Digree: Building A Distributed Graph Processing Engine out of Single-node Graph Database Installations*

Vasilis Spyropoulos
Athens University of Economics and Business
Athens, Greece
vasspyrop@aueb.gr

Yannis Kotidis
Athens University of Economics and Business
Athens, Greece
kotidis@aueb.gr

ABSTRACT

In this work we present *Digree*, a system prototype that enables distributed execution of graph pattern matching queries in a cloud of interconnected graph databases. We explain how a graph query can be decomposed into independent sub-patterns that are processed in parallel by the distributed independent graph database systems and how the results are finally synthesized at a master node. We experimentally compare a prototype of our system against a popular big data engine and show that *Digree* provides significantly faster query execution.

1. INTRODUCTION

Attempts to utilize relational databases and big data systems for storing and querying graph datasets are often hindered by the fact that neither technology natively supports navigational primitives over the graph structure. For instance, evaluating simple path expressions requires costly joins between tables storing adjacency list data in a relational system. Native graph databases permit much faster execution of navigational primitives because they promote object relationships as first class citizens in their storage model. Moreover, they offer declarative access to the underlying graph via high-level languages like Cypher.

In this work we utilize graph databases as local worker nodes in a distributed system, *Digree*, which is used to manage large graph datasets. User queries, in the form of graph patterns are decomposed into smaller elements, utilizing the capabilities of the underlying graph databases to process path expressions. These expressions are executed in parallel by all worker nodes and their results are transmitted to a master node, where intermediate answers are consolidated in order to form the final result set to the user query. Key to the success of the proposed architecture are (i) the efficiency of the native local graph databases in processing path expressions and (ii) the increased parallelism offered by the query decomposition process that enables all worker nodes to contribute in evaluating a user expression.

In our prior work [1], we have formally described the query decomposition phase and the subsequent synthesis of the intermediate results and proven their correctness. In this paper, we first illustrate these processes using a simple running example. We then discuss a new greedy heuristic that leads to an efficient decomposition of a user pattern query. We additionally present new optimizations that we employ in order to expedite the execution of complex graph patterns. The first optimization termed early termination is used to detect when the distributed execution will return an empty result before all distributed processing is concluded. This is important since often query patterns cannot be matched against the data graph and early identification of such scenarios helps avoid unnecessary utilization of resources in the distributed system. The second optimization is used to push additional filters towards the local workers in order to reduce selectivity and, consequently, the sizes of intermediate results. We then describe the architecture of our *Digree* system prototype and compare its performance against a popular big data system extended to support graph pattern matching queries.

2. OVERVIEW

2.1 Data Model

The scale of modern graph datasets such as those encountered in social network applications, easily overwhelms single node installations. This necessitates multi-node deployments that partition the large graphs intro smaller chunks that are managed

*This research is financed by the Research Centre of Athens University of Economics and Business, in the framework of the project entitled 'Original Scientific Publications'

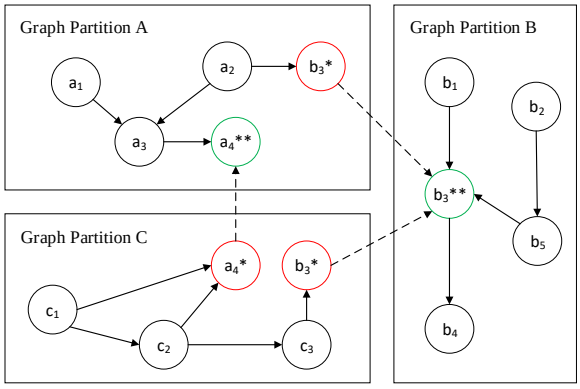


Figure 1: Example of a distributed Graph Dataset. A single asterisk by the name of a node indicates the special label REF. A double asterisk indicates the label $REFED$.

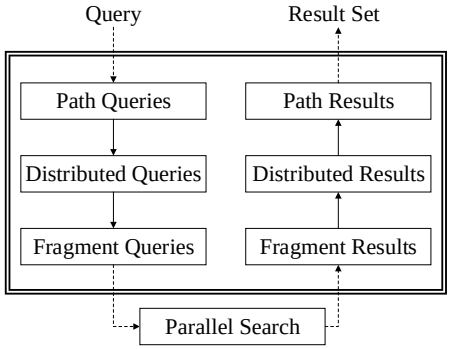


Figure 2: A high-level overview of the process we follow in order to answer a graph pattern query. Downward and upward directions respectively show the query decomposition and results combination processes.

by different servers [2]. Following this premise, *Digree* manages graph datasets that are partitioned across a number of worker nodes. Each graph partition is managed by a local graph database, which in our implementation is Neo4j. From an architecture perspective, *Digree* can utilize any graph database engine or combination of graph databases running at the worker nodes, as long as they implement a basic API for querying path expressions.

