VisualNeo: Bridging the Gap between Visual Query Interfaces and Graph Query Engines

Kai Huang
Hong Kong Univ. of Sci. & Tech.
ustkhuang@ust.hk

Houdong Liang[§]
Hong Kong Univ. of Sci. & Tech.
hliangam@connect.ust.hk

Chongchong Yao[§]
Hong Kong Univ. of Sci. & Tech.
cyaoad@connect.ust.hk

Xi Zhao
Hong Kong Univ. of Sci. & Tech.
xzhaoca@connect.ust.hk

Yue Cui
Hong Kong Univ. of Sci. & Tech.
ycuias@cse.ust.hk

Yao Tian
Hong Kong Univ. of Sci. & Tech.
ytianbc@cse.ust.hk

Ruiyuan Zhang
Hong Kong Univ. of Sci. & Tech.
zry@ust.hk

Xiaofang Zhou
Hong Kong Univ. of Sci. & Tech.
zxf@ust.hk

ABSTRACT

Visual Graph Query Interfaces (VQIs) empower non-programmers to query graph data by constructing visual queries intuitively. Devising efficient technologies in Graph Query Engines (GQEs) for interactive search and exploration has also been studied for years. However, these two vibrant scientific fields are traditionally independent of each other, causing a vast barrier for users who wish to explore the full-stack operations of graph querying. In this demonstration, we propose a novel VQI system built upon Neo4j called VisualNeo that facilities an efficient subgraph query in large graph databases. VisualNeo inherits several advanced features from recent advanced VQIs, which include the data-driven GUI design and canned pattern generation. Additionally, it embodies a database manager module in order that users can connect to generic Neo4j databases. It performs query processing through the Neo4j driver and provides an aesthetic query result exploration.

PVLDB Reference Format:
Kai Huang, Houdong Liang, Chongchong Yao, Xi Zhao, Yue Cui, Yao Tian, Ruiyuan Zhang, Xiaofang Zhou. VisualNeo: Bridging the Gap between Visual Query Interfaces and Graph Query Engines. PVLDB, 16(12): 4010-4013, 2023.
doi:10.14778/3611540.3611608

1 INTRODUCTION

Visual graph query interfaces (VQI) enable non-experts to compose graph queries effortlessly without writing any textual query language, which broadens the usability of graph querying frameworks. Consequently, numerous academic and commercial frameworks for querying large graph databases adopt VQIs for composing subgraph queries. For example, *PubChem* [15] provides a VQI for researchers in the chemistry domain to perform chemical compound searches. The GUI of *PubChem* [15] includes a label panel containing a set of chemical elements and a pattern panel with carboxyl groups, a benzene ring, etc. However, the contents of the panel are typically chosen manually by domain experts.

To address this challenge, data-driven VQI design has attracted considerable attention in recent years [2, 5, 6, 11]. Given a graph database D, data-driven VQIs automatically populate panels (*e.g.,* label panel, pattern panel) of the GUI from D. While the label and property panels can be populated without many difficulties by traversing the underlying databases, selecting useful patterns is an NP-hard problem [4, 8, 9]. A few algorithms have been proposed to conduct the selection of these patterns [3, 4]. TATTOO framework [4] performs data-driven selection of canned patterns for large networks while TED framework [3] works for a large collection of small- or medium-size data graphs. Several data-driven VQI systems have been developed as well [2, 11].

Despite the amount of progress made in the development of data-driven VQIs, a potential direction is overlooked, which is the connection between VQI and Graph Query Engines (GQEs). A GQE acts as an abstraction layer between the user application and the database. It receives database operations in the form of query language and outputs specific data to the user application. Therefore, VQI complements GQE in the way that it provides an easy-to-use interface for non-professional users to construct their queries, which are subsequently sent to GQE for processing and result exploration. Traditional data-driven VQI design focus on efficient subgraph query processing [12] and data-driven selection of canned patterns [2, 11], but ignores the potential of GQEs in terms of abilities to process queries.

