On Processing Spatial Queries in Graph Database Management Systems

by

Yuhan Sun

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved March 2021 by the
Graduate Supervisory Committee:

Mohamed Sarwat, Chair
Hanghang Tong
Kasim Candan
Ming Zhao

ARIZONA STATE UNIVERSITY

March 2021

ABSTRACT

Spatial data is fundamental in many applications like map services, land resource management, etc. Meanwhile, spatial data inherently comes with abundant context information because spatial entities themselves possess different properties, e.g., graph or textual information, etc. Among all these compound spatial data, geospatial graph data is one of the most challenging for the complexity of graph data. Graph data is commonly used to model real scenarios and searching for the matching subgraphs is fundamental in retrieving and analyzing graph data. With the ubiquity of spatial data, vertexes or edges in graphs are enriched with spatial location attributes side by side with other non-spatial attributes. Graph-based applications integrate spatial data into the graph model and provide more spatial-aware services. The co-existence of the graph and spatial data in the same geospatial graph triggers some new applications. To solve new problems in these applications, existing solutions develop an integrated system that incorporates the graph database and spatial database engines. However, existing approaches suffer from the architecture where graph data and spatial data are isolated.

In this dissertation, I will explain two indexing frameworks, GeoExpand and Riso-Tree, which can significantly accelerate the queries in geospatial graphs. GeoExpand includes a query operator that adds spatial data awareness to a graph database management system. In GeoExpand, the neighborhood spatial information is summarized and stored on each vertex in the graph. The summarization includes three different structures according to the location distribution. These spatial summaries are utilized to terminate the graph search early. Riso-Tree is a hierarchical tree structure where each node is represented by a minimum bounding rectangle (MBR). The MBR of a node is a rectangle that encloses all its children. A key difference

between Riso-Tree and R-Tree is that Riso-Tree contains pre-materialized sub-graph information to each index node. The sub-graph information is utilized during the spatial index search phase to prune search paths that cannot satisfy the query graph pattern. The Riso-Tree index reduces the search space when the spatial filtering phase is performed with relatively light cost.

TABLE OF CONTENTS

CHAPTER Page

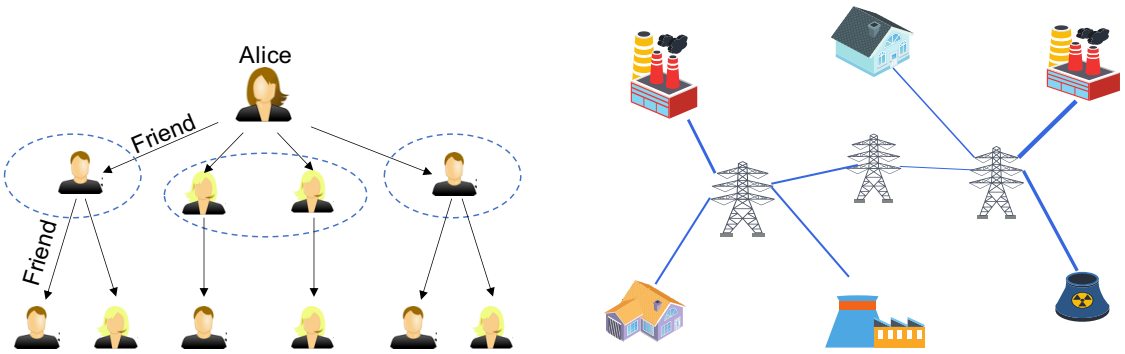
Chapter 1

INTRODUCTION

Spatial data is ubiquitous and any data with spatial information can be categorized into spatial data. Spatial data is fundamental in many applications like map services, land resource management, etc. These applications ignite the demand for accessing and analyzing spatial data efficiently using some basic spatial query types, e.g., spatial range query, spatial join query and k-NN query. Different techniques are proposed for spatial queries from data structure and algorithm perspective [7, 24, 39] and system perspective [59, 14, 1].

Meanwhile, spatial data inherently comes with abundant context information because spatial entities themselves possess different properties. For instance, geospatial graph data and spatial textual data are formed by enriching spatial data with graph and textual information respectively. Among all these compound spatial data, geospatial graph data is one of the most challenging for the complexity of graph data.

Graph data is commonly used to model real scenarios, including knowledge graph [19, 34, 57, 10, 58, 21], social network [15], protein network [23, 45, 37, 5], utility network [33], etc. Searching for the matching subgraphs is fundamental in retrieving and analyzing graph data. The application includes but is not limited to chemical and protein structures analysis, image processing [46], and social network community search. Figure 1 shows two graphs in real applications. Figure 1a is a social graph where people are modeled as vertexes and friendship relationships are modeled as edges. In such a social graph, searching the friends of friends' is a common query for friends recommendation. A utility graph (shown in Figure 1b) consists of utility



(a) Social Graph (b) Utility Graph

Figure 1: Graph Examples

facilities, like power station, transformer, meters. These facilities are connected through transmission lines and distribution lines. On top of a utility graph, the administrator of the utility management system may issue a query like searching all impacted meters if some transformers are broken.

Thanks to the popularity of location-aware mobile devices, spatial data is proliferating and easy to access at a low cost. With the ubiquity of spatial data, vertexes or edges in graphs are enriched with 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 $\approx$13% of them have spatial location attributes [48]. Social graph applications, like Facebook, Twitter, etc., also allow users to check in the location when publishing posts or pictures.

Graph-based applications integrate spatial data into the graph model and provide more spatial-aware services. The co-existence of graph and spatial data in the same geospatial graph triggers some new applications [4, 47, 41, 16, 8, 11, 12, 17, 30, 42, 49, 50, 56, 6, 44, 13, 36]. These new applications allow users to issue graph queries with spatial awareness:

- Q1:`Find all places in Seattle that are co-visited by at least two friends or friends of friends.`
- Q2:`Find single-family houses in Tempe, AZ that receive power from a distribution site connected to the central power station in Maricopa County.`

Compared with the queries on top of the applications in Figure 1, these queries are more than just graph queries but enriched with spatial relevance. These queries consist of graph query components and spatial predicates. The returned results are the subgraphs that match both graph and spatial constraints in the query. Even to find the subgraphs that match only the graph component of the given query is nontrivial because of the inherent complexity of graph data. The addition of the spatial component to the graph query makes the query more complex.

When looking at the graph component and spatial component of such queries independently, there are existing solutions for both of them. Graph database management systems (GDBMS) such as Neo4j [32] and Titan [52] provide an efficient way for managing and querying graph data. Spatial database systems can utilize the spatial index [7, 24, 39], *e.g.,* R-Tree to efficiently answer different categories of spatial queries, like spatial range, join and KNN queries. So a straightforward solution is to develop an integrated system that incorporates the existing graph database and spatial database engines. Figure 2 illustrates the architecture of such a system. Vertexes and edges of a graph are seamlessly transplant and stored in GDBMS. Attributes of vertexes and edges are normally stored as key-value pairs. Spatial attributes can be managed in the same manner as any other ordinary attribute. At the same time, entities that have spatial attributes in the graph are stored and indexed in the spatial database. When

