Digree: A Middleware for a Graph Databases Polystore

Vasilis Spyropoulos, Christina Vasilakopoulou, Yannis Kotidis
Department of Informatics, Athens University of Economics and Business, Athens, Greece
vasspyrop@aueb.gr, cvasilak@aueb.gr, kotidis@aueb.gr

Abstract— **In this paper we present *Digree*, an experimental middleware system that can execute graph pattern matching queries over databases hosting voluminous graph datasets. First, we formally present the employed data model and the processes of re-writing a query into an equivalent set of subqueries and subsequently combining the partial results into the final result set. Our framework guarantees the correctness and completeness of the produced answers. Then, we present a prototype implementation of *Digree*, which is agnostic to the underlying data processing engines used at the endpoints. As the experimental results show, in many cases *Digree* outperforms a single node graph database deployment in execution speed, up to 20 times depending on the query type.**

Keywords-**graph pattern matching; graph databases**

I. Introduction

Graph data processing is a notable example of the "*one size does not fit all*" paradigm. This is due to both the inherent heterogeneity of the graph data and the diversity of the different computations that can be performed on them. Consequently, existing approaches utilize relational databases [1], [2], big data systems [3], [4], [5], RDF triple stores [6], [7], or a combination of the above. In this fragmented environment, it is quite possible that applications will have to access graph data lying on different ecosystems, using different underlying storage representations and offering different query languages to access this data.

In this work, we present a middleware approach that permits execution of pattern matching queries over distributed or interlinked big-graph datasets hosted by such a network of independent data sources called endpoints. The middleware receives graph pattern matching queries, splits them in a suitable manner that permits their efficient parallel execution and then assembles all partial results in order to generate the final result set. We introduce a solid theoretical framework that ensures the correctness of this process. This framework is generic, in the sense that it is not tied to a particular implementation or query language.

Our prototype system, termed *Digree*, adopts a flexible architecture where requests for local data processing on the endpoints are made by implementing a very basic interface so as to be able to ask for computation of simple path expressions. Such interfaces can be implemented for native graph data management systems [8] but also for relational or big data deployments as well. The proposed middleware in our prototype is built around a DBMS that is used to temporarily store the partial results and perform the required operations so as to produce the final result set. The underlying endpoints storing the graph data remain fully functional and can, at the same time, continue to run as standalone systems.

In the evaluation of our prototype, we used native graph databases at the nodes and a PostgreSQL DBMS at the middleware. We are currently working to extend support to other types of endpoints, such as relational databases and big data systems like Spark [9]. The latter can also be used to implement the functionality of our middleware.

Our contribution can be summarized as follows:

- We study the evaluation of a general graph pattern matching query in a polystore [10] integrating interlinked graph databases. We formalize the process of decomposing a query into a set of smaller patterns, via a series of transformations. These subqueries can be executed in parallel, and their results are used to form the final result set.
- We present a prototype middleware system termed *Digree* that implements the aforementioned ideas in order to orchestrate the query decomposition and result set composition tasks. We describe a modular implementation that enables us to use different graph data implementations at the distributed endpoints.
- We present a series of experiments running on a prototype implementation of *Digree*. We run a set of pattern matching queries against three different datasets and discuss the scalability and capabilities of *Digree*.

II. Overview

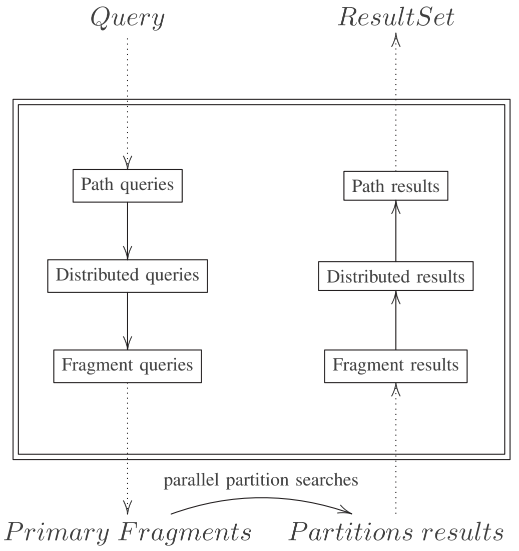
Our work concerns the process of efficiently querying a polystore of interlinked graph databases. The graph data model that we employ is one of the most widely adopted models, namely the *labeled property graph model*, which is made up of nodes, relationships, properties and labels. Consequently, given a *pattern query*, i.e. a directed graph with vertices and edges possibly with labels and properties, the fundamental task is to find subgraphs of the database that are isomorphic (structurally and semantically) to the pattern query. This belongs to the *(exact) pattern matching* problem, specifically in terms of *subgraph isomorphism* [11].

The middleware system that we propose takes as input a pattern query and essentially divides it into all required smaller parts that are executed in parallel over all graph

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

database partitions [12],[13]. The middleware then appropriately combines the partial results to produce the global result set. As it will be made clear from the discussion, *Digree* can utilize any graph database system or combination of those hosting the graph partitions. All that is required is the existence or implementation of the respective basic API calls for querying path expressions in the underlying systems.

Bellow, we can see the operations that decompose the input query and re-synthesize the partial results:



Section III mathematically formalizes the above operations and illustrates the process via a running example.

III. Query Rewrite and Results Combination

A. Preliminaries

A (finite) *directed graph* is a pair $G = (V, E)$ where V is the finite set of vertices and $E \subseteq V \times V$ is the finite set of edges; the first component of an edge pair is the *source* and the second is the *target*. The basic graph theoretic definitions given below can be found e.g. in [14], [15].

An *edge partition* $\{E_i\}$ of G is a set of non-empty disjoint subsets whose union gives E. If we define $\{V_i\}$ to be the set of source and target nodes of edges in $\{E_i\}$, every $G_i = (V_i, E_i)$ is a graph on its own. The family $\{G_i\}$ of edge-disjoint subgraphs is a *decomposition* of G,

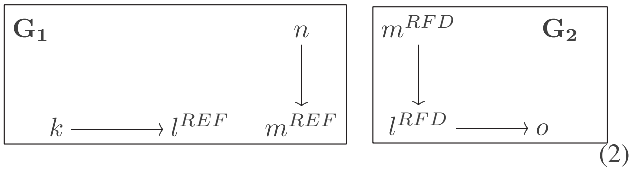
$$G = G_1 \cup ... \cup G_m = (\cup_i V_i, \cup_i E_i).$$

Notably, $\{V_i\}$ are not disjoint in principle: if two adjacent edges are located in different partitions, the vertex in-between is 'duplicated' and exists in both respective subgraphs. Those elements lie in the set

$$\mathcal{I} = \bigcup_{1\leq i,j\leq m}^{i\neq j} (V_i \cap V_j) = (V_1 \cap V_2) \cup (V_1 \cap V_3) \cup ... \cup (V_{m\text{-}1} \cap V_m). \tag{1}$$

The structure of the distributed graph database is as follows. Starting with the whole graph $G = (V, E)$, vertices are typically partitioned via some algorithm, and edges' location is determined by their source vertex. For our purposes, we create a duplicate of the end vertex of the cross-partition edges, labeled with REF; the original vertex obtains a label RFD, so that the graph database is in fact decomposed in parts $G_1, ..., G_m$ that constitute an edge partition.