In this demonstration, we present a novel data-driven visual subgraph query system called VisualNeo. The system is built upon Neo4j [14], a popular graph query engine. It possesses a *Database Manager* module with which users can connect to local or remote Neo4j graph databases by providing authentication information. It supports generic Cypher query processing via Neo4j driver, which includes nodes, relationships, labels, and properties. Therefore, VisualNeo builds a bridge between VQI and GQE, introducing a new direction for these two scientific fields.

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611608

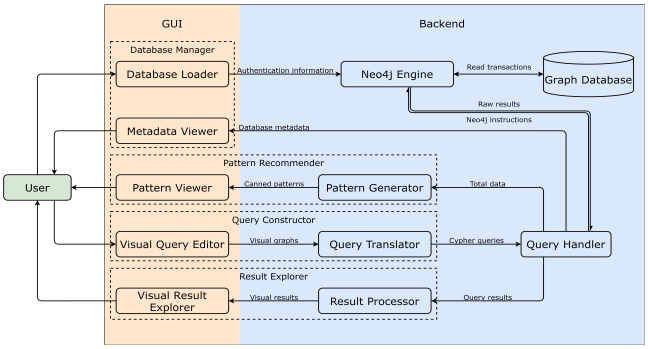

Figure 1: The architecture of VisualNeo.

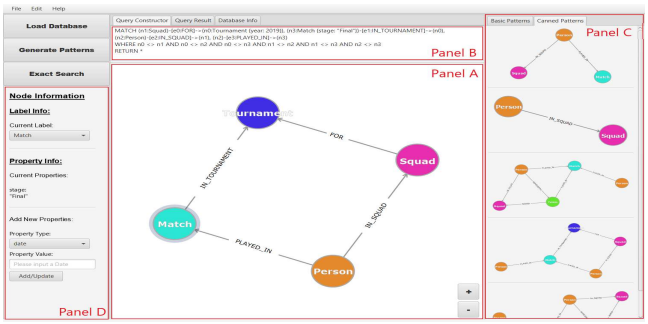

Figure 2: The *Query Constructor panel.*

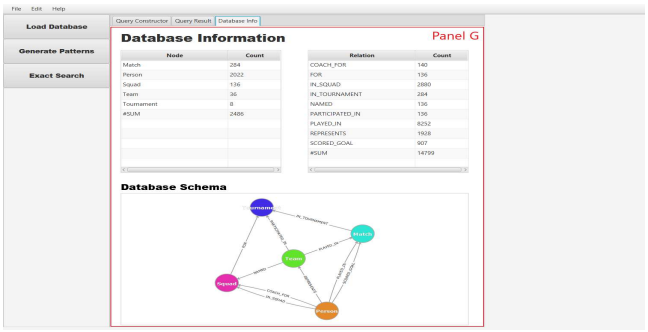

Figure 3: The *Database Info panel.*

VisualNeo also inherits several state-of-the-art features from recent data-driven VQI designs. First, it generates a set of diversified patterns called TED patterns (*i.e.,* Top-k Edge-Diversified patterns) [3], which summarize the characteristics of the underlying database and thus facilitate the query formulation. In particular, TED patterns have a theoretical guarantee of the edge coverage approximation ratio and its generation process requires limited memory. Second, it embodies an aesthetic query results explorer which adopts the Fruchterman-Reingold algorithm [7] to display the retrieved results.

VisualNeo embraces innovative features as well. To ensure user-friendliness, VisualNeo provides adequate support during the query formulation process. It displays metadata information of the underlying database and provides real-time query translation to guide users to perform exploratory searches where users are unsure about their initial goals or ways to achieve their goals.