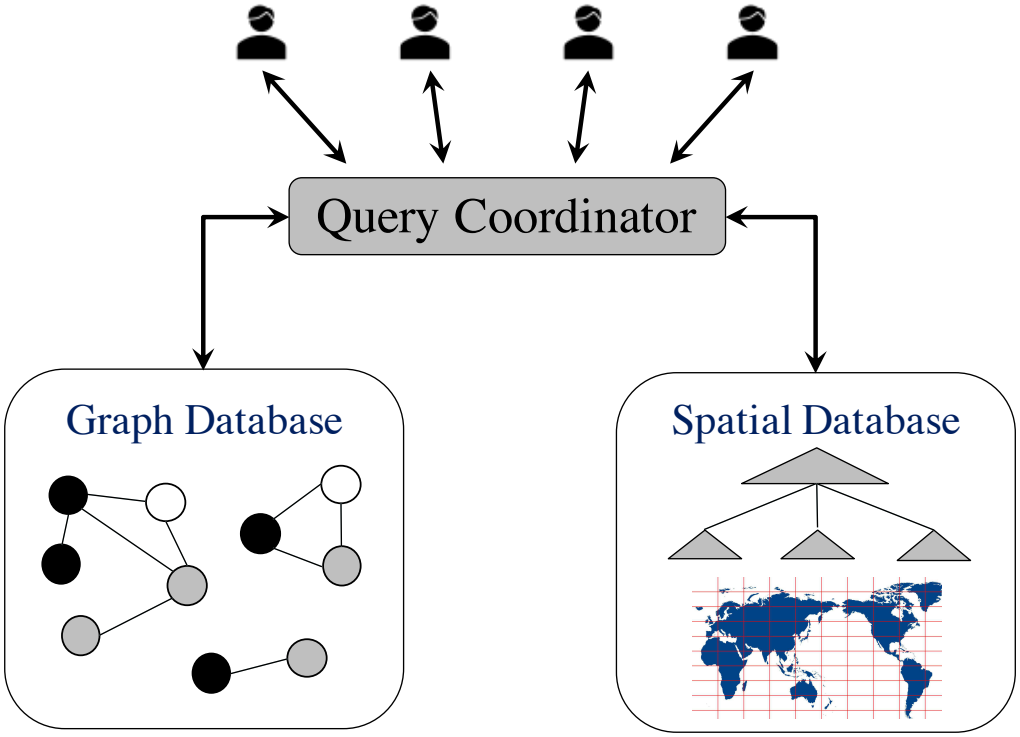


Figure 2: Isolated Architecture

queries are issued from users, Query Coordinator decomposes the query into the graph and spatial components. The query components are sent to corresponding database engines and answered by the corresponding database engines. The intermediate result is returned back to Query Coordinator and the final result is formed by combining the intermediate result. On top of this system architecture, two fundamental strategies (i.e., GraphTraverse and SpaIndex) can be taken to solve the queries in geospatial graphs. GraphTraverse first access the graph database to find the matched graph patterns and then evaluates the spatial predicates. In the evaluation, the vertexes that cannot satisfy the spatial predicate (*e.g.,* places outside the extent of Seattle in Q1) are filtered out. SpaIndex strategy 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 search the graph for the query graph pattern starting from the set of spatial vertexes.

Such an architecture takes the best from the existing systems. The developers only need to implement Query Coordinator and do not need to care about how the query components are processed. The state-of-the-art techniques applied to either graph database or spatial database can be seamlessly applied to such a system. However, the graph database and spatial database are isolated components in the architecture. Spatial entities are stored as an isolated community from the whole graph. Graph vertexes and spatial entities are accessed and searched in a completely separate fashion. Due to such a characteristic, both strategies will perform the unnecessary search. GraphTraverse will fetch many intermediate subgraph results that cannot satisfy the spatial predicate. This not only harms the performance of the graph search but also increase the computation cost of the spatial predicate evaluation. Similarly, SpaIndex strategy will generate many spatial entities that cannot form any final result after the graph search.

To improve the performance of GraphTraverse strategy, I propose GeoExpand framework, which augments the graph with Spatial Indexing Properties (SIP) to form what we call SPatially-Augmented Graph (SPA-Graph). For each vertex, its neighboring spatial information is stored in an efficient way as its property. SIP has three different formats. The framework will choose the appropriate format according to the neighboring spatial data distribution and pre-defined parameters. When the graph search is performed, the algorithm will validate the spatial predicate by using SIP to stop the unpromising earlier.

I also propose another technique, Riso-Tree to accelerate the SpaIndex strategy. Similar to the R-Tree, Riso-Tree is a hierarchical tree structure where each node is represented by a minimum bounding rectangle (MBR). The MBR of a node is a rectangle that encloses all its children. A key difference between Riso-Tree and R-Tree

is that Riso-Tree contains pre-materialized sub-graph information to each index node. The sub-graph information is utilized during the spatial index search phase to prune search paths that cannot satisfy the query graph pattern. The Riso-Tree index reduces the search space when the spatial filtering phase is performed with relatively light cost.

In the rest of the thesis, I will first introduce some background information, and the literature review is given to summarize the related work of the problem. In Chapter 3 and 4, I demonstrate the technical details of the GeoExpand and Riso-Tree framework. In Chapter 5, the maintenance method is introduced for the proposed framework. A geospatial knowledge management system is demonstrated in Chapter 6. The conclusion is given in Chapter 7.

Chapter 2

BACKGROUND AND LITERATURE REVIEW

2.1 Geospatial Graph Data Model

A labeled property graph $G = (V, E, \varphi, \psi, \phi)$ can be defined as follows:

1. V is a set of vertexes,
2. E is a set of edges,
3. φ is a mapping function $\varphi : V \to \mathbb{P}(\mathcal{L}_V)$, where $\mathcal{L}_V$ is the set that consists of all the vertex labels, $\mathbb{P}(\mathcal{L}_V)$ denotes the power set of $\mathcal{L}_V$. So φ maps each vertex to a set of multiple labels,
4. ψ is a mapping function $\psi : E \to \mathcal{L}_E$, where $\mathcal{L}_E$ is the set consists of all edge labels,
5. ϕ is a mapping function that maps each vertex to a set of *(key, value)* attributes.

A geospatial graph is a labeled property graph where some vertexes possess spatial location attributes, *e.g.,* point, polygon. The spatial attribute of a vertex is denoted as $v.loc$.

2.2 Geospatial Graph Query

Given a graph $G = (V, E, \varphi, \psi, \phi)$, a graph query $G_q = (V_q, E_q, \varphi_q, \psi_q)$ finds all the maps f from G_q to G such that: (1) $\forall v \in V_q$, $\varphi_q(v) \subset \varphi(f(v))$; and (2) $\forall (u, v) \in E_q$, $\psi_q(u, v) = \psi(f(u), f(v))$. A graph query with spatial predicates (*abbr.* GraSp) is a normal graph query except that it has spatial predicates specified on some query

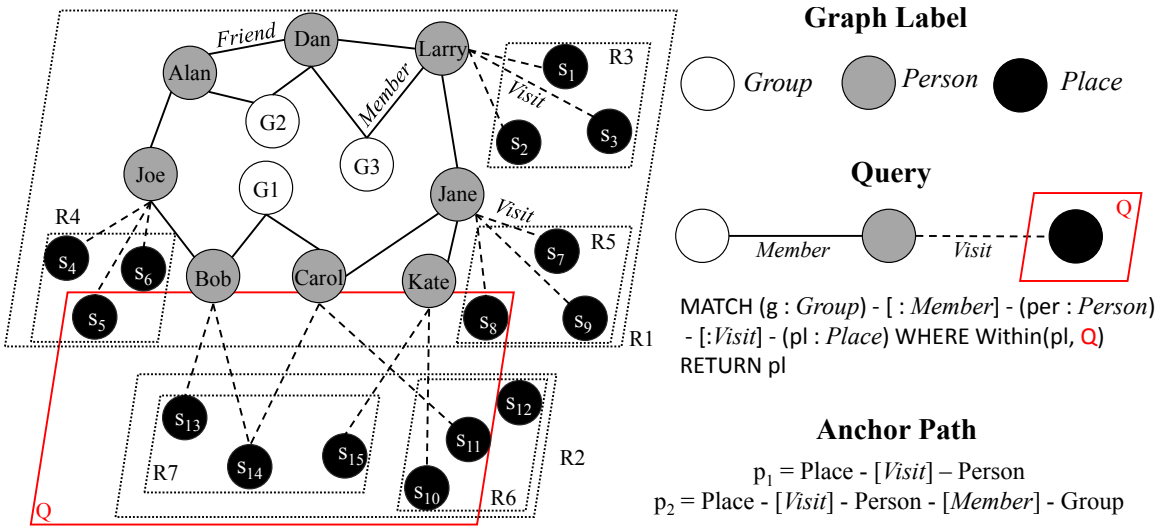


Figure 3: Running Example

vertexes. A GraSp query can be denoted as $\langle G_q, SP_q \rangle$ where $G_q = (V_q, E_q, \varphi_q, \psi_q)$ and SP_q is a spatial predicate. The returned maps and sub-graph should satisfy both graph pattern constraint G_q and the spatial predicate SP_q.

