same group of high-degree nodes (*e.g.*, in a social network, people who like rock music tend to *follow* a highly overlapping group of popular rock stars). To this end, we introduce special nodes in the graph, that we call *compressor nodes*, representing common connections of clusters of related nodes to high-degree nodes. Many redundant edges can, in this way, be removed from $\tilde{G}$.

Our dedensifcation technique is similar to the virtual node compression technique from the Web search community [11]. However, our goal in this paper is not to maximize compression — rather it is to accelerate query processing. Therefore we introduce a technique that constrains the formation of compressor nodes in order to provide global guarantees about the structure of the compressed graph. We use these guarantees to create query processing algorithms that enable direct querying of the compressed (dedensified) graph, and that leverage these guarantees to accelerate query processing. Unlike the approximate querying approaches mentioned above, our solution enables processing pattern matching queries in an exact way — returning the same results as queries over the original graph. Since the size of the whole graph is reduced and the intermediate results produced by edge-joins through high-degree nodes decrease, the performance of many types of graph pattern matching queries improves. Furthermore, the technique flattens the peaks in node degree distribution, which leads to a sub-linear increase of peak node degrees as the size of the graph increases. This facilitates scalable query processing over large graphs.

Dedensification and indexing are complementary techniques to improving pattern matching query performance over graphs. They can be used independently or together to further improve performance. However, while indexing improves query performance at the cost of increasing the space utilized by the database, dedensification almost always *reduces* the size of a dataset. Thus, dedensification does not suffer from the space-time trade-off of indexing.

2. DEDENSIFICATION

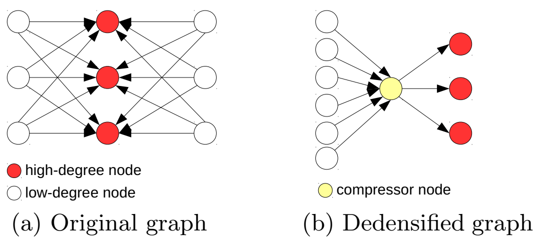
In this section we introduce the concept of *dedensification*. We parameterize our algorithms via a threshold, τ, that indicates the minimum number of adjacencies that a node should have for it be considered "high degree". Dedensification can be applied to both undirected and directed graphs. In the second case, the dedensification can be performed on both incoming and outgoing edges of the graph.

Many real-world graphs, especially those in the social network domain, have more density problems for incoming edges than outgoing edges. Therefore, in the following, we focus, without losing generality, on directed labeled graphs $G = (N, E)$ (N is the set of nodes and E is the set of edges). The nodes in N are labeled with a unique identifier, while the labels of the edges in E do not have to be unique (this type of labeling is the most common way to represent graph databases such as RDF graphs). A node is high-degree if it has at least τ incoming edges sharing the same label. The same node can be high-degree with respect to one edge label and low-degree with respect to a different edge label.

Although our techniques generalize easily to graphs with many different edge labels (this is described in more detail below), for ease of explanation, we will first consider the scenario where the data graph and the query graphs have only one edge label (*e.g.,* the "follows" label in the Twitter graph). This is semantically equivalent to an unlabeled graph, and makes query processing more challenging since this prevents filters on edge labels.

Given τ, we consider a bi-partition over the N nodes into two sets, N_h and N_l, representing the high-degree and the low-degree nodes, respectively. We call this graph G_τ.

In many graphs — especially those that follow the power law mentioned above — nodes in the graph are typically connected to many high-degree nodes. It is often the case that clusters of related nodes are connected to the same group of high-degree nodes (*i.e.*, they follow a *preferential attachment* [1]). We leverage this by repurposing the virtual node technique from the Web search community [11], and adding *compressor* nodes that summarize multiple connections of the same kind to high-degree nodes. In other words, rather than allowing the space around a node to be densely filled with incoming edges from all over the graph, we decouple high-degree nodes from their incoming connections by means of intermediate nodes that summarize multiple connections at once. We call this process "dedensification".



(a) Original graph (b) Dedensified graph

Figure 1: The dedensification operation.

For example, in the graph shown in Figure 1(a), six low-degree nodes are shown in white and three high-degree nodes are shown in red. Note that the low-degree nodes have outgoing edges to this same set of three high-degree nodes. Therefore, we can say that there exists a general "type of node" that has outgoing edges to this particular set of three high-degree nodes. We indicate that this "type of node" exists by creating a new node (shown in yellow on Figure 1(b)) that has outgoing edges to this set of three high-degree nodes. Then, we remove the edges that connect the white nodes to this set of high-degree nodes, and instead create a single edge from each white node to the new yellow node.

This new yellow node is called a "compressor node" because it has the potential to reduce the total number of edges in the graph. In particular, every node that can be classified into the "type of node" that the compressor node represents will have three edges removed, and replaced with a single edge that connects it to the compressor node. Thus, if there are 1000 nodes in the graph that connect to this particular set of three high-degree nodes, the compressor node will cause the 3000 edges incoming to these three high-degree nodes to be deleted and replaced with 1000 edges incoming to it. In addition to reducing the total number of edges in the graph, it has the additional benefit of reducing the congestion around the high-degree nodes.

The compressor nodes that we add to a graph are placed in a new set, N_c. (The original graph has $N_c = \emptyset$.) For simplicity, we indicate the graph as a four-tuple $G_\tau = (N_h, N_l, N_c, E)$. We represent the graph with $\tilde{G_\tau}$ if $N_c \neq \emptyset$.

Unlike the virtual node technique cited above, our focus in this paper is on accelerating query performance and not optimizing for compression. Therefore, we place constraints on how and when dedensification occurs, so that the query executer is able to leverage these constraints in order to reduce the scope of compressor handling during query processing. In particular, the dedensification (i.e. Algorithm 1) only creates a new compressor node if CONSTRAINT 1 holds on a set of high-degree nodes H and other nodes M:

CONSTRAINT 1 (DEDENSIFICATION PRE-CONDITION). *Given two input sets of nodes M and H, every node $n_i \in M$ has a directed edge to each node of H (*i.e., $\forall\ n_i \in M$ *and* $\forall\ h \in H$ *we have that* $(n_i, h) \in E$*).*

Therefore, for the graph G_τ of Figure 1(a), if the white nodes comprise M and the red nodes comprise H, the CONSTRAINT 1 is satisfied since all the white nodes are connected to each of the red nodes.

