Demonstrating Spindra: A Geographic Knowledge Graph Management System

Yuhan Sun
Arizona State University
Tempe, USA
ysun138@asu.edu

Jia Yu
Arizona State University
Tempe, USA
jiayu2@asu.edu

Mohamed Sarwat
Arizona State University
Tempe, USA
msarwat@asu.edu

***Abstract*—Knowledge Graphs are widely used to store facts about real-world entities and events. With the ubiquity of spatial data, vertexes or edges in knowledge graphs can possess spatial location attributes side by side with other non-spatial attributes. For instance, as of June 2018 the Wikidata knowledge graph contains $48,547,142$ data items (i.e., vertexes) to date and ≈13% of them have spatial location attributes. The co-existence of graph and spatial data in the same geographic knowledge graph allows users to search the graph with local intent. Many location-based services such as UberEats, GrubHub, and Yelp already employ similar knowledge graphs to enhance the location search experience for their end-users. In this paper, we demonstrate a system, namely Spindra, that provides efficient management of geographic knowledge graphs. We demonstrate the system using an interactive map-based web interface that allows users to issue location-aware search queries over the WikiData knowledge graph. The Front-end will then visualize the returned geographic knowledge to the user using OpenStreetMap.**

I. Introduction

Knowledge Graphs are widely used to store facts about real-world entities and events. With the ubiquity of spatial data, vertexes or edges in knowledge graphs can possess spatial location attributes side by side with other non-spatial attributes. For instance, as of June 2018 the Wikidata knowledge graph contains $48,547,142$ data items (i.e., vertexes) to date and ≈13% of them have spatial location attributes [7]. Figure 1 depicts a knowledge graph of restaurants, and the type of dishes they have on their menus. Many location-based services such as UberEats, GrubHub, and Yelp already employ similar knowledge graphs to enhance the location search experience for their end-users. For example, in Figure 1, the co-existence of graph and spatial data in the same geographic knowledge graph allows users to search the graph with local intent. An example query is:

`'Q1: Search the geographic knowledge graph for Asian restaurants that are located within the geographical region of Downtown Macau'`

A graph database management system (GDBMS) such as Neo4j [3] and Titan [6] can be leveraged to efficiently manage and access a geographic knowledge graph. For instance, a user can issue graph queries in Neo4j using the Cypher or Gremlin query language. The system will then optimize and execute such a query on the knowledge graph. To execute the query, the graph database system may apply one of two main strategies, GraphTraverse and SpaIndex.

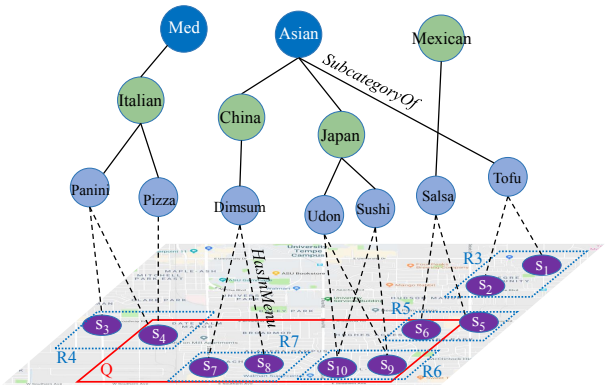
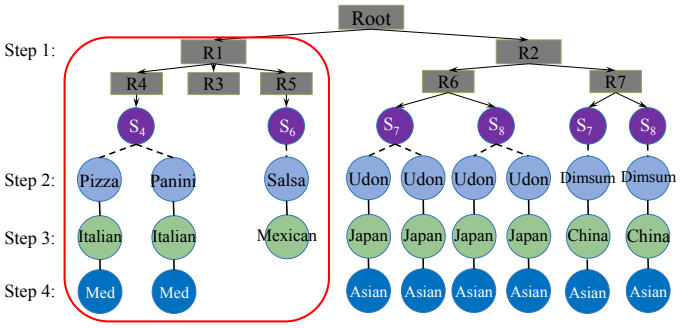