We consider three main categories of spatial predicates, including spatial range, KNN and spatial join predicates. A spatial range predicate has a format of $\langle s, Q \rangle$, where s is a query vertex in G_q and Q is a query rectangle. It means that the query vertex s needs to be within Q. A KNN predicate $\langle s, loc, K \rangle$ means s needs to be the top-K closet spatial vertexes from loc that satisfy G_q. A spatial join predicate $\langle s, t, P_{join} \rangle$ means s and t should satisfy the spatial relationship predicate P_{join}. Here P_{join} can be any relationship between two spatial geometries, like intersect, disjoint, within distance, etc. The corresponding queries with different categories of spatial predicates are denoted as GraSp-Range, GraSp-KNN and GraSp-Join respectively.

An example of a graph query with range predicate (GraSp-Range) is depicted in Figure 3. The graph with three types of vertexes, `Group`, `Person` and `Place`, depicted with white, gray and black respectively. Spatial vertexes (`Place`) are denoted with s_i

where $i = 1, 2, ..., 15$. Each spatial vertex has a location in the 2-D space. There are three types of edges, `Member`, `Friend` and `Visit`. A `Member` edge means a person is a member of a group. A `Friend` edge means the two persons are friends. A `Visit` edge means a place is visited by a person.

The example GraSp= $\langle G_q, SP_q \rangle$ has two components. The G_q component of the GraSp is $Group{-}[Member]{-}Person{-}[Visit]{-}Place$. The predicate is SP_q ={(`Place`, Q)} where Q is the spatial range enclosed with a red rectangle in the figure. Such query should return all possible mappings, including f_1 = {(`Group`, G1), (`Person`, Bob), (`Place`, s_{13})}, f_2 = {(`Group`, G1), (`Person`, Bob), (`Place`, s_{14})}, f_3 = {(`Group`, G1), (`Person`, Carol), (`Place`, s_{11})}, f_4 = {(`Group`, G1,), (`Person`, Carol), (`Place`, s_{14})}.

2.3 Literature Review

2.3.1 Graph Query

Generally, graph queries consist of any query that can be answered on top of a property labeled graph. Graph queries can be categorized into three main types based on their different purposes and execution methods.

2.3.1.1 Graph Pattern Matching Query

Graph pattern matching query takes input as a pattern and outputs a set of subgraphs that 'match' the given pattern. The purpose of this type of query is to select a relatively smaller subset of the original graph. The reason behind it is that

the original graph is so big that it is impossible to visualize or explore the whole graph at one time. Moreover, users may only be interested in part of the graph. Such a query facilitates users to narrow down the graph to their main focus. The pattern here is a labeled graph. The word 'match' can be tricky. The match can be exact or approximate, which allows some relaxation on the matching criteria.

Exact Match. An exact match requires there is a one-to-one projection (or mapping) between the returned subgraph and the query graph, including each vertex and edge. So the subgraph will have the same number of vertexes, edges, and the same topology structure. Subgraph Isomorphism is often used to find the exact matching subgraphs. It is defined as follows: Given a graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E, \varphi, \psi)$ and a graph query $G_q = (V_q, E_q, \varphi_q, \psi_q)$, find all the maps f from G_q to G such that:

1. $\forall v \in V_q$, $\varphi_q(v) = \varphi(f(v))$,
2. $\forall (u, v) \in E_q$ then $\psi_q(u, v) = \psi(f(u), f(v))$.

Approximate Match. Such a category of pattern matching queries does not require the returned subgraphs to exactly match the query pattern. It may define a rule to determine if the given subgraph can match the query graph pattern. It can also define a metric to evaluate the similarity between the subgraph and the query pattern. The returned subgraphs are either the top-K most similar to the query pattern or their similarities are larger than a given threshold. For different applications, the similarity metrics are defined in various ways to reflect the different purposes of the applications. Generally, the graph similarity (or distance) can be defined in two different ways [40]:

1. Feature-based distances. A set of features or invariants is established from a structural description of a graph, and these features are then used in a vector representation to which various distance or similarity measures can be applied.

2. Cost-based distance. The distance or similarity between two graphs reflects the number of prescribed edit operations that are required in order to transform one graph into the other.

2.3.1.2 Graph Algorithm Query

The graph algorithm queries not only select a subgraph but runs some algorithms consisting of sorting, categorizing, or some other techniques to generate more insight into the graph data. The examples include the shortest path query and finding the 10 most influential people in a social graph by using the PageRank algorithm.

2.3.1.3 Graph Analytic Query

This category of queries returns some information about the graph itself. This information describes some status or characteristic of the graph. Normally, all the nodes in the graph will be accessed during such queries. For instance, get the average degree of the graph. The example is trivial but a graph analytic query can be complex depending on the user's requirement.

2.3.1.4 Mixed Graph Query

Besides all the above-mentioned categories, a graph query can also be the mixture of these categories. The graph algorithm query or graph analytic query can be combined with graph pattern matching to form a more complex query. For instance, find the top-10 influential research institution established by European companies.

2.3.2 Framework For Subgraph Isomorphism

In the thesis, we mainly focus on the exact graph pattern matching query, i.e., subgraph isomorphism. The existing solutions for subgraph isomorphism can be divided into two different categories, i.e., memory-based and disk-based based on how the data is stored and accessed. Memory-based methods assume the whole graph is stored in memory while disk-based methods assume the graph data resides on disk. In the following, we introduce the two categories of methods in detail.

2.3.2.1 Memory-based Solutions

Memory-based approaches assume that the whole graph, including nodes and edges can be loaded into memory. Before the query process starts, the graph data will be loaded into memory once. Then the graph queries are performed by accessing the graph data in memory. These approaches basically follow a common computing algorithm schema [27]:

Algorithm 1 GenericQueryProc

```
1: M ← ∅
2: for u ∈ V_q do
3:    C(u) ←FilterCandidates(G, G_q, u)
4:    if C(u) = ∅ then
5:       return
6:    end if
7: end for
8: SubgraphSearch(G,G_q,M)
```

Algorithm 1 shows a generic subgraph isomorphism algorithm, GenericQueryProc. Its inputs are a query graph G_q and a data graph G, and its output is a set of subgraph isomorphisms (or embeddings) of G_q in G. Here, to represent an

Algorithm 2 SubgraphSearch

```
1: if |M| = |V_q| then
2:    return M
3: else
4:    u ←NextQueryVertex (. . .)
5:    C_R ←RefineCandidates(M,u,C(u),...)
6:    for each v ∈ C_R that is not yet matched do
7:       if IsJoinable (q, g, M, u, v, . . .) then
8:          UpdateState (M, u, v, . . .)
9:       end if
10:      SubgraphSearch (q, g, M, . . .)
11:      RestoreState(M, u, v,...)
12:   end for
13: end if
```

embedding, we use a list M of pairs of a query vertex and a corresponding data vertex. For each vertex u in V_q, GenericQueryProc first invokes FilterCandidates to find a set of candidate vertices $C(u)$ ($\subset V$) such that $\varphi_q(u) \subset \varphi_q(v)$ (Line 3). If $C(u)$ is empty, we can safely exit, making early termination possible (Line 5). After that, GenericQueryProc invokes a recursive subroutine, SubgraphSearch, to find mapping pairs of a query vertex and matching data vertices at a time (Line 8).