2 SYSTEM ARCHITECTURE

Figure 1 shows the architecture of VisualNeo. It consists of five modules, *Database Manager*, *Pattern Recommender*, *Query Constructor*, *Query Handler*, and *Result Explorer*. The *Database Manager* module first establishes a connection to a graph database server with user authentications and displays its metadata obtained by the *Query Handler* module. The *Query Handler* module also exports the whole database such that the *Pattern Recommender* module can then utilize METIS [13] (for graph partitioning) and TED [3] (for pattern generation) to produce diversified and high-coverage patterns. With the aid of these two modules, the *Query Constructor* module enables the user to form visual query graphs effortlessly. To integrate with the GQE (*i.e.,* Neo4j), the query graphs are further translated into formal query languages and fed to the *Query Handler* module. Next, the *Query Handler* module instructs the GQE to execute read transactions and retrieve the desired results. Finally, the *Result Explorer* module converts the unprocessed results into navigable visual graphs for the user to investigate.

Database Manager module. This module establishes a connection to a graph database server and displays its metadata. When the user clicks the "Load Database" button shown in Figure 2 and specifies the authentication information, the *Database Manager* module sends a connection request to the server. VisualNeo is compatible with all standard Neo4j servers and only requires read authority.

After connection, the *Query Handler* module automatically executes a sequence of queries to retrieve database metadata, including node/relationship counts, node/relationship labels, node/relationship property keys & data types, and the schema graph that defines the topology among different classes of nodes and relationships. These queries either directly access the Neo4j count store or invoke database procedures. Therefore, such searches have $O(1)$ time complexity for each class/property. The metadata is then displayed in the Database Information panel (*Panel G*) shown in Figure 3 and is also used to facilitate query formulation by providing label and property constraints in the *Query Constructor* module.

Pattern Recommender module. This module generates TED patterns and populates the canned pattern panel when the user clicks the "Generate Pattern" button shown in Figure 2 and specifies the constraints of desired patterns.

*Definition 2.1 (**Top-k Edge-Diversified Patterns (TED patterns)**).* *Given a graph G and an integer k, the top-k edge-diversified patterns is the set of k connected subgraphs $\mathcal{P} = \{p_1, p_2, \ldots, p_i, \ldots p_k\}$ in G such that the total coverage of $\mathcal{P}$ over G (denoted by $|Cov(\mathcal{P}, G)|$),* i.e., $|\cup_i Cov(p_i, G)|$, *is maximized, where $Cov(p_i, G)$ is the cover set of p_i over G* (i.e., *the covered edges of p_i over G).*

The Pattern Recommender module consists of two components: METIS graph partitioning [13] and TED pattern generation [3]. METIS partitions the large network into a large collection of small- or medium-sized graphs (*e.g.,* tens of nodes per graph). Next, the TED algorithm greedily searches for a set of patterns with maximum edge coverage. Given a set of partitions $D = \{G_1, G_2, \ldots G_n\}$

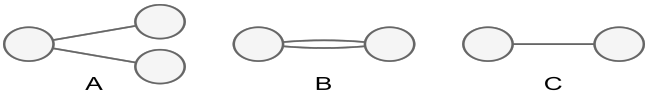

Figure 4: Indistinguishable patterns without node isomorphism (A and B) or relationship isomorphism (A and C).

and an integer k, the TED algorithm first enumerates all 1-sized subgraphs (*i.e.,* edges) and appends them into the set $\mathcal{P}$. Then, an iterative process is performed to enumerate τ-sized ($\tau \geq 2$) subgraphs by performing the right-most extension [16] and appends them into the set $\mathcal{P}$. When the size of $\mathcal{P}$ is k and a new subgraph g is generated, a swapping-based process is developed to determine if g should be swapped into $\mathcal{P}$. In particular, it first calculates the loss score (*i.e.,* decreased coverage if p is swapped out) for $p \in \mathcal{P}$ and then records the pattern p_t and its pattern score Score_L such that p_t has the minimum loss score. The benefit score (*i.e.,* increased coverage if g is swapped in) Score_B of g is also recorded. The subgraph g is considered as a promising candidate and swapped into $\mathcal{P}$ if $\text{Score}_B > (1+\alpha)\text{Score}_L + (1-\alpha)|Cov(\mathcal{P}, D)|/k$ is satisfied, where $\alpha \in [0, 1]$ is a *swapping threshold* for balancing loss score Score_L and the average coverage of patterns in $\mathcal{P}$. The approximation ratio (w.r.t., total coverage) of patterns $\mathcal{P}$ is bounded by $|Cov(\mathcal{P}, D)|/|Cov(\mathcal{P}_{opt}, D)| \geq \frac{1}{4}$ where $\mathcal{P}_{opt}$ is the optimal solution, which can be obtained by enumerating all subgraphs and generating all possible combinations of k subgraphs.