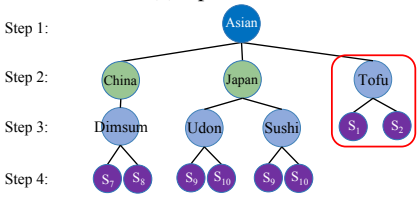
Fig. 1: Geographic Knowledge Graph

GraphTraverse first traverses the graph to find the matched graph patterns and then evaluates the spatial predicates. Then, the vertexes that cannot satisfy the spatial predicate (e.g., places within the extents of Q) are filtered out. Although GraphTraverse answers queries correctly, it may possibly visit many unnecessary vertexes/edges during the graph traversal. Another approach, namely SpaIndex, initially builds a spatial index [1], [2], [4], e.g., R-Tree, over the spatial vertexes in the graph. Neo4j is equipped with a spatial extension for managing spatial data which supports basic spatial range, nearest neighbor and distance queries. The query processor in SpaIndex runs in two steps: (1) Step I applies the index to find all spatial vertexes that can satisfy the spatial predicate; (2) Step II then traverses the graph starting from the set of spatial vertexes. Step I may retrieve spatial vertexes that satisfy the spatial predicate but do not match the query graph; That will unnecessarily traverse graph paths. In conclusion, both approaches exhibit unacceptable performance in applications that need to query the geographic knowledge graph in real-time or near real-time. Figure 2 shows the steps of answering Q1 using the two strategies.

In this paper, we demonstrate a system, namely Spindra, that provides efficient management of geographic knowledge graphs. The system is equipped with a geographic knowledge graph storage and indexing module that extends the core functionality of a graph database system (i.e., Neo4j) to efficiently store location facts and relationships among



(a) SpaIndex



(b) GraphTraverse

Fig. 2: Figure 2a depicts the steps the SpaIndex approach takes to process the example query given: *Step 1:* Search R-Tree to filter out the objects that are not located within the region Q. $R3$ is not accessed because it does not overlap with Q. *Step 2:* Traverse the graph from each spatial object to obtain the food type that the restaurant has menu in. *Step 3, 4:* Search the graph by following *SubcategoryOf* to find the Asian food vertex. Figure 2b shows how GraphTraverse processes the same query: *Step 1:* It first obtains vertex that represents Asian food. *Step 2:* Search for all types which are subcategory of Asian food. *Step 3, 4:* For each type, find all restaurants that have menu in it and are located within Q.

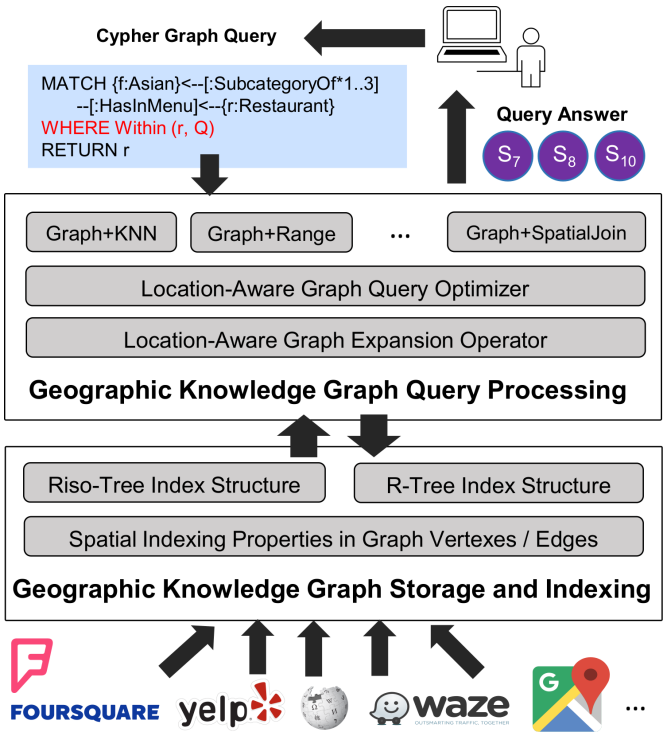


Fig. 3: System architecture overview

them as vertexes and edges. The system also optimizes and processes queries issued on the geographic knowledge graph. We demonstrate the system using an interactive map-based web interface that allows users to issue location-aware search queries over the WikiData knowledge graph. The Front-end will then visualize the returned geographic knowledge to the user using OpenStreetMap.

