PatMat: A Distributed Pattern Matching Engine with Cypher

Kongzhang Hao
UNSW, Sydney
khao@cse.unsw.edu.au

Zhengyi Yang
UNSW, Sydney
zyang@cse.unsw.edu.au

Longbin Lai
UNSW, Sydney
llai@cse.unsw.edu.au

Zhengmin Lai
ECNU, Shanghai, China
zmlai@stu.ecnu.edu.cn

Xin Jin
ECNU, Shanghai, China
xinjin@stu.ecnu.edu.cn

Xuemin Lin
UNSW, Sydney
lxue@cse.unsw.edu.au

ABSTRACT

Graph pattern matching is one of the most fundamental problems in graph database and is associated with a wide spectrum of applications. Due to its computational intensiveness, researchers have primarily devoted their efforts to improving the performance of the algorithm while constraining the graphs to have singular labels on vertices (edges) or no label. Whereas in practice graphs are typically associated with rich properties, thus the main focus in the industry is instead on powerful query languages that can express a sufficient number of pattern matching scenarios. We demo PatMat in this work to glue together the academic efforts on performance and the industrial efforts on expressiveness. To do so, we leverage the state-of-the-art join-based algorithms in the distributed contexts and Cypher query language - the most widely-adopted declarative language for graph pattern matching. The experiments demonstrate how we are capable of turning complex Cypher semantics into a distributed solution with high performance.

KEYWORDS

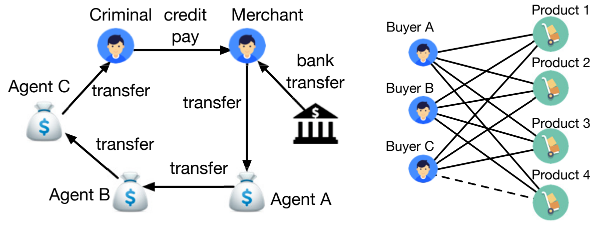
Graph pattern matching; Cypher; distributed processing; join optimization; graph database

ACM Reference Format:
Kongzhang Hao, Zhengyi Yang, Longbin Lai, Zhengmin Lai, Xin Jin, and Xuemin Lin. 2019. PatMat: A Distributed Pattern Matching Engine with Cypher. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM '19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3357384.3357840

1 INTRODUCTION

We study graph pattern matching in this work as a problem to find all embeddings (matches) of a small pattern graph in a very large graph database (data graph). Subgraph matching is one of the most fundamental problems in graph database and is associated with a wide spectrum of applications in the areas of finance, e-commerce, cyber security, bioinformatics, chemistry, social science, etc. Below we present two real-life scenarios that rely on graph pattern matching.

Scenario I. Figure 1a demonstrates a credit-card fraud case in a third-party payment network [16], in which the accounts are vertices and the transactions in between are edges. In this case, "Criminal" attempts to cash out short-term credits from the bank. To do so, the "Criminal" conspires with a "Merchant" (online shopping) to fake a buying using the credits. Once the "Merchant" receives money from the bank, he/she transfers this money back to the "Criminal" via three intermediate agents. We can detect such fraud by querying a 5-cycle pattern graph in the payment network.



(a) Credit-card fraud **(b) Recommendation**

Figure 1: The real-life pattern matching scenarios.

Scenario II. An online shopping site wants to recommend products to the buyers using the strategy presented in Figure 1b. In this case, three buyers have all purchased "Product 1-3" recently, while only "Buyer A" and "Buyer B" have bought "Product 4". It is reasonable to infer that "Buyer C" may also be interested in "Product 4", and recommendation is made accordingly [11]. We can identify all such links for recommendation by querying a "3-4" bi-clique (3 buyers connects to all 4 products at the same time) with one missing edge as a pattern graph in the buyer-product network.

Motivations. Despite its usefulness, graph pattern matching typically involves a computation-intensive operation, known as *subgraph isomorphism* [18]. In addition, data graph nowadays is growing beyond the capacity of one single machine. Therefore people are seeking efficient and scalable algorithms in the distributed context. Most such efforts model graph pattern matching as joins of edge tables, and the problem is accordingly turned into solving the optimal join plan that minimize the communication cost [2, 3, 9, 15, 19]. The state-of-the-art algorithms are capable of handling graph pattern matching with trillions of results in small-scale (~10 machines) cluster, and they show promising trends of scalability. However, these works were proposed without considering node and edge labels in order to better benchmark the performance. Although there are works that handle graph with labels [4, 6], they were typically developed in the sequential setting and were only applied to graphs whose vertices (edges) only contained singular labels.