For example, consider a distributed graph database with two partitions, $G_1 \cup G_2 = G$, and a subgraph $H \subseteq G$ with $V_H = \{k, l, m, n, o\}$ and $E_H = \{(k, l), (l, o), (n, m), (m, l)\}$. If the partition algorithm sends $\{k, n\}$ to G_1 and $\{m, l, o\}$ to G_2, then $\{(k, l), (n, m)\} \in G_1$ and $\{(m, l), (l, o)\} \in G_2$. A graphic representation is



(2)

Some basic properties of the above procedure are the following:

- A source node is always in the same partition as its edge, whereas its target inside the same partition may be a REF copy of the RFD vertex placed elsewhere.
- REF vertices necessarily have no outgoing edges.
- The set of all REF nodes, as well as that of all RFD nodes, equals $\mathcal{I}$; each RFD node may have REF duplicates in more than one partition.

A *path cover* is a set of disjoint paths in G which together contain all vertices; we denote a k-path as $P = (x_1, ..., x_k)$. If we don't allow paths of length 0, namely single vertices, a path cover $\{G_i\}$ forms an edge partition of G. The elements in the intersection $\mathcal{I}$ of the V_i's are now called *join* vertices.

A *weakly connected graph* is a graph where there exists an undirected path between every pair of vertices. In what follows, our initial pattern query will always be weakly connected, since otherwise we could take its weakly connected components and perform the transformations separately.

Our basic task is to identify subgraphs of the graph database which are *graph-isomorphic* to an input query. Two graphs G, H are isomorphic when there is an edge-preserving bijection $f : V_G \cong V_H$, i.e. such that $(x, y) \in E_G \Leftrightarrow (f(x), f(y)) \in E_H$ (hence also $E_G \cong E_H$).

As mentioned in the overview, our system is focused on the *labeled property graph* data model, elsewhere called directed labeled typed/attributed graph. Vertices can be viewed as tuples with a unique id, certain labels and properties (attributes), whereas edges have a source and target vertex, labels and properties. However, in order to emphasize the underlying general techniques and ideas, presently we employ the abstract representation of a plain directed graph.

B. Graph query rewrites

1) Pattern query to path pattern queries: Consider an arbitrary pattern query $Q = (V, E)$. The initial transformation decomposes the query into a list of edge-disjoint paths, i.e. specifies a path cover for Q. Any path covering algorithm from the literature will do and the choice is orthogonal to our techniques. We chose to use an all-paths algorithm to discover all possible paths between outer vertices or join candidate vertices and then select the largest possible paths that constitute a path cover of the query. The choice is in a sense independent of the rest of the method: changing it only affects the intermediate steps and not the final results. We thus obtain specific simple paths (in a specific order)

$$Q^1, Q^2, ..., Q^n : \bigcup_{1\leq i\leq n} V_i = V \text{ and } \bigcup_{1\leq i\leq n} E_i = E$$

Therefore this well-defines a function

$$\begin{aligned} F:\ PQ &\longrightarrow PPQ^n \\ Q &\longmapsto (Q^1, ..., Q^n) \end{aligned} \tag{3}$$

where PQ is the set of all (weakly connected) *pattern queries* and PPQ is the set of *path pattern queries*. The n-th cartesian product $PPQ^n = PPQ \times ... \times PPQ$ is as usually defined as the set of n-tuples. Notice that this function, like all functions that follow, is 'stable under isomorphism': for $R \cong Q$, $F(R) = (R^1, ..., R^n)$ with $R^i \cong Q^i$.

As a demonstrating example, consider a pattern query Q with $V = \{a, b, c, d, e\}$ and $E = \{(a, b), (b, c), (e, d), (d, b)\}$

$$\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \\ & d \arrow[u] & \\ & e \arrow[u] & \end{tikzcd} \tag{4}$$

The path decomposition operation produces the simple path queries $Q^1 = (e, d, b, c)$, $Q^2 = (a, b)$ and the single join vertex is $\mathcal{I} = V_{Q^1} \cap V_{Q^2} = \{b\}$. For simplicity, we denote the 4-path (e, d, b, c) as $Q^1 = (x_1, x_2, x_3, x_4)$ and the 2-path (a, b) as (y_1, y_2). Hence the image of Q under the function $F : PQ \to PPQ \times PPQ$ is

$$\begin{aligned} F(Q) = (Q^1, Q^2) &= (e \to d \to b \to c, a \to b) \\ &= ((x_1, x_2, x_3, x_4), (y_1, y_2) \mid x_3 \equiv y_2) \end{aligned}$$

2) Path to Distributed pattern queries: Suppose we have a path pattern query $P = (x_1, ..., x_k)$. The next transformation determines all the 'breakpoints' of the path, in order to identify all its possible separated sub-parts inside the partitions of the distributed graph database.

Consider the set $V_P \backslash \{x_1, x_k\} = \{x_2, ..., x_{k\text{-}1}\}$ with size k-2. This contains precisely the candidate nodes where the path can be split – with minimal part a single edge – due to the structure of the distributed graph database: the start node is necessarily in the same partition as the first edge of the path, whereas the end node appears inside the last edge's partition, even with a REF label. Thus we do not need to consider $\{x_1, x_k\}$ as breakpoints when splitting a path in all possible non-zero smaller paths.

Since a path can be split up at multiple nodes simultaneously, there are as many *distributed queries*, i.e. our initial path query together with a specific choice of breakpoints, as the size of the *powerset* $\mathcal{P}(V_P \backslash \{x_1, x_k\})$. This contains all possible subsets of $\{x_2, ..., x_{k\text{-}1}\}$, and its size is $2^{k\text{-}2}$. For simplicity of notation, we denote

$$\{x_{i_1}, x_{i_2}, ..., x_{i_v}\} \subseteq \{x_2, ..., x_{k-1}\} \text{ as}$$
$$s_{(i_1 i_2 .. i_v)} \in \mathcal{P}(V \backslash \{x_1, x_k\}).$$

Write $s_\emptyset = \emptyset$, which corresponds to the whole path (no breakpoints). We can now define a function

$$\begin{aligned} G:\ PQ &\longrightarrow DPQ^{2^{k\text{-}2}} \\ P &\longmapsto ((P, s_\emptyset), (P, s_{(2)}), .., (P, s_{(23)}), .., (P, s_{(23...k\text{-}1)})) \end{aligned} \tag{5}$$

where DPQ is the set of all distributed queries. If we start with F as in (3), we can compose it with G for each path $Q^1, ..., Q^n$:

$$\mathbf{G} : PPQ^n \xrightarrow{G_1 \times .. \times G_n} DPQ^{2^{k_1\text{-}2}} \times ... \times DPQ^{2^{k_n\text{-}2}} \tag{6}$$

with mapping

$$(Q^1, ..., Q^n) \mapsto$$
$$((Q^1, s_\emptyset), .., (Q^1, s_{(2..k_1\text{-}1)}), .., (Q^n, s_\emptyset), .., (Q^n, s_{(2..k_n\text{-}1)})).$$

Explicitly, given an arbitrary pattern query, F first decomposes it into simple paths of lengths $k_1, ..., k_n$ and then $G_1 \times ... \times G_n$ produces all acceptable combinations of breakpoints of all path queries.