SubgraphSearch takes as parameters a query graph G_q, a data graph G, and a partial embedding M and reports all embeddings of G_q in G. The recursion stops when the algorithm finds the complete solution (i.e., when $|M| = |V_q|$) (Line 2). Otherwise, the algorithm calls NextQueryVertex to select a query vertex $u \in V_q$ which is not yet matched (Line 4). After that, it calls RefineCandidates to obtain a refined candidate vertex set C_R from $C(u)$ by using algorithm-specific pruning rules (Line 5). Next, for each candidate data vertex $v \in C_R$ such that v is not matched yet, the IsJoinable subroutine checks whether the edges between u and already matched query vertices of V_q have corresponding edges between v and already matched data vertices of G (Line 7). If v is qualified, it is matched to u, and

SubgraphSearch updates status information by calling UpdateState (Line 8), and the algorithm proceeds to match the remaining query vertices of q by recursively calling SubgraphSearch (Line 10). Next, all changes done by UpdateState are restored by calling RestoreState (Line 11). The algorithm terminates when all embeddings are found.

This framework includes the computation steps of graph search in memory. The cost of the search is evaluated by using the CPU computation cost. It does not consider the disk access cost and assume all the data is residing in memory. The computation cost is mostly determined by the IsJoinable operation. The more it is performed, the longer the graph search will take. Such an operation is achieved by implemented checking the existence of an edge between a given pair of vertexes. This operation can be executed at a low cost when the graph is in memory. However, it is not trivial when the graph data is residing on disk.

2.3.2.2 Disk-based Approaches

Disk-based approaches are proposed for the scenarios that all the data cannot be loaded into memory when the data volume is too large or there is a requirement that the data needs to be persistent on disk. The difference between disk-based and in-memory approaches lies in the following aspects:

- **Candidates Generation**. The memory-based approaches start with initializing the candidate sets $C(u)$ for all the query vertexes in the query graph G_q. Such a step needs no cost because all the data is stored in memory. In disk-based approaches, however, such an operation is not affordable because each time to

form a candidate set, it requires to at least fetch all the vertexes with such label, which will incur huge disk I/O.

- **IsJoinable**. In memory-based approaches, each time to match one more query vertex, the IsJoinable is called to check the existence of an edge. But for the disk-based scenarios, it is costly to check the existence of an edge. It requires the disk access of the vertexes or the edge with more than one I/O cost.
- **Cost Model** The in-memory approaches focus on minimizing the CPU computation cost. But for disk-based approaches, the goal is to reduce both the CPU computation cost and more importantly, the disk I/O because disk access is more expensive than CPU computation.

Due to these differences, disk-based approaches have a different set of operations. When graph data is stored on disk, normally the neighborhood information of a given vertex is stored and accessed altogether. So the unit operator is to expand from a vertex and fetch all its neighbors. Based on such a characteristic, disk-based approaches utilize NodeLabelScan, Expand, Filter and Join operators to perform the subgraph isomorphism.

- NodeLabelScan. The NodeLabelScan operator fetches all the vertexes in the data graph G that with a given label.
- Expand. The Expand operator takes input as a set of vertexes and performs a 1-hop graph traversal to fetch all the neighbors of the set of vertexes.
- Filter. The Filter operator takes a set of vertexes and decides whether each vertex satisfies the given constraint. The constraint can be any boolean function. In the subgraph isomorphism problem, the filter is to decide whether a vertex has the given label.

- JOIN. The JOIN operator takes two sets of vertexes and outputs the common nodes.

Based on these operators, a subgraph isomorphism query is executed by combining these operators. Different plans can be generated and executed for the same query. To determine the best execution plan, disk-based approaches come with a plan optimizer to determine the best execution plan and perform the operator in the plan in a specific order.

2.3.2.3 Summary

In-memory methods start performing the query until all the graph data is loaded into memory. It is much faster to access the in-memory data compared with accessing data reside on disk. However, the graph data size can be too huge to fit in the memory. Then these approaches cannot be directly used. Another disadvantage is that these approaches need to load the whole graph into memory every time they perform the graph query. There will be a long waiting time until the first query can be answered. When there is no query issued, the graph data will still occupy the system resources. These limitations do apply to disk-based approaches. In this thesis, we focus on disk-based approaches.

2.3.3 Systems for Geospatial Graph Data

Some existing graph database systems allow users to define spatial properties on graph elements, like other ordinary properties. Indexing is available for accelerating the search space filtering on spatial predicates.

Neo4j. Neo4j [32] is a native graph database system. It takes a graph model called Property Labeled Graph (PLG), where nodes and edges are differentiated by string labels and attached with a set of properties. The edges connected to the same node are managed in a double-linked list. The cost to fetch a neighbor for a node is always $O(1)$. Such an index-free storage schema makes it efficient for graph traversal algorithms. The flexibility to define labels and properties makes PLG suitable for applications with rich semantics. Spatial information can be stored as built-in spatial types, including geospatial points in 2D and 3D. It supports several spatial functionalities, including spatial range, spatial distance. These queries are accelerated by using a back-end spatial index. Neo4j supports Cypher [9], which is a declarative graph query language. A GraSp can be represented by a Cypher query and executed by the Neo4j Cypher query engine.

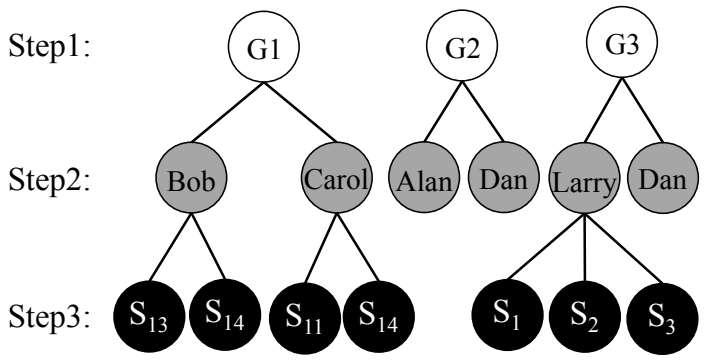
RDF. RDF is another popular graph database. The edges and node properties in the graph data model are stored as triples RDF is triple-based and each triple has the format of $\langle Subject, Predicate, Object\rangle$ (abbr. $\langle S, P, O\rangle$). Edges and properties are stored as tuples in RDF. Graph search is performed by using table join on the triple table. The spatial property for an entity is stored as a triple as well. GeoSPARQL [20], proposed by Open Geospatial Consortium (OGC), is a standard for representing and querying spatial data in RDF data using SPARQL. Geometry types of Geography Markup Language (GML) and well-known text representation of geometry (WKT) literals are supported. Simple Features, RCC8 and DE-9IM topological relationships are included in the query language. Such systems include but not limited to Virtuoso [53], GraphDB [22] and SRX [51]. Both Virtuoso and GraphDB exploit spatial indexing techniques for spatial queries. SRX encodes objects based on the Hilbert Curve Order

and proposes the corresponding query algorithms for spatial range, spatial join, and spatial KNN predicates.