Query Constructor module. This module provides the user with a editor for visual query formulation and subsequently translates the visual graphs into Cypher queries. As shown in Figure 2, it comprises five components: the *Graph Editor* panel (*Panel A*), the *Query View* panel (*Panel B*), the *Pattern View* panel (*Panel C*), and the *Element Constraint* panel (*Panel D*).

- Elements can be selected/added/removed in *Panel A* using mouse-keyboard operations, with navigation and zoom adjustment possible through dragging and scrolling.
- *Panel B* provides a real-time translation of the visual graphs in *Panel A* to Cypher statements.
- *Panel C* displays TED patterns generated by the *Pattern Recommender* module and basic patterns applicable to universal databases, which can be added to *Panel A* through drag-over operations.
- Elements' labels and properties can be viewed and edited in *Panel D*.
- Upon completing the graph, clicking the "Exact Search" button in Figure 2 sends the translated Cypher query to the Query Handler module.

Following each step of building a visual query, the resulting query is automatically translated into a formal Cypher query by traversing all relevant relationships. For each relationship, the translator generates a corresponding line in the `MATCH` clause in the form of `(startNode)-[thisRelation]-(endNode)`. Nevertheless, this approach is susceptible to isomorphic issue, meaning that the same node/relationship may be returned more than once for each matching record. For instance, if a user creates a visual query in the form of `(n2)-[r1]-(n1)-[r2]-(n3)` (Pattern A in Figure 4), it may result in a mismatch with Patterns B or C if the translated query lacks

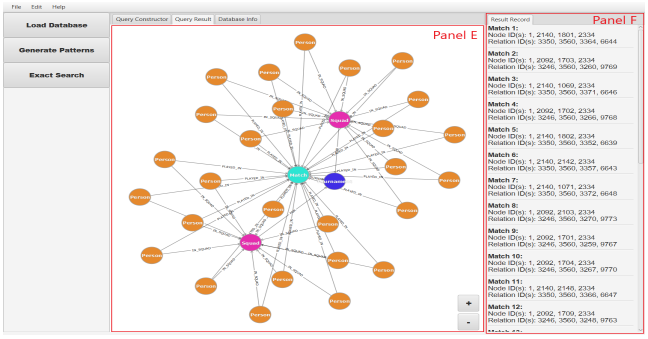

Figure 5: The *Query Result panel.*

appropriate isomorphic constraints. This highlights the need for additional measures to ensure accurate and reliable query translation. In practice, Neo4j Cypher utilizes relationship isomorphism for path matching, thereby preventing the same relationship from being returned more than once within a single result record. However, it does not assert node isomorphism. While this matching mechanism may assist programmers in managing complex queries, inexperienced users may be misled, as demonstrated by the fact that Pattern B in Figure 4 is considered a valid match for Pattern A. Most existing Cypher-oriented VQI (*e.g.,* Popoto.js [17]), do not handle node isomorphism because they do not connect to a GQE as a backend. To address this issue, we add additional inequality constraints on each node pair in the `WHERE` clause to ensure node isomorphism. To further enhance query efficiency, we eliminate trivial inequalities before feeding them into the constraints. For instance, if two nodes have distinct labels or properties, the corresponding inequality can be removed.