Moreover, people typically adopt property graph [17] model in practice, where each vertex and edge takes arbitrary number of properties (attributes) of various data types. For example, in Figure 1a, each vertex and edge must have a "label" property. Vertices

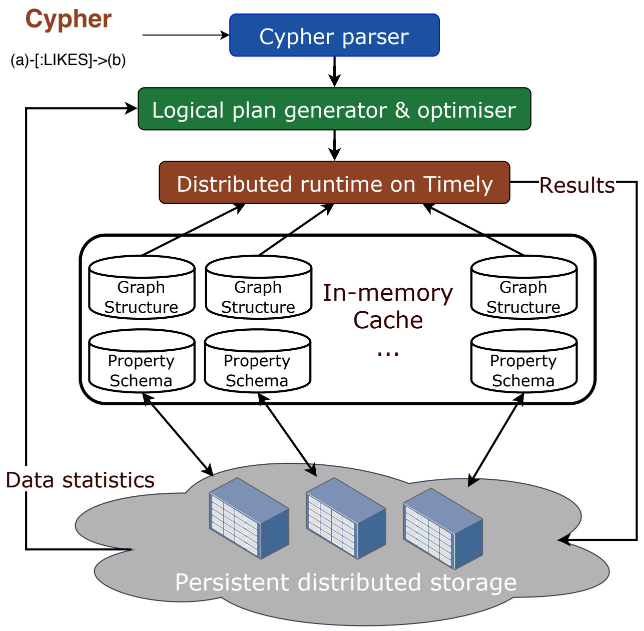


Figure 2: The PatMat's system architecture.

with "person" label can contain personalized profile and edges with "transfer" label can have a "time" slot to mark when the transaction happens. We can use such rich semantics to reduce false positives. For instance, one can limit the money transferring in the pattern to happen within a certain time span (e.g. 10 hours), and among people whose age is between 18 and 40. Apparently, we cannot directly benefit from the efforts in academia to solve pattern matching problems with the property graph model.

Meanwhile, people from the industry pioneer the development of declarative languages for graph pattern matching to express the rich semantics accompanied with property graph model. Cypher [1] is designed by Neo4j to visually recognize the graph patterns in data, and it is simple to learn and master. Neo4j is now working to standardize the open-sourced Cypher language [2], making it the most widely adopted graph querying language. As a result, we use Cypher as the query language in this work. Although Cypher is powerful in expressiveness, there lacks a system that meanwhile offers efficient and scalable backend and expressive Cypher frontend for graph pattern matching. Neo4j natively supports Cypher, but its free version runs in sequential context and cannot scale to large-scale graphs. Graphflow [8] extends Cypher to support dynamic query, but it is still developed as a single-machine solution with limited scalability. Morpheus [3] and Gradoop [7] are extensions that support Cypher query in Spark [20] and Flink [5] respectively. But their performance is not satisfactory due to the innate cost of the base systems.

These motivate us to propose PatMat to glue together the academic efforts on performance and the industrial efforts on expressiveness for graph pattern matching. We compare PatMat with Neo4j in the sequential context, and Morpheus and Gradoop in the distributed context to show the efficiency of PatMat.

[1]https://neo4j.com/developer/cypher-query-language/
[2]http://www.opencypher.org/
[3]https://github.com/opencypher/morpheus#morpheus-cypher-for-apache-spark

2 SYSTEM OVERVIEW

Figure 2 demonstrates the architecture of PatMat, where the arrows indicate the direction of data dependency. On the upper layer, we allow user to specify query using Cypher language, which is parsed into PatMat's internal representation consisting of *structure information* and *property constraints* using Cypher parser (Section 2.1). Logical plan generator (Section 2.2) is responsible to generate optimal plan based on the state-of-the-art algorithms, more specifically BinaryJoin [9] and WOptJoin [3]. A cost estimator is deployed to choose whether to use BinaryJoin or WOptJoin, or a hybrid strategy [1] (as one of the future works). Then the execution plan will be scheduled on the layer of distributed runtime, which is currently based on the Timely dataflow system [13], a general-purpose data-parallel distributed engine. On the bottom layer, the graph data (structure and properties) is persisted in a distributed storage (e.g. HDFS, TiKV [4]). Note that the storage layer also interacts with logical plan generator by offering data statistics for more accurate cost estimation, and maintains the results from the "distributed runtime" for further use. Between distributed runtime and storage layers, we place an in-memory cache in each machine of the cluster for quick access of graph structure and property schema (Section 2.3).