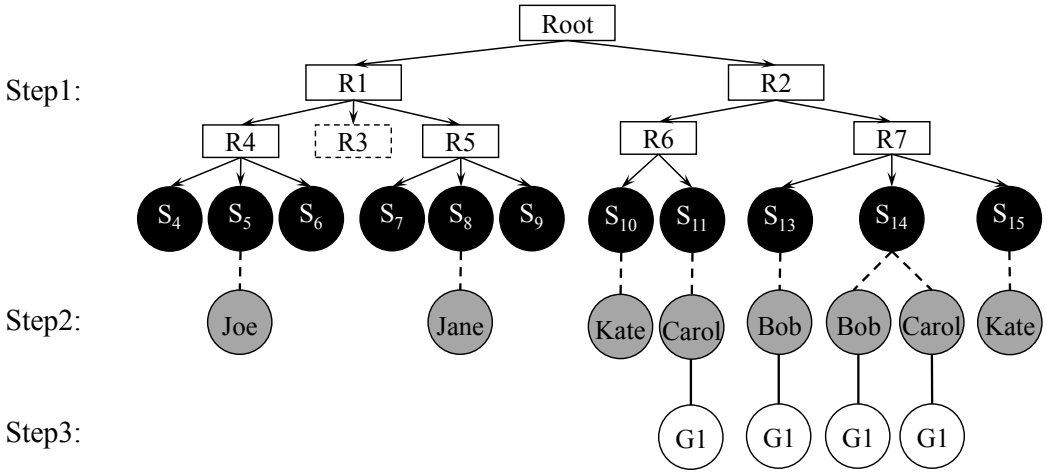
Some Others. Oracle Spatial and Graph [35] is a commercial database management system software for managing spatial and graph data. It can store both the RDF and property labeled graph data models. However, the spatial data and graph data management engines in Oracle Spatial and Graph are two separate modules. ArcGIS is a general-purpose GIS system. It is now equipped with an extension for managing utility network [2, 26]. It provides support for the network attributes and topology management that allows users to perform network analysis like utility network tracing and diagram generation. But it lacks the support for the general graph search.

2.3.4 Strategies for Geospatial Graph Query

Existing systems for geospatial graph data are designed following the architecture shown in Figure 2. The graph data is stored and managed by Graph Database while the spatial data is managed and indexed by a Spatial Database. Query Coordinator receive queries issued by users. Queries are parsed and decomposed into a graph and spatial components. The decomposed query components are qualified for being directly answered by Graph Engine and Spatial Index. Then Graph components and spatial components of Geospatial Graph Query (GraSp) are sent to Graph Engine and Spatial Index. Graph Engine and Spatial Index receive the query request and search the graph data and spatial data respectively. Query Coordinator combines and processes the results provided by Graph Engine and Spatial Index to generate the final results. The final results are returned back to the users.



(a) GraphTraverse



(b) SpaIndex

Figure 4: Figure 4b depicts the steps the SpaIndex approach takes to process the query given in Figure 3: *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 persons that visit such place. *Step 3:* Search the graph for all groups that each person is a member of. Figure 4a shows how GraphTraverse processes the same query: *Step 1:* It first obtains all groups. *Step 2:* For each group, it searches for all its members. *Step 3:* For each person, it finds all places that he/she visits and checks whether it is located within Q.

On top of this architecture, existing solutions follow two different strategies, GraphTraverse and SpaIndex.

2.3.4.1 GraphTraverse

GraphTraverse treats spatial location as nothing different from other attributes. It consists two main phases. In the first phase, GraphTraverse exploits the existing subgraph isomorphism algorithm to traverses the graph to find the matched subgraph patterns without considering the spatial predicates. In this phase, the approach does not consider the spatial predicate. In the second phase, it evaluates the spatial predicates. The spatial range predicate (u, Q) is evaluated by simply checking the location of the matched vertex of u towards the query range. The spatial join predicate (x, y, op) is evaluated by validating matched vertexes of x and y towards joining predicate op. The KNN predicate is different from the two spatial predicates mentioned above. The KNN predicate cannot be determined by the matched subgraph itself. It is determined by all the matched subgraphs. As a result, GraphTraverse strategy needs to first find all the subgraphs that satisfy the G_q. Then for the KNN predicate (u, loc) the distance from the matched vertex of u to the query is computed and sorted. The top-K closest subgraphs will be the final result.

Figure 4a shows how GraphTraverse processes the query given in Figure 3:

1. It first obtains all groups.
2. For each group, it searches for all its members.
3. For each person, it finds all places that he/she visits and checks whether it is located within Q.

2.3.4.2 SpaIndex

SpaIndex initially builds a spatial index [7, 24, 39], e.g., R-Tree, over the vertexes possessing spatial locations in the graph. Basically, SpaIndex runs in two steps:

1. *Spatial Filtering*: It first applies the index to find all spatial vertexes that can satisfy the spatial predicate.
2. *Graph Traversal*: It then traverses the graph starting from the set of spatial vertexes that satisfy the spatial predicate to find the matched subgraphs.

Regarding the *Spatial Filtering* phase, different spatial predicates are handled using the spatial index in different ways.

Algorithm 3 Range Predicate

```
1: Function RangeSearch(Node N, Rectangle Q, ResultSet S)
2: if N is a spatial object then
3:    if N.loc is within Q then
4:       S ← S ∪ {N}
5:    end if
6: else
7:    if N.mbr overlaps with Q then
8:       for each child n of N do
9:          RangeSearch(n, Q, S)
10:      end for
11:   end if
12: end if
```

Algorithm 3 demonstrates the steps of handling the spatial predicate using the spatial index. RangeSearch is a recursive function. It takes as inputs a node N in the R-Tree, the query rectangle Q and the result set S. The search starts from the root node. At each function call, it checks whether current node N is a spatial object or not. If it is a spatial object, it means the search has already reached the final layer of R-Tree. As a result, it checks whether $N.loc$ is located inside the query rectangle

Algorithm 4 KNN Predicate

```
1: Function KNNSearch(Query vertex u, Query location loc, Root node root)
2: result ← ∅
3: q ←NewPriorityQueue()
4: Enqueue(q, root_s, 0)
5: while |result| ≤ K and q ≠ ∅ do
6:    e ← Dequeue(q)
7:    if e is a spatial object then
8:       result.Add(e)
9:    else
10:      for each child of e do
11:         Enqueue(q, child, Dist(loc, child))
12:      end for
13:   end if
14: end while
```

Q. If it is true, N is inserted into the result set S. If N is not a spatial object, it checks whether $N.mbr$ overlaps with the query rectangle Q. If $N.mbr$ overlaps with Q, it implies that N might contain the result objects. As a result, each children node of N is searched by following the same procedure.

Algorithm 4 shows the pseudo-code of handling the KNN predicate. The algorithm keeps a priority queue q to store all the R-Tree nodes. The nodes are sorted in the queue based on their distances to the query location *loc*. Each time the head node e in the queue is popped up. If it is a spatial object, it is added to the result set. If not, all the children of e are obtained and inserted into the queue again. Such procedure will continue until the result set has the size of K. It means that the algorithm has found all the K spatial objects that are closest to the query location.

Algorithm 5 shows the main steps to solve the spatial join predicate by using R-Tree index structure. It keeps pairs of nodes in the R-Tree in a queue q. It starts by pushing the root nodes of R-Tree into q. Each time, a pair of R-Tree nodes $\langle node_a, node_b \rangle$ is dequeued. For each child of $node_a$ and $node_b$, the algorithm checks whether the predicate *op* is satisfied or not. The pairs that can satisfy the *op* are

Algorithm 5 Spatial Join Predicate

```
1: Function SpatialJoinSearch(Join Vertex s, Join Vertex t, Join Operator op)
2: Queue q ← ∅
3: q.Enqueue(⟨root_s, root_t⟩)
4: while q ≠ ∅ do
5:    ⟨node_s, node_t⟩ ← q.Dequeue()
6:    if node_s and node_t are not spatial objects then
7:       for each child_s of node_s and child_t of node_t do
8:          if op(node_s, node_t) = true then
9:             q.Enqueue(⟨node_s, node_t⟩)
10:         end if
11:      end for
12:   else
13:      Add ⟨node_s, node_t⟩ into the candidate set
14:   end if
15: end while
```

enqueued. If the pair are spatial objects rather than R-Tree nodes, this pair of spatial vertexes will be added to the result set.