As an example, we identify the distributed queries for Q as in (4). For the path $Q^1 = (x_1, x_2, x_3, x_4)$, $V_{Q^1} \backslash \{x_1, x_4\} = \{x_2, x_3\}$ so there exist k_1-2 $= 2$ possible breakpoints. Their $2^2 = 4$ combinations are $\{\emptyset, \{x_2\}, \{x_3\}, \{x_2, x_3\}\}$, thus the function producing its distributed queries is

$$\begin{aligned} G_1:\ PPQ &\longrightarrow DPQ \times DPQ \times DPQ \times DPQ \\ Q^1 &\mapsto \big((Q^1, \emptyset), (Q^1, \{x_2\}), (Q^1, \{x_3\}), (Q^1, \{x_2, x_3\})\big) \end{aligned}$$

For the path $Q^2 = (y_1, y_2)$, we have $V_{Q^2} \backslash \{y_1, y_2\} = \emptyset$, i.e. no possible breakpoints. Hence $2^0 = 1$ and $G_2 : PPQ \to DPQ$ with $G_2(Q^2) = (Q^2, \emptyset)$. In total, we have the composite function $\mathbf{G} \circ F = PQ \to PPQ^2 \to DPQ^{4+1}$ mapping Q to

$$\begin{aligned} \mathbf{G}(F(Q)) = &\big((Q^1, \emptyset), (Q^1, \{x_2\}), (Q^1, \{x_3\}), (Q^1, \{x_2, x_3\}), \\ &(Q^2, \emptyset) \mid x_3 = y_2\big) \end{aligned} \tag{7}$$

These are the two paths with chosen breakpoints:

$$(x_1 \to x_2 \to x_3 \to x_4, x_1 \to \mathbf{x_2} \to x_3 \to x_4,$$
$$x_1 \to x_2 \to \mathbf{x_3} \to x_4, x_1 \to \mathbf{x_2} \to \mathbf{x_3} \to x_4, y_1 \to x_3).$$

3) Distributed to Fragment pattern queries: The transformation that follows employs the paths' breakpoints information to actually split them into smaller paths. Observe how each element of $\mathcal{P}(V_P \backslash \{x_1, x_k\})$ uniquely corresponds to a specific path cover of the simple path P, e.g. $\{x_m\} \leftrightarrow \{(x_1, .., x_m), (x_m, .., x_k)\}$. Hence any distributed query can equivalently be written as $(P_1, .., P_r)$, where all P_i's are subpaths such that $\cup P_i = P$, and the end node of each P_i concides with the start node of P_{i+1}.

Based on that, we can describe the *fragment pattern queries* (or just fragments) in which every distributed query divides into. Some general facts are the following.

- # fragment queries= # breakpoints+1.
- # distributed queries with $v+1$ fragments= $\binom{k\text{-}2}{v}$, e.g.
  - $\binom{k\text{-}2}{0} = 1$ distributed query with 1 fragment, for choosing 0 breakpoints, i.e. the whole P;
  - $\binom{k\text{-}2}{1} = k$-2 distributed queries with 2 fragments, for choosing all singletons as breakpoints;
  - $\binom{k\text{-}2}{k\text{-}2} = 1$ distributed query with k-1 fragments, for choosing k-2 breakpoints, i.e. decomposing into all its edges.
- # all distributed queries is recovered to be $2^{k\text{-}2} = \binom{k\text{-}2}{0} + \binom{k\text{-}2}{1} + \binom{k\text{-}2}{2} + ... + \binom{k\text{-}2}{k\text{-}2}$.

We can thus express distributed queries in terms of fragments

$$\begin{array}{rl} H: & DPQ \longrightarrow PPQ^r \\ & (P, s_{(-)}) \longmapsto (P_{(-)1}, ..., P_{(-)r}) \end{array} \tag{8}$$

where $P_{(-)i}$ is determined by the chosen breakpoints $s_{(-)}$:

$$(P, s_{(i_1..i_v)}) \leftrightarrow (P_{(i_1..i_v)1}, P_{(i_1..i_v)2}, .., P_{(i_1..i_v)v+1}).$$

Starting with the function G as in (5), we can compose it with the cartesian product $H_1 \times .. \times H_{2^{k\text{-}2}} = \widetilde{H}$ for all different subsets $s_{(-)}$, namely

$$DPQ^{2^{k\text{-}2}} \rightarrow \overbrace{PPQ \times PPQ^{2\binom{k\text{-}2}{1}} \times PPQ^{3\binom{k\text{-}2}{2}} \times .. \times PPQ^{k\text{-}1}}^{2^{k\text{-}2}}$$

with mapping

$$((P,s_\emptyset),(P,s_{(2)}),..,(P,s_{(23)}),..,(P,s_{(2..k\text{-}1)}))\mapsto$$
$$(P,(P_{(2)1},P_{(2)2}),..,(P_{(23)1},P_{(23)2},P_{(23)3}),..,(P_{(2..k\text{-}1)1},..,P_{(2..k\text{-}1)k\text{-}1})).$$

Combining the transformations (3),(5) and (8), we can compose $G_1 \times ... \times G_n$ as in (6) with $\mathbf{H} = \widetilde{H}_1 \times ... \times \widetilde{H}_n$

$$\begin{tikzcd} DPQ^{2^{k_1\text{-}2}} \times ... \times DPQ^{2^{k_n\text{-}2}} \arrow[d, "\mathbf{H}"] \\ (PPQ \times ... \times PPQ^{k_1\text{-}1}) \times ... \times (PPQ \times ... \times PPQ^{k_n\text{-}1}) \end{tikzcd} \tag{9}$$

with mapping, for some pattern query Q,

$$\begin{tikzcd} ((Q^1, s_\emptyset), .., (Q^1, s_{(2..k_1\text{-}1)}), .., (Q^n, s_\emptyset), .., (Q^n, s_{(2..k_n\text{-}1)})) \arrow[d, mapsto] \\ (Q^1, .., Q^1_{(2..k_1\text{-}1)k_1\text{-}1}, .., Q^n, .., Q^n_{(2..k_n\text{-}1)k_n\text{-}1}). \end{tikzcd}$$

Notice that if $T = 1\binom{k\text{-}2}{0} + 2\binom{k\text{-}2}{1} + 3\binom{k\text{-}2}{2} + ... + (k\text{-}1)\binom{k\text{-}2}{k\text{-}2}$,

$$PPQ \times PPQ^{2\cdot(k\text{-}2)} \times PPQ^{3\cdot(k\text{-}2)} \times ... \times PPQ^{k\text{-}1} \cong PPQ^T$$

hence the image of $\mathbf{H}$ is $PPQ^{T_1} \times .. \times PPQ^{T_n}$.

Back to the example query (4), having computed its distributed queries in (7), we can now identify its fragment queries. For Q^1, we have $k_1 = 4$ so there are $\binom{2}{0} = 1$, $\binom{2}{1} = 2$, $\binom{2}{2} = 1$ distributed queries with $0+1 = 1$, $1+1 = 2$, $2+1 = 3$ fragment queries respectively. The function $\widetilde{H}_1 : DPQ^4 \rightarrow PPQ \times PPQ^2 \times PPQ^2 \times PPQ^3$ has as image the list of fragment queries

$$\begin{gathered}(Q^1, (Q^1_{(2)1}, Q^1_{(2)2}), (Q^1_{(3)1}, Q^1_{(3)2}), (Q^1_{(23)1}, Q^1_{(23)2}, Q^1_{(23)3})) = \\ \big((x_1, x_2, x_3, x_4), (x_1, x_2), (x_2, x_3, x_4), (x_1, x_2, x_3), \\ (x_3, x_4), (x_1, x_2), (x_2, x_3), (x_3, x_4)\big)\end{gathered}$$

For Q^2 we have $k_2 = 2$ so $\widetilde{H}_2 : DPQ \rightarrow PPQ$ with image the only fragment query $Q^2 = (y_1, y_2)$. The product of those two functions is composed with $\mathbf{G} \circ F$ to give

$$PQ \xrightarrow{F} PPQ^2 \xrightarrow{\mathbf{G}} DPQ^5 \xrightarrow{\mathbf{H}} PPQ^{(1+2+2+3)+1}$$

with mapping the total list of 10 fragment queries

$$\begin{gathered}\big(x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4, x_1 \rightarrow x_2, x_2 \rightarrow x_3 \rightarrow x_4, \\ x_1 \rightarrow x_2 \rightarrow x_3, x_3 \rightarrow x_4, x_1 \rightarrow x_2, x_2 \rightarrow x_3, \\ x_3 \rightarrow x_4, y_1 \rightarrow y_2 \mid y_2 \equiv x_3\big)\end{gathered} \tag{10}$$

4) Fragment queries to Primary Fragments: In the final list of all fragments $(P, P_{(2)1}, ..., P_{(2..k\text{-}1)k\text{-}1})$ for a path, some entries turn out to be identical, e.g. $P_{(2)1} = (x_1, x_2) = P_{(23)1}$. In this section, the described transformation distinguishes all the unique elements from that list.