This enables the dedensification to transform the input graph to the graph shown in Figure 1(b), with the new compressor node added (in yellow) that decouples the multiple connections of the white nodes to the common subset of high-degree nodes, as described above. Although in our example, the white nodes comprising M consist only of low-degree nodes, note that in general M may also contain high-degree nodes since high-degree nodes may have outgoing edges to other high-degree nodes. However, even in the cases where M and H overlap, CONSTRAINT 1 is still valid and the compression technique works in the same way[2].

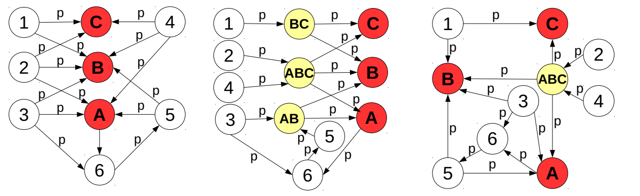
The dedensification algorithm computes an edge-cut that bi-partitions the graph by shielding **all** outgoing connections to high-degree nodes with compressor nodes. The pseudocode is shown in Algorithm 1. The loop in lines 4-11 chooses for how to find node sets H and M (i.e. a clustering on the high-degree nodes), while the loop in lines 12-19 computes the actual dedensification over H and M.

For example, consider the graph G_3 in Figure 2(a), which has three high-degree nodes depicted in red and six low-

[2]From now on we consider the terms dedensification and compression equivalent.

Algorithm 1: Dedensification of a graph.

```
Input  : A graph database $G_\tau = (N_h, N_l, \emptyset, E)$.
Output: A dedensified graph database $\tilde{G}'_\tau$.
1  $W \leftarrow \emptyset$; // an auxiliary map for CONSTRAINT 1
2  $N_c \leftarrow \emptyset$;
3  $E' \leftarrow E$;
4  foreach $n_i \in (N_h \cup N_l)$ do
5      $H \leftarrow$ select the outgoing high-degree nodes of $n_i$;
6      if $H \neq \emptyset$ then
7          $M \leftarrow W$.get($H$);
8          if $M = \emptyset$ then
9              $W \leftarrow W \cup (H, \{n_i\})$;
10         else
11             $M \leftarrow M \cup \{n_i\}$;
12 foreach $< H_i, M_i >$ in $W$ do
13     $n_c \leftarrow$ newNode();
14     foreach $n_i \in M_i$ do
15         $E' \leftarrow E' \setminus \{(n_i, h)\} : h \in H_i$;
16         $E' \leftarrow E' \cup \{(n_i, n_c)\}$;
17     foreach $h \in H_i$ do
18         $E' \leftarrow E' \cup \{(n_c, h)\}$;
19     $N_c \leftarrow N_c \cup \{n_c\}$;
20 return $\tilde{G}'_\tau = (N_h, N_l, N_c, E')$;
```



(a) A graph G_3 (b) $\tilde{G_3}$ (c) Non-expansive $\tilde{G_3}$

Figure 2: Dedensification of graphs.

degree nodes in white, all connected via the same type of edge `p`.

The result of the dedensification is in Figure 2(b). In this case, the algorithm finds three groups of high-degree nodes sharing the same incoming connections, namely `{A, B}`, `{B, C}` and `{A, B, C}`. They create three corresponding compressor nodes (in yellow). The outgoing nodes of the white nodes change accordingly. Note that after running Algorithm 1, every node has at most one outgoing edge to a compressor node, and every high-degree node has incoming edges coming only from compressor nodes.

We remark that the dedensification is a lossless compression of the graph since we can reconstruct the initial graph G from $\tilde{G}$ just by iterating on each compressor $n_c \in N_c$ and connecting each incoming node to n_c to all the outgoing nodes of n_c. Then, we empty N_c.

Non-expansive Dedensification. The dedensification in Algorithm 1 does not necessarily reduce the number of edges in the graph. This happens when the number of compressor nodes is extremely high due to a poor tuning of τ. Consider, for example, the non-compressed graph of Figure 2(a), which has fifteen edges and nine nodes (three of them are high-degree nodes with τ equal to three). The dedensification of such a graph results in a graph with no space gain since the number of edges is unchanged and the number of nodes increases by three (see Figure 2(b)). To avoid this problem, it is possible to perform a dedensification only if the number of removed edges is higher than the number of created edges (i.e. we have a space gain). Figure 2(c) shows a graph of Figure 2(a) dedensified with this non-expansive strategy.

Due to space constraints, we do not discuss this strategy further in this paper. However, in order to get the non-expansive strategy to work, slight modifications are necessary to the query algorithms presented in the next section. Nonetheless, we found these modifications caused only small differences in query performance, and we focus on the simpler version of the algorithms in this paper.

Dedensification of multi-edge graphs. Dedensification works with multi-edge graphs by considering a node high-degree if it has at least τ incoming edges sharing the same edge label. In this case, the created compressor nodes are associated to a specific given edge label.

3. GRAPH PATTERN MATCHING

In this Section we describe how graph pattern matching queries are processed over compressed graphs. Our solution is built as an extension of traditional pattern matching systems, leveraging their existing query execution modules. Therefore, before describing our mechanisms for performing pattern matching over compressed graphs, we first discuss traditional approaches to performing pattern matching queries over uncompressed graphs with a particular focus on the parts that our integration will leverage.

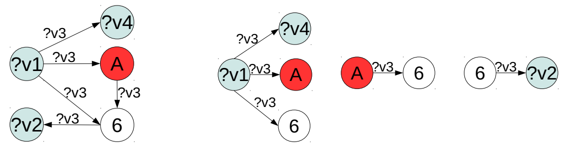
In general the process of graph pattern matching involves finding one or more instances of a query pattern in a graph dataset. A query pattern is itself a graph — a set of nodes and edges. These nodes and edges in the query pattern can either have labels (in which case these same labels must appear in the graph dataset in order for a match to occur) or they can be "variables" which means that they can match with any node (or edge) in the graph dataset.

For example, Figure 3(d) shows a very simple query pattern: a node whose label is `6` that is connected to another node (which is a variable) via a directed edge (whose label is also a variable). This query will return all possible values of `?v`$_2$ and `?v`$_3$ that match this pattern — in other words, all nodes to which node `6` has an outbound edge, along with the edge labels.

As another example, Figure 3(b) shows a slightly more complicated query pattern. This query returns all nodes that are connected to a node labeled `A` and another one labeled `6`, and a third node that has an unspecified label, as long as the three edges that connect it to `A`, `6`, and the third node all have the same label (which is represented using the same variable name `?v`$_3$).