Figure 4b depicts the steps the SpaIndex approach takes to process the query given in Figure 3:

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.
2. Traverse the graph from each spatial object to obtain persons that visit such a place.
3. Search the graph for all groups that each person is a member of.

2.3.4.3 Summary

Such an architecture can take advantage of existing database systems without modifying their internal implementation. The system developer only needs to focus on Query Coordinator. Any future modification will also only happen on Query

Coordinator if necessary. This reduces the complexity of developing and extending the system. Even though the most up-to-date techniques can be applied to Graph Engine and Spatial Index, which ensures the fast processing of graph or spatial data search, it still can lead to bad performance due to the inherent shortcoming of system architecture. Graph Engine and Spatial Index only communicate with Query Coordinator and they are unaware of each other. When Graph Engine performs the graph search, the spatial components of the query are not considered and vice versa. This leads to that Graph Engine and Spatial Index perform many unnecessary search.

2.3.5 Methods for Geospatial Graph Query

A system, GeoSN, is proposed to answer some geospatial graph queries in a social network [3]. The social network is simple and it only consists of users and friendship relations. The spatial information only exists for the user vertexes that each user possesses a spatial location. It defines several primitive social and geospatial queries as the fundamental components for solving more complex geospatial graph queries. $GetFriends(u)$ and $AreFriends(u_i, u_j)$ are two primitive queries that will be answered directly by the graph database component. $GetUserLocation(u)$, $RangeUsers(q, r)$ and $NearestUsers(q, k)$ are the three primitive queries supported by the spatial database component. The work focuses on three more complex geospatial graph queries, i.e., Range Friends, Nearest Friends and Nearest Star Group queries. These queries can be solved by combining the defined primitive queries. Graph database and spatial database components are utilized to answer the corresponding primitive queries. In the paper, the performance of the queries is evaluated in different storage schemes and machine architectures, including disk-based, memory-based, centralized

and distributed. However, the paper only discusses the solution for the defined queries and it does support arbitrary GraSp queries issued by users. Also, its solution is limited to the defined social network structure. There are attempting works in RDF data model [29, 54]. Encoding [29] is an encoding-based method, which extends the RDF search engine with spatial query support. Each spatial object is encoded with a unique ID according to its spatial location. When a GraSp is issued, the system will utilize the ID to prune the unpromising spatial objects without accessing the real locations of the objects. A filter phase will be applied after the pruning on ID. Spatial range and join predicates are the supported spatial query types. Similarly, [54] proposes an encoding-based method BRDF-First for spatial-temporal graph queries. The encoded ID is determined by the spatial-temporal information of the entity. However, these algorithms are designed for the RDF data engine and cannot be extended to the labeled property graph model. Moreover, none of them can support the KNN spatial predicate. In a nutshell, these approaches still belong to Isolated Architecture. Their query strategy is either GraphTraverse or SpaIndex. No awareness exists between the graph database and spatial database.

Augmented Spatial Index. Spatial indexes are augmented with different categories of information to solve different types of geospatial-related queries [60, 25, 18, 28]. In [60], two hybrid indexes are proposed for geospatial document search. One is an inverted file on top of (Keyword-First). Another is an R-Tree on top of inverted files (RTree-First). To search for the documents satisfying both spatial and textual constraints. Keyword-First index can be used to locate the satisfying documents based on search keywords first and then based on locations. RTree-First is exploited in a reversed method. However, this framework treats textural index and spatial index separately. KR*-tree [25] is another index structure extended from R-Tree for

geospatial keyword document search. It is different from RTree-First in that it allows internal tree nodes to store keywords. Similarly, IR^2-tree [18] extends R-Tree with signature files, rather than directly using the keywords. The signature files stored on a node can determine whether a keyword is contained by the documents belonging to this node. The usage of the signature files reduces the storage overhead. IR-Tree [28] is proposed to solve top-K geospatial and textual document search problem. IR-Tree is similar to IR^2-tree that it augments R-Tree with additional textual information. But IR-Tree stores document summary, including document frequency and term frequency, which are important in computing the tf-idf. However, IR-Tree is designed specifically for the scenario that the textual similarity is computed by using tf-idf. So the structure stored on top of R-Tree is an abstract representation of the textual information of the documents. However, graph data is more complex than document keyword data. Every spatial vertex can be connected to many different vertexes through different paths. So the technique cannot be applied to the geospatial graph data.

Table 1: Comparison of different approaches

Method	GeoSN	Encoding	BRDF-First	GeoExpand & Riso-Tree
General Graph Query	×	✓	✓	✓
Graph Query Language	×	✓	✓	✓
Compatibility	✓	×	×	✓
Range Query	✓	✓	✓	✓
Join Query	×	✓	×	✓
KNN Query	×	×	×	✓
Hybrid Index	×	×	×	✓
Support Updates	×	✓	✓	✓

Summary. Table 1 summarizes the characteristics of different approaches related to geospatial graph query. These characteristics include but not limit to:

1. Support general graph query;

2. Allow users to interactive through graph query language;
3. Compatible with different graph databases;
4. Support wide range of spatial predicates;
5. Support data updates.

Among all the approaches, GeoSN is designed for social graph with specific structure. It cannot support general input graph query. Only limited query types can be supported. No graph query language is supported by GeoSN either. Encoding and BRDF-First are proposed for RDF data. So they can support general RDF queries and the query language for RDF data. GeoSN is a framework compatible with different databases. The primitive queries defined in GeoSN, including $GetFriends(u)$ and $AreFriends(u_i, u_j)$, are simple and can be easily implemented in different graph databases. The operations for spatial queries are also popular query types, e.g., spatial range query, nearest neighbor query, which are supported by almost all the spatial databases. So GeoSN can facilitate a wide range of database systems. However, Encoding and BRDF-First are built on top of RDF engines. It is nontrivial to extend these two approaches to database systems other than RDF engines. Regarding the supported query types, none of the them can cover all the spatial predicates, including spatial range, join and KNN predicates, which are important and fundamental spatial query types. Moreover, as we can see, none of the approaches use the hybrid index structure. They all belong to the isolated architecture, which treats graph component and spatial component separately from both data and query perspective. When searching the graph, these approaches are not aware of the future processing of the spatial predicates, and vice versa. This makes it impossible to take advantage of the pruning power of both graph and spatial predicate. The data updates are not discussed in GeoSN. The updates for Encoding and BRDF-First are supported by

directly utilizing the RDF engine. No advanced methods are discussed either. I also discussed some augmented tree-based spatial indexes. These approaches are proposed for answering geospatial textual queries. However, graph data is more complicated than textual data in that it has complex internal structures, which makes it necessary to have more advanced and complex index structures. Although these approaches are proposed for geospatial keyword queries, they still provide a potential direction for solving spatial queries with rich semantic context by extending spatial index with augmented data structures.

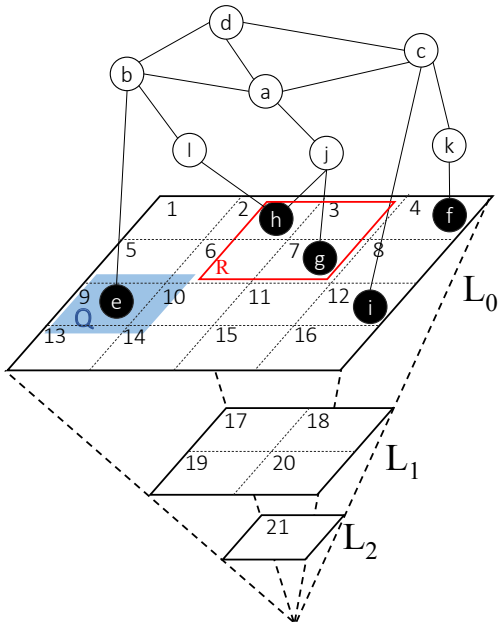
Chapter 3

SPATIAL-AWARE GRAPH SEARCH

Due to the isolation of the graph database and spatial database, the graph search is unaware of the spatial information of vertexes to be visited in the future. The graph search will access many branches that will be filtered by the spatial predicate. In this chapter, we demonstrate GeoExpand, which accelerates graph queries that incorporate spatial predicates. GeoExpand allows spatial-aware graph search by employing a light-weight technique that augments the underlying graph data with Spatial Indexing Properties.