We thus consider the *primary fragments* of a path, i.e. all distinct subpaths that are essential to build it up in all ways; for $P = (x_1, .., x_k)$ they are the following:

· #(k-1) 2-paths $(x_1, x_2), (x_2, x_3), \ldots, (x_{k\text{-}1}, x_k)$;
· #(k-2) 3-paths $(x_1, x_2, x_3), \ldots, (x_{k\text{-}2}, x_{k\text{-}1}, x_k)$; $(\ldots)$
· $\#\big(k\text{-}(k\text{-}2) = 2\big)$ $[k\text{-}1]$-paths $(x_1, .., x_{k\text{-}1}), (x_2, .., x_k)$;
· $\#\big(k\text{-}(k\text{-}1) = 1\big)$ k-path $(x_1, .., x_k)$.

In total, we have the above $(k\text{-}1) + (k\text{-}2) + ... + 1 = \frac{(k\text{-}1)k}{2}$ subpaths, which in combination make up all fragment queries.

In order to correspond these to fragments of the previous general form $P_{(-)i}$, so as to be able to discriminate the latter between primary and non-primary, we make the following choices which focus on their source/target:

- subpaths of the form $(x_1,..,x_i)$ or $(x_i,..,x_k)$ — including the start or end path vertex — are $P_{(i)1}$ or $P_{(i)2}$;
- subpaths of the form $(x_i,..,x_j)$ where $1 < i < j < k$ — including only intermediate vertices — are $P_{(ij)2}$;
- $(x_1,..,x_k)$ is just P.

Therefore the above collections of primary fragments are the 2-paths $P_{(2)1}, P_{(23)2}, ..., P_{(k\text{-}2\ k\text{-}1)2}, P_{(k\text{-}1)2}$, the 3-paths $P_{(3)1}$, $P_{(24)2}, ..., P_{(k\text{-}3\ k\text{-}1)2}, P_{(k\text{-}2)2}$ and so on. We can now formulate all fragment queries only using primary fragments, as claimed: for breakpoints $\{x_{m_1}, ..., x_{m_v}\}$, we have

$$\begin{aligned}(P_{(m_1..m_v)1}, P_{(m_1..m_v)2}, .., P_{(m_1..m_v)v+1}) \equiv \\ (P_{(m_1)1}, P_{(m_1 m_2)2}, P_{(m_2 m_3)2}, .., P_{(m_v)2}).\end{aligned} \tag{11}$$

Conclusively, out of the full list of fragments for P, the primary ones are the path itself, both fragments from choosing any <u>single</u> breakpoint (of the form $P_{(i)1}$, $P_{(i)2}$) and the second fragments from choosing any <u>two</u> breakpoints (of the form $P_{(ij)2}$). Hence the set of primary fragments is precisely

$$PF_P = \{P, P_{(i)1}, P_{(i)2}, P_{(ij)2} \mid 2 \le i \le k\text{-}1, i < j \le k\text{-}1\}. \tag{12}$$

We can now define a function

$$\begin{aligned} K : PPQ^T &\longrightarrow \mathbb{PF} \\ (P, P_{(2)1}, P_{(2)2}, ..., P_{(2..k\text{-}1)k\text{-}1}) &\longmapsto PF_P \end{aligned} \tag{13}$$

where $\mathbb{PF}$ is the set $\{PF_Q \mid \text{any } Q \in PQ\}$ of all sets of primary fragments for pattern queries. Combining K with all the previous transformations, we can compose (9) with

$$PPQ^{T_1+...+T_n} \xrightarrow{\mathbf{K}} \mathbb{PF} \tag{14}$$

which maps all fragments from all path queries $Q^1, ..., Q^n$ to the *union* of their primary fragments:

$$\begin{gathered}(Q^1, Q^1_{(2)1}, .., Q^1_{(2..k_1\text{-}1)k_1\text{-}1}, .., Q^n, .., Q^n_{(2...k_n\text{-}1)k_n\text{-}1}) \\ \downarrow \\ \{Q^u, Q^u_{(i_u)1}, Q^u_{(i_u)2}, Q^u_{(i_u j_u)2} \mid {\scriptstyle 2 \le i_u \le k_u\text{-}1, i_u < j_u \le k_u\text{-}1}\}_{1 \le u \le n}.\end{gathered}$$

The image of this final step is $PF_Q = PF_{Q^1} \cup ... \cup PF_{Q^n}$. Notice how, since all $Q^1, .., Q^n$ are edge-disjoint, the sets PF_{Q^u} are all distinct from each other.

Proposition 1: The total number of primary fragments for an arbitrary pattern query Q is

$$\sum^{1 \le u \le n} \frac{k_u(k_u - 1)}{2}$$

where n is the number of simple paths it decomposes into, and $k_1, .., k_n$ are the respective path lengths.

We can now compose all defined functions (3,6,9,14) in order to obtain the transformation $\mathcal{M}$; given an arbitrary pattern query Q, it produces the set of its primary fragments $\mathcal{M}(Q) := PF_Q$. Graphically,

$$\begin{tikzcd}
PQ \arrow[r, "F"] \arrow[rrdd, dotted, "\mathcal{M}"'] & PPQ^n \arrow[r, "\mathbf{G}"] & DPQ^{2^{k_1\text{-}2}} \times .. \times DPQ^{2^{k_n\text{-}2}} \arrow[d, "\mathbf{H}"] \\
 & & PPQ^{T_1} \times ... \times PPQ^{T_n} \arrow[d, "\mathbf{K}"] \\
 & & \mathbb{PF}.
\end{tikzcd}$$

This fulfills the purpose of the current section; an operation which decomposes any pattern query into all fragments necessary to reconstruct it is established.

For our example pattern query (4), we have

$$\begin{aligned} PF_{Q^1} = &\{Q^1, Q^1_{(2)1}, Q^1_{(2)2}, Q^1_{(3)1}, Q^1_{(3)2}, Q^1_{(23)2}\} = \\ &\{(x_1, x_2, x_3, x_4), (x_1, x_2), (x_2, x_3, x_4), \\ &(x_1, x_2, x_3), (x_3, x_4), (x_2, x_3)\} \\ PF_{Q^2} = &\{Q^2\} = \{(y_1, y_2)\} \end{aligned}$$

so via $PQ \xrightarrow{F} PPQ^2 \xrightarrow{\mathbf{G}} DPQ^5 \xrightarrow{\mathbf{H}} PPQ^{10} \xrightarrow{\mathbf{K}} \mathbb{PF}$ we have the full set of primary fragments for Q

$$\begin{aligned} \mathcal{M}(Q) = PF_Q = PF_{Q^1} \cup PF_{Q^2} = \{x_1 \to x_2 \to x_3 \to x_4, \\ x_1 \to x_2, x_2 \to x_3 \to x_4, x_1 \to x_2 \to x_3, \\ x_3 \to x_4, x_2 \to x_3, y_1 \to y_2 \mid y_2 \equiv x_3\} \end{aligned} \tag{15}$$

with precisely $\frac{4(4-1)}{2} + \frac{2(2-1)}{2} = 6 + 1 = 7$ elements; compare with the list (10) of 10 elements, some of them unneeded duplicates.