In general, a query pattern can be decomposed into a series of node-edge-node "triples" — one triple corresponding to each edge in the query pattern. For example, Figure 3(b) can be decomposed into three triples: (`?v`$_1$, `?v`$_3$, `?v`$_4$), (`?v`$_1$, `?v`$_3$, `A`), and (`?v`$_1$, `?v`$_3$, `6`). The first element of each triple can be referred to as the "source" of the edge, or alternatively called the "subject" s. The second element is the "label" of the edge, or alternatively called the predicate p. The third element can be referred to as the "destination" of the edge, or alternatively called the "object" o.

There has been a significant amount of research and proposed algorithms for processing pattern matching queries in graph database systems, many of which have innovative and varied techniques for query processing (see Section 5). Our work is designed to interface with the most prevalent algorithm that exists in currently available systems for processing pattern matching queries. This algorithm proceeds as follows: each triple pattern $t_1, t_2, \ldots, t_n$ of q corresponds to a selection on the graph database G (*i.e.* $t_1(G), t_2(G), \ldots, t_n(G)$) and each intersection between two triple patterns corresponds to a join between two intermediate sets of triples (*e.g.*, $t_1(G) \bowtie t_2(G)$ if t_1 and t_2 have a node in common). Once all the selections and joins on shared nodes have been performed, the output of these operations is the complete answer set to q; that is $q(G) = t_1(G) \bowtie \ldots \bowtie t_n(G)$ [4,5,12].



(a) A query q_i over G_3 (b) q'_i (c) q''_i (d) q'''_i

Figure 3: Query decomposition.

Since joins of this type are commutative, these joins can be performed in any order. However, the common practice is to group them together by source node. In other words, if a node in the query graph has multiple edges emerging from it, the joins of the triples corresponding to these edges are done together, sequentially (or concurrently if the query processor supports multi-way joins) in the query plan. This pattern (where a node in the query graph has multiple edges emerging from it) is called a star. Query processing thus proceeds by decomposing a query graph q into star sub-queries $q', q'', \ldots, q^{(n)}$, which can range from simple triple patterns $q^{(i)} = \{(s,p,o)\}$ to wider stars $q^{(i)} = \{(s,p_1,o_1),(s,p_2,o_2),\ldots,(s,p_n,o_n)\}$ [13,14]. In other words, the query processor first joins together all edges within each star pattern in the query, and only after this does it join stars with each other.

For example, the graph pattern matching query q_i in Figure 3(a) contains two constant nodes (one of them, node `A`, is high-degree) and three variable nodes (`?v`$_1$, `?v`$_2$ and `?v`$_4$), all connected through the same kind of edge, here expressed with the variable `?v`$_3$. q_i can be decomposed into the three stars (q'_i, q''_i and q'''_i) shown in Figure 3(b), Figure 3(c) and Figure 3(d), respectively.

The optimizer is responsible for finding as good an execution plan as possible, which involves selecting a "good" order for joining the stars in q (*e.g.*, $q(G) = q'(G) \bowtie \ldots \bowtie q^{(n)}(G)$) [5,13]. For example, Figure 4(a) shows a possible execution plan for q_i where first q' and q'' are joined on node `A`, and then the resulting partial result is joined with q''' on node `6`. Alternatively, Figure 4(b) shows another admissible plan that first joins q'' and q''' on `6` and then joins the results with q'.

All plans for join ordering produce the same final result — they differ only in the performance and size of intermediate result sets. The two plans in Figure 4(a) and in Figure 4(b) for $q_i(G_3)$ produce two answers $a_1 = \{$`v`$_1$=`3`, `v`$_2$=`5`, `v`$_3$=`p`, `v`$_4$=`A`$\}$ and $a_2 = \{$`v`$_1$=`3`, `v`$_2$=`5`, `v`$_3$=`p`, `v`$_4$=`B`$\}$. Analogously, the optimizer also has to determine the execution plan for each star. For example, the star q' of q can be processed using the execution plan in Figure 4(c).

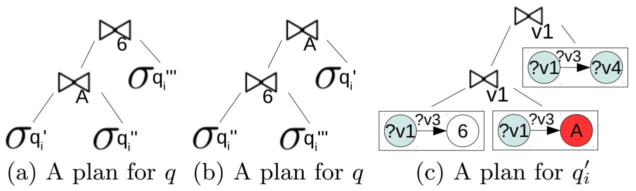


Figure 4: Query planning and execution.

Many sophisticated graph query optimizers have been created in the context of semi-structured data such as RDF data [5, 13, 14]. These optimizers usually choose a plan by minimizing a metric, via dynamic programming, that estimates the cost of the plan. The estimation makes use of a graph synopsis that stores statistics about the graph. For example, RDF-3X keeps track of the number of occurrences of the unary and binary triple projections [5]. RDF-3X synopses have been then enriched with *characteristic sets* and *characteristic pairs* to keep cardinality information on how the "type of stars" are connected to each other [13]. In particular, the characteristic pairs highlight how a decomposition of the query into star and chain subqueries is beneficial for query optimization and execution.

Our solution sits as a layer above a traditional database system supporting pattern-matching. Therefore, it does not attempt to optimize the ordering of the joins — rather, it leverages the execution planner of the underlying system to perform actual query optimization and execution. Rather, the goal of our solution is to make graph pattern matching more scalable by reducing the workload of the selection and join operations. In particular, we enable the optimizer of the underlying system to directly access the compressed data without first decompressing it. This is possible since the compressed data is itself a graph, $\tilde{G}$. Our solution processes a query q over G by rewriting q to a new query pattern $\tilde{q}$ over the compressed graph $\tilde{G}$, such that $q(G) = \tilde{q}(\tilde{G})$. Thus the underlying system does not need to be aware of the original query or the original graph. It simply needs optimize the processing $\tilde{q}$ over $\tilde{G}$.

In the following we show how star queries are modified for execution over a compressed graph $\tilde{G}$ when dedensification has been applied. These star queries can then be joined together using standard techniques in the underlying system to arrive at the complete query rewrite $\tilde{q}$.

In graphs compressed with dedensification, there are no direct connections from low-degree nodes to high-degree nodes. Instead, special compressor nodes are always present in between low-degree nodes and high-degree nodes. We leverage this fact to generate the algorithm in Figure 5, which shows the computation of all types of star joins over a dedensified graph database. We make use of a flowchart to explain the algorithm more intuitively, referring to each block using the letter enclosed within a green circle associated with that block. The figures embedded in the dashed line contain queries, whose color notation is the same used for Figure 1 and Figure 3. These queries are submitted to the graph database $\tilde{G}$, represented as a set of edges $\{(s, o)\}$.