Query Handler module. This module's features include executing Cypher queries via the Neo4j driver, receiving query results, extracting desired data from returned records, converting raw data into Java-typed data, and boxing data into proper containers for further processing. All sessions and transactions are read-only to avoid modification and undesired authentication errors. Besides, consecutive transactions (e.g., metadata queries) are bundled into a single session to enhance efficiency.

Result Explorer Module This module is responsible for processing query results and presenting them in an aesthetically pleasing and navigable manner. It has two components: *Panel E* displays the result graph, while *Panel F* lists all matching records. With a click on an item in *Panel F*, *Panel E* immediately navigates to the corresponding pattern and highlights it.

In cases where the query involves nodes with high centrality or relationships with high betweenness, it is likely that the result contain duplicate elements. This occurrence of redundant information can cause high data transfer traffic and memory cost at the local device. To avoid such inefficiencies, we generate an ID reference list with Cypher for each query and only keep the information of distinct elements. This approach ensures that the final result is devoid of any information loss, and simultaneously prevent unnecessary data transfer or memory usage at the local device.

To arrange the results in an orderly manner, we employ a modified version of the Fruchterman-Reingold (FR) force-directed graph

layout algorithm [7]. First, we prune self-loops and retain only one relationship between each unordered pair of nodes to avoid unnecessary computations and excessive proximity between heavily connected nodes. Furthermore, we specify a constant optimal distance between connected node pairs and remove boundaries around the graph to enhance visual effects and reduce computation cost. Additionally, we establish a maximum distance on repulsive forces to prevent disjoint subgraphs from repelling each other to infinity. Finally, we center the centroid of all nodes at the origin after the force simulation to cancel the universal offset. The additional $O(|V|)$ computation cost for computing the centroid once for each graph is negligible compared to the overall $O(K(|V|^2 + |E|))$ time complexity, where $G = (V, E)$ is the graph and K is the number of simulation iterations.

3 RELATED SYSTEMS AND NOVELTY

Graph Query Engines (GQEs) such as Neo4j [14] assume that a user has programming and debugging expertise to formulate queries correctly with query languages (*e.g.,* Cypher). This assumption makes it harder for non-programmers to take advantage of a graph querying framework. Although existing Visual Query Interfaces (VQIs) such as PLAYPEN [2] and VINCENT [11] can alleviate this problem by enabling users to visually formulate queries, they ignore the potential of GQEs in terms of abilities to process queries. In contrast, VisualNeo has the following advantages. First, VisualNeo supports not only visual query formulation but also efficient graph query processing by leveraging the strength of GQEs. That is, it bridges the gap between VQIs and GQEs. Second, the patterns generated for visual query formulation can achieve a guaranteed approximation ratio of edge coverage and the generation process requires limited memory. Third, in contrast to VINCENT's hierarchical layout, VisualNeo utilizes the Fruchterman-Reingold algorithm [7] to display a force-directed graph drawing, which better suits general databases whose structure is more uncertain. Lastly, VisualNeo supports queries in an attributed graph and query translation, while both PLAYPEN and VINCENT only support queries in a simple graph.

Existing VQI libraries for Neo4j such as Popotojs [17] and tools such as Graphileon [18] only support iterative constructions of edges one-at-a-time (*i.e.,* edge-at-a-time mode), while VisualNeo enables a user to construct multiple nodes and edges in a subgraph query by performing a single click-and-drag action (*i.e.,* pattern-at-a-time mode) and thus facilitate efficient query formulations.

4 DEMONSTRATION OVERVIEW

VisualNeo is implemented in Java JDK 17 and JavaFX 19. In the demonstration, it will be loaded with a few real-world databases (*e.g.,* Women's World Cup 2019) from Neo4j Sandbox [1]. Example query graphs that can be constructed using patterns will be presented for formulation. Users can write their own ad hoc queries through our visual query editor as well. The key objective of the demonstration is to lead users through the full-stack operations of graph querying with the aid of data-driven VQIs and GQEs. In particular, it enables the audience to experience the following:

Scenario 1: Database loading and metadata information display. The "Load Database" button shown in Figure 2 enables the audience to connect to local or remote databases. After the database is loaded, the metadata information can be viewed in Figure 3. Users can obtain a macroscopic understanding of the underlying database by looking into the displayed node/relationship label table and schema graph.