C. Partition results combination

1) Primary fragments to Fragment Result Sets: The next step is to execute searches in the partitions $G_1, ..., G_m$ of our distributed graph database G, in order to identify paths isomorphic to the primary fragments, and then appropriately 'join' them in order to reconstruct some part of the initial query. Due to the REF and RFD characteristic discussed in III-A, we can add further restrictions for the start/end vertices of the fragment paths — in order to reduce the candidate vertices — based on whether a fragment located in some partition is destined to be joined with another of a different partition.

For some partition $G_w \in \{G_1, ..., G_m\}$, define the sets of vertices $R_w = \{x \in V_w \mid x \text{ is } REF\}$, $D_w = \{x \in V_w \mid x \text{ is } RFD\}$. Notably $R_w \cap D_w = \emptyset$, since the RFD label pins the 'real' location of the vertex based on the partitioning algorithm, whereas all REF-labeled vertices are copies in different partitions. If we consider all such vertices from all partitions,

$$R_G = R_1 \cup ... \cup R_m = \mathcal{I} = D_1 \cup ... \cup D_m = D_G$$

for $\mathcal{I}$ the set of all duplicated vertices as in (1).

Define the following four classes of *partition fragment result sets*, corresponding to the four types of primary fragments for each path Q^u as in (12):

$$\begin{aligned} FRS_{G_w}(Q^u) =& \{\bar{Q} \in G_w \mid \bar{Q} \cong Q^u\} \qquad (16)\\ FRS_{G_w}(Q^u_{(i)1}) =& \{\bar{Q} \in G_w \mid \bar{Q} \cong Q^u_{(i)1} {\wedge} e_{\bar{Q}} {\in} R_w\}\\ FRS_{G_w}(Q^u_{(i)2}) =& \{\bar{Q} \in G_w \mid \bar{Q} \cong Q^u_{(i)2} {\wedge} s_{\bar{Q}} {\in} D_w\}\\ FRS_{G_w}(Q^u_{(ij)2}) =& \{\bar{Q} \in G_w \mid \bar{Q} \cong Q^u_{(ij)2} {\wedge} e_{\bar{Q}} {\in} R_w {\wedge} s_{\bar{Q}} {\in} D_w\} \end{aligned}$$

For each primary fragment of each path of Q, we take the union of these sets to form the following $\sum_{u=1}^{n} \frac{k_u(k_u\text{-}1)}{2}$ *fragment result sets*

$$FRS^u_{-} \equiv FRS(Q^u_{-}) := FRS_{G_1}(Q^u_{-}) \cup ... \cup FRS_{G_m}(Q^u_{-}). \qquad (17)$$

Specifically, we have $FRS^u = \{\bar{Q} \in G \mid \bar{Q} \cong Q^u\}$, $FRS^u_{(i)1} = \{\bar{Q} \in G_w \mid \bar{Q} \cong Q^u_{(i)1} \wedge e_{\bar{Q}} \in R_w$ for some $w\}$, $FRS^u_{(i)2} = \{\bar{Q} \in G \mid \bar{Q} \cong Q^u_{(i)2} \wedge s_{\bar{Q}} \in D_G\}$ and $FRS^u_{(ij)2} = \{\bar{Q} \in G_w \mid \bar{Q} \cong Q^u_{(ij)2} \wedge e_{\bar{Q}} \in R_w$ for some $w\}$. These sets contain all isomorphic instances of the primary fragments in the database, all together in large collections, where the partition from which each of them originates is forgotten.

In general, it is convenient to think of these sets FRS^u_{-} as *relational tables*, with attributes the 'names' of the vertices of Q they refer to, and tuples the graph-isomorphic paths appearing in the partitions.

We can now define the full set of fragment result sets for a specific pattern query Q:

$$FRS_Q = \{FRS^u, FRS^u_{(i)1}, FRS^u_{(i)2}, FRS^u_{(ij)2}\}$$

with $1 \leq u \leq n$ and appropriate ranges for i, j. If $\mathbb{FRS} = \{FRS_Q \mid \text{any } Q\}$ is the set of all sets of fragment result sets for arbitrary pattern queries, consider the function

$$\begin{aligned} \mathfrak{F} : \mathbb{PF} &\xrightarrow{\qquad\qquad} \mathbb{FRS} \qquad (18)\\ PF_Q = \{Q^u_{-}\} &\longmapsto FRS_Q = \{FRS^u_{-}\} \end{aligned}$$

Clearly, the size of FRS_Q is also $\sum \frac{k_u(k_u-1)}{2}$.

For our example pattern query as in (4), the 7 fragment result sets are of the form

$$\begin{aligned} FRS_1 := FRS^1 =& \{[x_1 \to x_2 \to x_3 \to x_4]\}\\ FRS_2 := FRS^1_{(2)1} =& \{[x_1 \to x_2] \mid x_2 \text{ is } REF\}\\ FRS_3 := FRS^1_{(2)2} =& \{[x_2 \to x_3 \to x_4] \mid x_2 \text{ is } RFD\}\\ FRS_4 := FRS^1_{(3)1} =& \{[x_1 \to x_2 \to x_3] \mid x_3 \text{ is } REF\}\\ FRS_5 := FRS^1_{(3)2} =& \{[x_3 \to x_4] \mid x_3 \text{ is } RFD\}\\ FRS_6 := FRS^1_{(23)2} =& \{[x_2 \to x_3] \mid x_2 \text{ is } RFD \wedge x_3 \text{ is } REF\}\\ FRS_7 := FRS^2 =& \{[y_1 \to y_2]\} \end{aligned}$$

where $[-]$ denotes all the graph-isomorphic instances of the path arising in some partition of G. The mapping $\mathfrak{F}(PF_Q) = FRS_Q$ is the set containing all the above sets, with the condition that $y_2 = x_3$.

Remark 1: There may be cases when certain primary fragments of some pattern query Q are graph-isomorphic. It is then not required to search the partitions multiple times; rather it seems optimal to first identify them and only query once. For the above example, PF_Q as in (15) already includes some isomorphic fragments, e.g. $Q^1_{(2)1} \cong Q^1_{(3)2} \cong Q^1_{(23)2} \cong Q^2$: these are all (isomorphic) 2-paths. Hence if we populate the set/table FRS^2, we can then obtain $FRS^1_{(2)1}$, $FRS^1_{(3)2}$ and $FRS^1_{(23)2}$ by selecting the ones with the extra label restrictions, rather than querying three more times.

2) Fragment Result Sets to Distributed Result Sets: After querying the database to specify (populate, when viewed as tables) all the separate $FRS(Q^u_{-})$'s in the previous step, we can form sets of isomorphic instances of the distributed queries for Q, which will be constructed 'algebraically' from the fragment result sets.

An important advantage of viewing result sets as relational tables is that we can *join* them in the usual sense, i.e. employ extension of natural join $\bowtie$ of binary relations, as described in [16]. We can construct various joins of the fragment result sets of some path P, e.g.

$$\begin{aligned} & FRS_{(i)1} \bowtie FRS_{(i)2} = \{(y_1, ..., y_k) \mid \\ & (y_1, ..., y_i) \in FRS_{(i)1} \wedge (y_i, ..., y_k) \in FRS_{(i)2}\}. \end{aligned}$$

The other cases needed for our purposes are $FRS_{(i)1} \bowtie FRS_{(ij)2}$, $FRS_{(ij)2} \bowtie FRS_{(jk)2}$ and $FRS_{(ij)2} \bowtie FRS_{(j)2}$, for all appropriate i, j, k.

