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

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

(b) Recommendation

Figure 1: The real-life pattern matching scenarios.
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 users to specify queries 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 for generating optimal plans 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 first 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 $\lor, \land, \neg$, etc., and Arithmetic defines arithmetic operations such as $+, -, \ast, \div$, 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:

Constant: 30
Predicate: >
Constant: 10
Arithmetic: +
Variable: age

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 $R(v_1, v_2, v_3) = E(v_1, v_2) \land E(v_2, v_3)$...


\[ E(v_2, v_3) \sim E(v_1, v_2) \], where \( E(\cdot) \) 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 joinunits 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 (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.](image)

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

<table>
<thead>
<tr>
<th></th>
<th>Q1/s</th>
<th>Q2/s</th>
<th>Q3/s</th>
<th>Q4/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neo4j</td>
<td>87</td>
<td>594</td>
<td>236</td>
<td>182</td>
</tr>
<tr>
<td>PatMat</td>
<td>12</td>
<td>24</td>
<td>17</td>
<td>256</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>Q1/s</th>
<th>Q2/s</th>
<th>Q3/s</th>
<th>Q4/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gradoop</td>
<td>0OM</td>
<td>OT</td>
<td>0OM</td>
<td>0OM</td>
</tr>
<tr>
<td>Morpheus</td>
<td>OT</td>
<td>OT</td>
<td>OT</td>
<td>OT</td>
</tr>
<tr>
<td>PatMat</td>
<td>2.6</td>
<td>9.4</td>
<td>5.3</td>
<td>77.3</td>
</tr>
</tbody>
</table>

---

**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 Q4, 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.

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
Xuemlin Lin is supported by 2019DH0ZX01, 2018-YFB1003504, NSFC 61225206, ARC DP180103096 and DP170101628.

REFERENCES

\(^8\)Demo video available at https://www.youtube.com/watch?v=df5bvs0AHHU