II. System Overview

Figure 3 shows the architecture of Spindra. Spindra consists of three main components, Web Interface, Query Processing Coordinator, and Data Store and Indexing. In the following, we demonstrate each of the components.

A. Data Store and Indexing

The backend of the system stores the geographic knowledge graph data and the index structure. The data source can be existing well-known datasets, such as Foursquare, Yelp, Wikipedia, etc. The graph data is managed by the graph database. The information, like the location of an entity in the graph is stored as the property of a vertex. Two categories of indexes, SPA-Graph and spatial index are exploited by the system to accelerate location-aware graph queries (will be demonstrated later).

SPA-Graph is an augmented index structure on top of the graph data. It keeps all the information of the graph data but augments it with some spatial information to make the graph data spatial-aware. Figure 4 depicts the structure of SPA-Graph. The spatial information stored on each vertex v describes the spatial boundary of spatial vertexes that can be reached from v through a specific number of hops. Such spatial information has three categories:

- **GeoB:** An extra bit (i.e., boolean), called Spatial Reachability Bit (abbr. GeoB) that determines whether v can reach any spatial vertex ($u \in V_S$) in the graph at a specific hop number. GeoB of a vertex v is set to 1 (i.e., true) in case v can reach at least one spatial vertex at hop k and set to 0 (i.e., false) otherwise.
- **RMBR:** Reachability Minimum Bounding Rectangle (abbr. RMBR) represents the minimum bounding rectangle MBR(S) (represented by a top-left and a lower-right corner point) that encloses a set of vertexes S where S includes all spatial vertexes reachable from vertex v through k hops.
- **ReachGrid:** A list of spatial grid cells, called the reachability grid list (abbr. ReachGrid). Each grid cell C in ReachGrid(v) belongs to a hierarchical grid data structure that splits the total physical space into $r \times r$ spatial grid cells. Each spatial vertex $u \in V_S$ will be assigned a unique cell ID ($k \in [1, r \times r]$) in case u is located within the extents of cell k, noted as Grid(u) $= k$. Each cell $C \in$ ReachGrid(v) contains at least one spatial vertex that is reachable from v through exactly k hops.

For a query with the spatial range predicate, SPA-Graph can greatly reduce the graph search space due to its characteristic

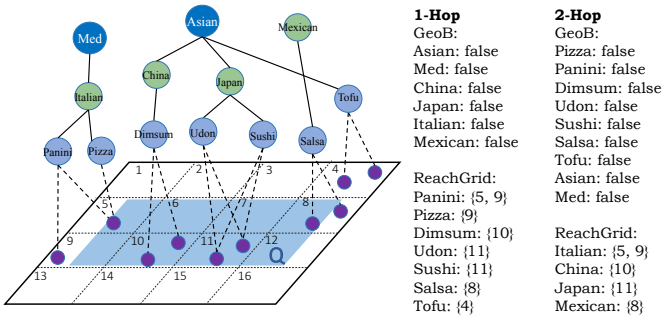


Fig. 4: SPA-Graph

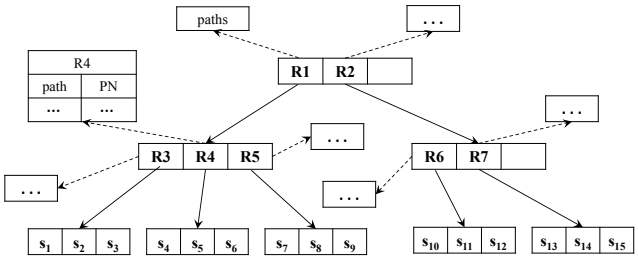


Fig. 5: Riso-Tree Structure

that for each vertex visited during the search, its future reachable spatial region can be predicted. If the reachable region of that vertex does not overlap with the query range predicate, the vertex can be safely pruned without really performing the search.