3.1 Augmented Graph Data Structure

For GeoExpand to achieve its goal, we augment a graph structure with Spatial Indexing Properties, to form what we call SPatially-Augmented Graph (SPA-Graph). To be generic, the system stores the newly added Spatial Indexing Properties (SIP) the same way other properties are stored in a graph database system. That way these Spatial Indexing Properties can be seamlessly and effortlessly integrated into existing graph databases. The structure of a SPA-Graph is similar to that of the original graph except that each vertex $v \in V$ in a SPA-Graph $G = \{V, E\}$ stores information of the reachable spatial vertexes. The proposed approach only considers spatial vertexes that are reachable within a distance limit B, which is a user-predefined parameter. For each $k \leq B$, a data structure (SIP(v,k)) will be stored as a property of v, which depicts the spatial boundary for the spatial vertexes that can be reached from v through k



1-Hop
GeoB:
a: false
d: false
e: false
f: false
g: false
h: false
i: false

RMBR:
j: R

ReachGrid:
b: {9}
c: {12}
k: {4}
l: {2}

2-Hop
GeoB:
a: true
b: false
j: false

RMBR:
g: R
h: R

ReachGrid:
c: {4}
d: {9, 12}
e: {9}
f: {4}
k: {12}
i: {12}
l: {9}

Figure 5: SPA-Graph Overview

hops. Each vertex will store from 1-hop to B-hop spatial reachable information in a compressed bitmap form. Such spatial reachability information (aka. Spatial Indexing Properties) has three alternatives as follows:

- **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 $\text{Grid}(u) = k$. Each cell $C \in \text{ReachGrid}(v)$ contains at least one spatial vertex that is reachable from v by k hops.

Intuition. The intuition of GeoExpand operator is to leverage the Spatial Indexing Properties (SIP) in the SPA-Graph, which includes GeoB, RMBR and ReachGrid to avoid expanding those vertexes that are guaranteed not to reach spatial range predicate in a specific number of hops. That way, GeoExpand cuts down the number of traversed graph vertexes and edges and hence significantly reduces the overall latency. Furthermore, GeoB, RMBR and ReachGrid are stored as just extra properties to the corresponding vertexes in the original graph. The simplicity of GeoExpand implementation makes it a very practical solution to efficiently support graph search in existing graph database systems.

3.2 Query Processing

3.2.1 Operator Replacement

The Expand operator takes as input a query vertex. When the plan is executed, for each graph vertex mapped to such query vertex, the system will obtain all its neighboring vertexes. By replacing the Expand operator in the execution plan with GeoExpand, the graph traversal can avoid many unpromising branches. The replacement may happen on the vertexes whose distance from the range predicate is less than or equal to B. The GeoExpand operator has two more arguments compared

to EXPAND, which are the number of hops from this query vertex to the spatial vertex and the query spatial range.

3.2.2 Operator Evaluation

The EXPAND operator traverses the graph from each vertex belonging to a set of vertexes (obtained from the previous step) by following the pattern specified by the input argument. If such a pattern cannot be satisfied for the current vertex, the traversal will not continue on such vertex. Algorithm 6 shows the pseudo-code of the query algorithm. It takes the input as a graph vertex v, an expansion pattern pat, a hop distance to the spatial predicate k and a query rectangle Q. For a specific GEOEXPAND operator , k-hop GEOEXPAND information stored on v will be checked towards Q. Since the SIP has three categories, they are treated differently as follows:

Case I (GeoB): If SIP(v,k) is GeoB, the algorithm checks whether GeoB value is true or false. If it is false, it means that v cannot reach any spatial vertexes by k hops and the algorithm prunes the traversal at v. Otherwise, the algorithm calls EXPAND(v, pat) to perform the traversal on v.

Case II (RMBR): An RMBR can either overlap with Q or not. If an RMBR does overlap with Q, the algorithm again executes EXPAND(v, pat) to traverse from v with respect to pat (9). Otherwise, the algorithm will not expand v.

Case III (ReachGrid): If any grid belonging to ReachGrid overlaps with Q, it indicates that v can reach a spatial vertex located in Q through k hops. Hence, the algorithm expands v (Line 13). Otherwise, the algorithm prunes the traversal at v.

Algorithm 6 GeoExpand Evaluation Algorithm

```
1: Function GeoExpand(v, pat, k, Q)
2: switch (Spatial Indexing Properties Type)
3: case GeoB:
4:    if v.GeoB(k) = false then
5:       Terminate()
6:    end if
7: case RMBR:
8:    if Q does not overlap with v.RMBR(k) then
9:       Terminate()
10:   end if
11: case ReachGrid:
12:   if All grid cells in v.ReachGrid(k) do not overlap with Q then
13:      Terminate()
14:   end if
15: default:
16:   Invoke Expand(v, pat)
17: end switch
```

3.2.3 Running Example

Let us consider an example query on top of the graph in Figure 5 as follows:

Q1: `MATCH (user`$_1$`:`$User$`{name:Alice})`

`-[:`$Follow$`]-(user`$_2$`:`$User$`)`

`-[:`$Visit$`]-(place:`$Venue$`)`

`RETURN place`

Figure 6 shows two execution plans of the example query. The two execution plans are constructed on top of the Expand and GeoExpand respectively. The left side plan is generated by the neo4j optimizer. When such plan is executed, it will first access the vertex a. Then the Expand operator will be processed and all the neighboring vertices of a will be fetched. For each neighbor of a, its visited spatial vertices will be fetched. Then their spatial locations are checked towards the query rectangle Q. The

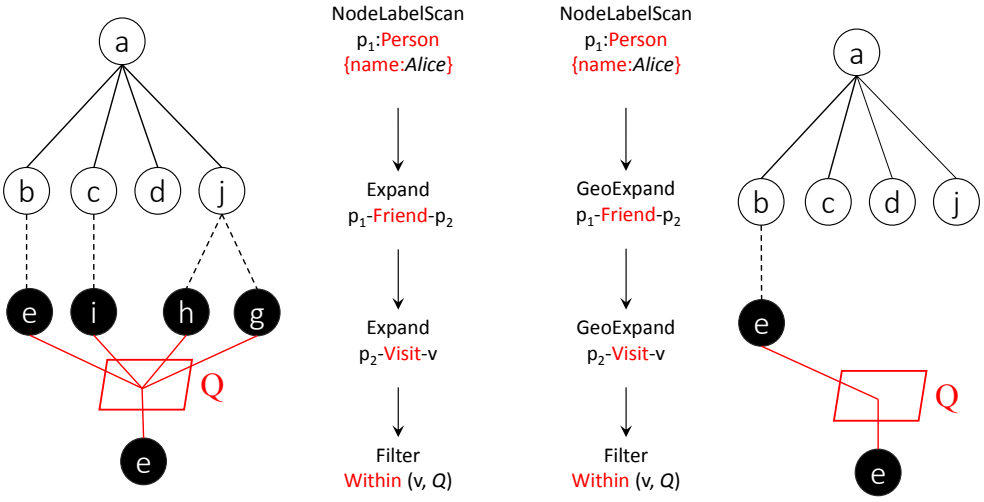


Figure 6: Processing steps by running the execution plan

difference lies in the two EXPAND operators. Recall that the first step of our approach is to replace the qualifying operators. In this case, assume that B is equal to 2. The query vertexes $user_1$ and $user_2$ are 1 and 2 hop distance from the spatial query vertex *place*. Both distances do not exceed B. So the EXPAND operators on $user_1$ and $user_2$ can be replaced by GEOEXPAND operators. When the new execution plan is processed, the first step will be the same with that in neo4j. The first GEOEXPAND operator has the format of GEOEXPAND($user_1$, $user_1 - Friend - user_2$, 2, Q). So the SIP(a, 2) will be checked towards Q. The 2-hop information stored on a is GeoB and its value is *true*. As a result, the EXPAND operator will be invoked. Then for each expanded neighbor of a, the GEOEXPAND($user_2$, $user_2 - Visit - place$, 1, Q) will be processed. These neighbors are b, c, d and j. Their SIP will be checked towards Q. Among all these vertices, c, d and j are pruned directly without being expanded because ReachGrid(c, 1) = {12} (Figure 5) does not overlap with Q. Similarly, GeoB(d, 1) = $false$ and RMBR(j, 1) do not overlap with Q. So all of them are pruned. So the total number of visited vertices is reduced by using the SPA-Graph.