2.1 Cypher Parser

We implement Cypher parser to transform user-specified Cypher query into our internal representation of pattern graph structure and property constraints. Cypher parser firstly uses `libcypher-parser` [5] to convert the Cypher query into a syntax tree, which will further be parsed into *structure information* and *property constraints*. The structure is simply maintained as lists of query vertices and edges, where query vertices are identified with continuous numbering, and edges are labelled with the source and target vertices' ids.

Property constraints are extracted and stored as an *expression tree*, which is a binary tree with four types of tree node, namely `Constant`, `Variable`, `Predicate` and `Arithmetic`. `Constant` and `Variable` must be tree leaves. `Predicate` and `Arithmetic` are used to define operations between the left and right child in the tree, where `Predicate` defines logical operations such as $>, <, \leq, \geq$, $\wedge, \vee$, etc., and `Arithmetic` defines arithmetic operations such as $+, -, *, /$ etc. To filter a vertex using expression tree, we recursively get the value of the tree nodes. In detail, `Constant` directly returns the constant value, while `Variable` refers to the value stored in the property schema which should be retrieved from cache or storage. `Predicate` and `Arithmetic` will return the results computed from their left and right child. Note that `Predicate` will always be the tree root that returns `Boolean` value for the filtering. Below gives an expression tree for the property constraint `v.age + 10 > 30`:

$$\begin{tikzcd}
 & \text{Constant: 30} & \\
\text{Predicate: >} \arrow[ru, dash] \arrow[rd, dash] & & \text{Constant: 10} \\
 & \text{Arithmetic: +} \arrow[ru, dash] \arrow[rd, dash] & \\
 & & \text{Variable: age}
\end{tikzcd}$$

2.2 Logical Execution Plan

It is natural to express the graph pattern matching with joins. For example, a triangle query can be written as $R(v_1, v_2, v_3) = E(v_1, v_2) \bowtie$

[4]https://github.com/tikv/tikv
[5]https://github.com/cleishm/libcypher-parser

$E(v_2, v_3) \bowtie E(v_1, v_3)$, where $E(*)$ stands for the edges of the data graph. In the distributed context, researchers are seeking optimal join plans to minimize communication cost during the execution. Among the efforts two algorithms (strategies) stand out, namely "Binary-join-based subgraph-growing algorithm" (BinaryJoin) and "Worst-case optimal vertex-growing algorithm" (WOptJoin).

BinaryJoin decomposes the pattern graph into a set of *join units* that are either star (a tree of depth 1) or clique (a complete graph). These join units can be independently computed in each partition of the graph, which are further joined together following a pre-computed order to produce the final results. WOptJoin is based on the worst-case optimal join algorithm [14] with a vertex-growing fashion. In detail, WOptJoin determines the matching order of the query vertices via a greedy heuristic that starts with vertex of the largest degree, and consequently selects next vertex which has most connections (id as tie breaker) with already-selected vertices. Following the matching order as $\{v_1, v_2, \ldots, v_n\}$, the algorithm first computes the results of $\{v_1, v_2\}$ (edges) that can end up in the final results, then grows to $\{v_1, v_2, v_3\}$, and so on until the results are constructed.

In recent experiments [10], we have found out that there are some pattern graphs that favour BinaryJoin algorithm, while some others favour WOptJoin. As a result, we propose to compute both join plans, then use the cost estimation proposed in [9] to evaluate both plans, and finally select the one with less cost (break tie with WOptJoin). In [1], the authors showed a hybrid solution that can leverage the advantages of both BinaryJoin and WOptJoin. While it requires further study in the distributed context, we leave it as an interesting future work.