In Figure 4, for instance, vertex *Tofu* has 1-hop spatial reachable information of a list $\{4\}$. This means traversing from vertex *Tofu* can only reach spatial vertexes within cell 4. By exploiting this information, the GraphTraverse strategy in 2b can be improved because the search space after *Tofu* can be avoided (highlight in red rounded rectangle). The reason is that cell 4 does not overlap with the query region Q. So the algorithm knows further traversal will not lead to any satisfying results and terminates the search at *Tofu*.

Another index structure category is spatial index. It includes R-Tree and Riso-Tree together to accelerate the search. Riso-Tree [5] is a graph-aware spatial index. Figure 5 demonstrates its abstract structure. Riso-Tree takes R-Tree as its skeleton but with graph information being attached to its nodes. For each non-leaf node in Riso-Tree, it is stored with all the paths connected to the spatial vertices belonging to this R-Tree node. Each leaf node is stored with not only the paths but also the vertices that can be reached through each path. In R-Tree, the pruning only happens on nodes whose MBRs do not overlap with the query region. By exploiting Riso-Tree, the search can skip some nodes in the tree if any desired path is not included in the current node's graph information besides pruning according to spatial overlap. So Riso-Tree provides more pruning power compared to R-Tree.

Riso-Tree can improve the performance of SpaIndex strategy. In Figure 2a, s_4 and s_6 lead to unnecessary search because they are not related to Asian food. But the algorithm can only drop s_4 and s_6 after searching the graph. By utilizing the label paths stored on R4 and R5, the algorithm can directly terminate the search here without triggering the traversal from either s_4 or s_6.

B. Query Processing Coordinator

Basically, the Query Processing Coordinator stands as the middle layer between users and the backend data store. It takes location-aware graph queries (GraSp) from users and returns the correct result.

Typical queries in GraSp include GraSp-Range, GraSp-KNN and GraSp-Join, etc. The example query mentioned previously is one GraSp-Range query. It has a graph constraint part and a range constraint part. GraSp-KNN asks for the top-K closest spatial objects which satisfy a given graph pattern from a given location. For instance, to find 10 closest restaurants that have menu in Asian food. GraSp-Join is a spatial join query with graph constraints. An example for GraSp-Join can be to search for all the restaurants that have menu in Asian food and are close to a resort (e.g., the distance less than 1 km).

The Location-Aware Graph Query Optimizer analyzes the given query and decides the best execution plan. The plan can be either GraphTraverse, SpaIndex or Compound strategy but with new operators.

Basically, two existing operators will be replaced. One is the spatial index search operator used in *Step 1* of SpaIndex strategy. It will be replaced with a graph-aware operator, which considers the paths in the query and searches Riso-Tree instead to reduce search space. Another one is the EXPAND operator used to expand from a vertex to fetch all its neighbors. The optimizer will replace it with GEOEXPAND. GEOEXPAND considers the information stored on SPA-Graph and expands a vertex only if its spatial reachable information shows the vertex can possibly reach the query region after the EXPAND. After the plan rewriting, the execution plan will be executed.

III. SCENARIO

In this section, we demonstrate real user scenarios by using Spindra. The WikiData graph dataset [7] is used as the data source. WikiData is a knowledge graph extracted from Wikipedia. It contains real-world objects and their relationships. The data is loaded into Neo4j graph database system. The storage backend, including SPA-Graph and Riso-Tree, are constructed after the data is loaded.

A real scenario can be described as follows: An ICDE 2019 participant plans to explore Macau. He/she is interested in visiting some museums in Macau. A query to search for nearby places that are *InstanceOf MUSEUM* is helpful in this case. Figure 6 shows the interface for managing the data in the backend data store. It can answer Cypher queries and visualize the graph data and the index information. The figure shows an overview of the related entities for the query in the dataset. Blue circles represent *PLACE* entities. They are *InstanceOf* different categories and categories can be