Scenario 2: Data-driven visual query formulation. Through the "Generate Pattern" button shown in Figure 2, users can set the hyperparameters for TED frameworks and launch the generation of the TED patterns. The TED patterns will then be displayed in *Panel C*. In the process of query formulation, users can create nodes/relationships directly or drag-and-drop basic patterns or TED patterns from *Panel C*.

Scenario 3: Real-time query translation. VisualNeo supports real-time query translation for users' reference. In the process of visual query formulation, users can get familiar with formal query languages by observing the translated Cypher query (see *Panel B*) of the graph query (see *Panel A*). Consequently, users are able to construct their desired queries effortlessly.

Scenario 4: Aesthetic query result exploration. After users click the "Exact Search" button shown in Figure 2, VisualNeo will start the query processing, display the result graphs in a force-directed way in *Panel E*, and enable users to iterate through the query results using matching records in *Panel F*.

A demonstration video is publicly available at https://youtu.be/th0LqEK-S3s.

ACKNOWLEDGMENTS

The work was conducted in the JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust.

REFERENCES

[1] Neo4j Graph Data Platform. Neo4j sandbox.
[2] Yuan Z, Chua H E, Bhowmick S S, et al. PLAYPEN: Plug-and-Play Visual Graph Query Interfaces for Top-down and Bottom-Up Search on Large Networks. *In SIGMOD*, 2022.
[3] Huang K, Hu H, Ye Q, Tian K, Zheng B, Zhou X. TED: Towards Discovering Top-k Edge-Diversified Patterns in a Graph Database. *In SIGMOD*, 2023.
[4] Yuan Z, Chua H E, Bhowmick S S, et al. Towards plug-and-play visual graph query interfaces: data-driven selection of canned patterns for large networks. *In PVLDB*, 2021.
[5] Bhowmick S S, Huang K, Chua H E, et al. AURORA: data-driven construction of visual graph query interfaces for graph databases. *In SIGMOD*, 2020.
[6] Huang K, Bhowmick S S, Zhou S, et al. Picasso: exploratory search of connected subgraph substructures in graph databases. *In PVLDB*, 2017.
[7] Fruchterman, Thomas MJ and Reingold, Edward M. Graph drawing by force-directed placement. *Software: Practice and experience*, 1991.
[8] Huang K, Chua H E, Bhowmick S S, et al. CATAPULT: data-driven selection of canned patterns for efficient visual graph query formulation. *In SIGMOD*, 2019.
[9] Huang K, Chua H E, et al. MIDAS: towards efficient and effective maintenance of canned patterns in visual graph query interfaces. *In SIGMOD*, 2021.
[10] Bonifati, Angela and Martens, Wim and Timm, Thomas. An Analytical Study of Large SPARQL Query Logs. *In PVLDB*, 2017.
[11] Huang K, Ye Q, Zhao J, et al. VINCENT: towards efficient exploratory subgraph search in graph databases. *In PVLDB*, 2022.
[12] Jin, Changjiu and Bhowmick, Sourav S and Choi, Byron and Zhou, Shuigeng. Prague: towards blending practical visual subgraph query formulation and query processing. *In ICDE*, 2012.
[13] Karypis, George and Kumar, Vipin. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.
[14] Neo4j Graph Data Platform. Neo4j. Available at: https://neo4j.com/.
[15] National Center for Biotechnology Information. PubChem Sketcher. Available at: https://pubchem.ncbi.nlm.nih.gov/edit3/index.html.
[16] Xifeng Yan, and Jiawei Han. gspan: Graph-based substructure pattern mining. *In ICDM*, 2002.
[17] NHOGS Interactive. Popoto.js. Available at: https://popotojs.com.
[18] Graphileon. Available at: https://graphileon.com/.