2.3 Graph Storage and Cache

For now we partition the graph regarding the vertices, where each vertex is placed together with all its neighbours (both incoming and outgoing for directed graphs) and the vertex properties. An edge is identified by the source vertex and target vertex, with its properties being placed in the source vertex's machine. With the simple partition strategy, graph structure of each partition is stored using compressed sparse row[6] in each machine's in-memory cache. The cache also contains frequently-used properties of vertices and edges up to a memory threshold. The property cache will be replaced using an LRU (least recently used) strategy. As "label" property presents in most queries, we directly maintain the vertex and edge "label" in-memory with the graph structure, where each label is automatically converted to an integer. This results in faster filtering on "label". As one important future work, we will investigate graph partition by injecting a "Placement driver" between the Cache layer and Persistent storage layer. A better cache replacement strategy will also be studied and applied in the future.

3 PERFORMANCE STUDIES

We use the LDBC social network benchmarking library (SNB) [7] to study the performance of PatMat. SNB provides a data generator that generates a synthetic social network of required statistics, as well as a document that describes benchmarking tasks consisting of pattern matching queries. We use four of them as shown in Figure 3, where the filtering constraints are listed beneath the query, with

[6]https://en.wikipedia.org/wiki/Sparse_matrix
[7]http://ldbcouncil.org/benchmarks

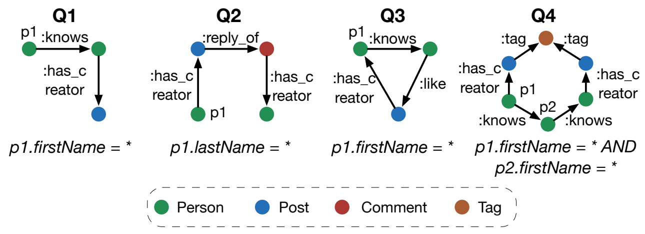


Figure 3: The LDBC queries.

	Q_1/s	Q_2/s	Q_3/s	Q_4/s
Neo4j	87	594	236	182
PatMat	12	24	17	256

Table 1: PatMat **vs. Neo4j: single server**

	Q_1/s	Q_2/s	Q_3/s	Q_4/s
Gradoop	OOM	OT	OOM	OOM
Morpheus	OT	OT	OT	OT
PatMat	2.6	9.4	5.3	77.3

Table 2: PatMat **vs. Gradoop and Morpheus: 10 machines**

"*" indicating some names randomly picked from the database. We adapt the queries in two ways: (1) using one-hop edge for multi-hop edges; (2) removing aggregators. We will support these query variants in the future version. We generate the benchmarking data graph using the "Facebook" mode with a duration of 3 years, and a scale factor of 60. The graph has 187.11 million vertices and 1246.66 million edges respectively, occupying over 65GB disk space in text file, and 170GB in Neo4j's database (including indices).

We deploy benchmarking on two systems: (1) a local cluster of 10 machines connected via one 10GBps switch, in which each machine has 64GB memory, 1 TB disk and 1 Intel Xeon CPU E3-1220 V6 3.00GHz with 4 physical cores; (2) a server of 2 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz (each has 20 cores 40 threads), 500GB memory and 2 TB disk. We allow 2 hours as the maximum running time for each test, with OT and OOM indicating a test case running out of the time limit and out of memory respectively.

Single Machine. We compare PatMat with Neo4j [8] on the server. As Neo4j can only run each query using one CPU core, we configure one worker for PatMat for a fair comparison. Table 1 shows the results. We make sure that both PatMat and Neo4j are caching the whole graph in memory. PatMat runs much faster than Neo4j in all queries except Q_4, for which PatMat is only around $1.4x$ times slower. PatMat's runtime efficiency is partly due to its WOptJoin ("vertex-growing") and BinaryJoin strategies [10], compared to Neo4j's "edge-growing" strategy which grows one edge each round until finding the results. The "edge-growing" strategy is proven to be sub-optimal [3] and may produce a large number of intermediate results. In comparison, WOptJoin ("vertex-growing") and BinaryJoin strategies [10] are guaranteed with worst-case optimality and they render much fewer intermediate results. Moreover, PatMat's runtime efficiency is relevant to two other factors: (1) PatMat is a prototype system which supports fewer features, hence is inherently more efficient. (2) While Neo4j is built on Java, PatMat is implemented by Rust which is a faster language. Note that we can easily configure PatMat to run in parallel on the server to scale out.