For Q decomposed in paths $\{Q^1, ..., Q^n\}$, we define the *distributed result sets*

$$\begin{aligned} & DRS^u_1 {\equiv} DRS(Q^u, \emptyset) = FRS^u \qquad (19)\\ & DRS^u_{(i)} {\equiv} DRS(Q^u, s_{(i)}) = FRS^u_{(i)1} \bowtie FRS^u_{(i)2}\\ & DRS^u_{(ij)} {\equiv} DRS(Q^u, s_{(ij)}) = FRS^u_{(i)1} \bowtie FRS^u_{(ij)2} \bowtie FRS^u_{(j)2}\\ & DRS^u_{(ijk)} {\equiv} DRS(Q^u, s_{(ijk)}) = FRS^u_{(i)1} \bowtie FRS^u_{(ij)2}\\ & \qquad \bowtie FRS^u_{(jk)2} \bowtie FRS^u_{(k)2} \qquad \text{etc.} \end{aligned}$$

where each set is formed following the reconstruction rules of fragment queries from primary fragments from (11). Obviously, for any such Q with lengths of paths $k_1, .., k_n$, the total number of the distributed result sets/tables is $\sum_{u=1}^{n} 2^{k_u\text{-}2}$.

We can now define the set of all distributed result sets

$$DRS_Q = \{DRS^u_1, DRS^u_{(i)}, DRS^u_{(ij)}, ... \mid 1 \leq u \leq n\}$$

as well as the set of all sets of distributed result sets for arbitrary pattern queries, $\mathbb{DRS} = \{DRS_Q \mid \text{any } Q\}$. We can now define a function using the join construction formulas

$$\begin{aligned} \mathfrak{D} : \mathbb{FRS} &\xrightarrow{\qquad\qquad} \mathbb{DRS}\\ FRS_Q = \{FRS^u_{-}\} &\longmapsto DRS_Q = \{DRS^u_1, DRS^u_{(i)}, ..\} \end{aligned} \qquad (20)$$

In our example (4), we have the following $2^2 + 2^0 = 5$ distributed result sets:

$$DRS^1_1 = \{[x_1 \to x_2 \to x_3 \to x_4]\}$$

$$DRS^1_{(2)} = \{x_1 \to x_2 \to x_3 \to x_4 | x_1 \to x_2 \in FRS^1_{(2)1} \wedge x_2 \to x_3 \to x_4 \in FRS^1_{(2)2}\}$$

$$DRS^1_{(3)} = \{x_1 \to x_2 \to x_3 \to x_4 | x_1 \to x_2 \to x_3 \in FRS^1_{(3)1} \wedge x_3 \to x_4 \in FRS^1_{(3)2}\}$$

$$DRS^1_{(23)} = \{x_1 \to x_2 \to x_3 \to x_4 | x_1 \to x_2 \in FRS^1_{(2)1} \wedge x_2 \to x_3 \in FRS^1_{(23)2} \wedge x_3 \to x_4 \in FRS^1_{(4)2}\}$$

$$DRS^2_1 = \{[y_1 \to y_2]\}$$

Notice that these sets contain multiple graph-isomorphic paths to Q^1 and Q^2 obtained in different ways, and the 'names' x_i are only used as attributes of the tables where the tuples of these sets are stored in. The image $\mathfrak{D}(\mathfrak{F}(PF_Q)) = DRS_Q$ contains all these sets, with the condition $y_2 = x_3$.

3) Distributed Result Sets to Path Result Sets: Having specified all isomorphic distributed queries arising in the database — which are in fact the paths $Q^1, ..., Q^n$ formed by joining fragments in different intermediate nodes each time — we now wish to assemble the whole result table for each path, ignoring the way they were constructed.

Define the *path result set* to be the union of those $DRS^u_{(-)}$ for each path $Q^u \in \{Q^1, ..., Q^n\}$ individually:

$$PRS^u = DRS^u_1 \cup DRS^u_{(2)} \cup DRS^u_{(3)} \cup .. \cup DRS^u_{(23..k\text{-}1)} \tag{21}$$

Consider the set $PRS_Q = \{PRS^1, ..., PRS^n\}$ of all path result sets. If $\mathbb{PRS} = \{PRS_Q \mid \text{any } Q\}$ contains all sets of path result sets for arbitrary pattern queries, we form the transformation

$$\begin{tikzcd} \mathfrak{P} : \mathbb{DRS} \arrow[r] & \mathbb{PRS} \\ DRS_Q = \{DRS^u_{(-)}\}^n_{u=1} \arrow[r, mapsto] & PRS_Q = \{PRS^u\}^n_{u=1}. \end{tikzcd} \tag{22}$$

For our example query (4), we obtain the following path result sets for Q^1 and Q^2:

$$\begin{aligned} PRS^1 = {} & DRS^1_1 \cup DRS^1_{(2)} \cup DRS^1_{(3)} \cup DRS^1_{(23)} = \{x_1 \to x_2 \to x_3 \to x_4 | \\ & x_1 \to x_2 \to x_3 \to x_4 \in FRS_1 \vee \Big(x_1 \to x_2 \in FRS_2 \wedge x_2 \to x_3 \to x_4 \in FRS_3\Big) \vee \\ & \Big(x_1 \to x_2 \to x_3 \in FRS_4 \wedge x_3 \to x_4 \in FRS_5\Big) \vee \\ & \Big(x_1 \to x_2 \in FRS_2 \wedge x_2 \to x_3 \in FRS_6 \wedge x_3 \to x_4 \in FRS_5)\Big\} \end{aligned}$$

$$PRS^2 = DRS^2_1 = \{[y_1 \to y_2]\}.$$

Hence the image of the composition of the transformations gives $\mathfrak{P}(\mathfrak{D}(\mathfrak{F}(PF_Q))) = PRS_Q = \{PRS^1, PRS^2 \mid x_3 \equiv y_2\}$.

4) Path Result Sets to Base Result Sets: Now that we have gathered all paths that are isomorphic to $Q^1, ..., Q^n$ inside the distributed graph database, we recall that the *join vertices* of the weakly connected pattern query Q constitute common 'attributes' of the path queries results when stored in tables. Thus we can form the *base result set*

$$BRS_Q = PRS^1 \bowtie PRS^2 \bowtie ... \bowtie PRS^n \tag{23}$$

where the natural join is performed over the common (join) vertices of the paths. Consequently, we can view the set BRS_Q as a table, with attributes the names of the discrete vertices of Q, and tuples all of its isomorphic subgraphs inside G. If $\mathbb{BRS} = \{BRS_Q \mid \forall Q \in PQ\} \subseteq \mathcal{P}(PQ)$ is the set of all sets of base result sets for arbitrary Q's, the above join defines the transformation

$$\begin{tikzcd} \mathfrak{B} : \mathbb{PRS} \arrow[r] & \mathbb{BRS} \\ PRS_Q = \{PRS^1, ..., PRS^n\} \arrow[r, mapsto] & BRS_Q. \end{tikzcd} \tag{24}$$

The full process of assembling the results of primary fragments in the partitions into isomorphic instances of the initial query is thus given by the composite transformation of (18,20,22,24)

$$\begin{tikzcd} \mathbb{PF} \arrow[r, "\mathfrak{F}"] \arrow[ddrr, dotted, bend right, "\mathcal{L}"'] & \mathbb{FRS} \arrow[r, "\mathfrak{D}"] & \mathbb{DRS} \arrow[d, "\mathfrak{P}"] \\ & & \mathbb{PRS} \arrow[d, "\mathfrak{B}"] \\ & & \mathbb{BRS} \end{tikzcd}$$

For the example query Q as in (4), we end up with

$$\mathfrak{B}(\mathfrak{P}(\mathfrak{D}(\mathfrak{F}(PF_Q)))) = PRS^1 \bowtie PRS^2 =$$
$$\{(x_1, x_2, x_3, x_4, y_1) | (x_1, x_2, x_3, x_4) \in PRS^1 \wedge (y_1, x_3) \in PRS^2\}$$

where the join is performed over the unique join vertex in $\mathcal{J} = \{x_3\}$. Notice that this result set BRS_Q of all isomorphic subgraphs to Q would contain also H of the distributed setting example (2):

$$(n, m^{REF}) \in FRS^1_{(2)1} \wedge (m^{RFD}, l, o) \in FRS^1_{(2)2} \Rightarrow$$
$$(n, m, l, o) \in DRS^1_{(2)} \subseteq PRS^1,$$
$$(k, l) \in FRS^2 = DRS^2 = PRS^2.$$

We conclude with a result which ensures the validity of our method (we omit the proof due to space restriction).