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. *Digree* assumes that when a node u in graph partition GP_1 has an outgoing edge to a node v that exists in another partition GP_2 then:

- in GP_1 we maintain a local reference v' to v. We apply to it the label REF, indicating that this node is a REFerence to "remote" node v.
- all nodes in GP_1 that have outgoing edges to v are using the same single reference v'.
- at GP_2 we append to node v the label $REFED$, indicating that the node is REFErenceD by a remote node.

An example is shown in Figure 1. We note that the partitioning process is happening during ingestion of a new dataset and is orthogonal to the techniques we present here. Any partitioning algorithm, including dynamic partitioning techniques [2, 3] can be used as long as special care is taken in order to assign the aforementioned labels into the boundary nodes. These nodes can be located by following cross-edges between the graph partitions in a static scheme or when performing node migration in a dynamic partitioner [2]. Moreover, this labeling is performed at the system level in a manner that is transparent to the applications using the data that need not be concerned with the details of the graph partitioning layout.

2.2 Query Processing

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 [4].

Digree takes as input a pattern query and decomposes it into smaller elements that are processed in parallel by the worker nodes. All computed results are shipped to a designated master node that combines the partial results to produce the global result set. Figure 2 presents an overview of the operations that decompose an input pattern query into smaller sub-patterns that are processed independently by the graph databases. The results to these sub-pattern queries are fused by *Digree* in order to compile the final result set. In what follows we use a running example to illustrate this process.

Taking as input the graph pattern query (from now on referred to as base query) depicted in Figure 3, *Digree* first decomposes it into a set of path queries. A possible decomposition consisting of two path queries is shown in Figure 4 and consists of path queries $pq1$ and $pq2$. Node B, depicted in orange indicates the location where results from these two path queries need to be "joined" in order to produce matching patterns to the input base query.

Next, we take an intermediate step where for each of the path queries we find out on which nodes

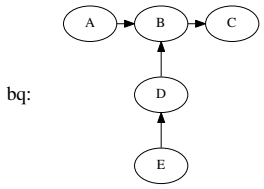


Figure 3: Running Example: The letters inside the depicted nodes are variable names so we can refer to specific nodes of the pattern query. Nodes may also have multiple labels and/or properties which we omit in this simplified example.

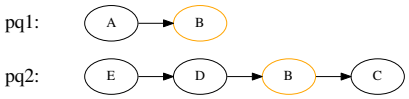


Figure 4: Two path queries obtained from the base query of Figure 3. Orange nodes denote locations where partial results need to be joined.

they should be "taken apart", in order to account for nodes that are possibly stored in different partitions. We refer to these nodes as break-points. This process forms a new set of queries that we call distributed queries and are shown in Figure 5 (the break-points are colored in blue). From path query $pq1$ we generate just one distributed query $dq1$, while path query $pq2$ gives us four different distributed queries, namely $dq2$, $dq3$, $dq4$ and $dq5$.

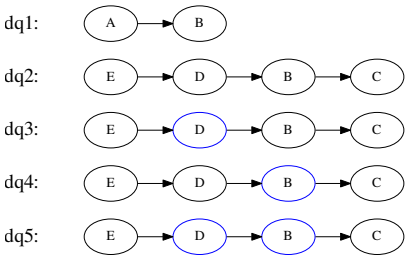


Figure 5: Resulting set of Distributed Queries. Nodes in blue are break-points. These nodes refer to possible locations where a path may be split and the resulting subpaths may be stored in different nodes.

As one can see, we take all combinations of break-point choices which, for every respective path query of length k, results in 2^{k-2} distributed queries, since break-points cannot exist at path start and terminal nodes. The results gathered for each distributed query should be unioned in order to generate the result set for the respective path query.

Each distributed query depending on the break-points it contains generates a number of fragment queries. These are the actual queries that will be submitted to the underlying graph databases. The fragment queries of our example are shown in Figure 6. Essentially, each distributed query represents a possible layout of the path it came from in the distributed setting. For example distributed query $dq4$ ($E \rightarrow D \rightarrow B \rightarrow C$), where the break-point is node B, aims to retrieve all instances of the path where the part $E \rightarrow D$ lies in one database while the fragment $B \rightarrow C$ lies in another one. Though, due to the data model, if such a result exists then node B should exist in both partitions, in the first labeled as REF and in the second as $REFED$. Thus, the fragment queries that will be generated will be $fq5$: $E \rightarrow D \rightarrow B^{REF}$ and $fq6$: $B^{REFED} \rightarrow C$. In the results combination process all fragment results for such fragment sets will be joined on their common node (in our example that one is B). One such joinable pair of fragment results for the example shown in Figure 1 would be $a1 \rightarrow a3 \rightarrow b3^{*}$ and $b3^{**} \rightarrow b4$.