The algorithm starts by taking as input the graph $\tilde{G}$ and the query q. If the query pattern q contains any ref-

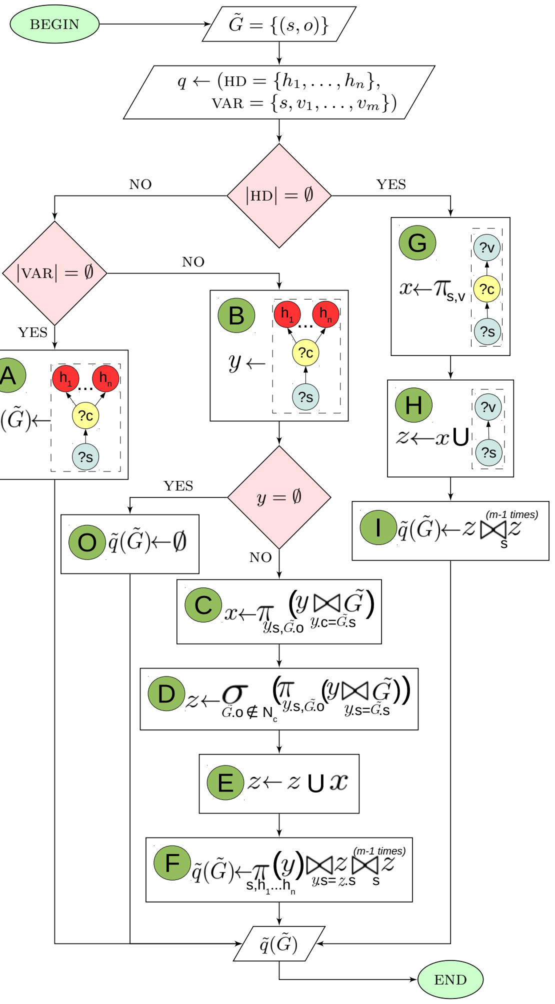

BEGIN
$\tilde{G} = \{(s,o)\}$
$q \leftarrow$ (HD $= \{h_1,\dots,h_n\}$, VAR $= \{s, v_1,\dots,v_m\}$)
$|\text{HD}| = \emptyset$
NO
YES
$|\text{VAR}| = \emptyset$
A $\tilde{q}(\tilde{G})\leftarrow$
B $y \leftarrow$
G $x \leftarrow \pi_{s,v}$
H $z \leftarrow x \cup$
$y = \emptyset$
O $\tilde{q}(\tilde{G})\leftarrow\emptyset$
I $\tilde{q}(\tilde{G}) \leftarrow z \bowtie_s^{(m-1\ times)} z$
C $x \leftarrow \pi_{y.s,\tilde{G}.o}(y \bowtie_{y.c=\tilde{G}.s} \tilde{G})$
D $z \leftarrow \sigma_{\tilde{G}.o \notin N_c}(\pi_{y.s,\tilde{G}.o}(y \bowtie_{y.s=\tilde{G}.s} \tilde{G}))$
E $z \leftarrow z \cup x$
F $\tilde{q}(\tilde{G}) \leftarrow \pi_{s,h_1\dots h_n}(y) \bowtie_{y.s=z.s} z \bowtie_s^{(m-1\ times)} z$
$\tilde{q}(\tilde{G})$
END

Figure 5: Star Join for Dedensification.

erences to constant low-degree node labels, the filters associated with matching these low-degree nodes are trivial, and are no different between uncompressed and dedensified graphs. Therefore, we ignore constant low-degree nodes in our discussion of the algorithm and only focus on the parts of q involving constant high-degree nodes (the set HD $= \{h_1, \dots, h_n\}$) and "variable nodes" (the set VAR $= \{s, v_1, \dots, v_m\}$, where s is the source and v_i are the destinations) which could be matched to either low-degree nodes or high-degree nodes.

For the purposes of the algorithm presented here, all non-variable edges within a star are uniform (if they are labelled, they have the same label). Edges with different labels in a query pattern are matched via different stars. Therefore, Figure 5 does not include edge-labels.

There are three macro-cases of star queries: (i) stars formed by only high-degree nodes in their fan-out (the left branch of Figure 5), (ii) stars formed by a mixed fan-out of variables and high-degree nodes (the central branch of Figure 5) and, (iii) stars formed by only variables in their fan-out (the right branch of Figure 5). Intuitively, all the cases follow the same algorithmic approach. They initially compute the portion of each answer formed by high-degree nodes. Then, the algorithm completes these partial answers with the remaining parts of the query. In practice, we apply a "push high-degree nodes down" rule in the query rewrite. The three cases are explained separately in the following.

High-Degree Nodes Only. When $\text{VAR} = \emptyset$ the query contains a fan-out with high-degree nodes only. This is the easiest case to compute because incoming edges to high-degree nodes only come from compressor nodes. Therefore, we simply have to search for any compressor node that connect to the high-degree nodes $h_1, \ldots, h_n$ in the query. If there are no such nodes, the empty set can be immediately returned. Otherwise all nodes connected to these compressor nodes via an outgoing edge from itself to the compressor node form the result set for $\tilde{q}(\tilde{G})$. This case can be computed with one query represented in the corresponding block A of Figure 5. It is equivalent to the input star with an additional compressor variable between the source and the destinations.

For example, in the case of a query $q_1 = (\text{HD} = \{\texttt{A,B}\}, \text{VAR} = \{s\})$ over the graph in Figure 2(b), the algorithm computes block A finding the compressors AB and ABC that are connected to both A and B. In this case the solutions are the incoming nodes to those compressors, namely nodes 2, 3, 4 and 5.

Mix of High-Degree and Variable Nodes. When the query has both variables (*i.e.* $\text{VAR} \neq \emptyset$) and constant high-degree nodes (*i.e.* $\text{HD} \neq \emptyset$), we first search for the partial stars containing the high-degree nodes that are specified in the query (*i.e.* "push high-degree nodes down"), assigning them to the set y (block B of Figure 5). If y is empty (*i.e.*, the condition $y = \emptyset$ holds), we do not need to search further because the final result set $\tilde{q}(\tilde{G})$ is empty as well (*i.e.*, $\tilde{q}(\tilde{G}) = \emptyset$ in `block 0`) since there does not exist any stars in G (or $\tilde{G}$) having this particular set of input high-degree nodes.

Otherwise, if y is not empty, we compute block C, block D, block E and block F in sequence to enrich the partial answers in y. Note that y already contains `all` possible sources `s` of $\tilde{q}(\tilde{G})$ (and these sources can be both low-degree and high-degree nodes). The goal of blocks `C-F` is to refine y by matching the variable parts of the query. This is complicated by the fact that variables can correspond to both low-degree and high-degree nodes.