Theorem 1: By performing the series of transformations $\mathcal{L} \circ \mathcal{M}$ depicted by the dotted line

$$\begin{tikzcd} PPQ^{T_1} \times .. \times PPQ^{T_n} \arrow[rr] & & \mathbb{PF} \arrow[d] \arrow[dddl, dotted, "\mathcal{L}"'] \\ DPQ^{2^{k_1\text{-}2}} \times .. \times DPQ^{2^{k_n\text{-}2}} \arrow[u] & & \mathbb{FRS} \arrow[d] \\ PPQ^n \arrow[u] & & \mathbb{DRS} \arrow[d] \\ PQ \arrow[u] \arrow[r, dashed, "\mathcal{L}\circ\mathcal{M}"] \arrow[uuurr, dotted, "\mathcal{M}"] & \mathbb{BRS} & \mathbb{PRS} \arrow[l] \end{tikzcd}$$

we obtain <u>all</u> isomorphic instances of the initial query Q in the distributed graph database.

IV. Experimental Evaluation

A. Implementation Details and Setup

The *Digree* prototype is implemented as a middleware system that spans over a set of distributed graph databases. It consists of two separate software units, the master and the slave. A basic *Digree* deployment is consisted of one master node, k slave nodes and an extra database node. Each slave node is assigned a graph partition that is loaded in a graph data management system (e.g. graph database, relational database or other). While in our prototype implementation we use Neo4j to manage each graph partition, this is not a restrictive choice. In order to use a different data store (or a combination of different data stores), all is required is the interfaces to the respective query languages or APIs. We deployed our system on a cluster consisting of 18 Linux virtual machines. Each machine had 4 cpus and 8 GB of memory. We used one machine to run the master process, one machine to host a PostgreSQL database server and the rest 16 machines to host the partitions of the graph database (one Neo4j database per node) and the slave processes. For comparison purposes we used a virtual machine of the same specifications to run the queries on a stand alone Neo4j instance loaded with the unpartitioned dataset (from now on refered to as single-node).

B. Datasets

We used three real world datasets for our experiments. The first two are the amazon product network[1] [17] and the youtube video graph.[2] The third dataset that we used is an anonymized twitter users graph parsed by our team. The details of the datasets are shown in Table I (size refers to a single partition Neo4j database size on disk). For partitioning the smaller datasets (amazon and youtube) we used the popular METIS [18] algorithm, while for the twitter dataset we used the online partitioning algorithm LDG [19] since METIS could not handle that large an input.

Table I
Datasets Overview

	#nodes	#edges	#labels	size
amazon	548.552	1,788,725	11	99.6 MB
youtube	155,513	2,969,826	14	161.8 MB
twitter	35,648,794	910,526,369	232	30.85 GB

C. Experiments

1) Pattern Queries: For each of the datasets we built a set of pattern matching queries. Specifically we used seven pattern queries used in [1] (depicted in Figure 1). For each of the datasets and for each pattern query we created ten instances, each with a different random set of node labels. We set a time limit of 1000 seconds for the queries to finish execution, in order to reduce the effect of strangling queries in the average computation for either system. In Figure 2(a) we provide the average execution time of the finished queries run in single-node and *Digree*, while in Figure 3(a) we present the percentage of the queries that returned their result set within the time limit. We can see that for the smaller datasets (amazon and youtube) the differences between the two implementation are not significant. Nevertheless, *Digree* manages to produce reduced query times because of the effect of parallel execution. For the larger dataset (twitter), where the graph cannot fit in the main memory of the single-node instance, the results are more interesting. Not only execution time is more than halfway down, but *Digree* also manages to answer more queries than the single-node (96% against 89%) within the time limit. For the rest of the experiments we present results only for the larger dataset (twitter) since *Digree* is aimed for big datasets that don't easily fit on main memory of a single node.

2) Mutated Pattern Queries: We continued by using the pattern queries from the previous experiment and creating a number of mutations for each of those. These mutations have been created by randomly choosing a vertex from the graph query and attaching a new vertex to it, randomly incoming or outgoing. We refer to the number of vertices added as the mutation degree. For each pattern query we created mutations with degree from 1 to 5 and for each one we created 10 query instances, each with a different random set of labels. We applied the same time limit as before and the results are shown in Figures 2(b) and 3(b). For this more challenging set of queries *Digree* reduces execution time to less than one third of the time required for single-node execution. Moreover, it manages in answering more queries within the time limit than the single-node.

3) Path Queries: Digree utilizes path queries as its basic tool of decomposing a complex graph pattern. In the final experiment we demonstrate the effect of parallel processing of path queries that result in the reduced query times observed in the previous experiments. We used path queries varying in length from 3 to 8 vertices. For each path length we created 100 query instances, each with a different random set of labels and we used the same time limit as before. The results are presented in Figures 2(c) and 3(c). *Digree* manages to execute the queries in a small fraction of the time requirered by single-node, showing a small increase in execution time as the length of the path increases. *Digree* is between 9 and 20 times faster that the single-node deployment.

V. Related Work

There exists a number of distributed graph databases. Trinity [20] is an in-memory distributed graph database used for online query processing and offline analytics on large graphs as well as for subgraph matching [21]. Titan is an open source graph database layered upon a distributed

[1] https://snap.stanford.edu/data/

[2] http://netsg.cs.sfu.ca/youtubedata/

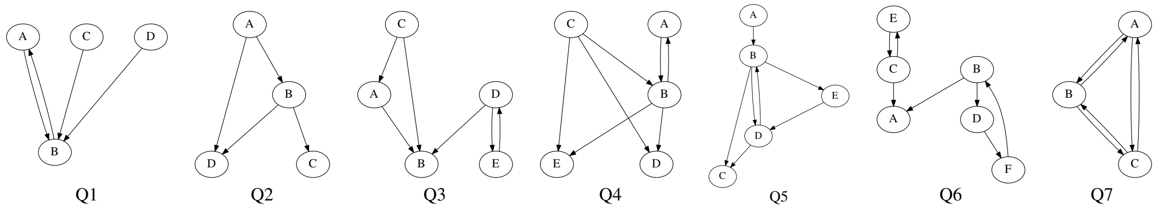
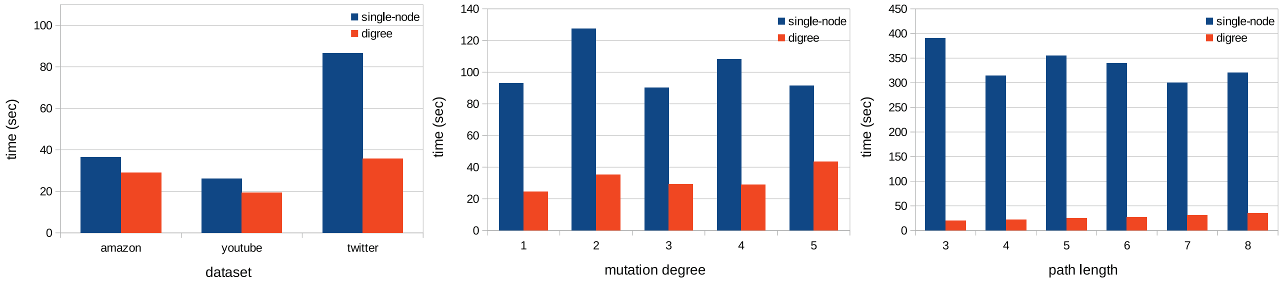