Distributed Context. In this experiment, we compare PatMat with Gradoop and Morpheus in a local cluster of 10 machines, each running 3 workers. We record the running time (of the slowest worker) of each system in Table 2. Gradoop and Morpheus failed all test cases due to either `OOM` or `OT`. This is most likely due to the "edge-growing" strategy adopted by both systems, which is sub-optimal [3] and may generate too many intermediate results during computation. Comparatively, PatMat uses WOptJoin and BinaryJoin strategies [10] which produce much fewer intermediate results. Furthermore, the base systems of Gradoop and Morpheus, namely Flink and Spark, have been shown to incur large system cost despite their scalability [12].

4 DEMONSTRATION SCENARIOS

The demonstration mainly presents: (1) the processing pipeline of PatMat; (2) comparison of PatMat with existing Cypher-compatible systems using a large dataset; (3) real-life applications. Throughout the demonstration, the attendee will be able to get familiar with the system architecture of PatMat as well as its performance advantage and practicality.

4.1 Processing Pipeline

In this scenario, we guide the attendee to experience the whole processing pipeline of PatMat[8], including:

- Cluster configuration: The attendee can configure the cluster to deploy PatMat;
- Graph construction: We allow the attendee to specify the sources of the graph data (local csv and remote data on S3) and load them to construct the PatMat graph DB. PatMat will persist the graph data in TiKV and initialize the cache at backend;
- Writing query using Cypher: The attendee will be provided with ER-diagram of the DB, and guided to write graph pattern query using Cypher. The query will be parsed and visualized so that attendee can review and modify;
- Generating execution plan: Once the query is specified, we will guide the attendee to configure options to generate an optimal execution plan to run in the configured cluster;
- Result and performance metrics demonstration: After execution, the attendee will see the results according to what he/she specified in the Cypher query. In addition, the performance metrics such as running time, communication cost, size of intermediate results and memory usage etc. will be displayed.

4.2 Comparing with Existing Systems

In this scenario, we will pre-load a large dataset in AWS cluster and allow attendee to specify one of the benchmarked queries. The query will be executed in the AWS cluster using Gradoop, Morpheus and PatMat respectively. The performance metrics will be delivered back to the scene and demonstrated to the attendee. We use this scenario to show PatMat's advantage over existing Cypher-compatible solutions in the distributed context.

[8]Demo video available at https://www.youtube.com/watch?v=df5bvs0AHHU

4.3 Real-life Application

We will show how PatMat can be used to recommend authors from DBLP network to attendees as potential collaborators. We construct the DBLP co-authorship network covering authors who have published papers in the past 5 years in top-tier DB/DM conferences including SIGMOD, VLDB, ICDE, KDD, ICDM and CIKM. The edges in network will record frequencies of co-authorship. The query is a pattern graph of 4-clique missing one edge from the attendee to the potential collaborator. The query further requires that each existing co-authorship (edge) has a frequency of at least 2.

5 ACKNOWLEDGEMENTS

Xuemin Lin is supported by 2019DH0ZX01, 2018-YFB1003504, NSFC 61232006, ARC DP180103096 and DP170101628.

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing *(SIGMOD '16)*. 431–446.
[2] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. 2013. Enumerating subgraph instances using map-reduce. In *Proc. of ICDE'13*.
[3] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-memory Dataflows. *Proc. VLDB Endow.* 11, 6 (2018), 691–704.
[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products *(SIGMOD '16)*. 1199–1214.
[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.
[6] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proc. of SIGMOD'13*.
[7] Martin Junghanns, Max Kiessling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. 2018. Declarative and Distributed Graph Analytics with GRADOOP. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 2006–2009.
[8] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database *(SIGMOD '17)*. 1695–1698.
[9] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (2016), 217–228.
[10] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (2019), 1099–1112.
[11] Sune Lehmann, Martin Schwartz, and Lars Kai Hansen. 2008. Biclique communities. *Phys. Rev. E* 78 (2008), 016108. Issue 1.
[12] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at What Cost? *(HOTOS'15)*.
[13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System *(SOSP '13)*. 439–455.
[14] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case Optimal Join Algorithms. *J. ACM* 65, 3 (2018).
[15] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: On Compression and Computation. *Proc. VLDB Endow.* 11, 2 (2017), 176–188.
[16] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.
[17] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph Databases: New Opportunities for Connected Data* (2nd ed.). O'Reilly Media, Inc.
[18] R. Shamir and D. Tsur. 1997. Faster subtree isomorphism. In *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*. 126–131.
[19] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel Subgraph Listing in a Large-scale Graph. In *SIGMOD'14*. ACM, 625–636.
[20] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 2–2.