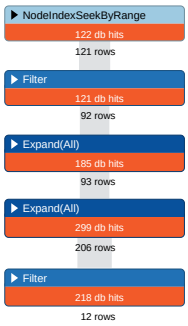


Fig. 6: Backend Data View Interface

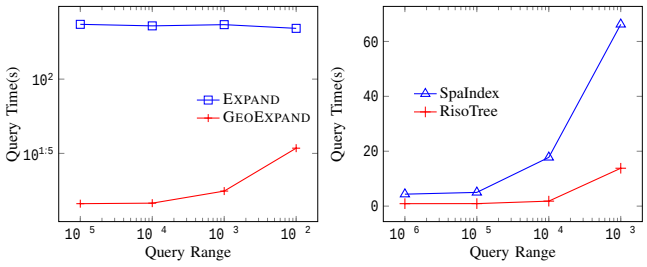
SubclassOf other categories. Normally, the query should only search for museums. However, it might happen that there is no museum nearby or the places whose categories are related to museum (e.g., amusement park) can be accepted and recommended as alternatives to increase the richness of the result. So the system will trigger a search to find all the places that are nearby and their categories are *SubclassOf* tourist attraction.

Figure 6 also shows the execution plan of the query by directly running a Cypher query in Neo4j. Because of the existence of the spatial index, the spatial filtering will take be performed. The first two boxes in the plan reflect such a step. Such step incurs 243 db hits and 92 spatial objects are within the search region. Then for each spatial object, the executor performs two *Expand(All)* operators. The first *Expand(All)* operator expands through the edge *InstanceOf* and fetches all nodes connected by such edge type. The second *Expand(All)* expands each node obtained from the previous step through edge *SubclassOf*. The two *Expand(All)* operators increase the number of rows from 92 to 206 because each node can have many neighbors. The final step is to check whether the current node is *amusement park*. At the last box in the execution plan, the number of rows decreases from 206 to 12 after the *Filter* operation. It is because many subgraphs do not contain a node of *tourist attraction*. So even 92 spatial entities are within the query region, only 12 of them can satisfy the graph constraint of being *InstanceOf* a category that is *SubclassOf* tourist attraction. In other words, many nodes visited by the *Expand(All)* operator are unnecessary.

When this query is issued in Spindra, the spatial filter phase will also be performed at first. But with the help of Riso-Tree, not all spatial objects within the query region are promising. Those spatial objects that do not have the required label paths will not be executed in the next step. So there will be far less than 92 rows. As a result, the execution time will be reduced.

Figure 7 shows a web interface which visualizes all the satisfying spatial objects in a map view. We can observe that not only museums but also some other spatial entities, such as Macau Tower, which is a perfect place for tourism are returned. In the web interface, users can move the searching

Fig. 7: Query Result Map View



(a) GEOEXPAND VS EXPAND (b) Riso-Tree VS R-Tree

Fig. 8: GraSp-Range query response time

center by dragging the red marker and change the search radius. The user can also change the search topic to Hotel, University, etc.

In order to test the influence of the query range selectivity in GraSp-Range, we run further experiments on Foursquare dataset. We vary the spatial selectivity, which is measured by the ratio of the number of spatial objects in the query region to the total number of spatial objects in the dataset. Figure 8a shows that by using the replaced GEOEXPAND operator, the query performance can be improved because spatially-unpromising subgraphs are pruned. Figure 8b shows that by using Riso-Tree, the execution time can be reduced by two orders of magnitude. It also reveals that the Riso-Tree can become especially effective when the selectivity is not too high.

REFERENCES

[1] BECKMANN, N., KRIEGEL, H., SCHNEIDER, R., AND SEEGER, B. The r*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD* (1990), pp. 322–331.
[2] GUTTMAN, A. R-Trees: A Dynamic Index Structure For Spatial Searching. In *SIGMOD* (1984).
[3] Neo4j graph database. https://neo4j.com/.
[4] SAMET, H. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.
[5] SUN, Y., AND SARWAT, M. A generic database indexing framework for large-scale geographic knowledge graphs. In *ACM SIGSPATIAL GIS* (2018), pp. 289–298.
[6] Titan distributed graph database. http://titan.thinkaurelius.com/.
[7] VRANDEČIĆ, D., AND KRÖTZSCH, M. Wikidata: A free collaborative knowledgebase. *Commun. ACM 57*, 10 (Sept. 2014), 78–85.