One can see that there are duplicate fragment queries, e.g. $fq3$ and $fq7$. It is sufficient to execute just one instance of those and use the results for all instances.

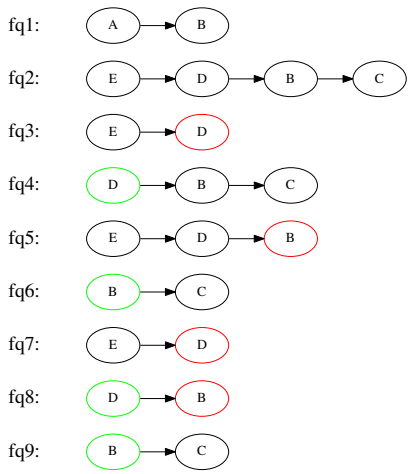


Figure 6: The Fragment Queries that will be submitted to the graph databases. For duplicate sets (such as $fq3$-$fq7$ and $fq6$-$fq9$) we submit just one query and reuse its results.

The full decomposition from the base query all the way down to it's fragment queries queries can be seen in Figure 7. Most of these tasks can be executed in parallel. Fragment queries are submitted to the worker nodes and when all fragment results that correspond to a distributed query's decomposition are gathered the computation of the distributed

query may start. Parallelization is also possible at the "higher" layer when all distributed results that are required to answer a path query are gathered.

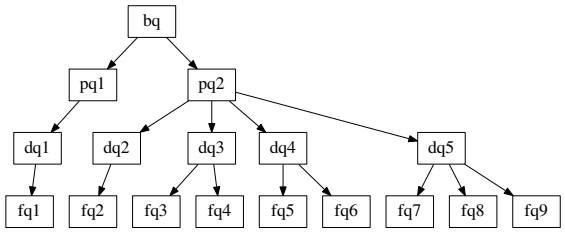


Figure 7: The decomposition mapping for Base Query into Path Queries, Distributed Queries and Fragment Queries (referring to Figures 3, 4, 5 and 6).

2.3 Key Intuition

While most of the approaches in the literature use some kind of decomposition, be it edge-level or subgraph-level of the user query and the subsequent combination of the partial results so as to answer distributed queries, there is a significant advantage in our setting. This is a result of the use of REF and $REFED$ labels at the partitions' boundary nodes. Due to the existence of these special labels we expect, and usually achieve, low selectivity while answering the fragment queries. Take for example fragment queries $E \rightarrow D$, $B \rightarrow C$ and $D \rightarrow B$ (Figure 6). These all contain nodes labeled either REF or $REFED$ and this not only allows for much more efficient execution of the query (based on available indexes at the local nodes) but may significantly reduce the number of results. Other systems that opt to get results for each edge of the input query and then join them using some execution plan [5] would execute the respective plain edge queries ($E \rightarrow D$, $B \rightarrow C$ and $D \rightarrow B$) and end up with a potentially much higher number of results to join at the next step. In our setting the query results that are local to a partition are collected by the execution of longer, more selective paths that contain them, e.g. local results for $E \rightarrow D$ are discovered by the execution of fragment queries $fq2$ ($E \rightarrow D \rightarrow B \rightarrow C$) and $fq5$ ($E \rightarrow D \rightarrow B$).

3. OPTIMIZATIONS

3.1 Path Queries Selection

If we consider the decomposition of the base query into path queries, we are faced with many alternative selections. Consider for example the base query from Figure 3. Instead of decomposing it into the

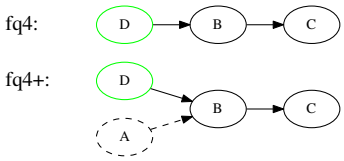


Figure 8: Fragment query $fq4$ and it's augmented version $fq4+$.

path queries $pq1$ and $pq2$ as shown in Figure 4 someone could decompose it into paths $A \rightarrow B \rightarrow C$ and $E \rightarrow D \rightarrow B$, a choice that would affect the further decomposition but also the efficiency of the execution. As the input graph query grows larger the number of choices is increasing and the cost analysis needed to select the best one is non trivial. The design and implementation of such a query optimizer is a work in progress of our own but preliminary results have shown that the system is favoured by the choice of longer path queries. Based on that, for the prototype of *Digree* we decided to use a heuristic algorithm that (i) creates an empty solution list, (ii) enumerates all possible (non-trivial) paths, (iii) adds them in a priority queue in decreasing order by their length, (iv) removes the first path from the queue and adds it to the solution list, and (v) until the base query is covered removes the next path from the queue and adds it to the list if it does not overlap with any of the paths already in the queue.