We thus check for variables matching the low-degree and high-degree nodes separately. To check for matches with high-degree nodes, we leverage the fact that every node `s` that is connected to one or more high-degree nodes has exactly one outgoing edge to a compressor node. Therefore the same compressor nodes (call them c) identified in the intermediate result set y must be used to connect `s` to any high-degree node that a variable will match with. Therefore, in `block C` we join the intermediate result set, y, with the full compressed graph $\tilde{G}$ on the compressor nodes (*i.e.* $y \bowtie_{y.c=\tilde{G}.s} \tilde{G}$), in order to generate a list of edges with the subjects s in y being the source for these edges, and the destination of these edges are the high-degree nodes that serve as potential matches for the variables in the query. We label this intermediate result set x.

Then, in block D, we search for potentially matching low-degree nodes. Unlike block C where the join is performed on the compressor nodes, in this case the *subject* nodes of y are joined with all edges of the compressed graph $\tilde{G}$ (*i.e.* $y \bowtie_{y.s=\tilde{G}.s} \tilde{G}$) in order to find all nodes that are directly connected via an edge from the subject nodes of y. These nodes fall in one of two categories: they are either (1) low-degree nodes or (2) compressor nodes. (They are guaranteed not to be high-degree nodes since high-degree nodes only have incoming edges from compressor nodes.) We therefore filter out the compressor nodes (*i.e.* $\tilde{G}.o \notin N_c$), which leaves the resulting match set, called z, containing edges from subjects in y to all low-degree nodes that are potential matches for a variable in the query.

Block E unions edges in x with the edges in z, and stores the result back in z. z thus contains edges from the subjects in y to all nodes that these subjects were directly connected to in the original (uncompressed) graph and can be used for matching the variables in the star query.

Finally, in block F, we use z to match all the variables in the query. We self-join z $m-1$ times (where m is the number of variables in the star to match), and join this result with y (*i.e.* $\tilde{q}(\tilde{G}) = \pi_{s,h_1,\ldots,h_n}(y) \bowtie_s z \bowtie_s^{(m-1\ times)} z$).

We now explain this case with an example query $q_2 = (\text{HD} = \{\texttt{A}\}, \text{VAR} = \{s, v_1\})$ over the graph in Figure 2(b). First, block B finds the compressors AB and ABC connected to A, together with their incoming nodes. y thus has the partial results (s=2, c=ABC, $\texttt{h}_1$=A), (s=3, c=AB, $\texttt{h}_1$=A) and so on.

Then, block C computes a query that retrieves the other high-degree nodes that can match the variables. They are the destinations of the outgoing edges from AB and ABC. We save them in x together with their corresponding sources, *e.g., x* contains (s=2, o=C), (s=3, o=B), etc. The subsequent block D finds edges starting from the source nodes of y and ending in a low-degree node. In our case we retrieve only the edge `(3, 6)` that we save into z. Block E then adds the edges in x to z. Finally, block F computes a join between y and z (z is not self-joined with itself since there is only one non-source variable in this query). For instance the element of y: (s=3, c=AB, $\texttt{h}_1$=A) joins with several edges in z including `(3, 6)` found in block D and `(3, B)` found in block C. This indicates that when the source variable is matched with 3, nodes 6 and B can be matched to variable ?$\texttt{v}_1$ from the query.

Variables Nodes Only. When $\text{HD} = \emptyset$, the case is equivalent to the previous case (mix of high-degree and variable nodes) except that there is no filter to perform on the high-degree constants, so the initial creation of the set y in the previous case is not done. Block G corresponds to block C in the previous case in that it finds the high-degree nodes that are potential matches for object variables in the query. However, unlike block C which only had to search the compressor nodes found in y, Block G must search the compressor nodes in the entire graph, since there is no initial filter on high-degree constants in this case. Block H corresponds to block D and block E from the previous case. It finds all the low-degree nodes that are potential matches for variables in the query (*i.e.* all nodes that have incoming edges from non-compressor nodes), and then unions the edges with these low-degree nodes as objects with the edges produced in block G that have high-degree nodes as objects. Block I corresponds exactly to Block F, without the join with y. The algorithm ends, in all the three cases, by outputting $\tilde{q}(\tilde{G})$.

4. EMPIRICAL EVALUATION

This section presents the results of some experiments we conducted to test the approach described in this paper. We want to understand: (i) how the approach compares to running normal queries over uncompressed graphs, (ii) the scalability of the approach on real-world graphs and (iii) whether the approach can complement existing indexing approaches.

As mentioned in Section 3, although there are many different proposed techniques for storing graphs and processing pattern matching queries over them, our work is designed to interface with the most common technique: storing the graph in tables containing one row for each edge in the graph (where each row contains attributes corresponding to the two nodes and the edge label connecting them), and processing pattern matching queries via self-joins on this table [5,13].

Therefore, we created a graph database system prototype that stores the datasets in PostgreSQL and we developed a set of stored procedures written in PL/pgSQL that (i) generate a compressed graph $\tilde{G}_\tau$ given a graph G and a threshold τ; (ii) implement all the algorithms described in Section 3, including those for splitting up star queries into subqueries and joining them together. We delegated query optimization to the DBMS, which optimizes each individual query component issued to the system from our stored procedures.

All the experiments were conducted on a single machine provided with 2.66GHz Inter Xeon (2 cores) running Linux RedHat with 4 GB DDR RAM and a 2-disk 1TB striped RAID array. We conducted cold and warm cache experiments. All the times reported are "end-to-end" query performance and include the outputting of the results. Every test was executed twice, and we report the average of these runs.

Dataset	Number of nodes	Number of edges
Twitter	81,306	1,115,532
Google	875,713	5,105,035
Barabasi	400,000	8,000,000
LiveJournal	4,847,571	68,993,773

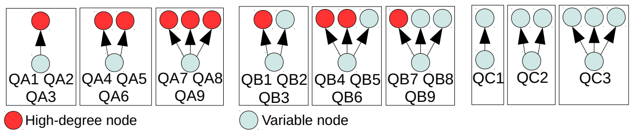
Table 1: Datasets properties.

Our experiments were performed on both real datasets (taken from the SNAP datasets [15]) and synthetic datasets. For real datasets we used the social networks Twitter and LiveJournal, and the Web graph Google. For the synthetic dataset, we used a generator included in JUNG[3](Java Universal Network/Graph Framework) for random Barabasi power-law graphs. This dataset is known to nicely approximate directed real-world graphs [1]. The sizes of the datasets are shown in Table 1.