Figure 1. The pattern queries used in the experiments

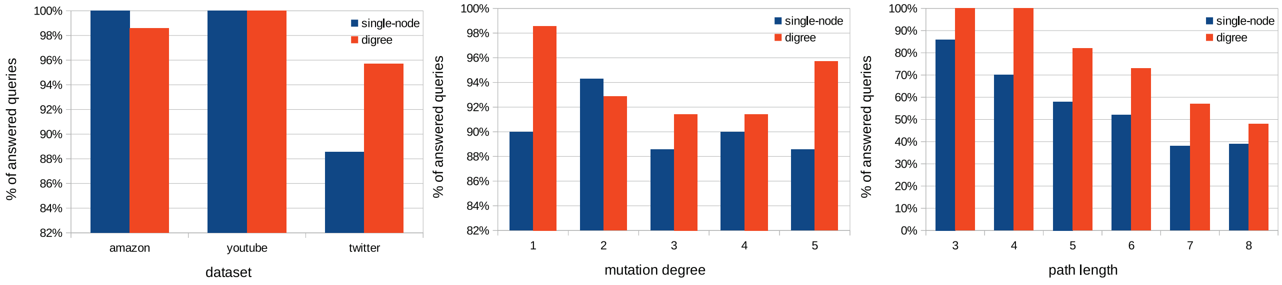


(a) pattern queries

(b) mutated pattern queries

(c) path queries

Figure 2. Average execution time



(a) pattern queries

(b) mutated pattern queries

(c) path queries

Figure 3. Percentage of queries answered within time limit of 1000 seconds

NoSQL database like HBase. ThingSpan [22] is a commercial graph analytics platform that handles very large graphs by utilizing a number of open source big data tools. *Digree* can act as a middleware so as to utilize any such system or combination of systems deployed at the graph partitions.

A number of systems were developed for distributed graph processing such as Google Pregel [4], it's open source alternative Apache Giraph [23] and GraphX [3]. These systems are not specialized in graph pattern matching but are frameworks that provide a programming model for the user to develop and deploy graph algorithms.

A work that concentrates on distributed pattern matching can be found in [1] where the authors use relational data storage and their work is closely related to traditional database system query optimization. *Digree* can easily operate over relational graph management solutions, by building the required API calls. In [24] the authors propose algorithms that use the message passing model and apply their ideas for graph simulation [25], while in [26] the authors propose a distributed solution for graph mining. In [27] the authors present GraphMat, a framework for writing high performance parallel graph algorithms taking advantage of multicore architectures in a single-node deployment. In [28] the authors present PSgL, a distributed solution for subgraph listing, which is a special case of matching when all vertices have the same label. A large amount of related work exists in the context of semantic web. In [6] the authors propose an approach where a SPARQL query is executed at all partitions and partial RDF matches of it are then assembled so as to build the cross-partition results, while in [29] the authors focus on data partitioning and propose a mapreduce based join solution for inter-partition query answering. In [30] the authors propose a partitioning of the RDF data that adopts to workload changes in order to better support queries. The authors in [7] focus on graph partitioning and how it affects their mapreduce based system performance. In [31] the authors examine how caching of SPARQL query results can favor execution of queries over large RDF graphs.

VI. Conclusion

In this paper we presented *Digree*, a middleware to handle graph pattern matching queries over a distributed graph database or inter-linked graph databases. We developed a solid theoretical base to rewrite a graph query so as to be executed in a distributed setting in parallel, but also to combine back the partial results into the final result set. We presented a prototype implementation of *Digree* and demonstrated its capacity in reducing execution times of complex pattern matching queries, especially for datasets that do not fit in main memory of a single node. As future work, we plan to compare with other systems such as GraphX [3] and to use a variety of systems to manage the graph partitions or the central processing engine of *Digree*.

References

[1] J. Huang, K. Venkatraman, and D. J. Abadi, "Query optimization of distributed pattern matching," in *ICDE*, 2014.

[2] D. Bleco and Y. Kotidis, "Graph analytics on massive collections of small graphs," in *Proceedings of EDBT*, 2014.

[3] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13, 2013.

[4] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. of the ACM SIGMOD*, 2010.

[5] V. Spyropoulos and Y. Kotidis, "Dynamic partitioning of big hierarchical graphs," in *Proceedings of the First International Workshop on Big Dynamic Distributed Data, 2013*.

[6] P. Peng, L. Zou, M. T. Özsu, L. Chen, and D. Zhao, "Processing sparql queries over distributed rdf graphs," *The VLDB Journal*, vol. 25, no. 2, pp. 243–268, Apr. 2016.

[7] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *PVLDB*, vol. 4, no. 11, 2011.

[8] (2016, aug) Neo4j. [Online]. Available: http://neo4j.com/

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[10] A. Elmore, J. Duggan, M. Stonebraker, M. Balazinska, U. Cetintemel, V. Gadepally, J. Heer, B. Howe, J. Kepner, T. Kraska, S. Madden, D. Maier, T. Mattson, S. Papadopoulos, J. Parkhurst, N. Tatbul, M. Vartak, and S. Zdonik, "A demonstration of the bigdawg polystore system," *Proc. VLDB Endow.*, Aug. 2015.

[11] B. Gallagher, "Matching structure and semantics: A survey on graph-based pattern matching," *AAAI FS*, vol. 6, 2006.

[12] I. Filippidou and Y. Kotidis, "Online and on-demand partitioning of streaming graphs," in *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, 2015.

[13] I. Filippidou and Y. Kotidis, "Online partitioning of multi-labeled graphs," in *Proceedings of the GRADES*, 2015.

[14] R. Diestel, *Graph Theory*, ser. Graduate Texts in Mathematics. Springer, 1997, no. 173.

[15] J.-A. Bondy and U. S. R. Murty, *Graph theory*, ser. Graduate texts in mathematics. New York, London: Springer, 2007.

[16] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[17] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Trans. Web*, 2007.

[18] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proceedings of IEEE/ACM SC Conf.*, 1995.

[19] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. of ACM SIGKDD*, 2012.

[20] Y. L. Bin Shao, Haixun Wang, "Trinity: A distributed graph engine on a memory cloud," in *Proc. of SIGMOD*, 2013.

[21] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proc. VLDB Endow.*, 2012.

[22] (2016, aug) Thingspan. [Online]. Available: http://www.objectivity.com/

[23] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proc. VLDB Endow.*, vol. 8, no. 9, May 2015.

[24] S. Ma, Y. Cao, J. Huai, and T. Wo, "Distributed graph pattern matching," in *Proceedings of WWW*, 2012, pp. 949–958.

[25] W. Fan, X. Wang, Y. Wu, and D. Deng, "Distributed graph simulation: Impossibility and possibility," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1083–1094, Aug. 2014.

[26] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proceedings of SOSP*, 2015.

[27] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015.

[28] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *Proc. of the 2014 ACM SIGMOD Int. Conference on Management of Data*.

[29] K. Lee and L. Liu, "Scaling queries over big rdf graphs with semantic hash partitioning," *Proc. VLDB Endow.*, vol. 6, no. 14, pp. 1894–1905, Sep. 2013.

[30] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning," *The VLDB Journal*, 2016.

[31] N. Papailiou, D. Tsoumakos, P. Karras, and N. Koziris, "Graph-aware, workload-adaptive sparql query caching," in *Proc. of ACM SIGMOD*, 2015.