3.2 Early Termination

Because of the way that the queries are broken into path queries we can assert that if any of the path queries result set is empty, then the result set for the base query is also empty since it is computed by joining the path results. Before submitting the fragment queries to the partitions we sort them by shortest ancestor path query. That way, we are able to get path results early in the query execution and increase the chance to find out that one of them, and consequently the base query, has an empty result set. In that case an interrupt signal is sent throughout the system and the empty result set answer is returned to the user.

3.3 Fragment Query Augmentation

When fragment queries are submitted to the local nodes, they are augmented with additional structural conditions from the base query in order to help the underlying graph database system to answer the query more efficiently. A node in a fragment query is in one of three states. It is either a REF node, a $REFED$ node, or a pure-local node. A pure-local node can be augmented by any other node (and the

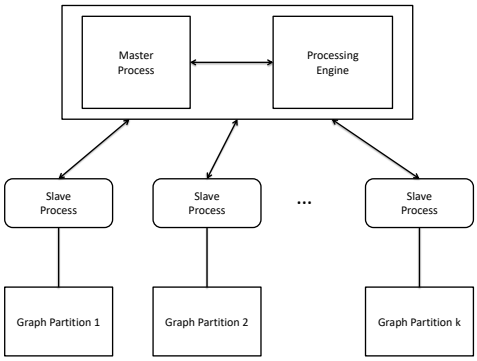


Figure 9: *Digree* architecture overview

related edge) from the base query that is its neighbour, connected either by an incoming or outgoing edge since all of them should exist in the same partition. A $REFED$ node can be augmented by its outgoing edges neighbours since due to our data model all of them should exist in the same partition, but not by its incoming neighbours since at least one of them lies in another partition and we have no way knowing about it. Last, the REF nodes cannot augment since they are actually pseudo-nodes referencing a node existing in another partition.

For example, consider the base query in Figure 3 and the fragment query $fq4$ in Figure 6. Applying this technique, we can augment $fq4$ by the addition of node A and the respective edge $A \rightarrow B$, as shown in Figure 8. It is not possible to further augment $fq4$ by the use of node E since this is an incoming node to node D (the respective edge is $E \rightarrow D$) which, in the context of $fq4$, is a $REFED$ node.

4. SYSTEM ARCHITECTURE

Digree is designed as a distributed system that consists of two main processes, namely the master process and the slave process, and a main processing engine. A deployment should consist of a single instance of the master process, one slave process for each of the managed databases/partitions and one processing engine. The processing engine is a layer over a data management system (e.g. a relational database system, a graph database or a big data system) and handles the temporary storage of the partial results and the operations (union, join) that need to be applied to them. An overview of *Digree* architecture is shown in Figure 9.

The master process receives the user graph query and performs the described decomposition to fragment queries that are then submitted to the slave processes. In order to manage a different graph storage engine one needs to implement an API so that it can communicate with it at least a simple path expression. When a fragment query is answered at a partition the slave process takes care that the results are transferred to the processing engine and also that a signal is sent to the master process. The master, depending on the signals it receives from the slaves, decides when a results combination process can start (e.g. union distributed results to compute a path result set) and triggers the appropriate operation at the processing engine.

Table 1: Datasets Overview

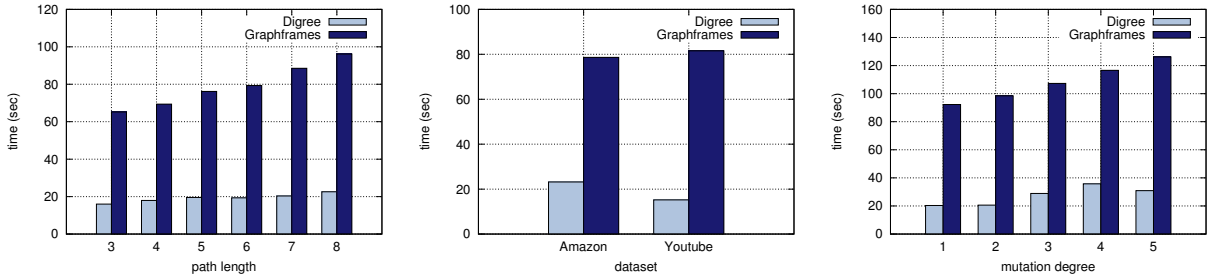
	#nodes	#edges	#labels
Amazon	548.552	1,788,725	11
Youtube	155,513	2,969,826	14