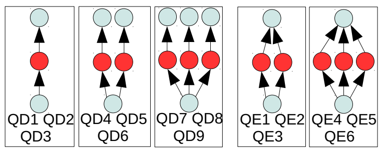
We compressed the dataset multiple times, using different threshold (τ) values. Each query on our benchmark was therefore run over the original uncompressed graph (indicated in the diagrams with "or"), and compared with the same query run over the dedensified versions of the graph (indicated with "dd").

Unfortunately there is no widely agreed upon benchmark that contains realistic pattern matching queries over graph data. Existing benchmarks have been heavily criticized as not being representative of real-word workloads. We do not aim to solve this problem and create a new benchmark containing real-world queries in this paper. Rather, as noted in Section 3, recent work by Gubichev et al. on graph pattern matching in the relational database context has shown that a good query processing strategy is to decompose a graph pattern matching query into stars, and perform all joins necessary to match these stars, and only afterwards join these stars together [13,14]. Since star queries are the basic building blocks for all queries — even complex ad-hoc real-world queries, it suffices to perform a detailed evaluation on these star queries, and note that the final joining together of the output of these star queries are equivalent in both the original and dedensified approaches.

[3]http://jung.sourceforge.net/



(a) *pattern A* (b) *pattern B* (c) *pattern C*



(d) *pattern D* (e) *pattern E*

Figure 6: Queries of the benchmark.

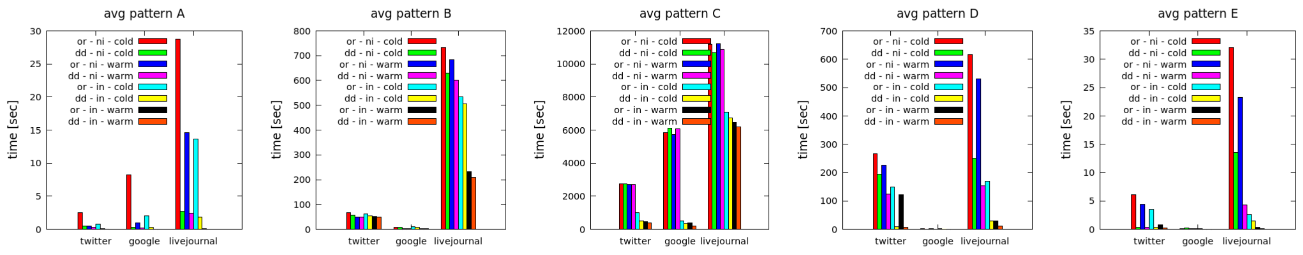
Figure 6 shows the star queries we used to evaluate our algorithms. At a high level, the benchmark is divided into five classes of queries, three of which — *pattern A*, *pattern B* and *pattern C* — correspond to three different types of basic stars: *pattern A* includes queries with only high-degree nodes, *pattern B* includes queries with a mixed fan-out of variables and high-degree nodes and *pattern C* includes queries with variables only. The source node s of the stars is a variable in all the queries, so that we cannot directly exploit an index to return a query result immediately. We do not include low-degree nodes in our star queries since they are performed equivalently in the original and dedensified systems, and can be matched quickly by the system. *pattern D* and *pattern E* show some interesting compositions of multiple stars through high degree nodes.

The red nodes in Figure 6 are explicit references to a particular high-degree node. In order to pick which high-degree nodes should be used in these queries, we used the following method: we find the five highest-degree nodes in the graph and then take the top three most frequent combinations of these 5 nodes. For example, queries QA4-QA5-QA6 and QA7-QA8-QA9 include the most frequent combinations of two and three high-degree nodes, respectively. In this way, we make sure that our benchmark tests the most challenging queries over the graph database.

4.1 Results

In this section we evaluate the results obtained using our graph DBMS prototype. Since we are more focused on query performance than data compression, in this section we use large τ values such that there are around 100 compressor nodes created per dataset: $\tau = 2,500$ for Twitter, $\tau = 5,000$ for Google, $\tau = 10,000$ for LiveJournal, and $\tau = 7,000 - 28,000$ for Barabasi (see Section 4.2 for why we used multiple τ values for Barabasi).

We ran two different sets of experiments. The first set does not use any indexing (indicated in the diagrams with "ni"), while the second set is representative of popular triple-



(a) pattern A (b) pattern B (c) pattern C (d) pattern D (e) pattern E

Figure 7: Average querying performance.

stores and uses exhaustive indexing of all combinations of attributes (indicated with "in"). The results obtained on real datasets (*i.e.* TWITTER, GOOGLE and LIVEJOURNAL) are presented in Figure 7.

As mentioned above, our benchmark consists of five classes of queries. Each member of a class are similar with each other — only differing from each other in the number of branches from a particular star. Therefore, members of a class all present similar performance trends. In order to reduce complexity of analysis, in this section we present only a single performance number for each class of query, which is derived by taking an average of the performance of each individual member of the class.

Performance without Indexing. We first discuss the results in Figure 7 where we did not create any indexes on the data, thereby forcing the DBMS to always access the data via a sequential scan.

Our implementation separates nodes associated with low degree nodes, high degree nodes, and compressor nodes. This distinction and co-location of related nodes allows PostgreSQL to achieve access locality for those parts of our plan tree shown in Figure 5 that focuses on low degree nodes, high degree nodes, and compressor nodes respectively. This explains why the advantage of densification is usually greater for the cold cache results than the warm cache results.

The other primary reason why the dedensified algorithms perform better than performing the query on the original graph is the reduction of the size of the intermediate results that are input to the joins required to construct the stars. For example, consider query A7 on TWITTER which requires three or more joins for both the dedensified and original executions. It involves joining edges to three high-degree nodes: h_1, h_2, and h_3. For the original graph, the first join on the raw graph produces 2,626 intermediate results. However, after the final join (with the edges of h_1), the result set drops dramatically to 22 results. In contrast, for the dedensified graph, the first join between the edges of h_2 and those of h_3 results in 4 intermediate results (all of them are compressor nodes) which are then joined with the h_1 edges to yield the same set of 22 final results. This final join is thus much faster to process.

For the above two reasons, the queries belonging to *pattern A* are always faster over the dedensified graphs than over the original graphs.

On *pattern C* queries, the dedensified queries do sometimes perform better than the uncompressed queries, but the difference is small. This is because the intermediate result sets are only slightly reduced in size for the compressed *pattern C* queries, since the joins generally involve all edges in the graph. At the scale of the graphs used in this experiment, this advantage is offset by the extra join that the dedensified queries must perform. However, for increasingly larger graphs experimented with in Section 4.2, the trade-off shifts, and the performance of dedensification starts to separate from the performance of querying the original graph.