5. EXPERIMENTAL EVALUATION

We deployed our system on a cluster consisting of 18 Linux virtual machines (VMs). Each VM had 4 cores and 8 GB of memory. We used one VM to run the master process, one VM to host a PostgreSQL database server acting as the processing engine and the rest 16 VMs to host the partitions of the graph database (one Neo4j database per node) and the slave processes. We compared *Digree* to the motif finding feature of Graphframes [6], a package for Apache Spark which provides DataFrame-based Graphs. Graphframes was deployed on the same cluster using the default Spark setup. We used two real world datasets for our experiments which are the Amazon product network[1] [7] and the Youtube video graph.[2] The details of the datasets are shown in Table 1. For partitioning the datasets we used the popular METIS [8] algorithm. All measurements are made after the datasets have been loaded and partitioned on the respective systems. In [1] we present additional experiments running *Digree* on a much larger twitter dataset consisting of over 35 million nodes and 900 million edges, having 232 different labels. Graphframes however, in the aforementioned cluster, could not handle that dataset so we do no present the respective results here.

In the first experiment of Figure 10(a), we evaluate how each system scales while answering simple path queries of increasing length. For each dataset and for path lengths from 3 to 8 we created 5 graph queries of random labels. In the figure we report the average execution time of these queries. We also used seven graph pattern matching queries from [5]. For each pattern query we created 5 randomly la-

[1]https://snap.stanford.edu/data/
[2]http://netsg.cs.sfu.ca/youtubedata/



(a) path length (b) datasets (c) mutation degree

Figure 10: Average execution time

beled instances and ran all of them on the two systems. We present average execution times for each dataset in Figure 10(b). In all cases, *Digree* is significantly faster.

We then took the pattern queries from the last experiment and created 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. As shown in Figure 10(c), *Digree* outperforms Graphframes which presents a steady linear increase in execution time with respect to the number of mutations.

6. RELATED WORK

A number of systems were developed for distributed graph processing such as Google Pregel [9], Apache Giraph [10] and GraphX [11]. However, none of these systems is specialized in graph pattern matching. In [5] the authors explore relational optimizations for graph pattern matching. The work of [2] also suggests bulding a distributed graph database out of local Neo4j installations. The authors propose a novel lightweigt dynamic repartitioner that increases data locality while maintaining load balance. This work is complementary to ours as it focuses on maintaining a good partitioning scheme in evolving datasets, while *Digree* focuses on parallel processing of complex query patterns over the resulting partitions.

7. CONCLUSION

In this paper we presented *Digree*, a distributed graph processing engine that exploits the efficient graph processing primitives provided by local graph databases, while at the same time benefits from the increased parallelism offered by the proposed query decomposition process. We have compared *Digree* against a popular big data system and shown that it consistently provides better performance.

8. REFERENCES

[1] V. Spyropoulos, C. Vasilakopoulou, and Y. Kotidis, "Digree: A Middleware for a Graph Databases Polystore," in *Proceedings of IEEE BigData*, 2016.

[2] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases," in *Proceedings of EDBT*, 2015.

[3] I. Filippidou and Y. Kotidis, "Online and On-demand Partitioning of Streaming Graphs," in *Proc. of the IEEE Big Data*, 2015.

[4] B. Gallagher, "Matching Structure and Semantics: A survey on Graph-based Pattern Matching," *AAAI FS*, vol. 6, 2006.

[5] J. Huang, K. Venkatraman, and D. J. Abadi, "Query Optimization of Distributed Pattern Matching," in *Proceedings of ICDE*, 2014.

[6] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "GraphFrames: An Integrated API for Mixing Graph and Relational Queries," in *Proceedings of GRADES*, 2016.

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

[8] G. Karypis and V. Kumar, "Analysis of multilevel graph partitioning," in *Proceedings of IEEE Supercomputing Conference*, 1995.

[9] 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 *Proceedings of ACM SIGMOD*, 2010.

[10] M. Han and K. Daudjee, "Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems," *Proc. VLDB Endow.*, vol. 8, May 2015.

[11] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A Resilient Distributed Graph System on Spark," in *GRADES*, 2013.