The results for *pattern B* are a hybrid of the results from *pattern A* and *pattern C*. Queries involving more constants than variables are more similar to *pattern A*, and queries involving more variables are more similar to *pattern C*.

The queries in *pattern D* and *pattern E* involve composition of stars through high degree nodes. These results are similar to those obtained on *pattern A*. The execution times on the GOOGLE dataset are close to zero because they return an empty result set for several queries.

Performance with Indexing. We now discuss the results in Figure 7 where the raw data was exhaustively indexed. Exhaustive indexing is adopted by many pattern-matching query systems [4, 5]. This configuration includes B$^+$-tree indexes on $(s,\ p,\ o)$, $(s,\ o,\ p)$, $(p,\ s,\ o)$, $(p,\ o,\ s)$, $(o,\ s,\ p)$, $(o,\ p,\ s)$, $(s,\ o)$, $(o,\ s)$, $(s,\ p)$, $(p,\ s)$, $(p,\ o)$ and $(o,\ p)$.

Although overall performance is faster when indexes can be used to accelerate matches to query constants, the relative performance of the three query schemes are nearly identical for this set of experiments as for the set of experiments without indexing. This is because dedensification and indexing are complementary. Indexes help to accelerate the match of the high-degree constants in the query, and can do so for both dedensified graphs and original graphs. Our dedensification algorithms change the structure of the graph, but do not change the fundamental representation of nodes and edges. This allows the dedensified graph to be indexed in the same way as the original graph. However, the dedensification strategies are able to maintain their advantage over the uncompressed strategy when indexes are used because indexes only help for first steps of query processing, and the main advantage of the dedensification strategies is that they keep the intermediate result set and join-input sizes small.

4.2 Evaluation with Evolving Graphs

We now present the results of experiments we ran in order to understand how the different pattern-matching techniques scale when the size of the graph increases. The Barabasi model simulates the evolution of a real-world network [1]. Therefore, we use the Barabasi model to generate a graph and measure the performance of pattern-matching queries over this graph at four points during its evolution: (1) when it has 100,000 nodes and 2,000,000 edges, (2) when it has 200,000 nodes and 4,000,000 edges, (3) when it has

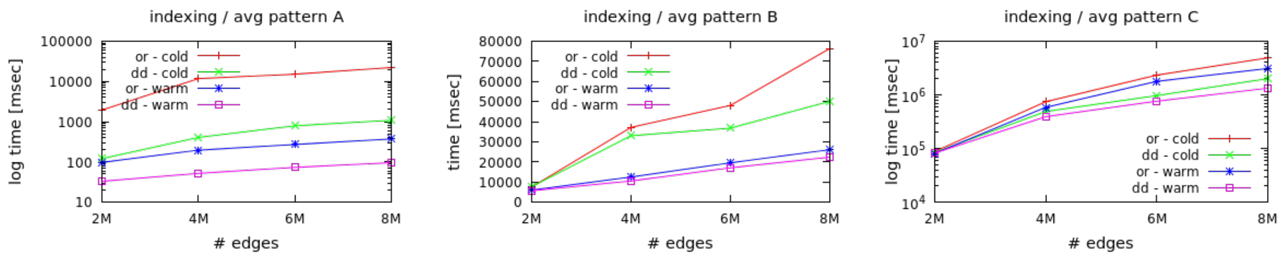


Figure 8: Evolving graph on experiments with indexing.

300,000 nodes and 6,000,000 edges, and (4) when it has 400,000 nodes and 8,000,000 edges.

Due to the preferential attachment formation of the graph [1], the most expensive queries involve the same high-degree nodes in all the four datasets, which allows us to compare exactly the same queries over the evolving network. However, as the size of graph changes, the definition of "high degree" changes along with it. For the first intermediate point in the Barabasi graph, we used $\tau = 7,000$; for the second: $\tau = 14,000$; for the third: $\tau = 21,000$; for the fourth: $\tau = 28,000$. Figure 8 presents the results of our experiments.

The behavior of queries of *pattern A* is linear with respect to the growth of the size of the dataset. However, the slope of the compressed graphs is smaller, with a behavior nearly-constant (note the log scale). This demonstrates that the advantages of dedensification for *pattern A* discussed in Section 4.1 scales with the graph size.

For *Pattern B*, the performance are similar for smaller graphs, but as the size of the graph increases, the advantages of dedensification yield large improvements in performance. This is because as the size of the graph increases, the cost of the joins in the query plan bottlenecks query performance, and the dedensification strategies keep the size of the input to these joins much smaller than the original strategy.

For *Pattern C*, the benefits of dedensification are reduced, as discussed in previous sections. However, the dedensification does scale better than the original strategy (note the log scale).

4.3 Tuning of Dedensification

Our approach has an input parameter, τ, that is used to define which nodes are "high-degree" in the graph and consequently which neighborhoods to dedensify. Figure 9 shows the number of compressor nodes added and edges from the graph when varying τ, for TWITTER (up to $\tau = 2,500$) and GOOGLE (up to $\tau = 5,000$) datasets. These two datasets represent the two extremes among the structure of our datasets: TWITTER is the densest while GOOGLE is the sparsest. In Figure 9, the compressor nodes added are on the left-hand y-axis and edges removed on the right-hand y-axis (both in the log scale).

The number of compressor nodes created by the dedensification process is inversely proportional to τ. The compression rate, expressed in terms of difference between the number of edges in the original graph and in the compressed graph, follows an heavy-tailed behavior. From our experiments (not shown in this version of the paper), we also witnessed that higher compression rates do not correspond to better performance. The important effect of dendensification is to reduce the density around the densest nodes in

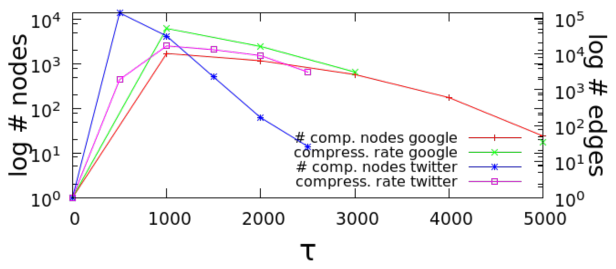


Figure 9: Compression statistics varying τ.

the graph. Compressing the intermediate nodes yield little marginal benefit.

5. RELATED WORK

Despite the fact that graph pattern matching is one of the most studied problems in the graph data management field, there is little research addressing the issues engendered by high-degree nodes. Nevertheless, below we outline related work divided into categories.

Improving graph pattern matching performance via graph reduction and compression. The impossibility of reducing the complexity of algorithms for graph query answering (*e.g.,* graph pattern matching) and the limited resources for computing queries over large graphs, has pushed many researchers to devise alternative solutions for improving performance. One of them is to reduce the size of the input by transforming the original graph G into a smaller graph $\tilde{G}$ [6–9, 11]. Fan et al. [7] propose to compute approximate query answering over a reduced portion of the graph, where the compression ratio and the query are given upfront. In practice, the portion of the graph that would be untouched by the query (or a set of them) is pruned a priori. A similar idea is applied for approximating the results of several classes of first-order logic queries by accessing a bounded amount of data [10].

Satuluri et al. modify the graph for an efficient clustering by reducing the edgeset with a heuristic that retains only those edges that are likely to be part of the same cluster [8]. *Spectral sparsification* was proposed by Spielman et al. to approximate important properties of a weighted undirected graph G by simply checking them on $\tilde{G}$ in a reduced time complexity [9]. These approaches are different than our approach because they are *lossy* and therefore unsuitable for graph database systems that are designed for exact query answering.

Our utilization of compressor nodes is similar to the "virtual node" concept from Buehrer et. al. [11] which was developed for community detection in web graphs. However, the Buehrer et. al. technique focuses on compression and graph processing, but not pattern-matching queries. Therefore, they extract all meaningful connectivity formations and optimize for compression instead of query execution. As a result, they do not distinguish between high-degree and low-degree nodes, and the query processing technique we introduce in Section 3 cannot be applied over virtual nodes.

Work by Fan et. al. [6] has similar motivation to ours. It proposes a lossless compression technique and query pattern matching directly over compressed data. The compression uses bisimulation equivalence to merge into the same hypernode many original nodes, all sharing type and connections. In this case the queries have to be expressed via bounded simulation rather than graph isomorphism. These kind of queries work well on in-memory databases but differ from queries normally employed by disk-based graph database systems.

Other main memory approaches compress the graph adjacency matrix for network/link analysis, such as the page rank algorithm, which are computed through matrix multiplications [16–18]. They aim at finding a suitable ordering of the rows of the matrix in order to create uniform blocks of 1s and 0s in the matrix, thus allowing a higher-rate of (bitwise) compression. The WebGraph framework [16] introduces a compression mechanism that relies on the properties of similarity and locality, which characterize the web graph. They are able to use only 3 bits per edge in a giant graph. Chierichetti et al. [17] extend the WebGraph ideas to social networks by exploiting the link reciprocity property. SlashBurn [18] directly exploits the high-degree nodes in the graph to define the communities of nodes (*i.e.* adjacent rows of the matrix). All of this work focuses on improving particular in-memory graph operations using compression, but differs from our contributions of using compression to accelerate pattern matching queries for disk-based graph database systems.

Improving graph pattern matching performance via graph indexing. Most database systems rely extensively on indexing to compute graph pattern queries [4, 5, 14, 19]. (An exception to this style of query processing are in-memory graph database systems that take advantage of graph exploration instead of joins such as the Trinity system [20].) RDF-3X [5] and Hexastore [4] exploit an exhaustive indexing scheme comprising the six clustered permutations of the triples and of their binary and unary projections. gIndex [19] indexes frequent sub-graphs, proving that this helps the execution of complex graph queries. gStore [14] is a SPARQL engine that makes use of a height-balanced tree index (called VS*-tree) and materialized views to speed up aggregate and wildcard-based queries. Reachability indexes have also been proposed for pattern matching: Cheng et al. [21] use a 2-hop indexing scheme to accelerate reachability tests that are used for graph pattern matching.

Our approach is complementary to these approaches. The compressed graphs that we create look like regular graphs to the underlying system, and can be indexed in the same way as the raw graphs (as we found in Section 4). Furthermore, materialized views can be created on these graphs to accelerate reachability and wildcard-based queries. To summarize: our work does not focus on performing selections on the dataset faster, but rather on pruning a considerable amount of intermediate results around nodes involved in multiple dense connections.

6. CONCLUSION

In this paper, we introduced a dedensification mechanism for graph databases whose goal is to overcome an important problem that prevent graph database systems from achieving scalable query performance: high-degree nodes. Dedensification discovers information redundancy in graphs resulting from multiple nodes connecting to the same set of high-degree nodes, and removes this redundancy via the creation of new "compressor" nodes. Our experiments show that dedensification improves performance for queries involving high-degree nodes, sometimes by an order of magnitude.

Acknowledgements. This work was sponsored by the NSF under grant IIS-1527118.

7. REFERENCES

[1] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, 1999.

[2] J. Leskovec, J. M. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *TKDD*, vol. 1, no. 1, 2007.

[3] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web - revisited: a trick of the heavy tail," in *WWW*, 2014.

[4] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, 2008.

[5] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[6] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *SIGMOD*, 2012, pp. 157–168.

[7] W. Fan, X. Wang, and Y. Wu, "Querying big graphs within bounded resources," in *SIGMOD*, 2014, pp. 301–312.

[8] V. Satuluri, S. Parthasarathy, and Y. Ruan, "Local graph sparsification for scalable clustering," in *SIGMOD*, 2011.

[9] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM J. Comput.*, vol. 40, no. 6, 2011.

[10] W. Fan, F. Geerts, Y. Cao, T. Deng, and P. Lu, "Querying big data by accessing small data," in *PODS*, 2015.

[11] G. Buehrer and K. Chellapilla, "A scalable pattern mining approach to web graph compression with communities," in *WSDM*, 2008.

[12] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "SW-Store: a vertically partitioned DBMS for semantic web data management," *VLDB J.*, vol. 18, no. 2, 2009.

[13] A. Gubichev and T. Neumann, "Exploiting the query structure for efficient join ordering in SPARQL queries," in *EDBT*, 2014.

[14] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gStore: a graph-based SPARQL query engine," *VLDB J.*, vol. 23, no. 4, 2014.

[15] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data.

[16] P. Boldi and S. Vigna, "The webgraph framework I: compression techniques," in *WWW*, 2004, pp. 595–602.

[17] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *KDD*, 2009.

[18] Y. Lim, U. Kang, and C. Faloutsos, "SlashBurn: Graph compression and mining beyond caveman communities," *TKDE*, vol. 26, no. 12, pp. 3077–3089, 2014.

[19] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *SIGMOD*, 2004, pp. 335–346.

[20] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in *SIGMOD*, 2013.

[21] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," in *ICDE*, 2008, pp. 913–922.