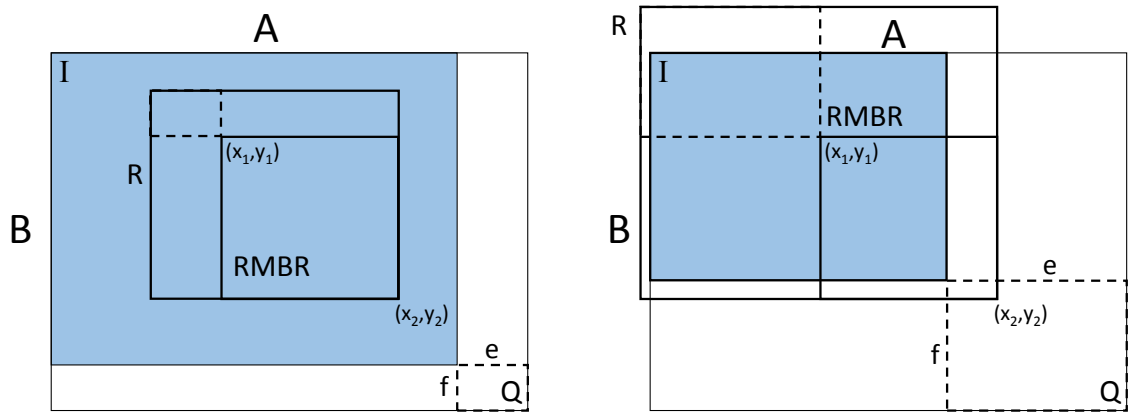


Figure 7: Pruning power of RMBR

3.2.4 SPA-Graph Analysis

This section analyzes each category of SIP from two perspectives: (1) Storage Overhead: the amount of storage overhead that each vertex type adds to the system (2) Pruning Power: the probability that the query processing algorithm terminates when a vertex of such type is visited.

GeoB. When GeoB(v, k) is accessed and its value is false, the algorithm stops the graph traversal originating at v. That is due to the fact that v cannot reach any spatial vertex in the graph given hop number k. Otherwise, the query processing algorithm continues expanding from v. As a result, pruned power of a GeoB lies in the condition that GeoB(v, k) is false. GeoB with true value does not contribute to the pruning power. Given a hop number k, the number of vertices that cannot reach spatial vertex through k hops is a certain value. Denote such number as $|V_{k,false}|$. The probability of a GeoB being false value is $\frac{|V_{k,false}|}{|V|}$. This is also the pruning power of GeoB.

RMBR. The GeoExpand operator will be stopped at v only if RMBR(v, k) does not overlap with the query region. Figure 7a shows the condition of not overlapping. The width and height of the whole space are denoted as A and B, respectively. Assume that the query rectangle can be located anywhere in the space with equal probability. We use (x_1, y_1) and (x_2, y_2) to represent the RMBR's top-left corner and lower-right point coordinates, respectively. The query rectangle is denoted as Q in the figure. Without loss of generality, we consider the top-left vertex of Q as its location. Then the domain of Q's location should be part of the total space, denoted as I (the blue shadowed area in Figure 7a). Its area is determined by size of Q. The width and height of Q are denoted as e and f respectively. So the area of I, $A_I = (A - e) \times (B - f)$. If Q does not overlap with RMBR, Q must lie outside rectangle R which forms the overlap region (drawn with solid line in Figure 7b). Area of R (denoted as A_R) is obviously determined by the RMBR location and size of Q. It can be easily observed that $A_R = (x_2 - (x_1 - e)) \times (y_2 - (y_1 - f))$. Another possible case is demonstrated in Figure 7b. In such case, if we calculate R in the same way, the range of R will exceed I. As a result, $A_R = A_I$ in this case. As we can see, the area of overlapped region is determined by the range of R and I altogether. Then we can have a general representation of the overlap area $A_{Overlap}$:

$$A_{Overlap} = (min(A-e, x_2) - max(0, x_1-e)) \times (min(B-f, y_2) - max(0, x_2-f)). \quad (3.1)$$

The *No Overlap* area is $A_I - A_{Overlap}$ and the probability of having such case is as follows:

$$P_{NoOverlap} = \frac{A_I - A_{Overlap}}{A_I} = 1 - \frac{A_{Overlap}}{A_I}. \quad (3.2)$$

When comparing RMBR with GeoB, we need to mention that RMBR(v, k) only exists when v can reach some spatial vertices through k hops. So we are comparing RMBR with GeoB with true value. Actually, GeoB with true value can be imagined as a specific RMBR whose region is the whole space. So no matter where the query rectangle is, it must overlap with such a whole-space RMBR. It means that GeoB with true value does not have any pruning power. So RMBR can outperform GeoB. But how much it can outperform is determined by $P_{NoOverlap}$. If A_{RMBR} is large, $P_{NoOverlap}$ is also small according to Equation 3.2.

When the storage overhead of an RMBR is considered, coordinates of RMBR's top-left and lower-right vertices should be stored. Thus its storage will be at least four bytes depending on the spatial data precision. It means tha the storage overhead of an RMBR is always higher than that of a GeoB.

ReachGrid. For a high resolution grid, it is of no doubt that a ReachGrid possesses high pruning power. However, it costs higher storage overhead because a ReachGrid can contain more grid cells. So blindly increasing the resolution will not definitely improve the query time performance. The reason is that the cost to access each vertex also becomes higher caused by the increasing size of ReachGrid.

When a ReachGrid is compared with an RMBR, RMBR can be seen as one simplified cell for which the resolution is equal to that of RMBR and it is the only cell. Since the area of an RMBR is larger than a grid cell most of the time, RMBR will have less pruning power. One extreme case of RMBR is that the vertex can reach only one spatial vertex. In such case, RMBR is a point whose location is the spatial vertex. Such RMBR can still be counted as a ReachGrid whose grid size $x \to 0$. According to the analysis of RMBR, it should be with higher storage overhead and more accuracy.

Actually, storing it as a ReachGrid in this case, however, will cost an integer while any RMBR requires storage for four floats or even doubles.

3.3 Initialization

This section describes how to initialize and maintain the SIP in a graph database. Before we go into the details of the initialization algorithm, we first propose a lemma concerning the reachable vertices.

Theorem 1 *Use V_u^k to denote the set of vertices that are reachable from u through k hops. If we define $V_u^0 = u$, then the k-hop reachable vertices from a given vertex u is the union set of all the $k-1$ hop reachable vertices of the neighbors of u, $V_u^k = \bigcup_{u \rightsquigarrow v} V_v^{k-1}$, $k > 0$.*

Proof 1 *Given a vertex u and any vertex v that $u \rightsquigarrow v$, for any vertex $w \in V_v^{k-1}$, w can be reachable from v through $k-1$ hops. Meanwhile, u is 1 hop from v, so w is k hops from u. Then $w \in V_u^k$.*

Then we discuss the relationship between different categories of SIP of a vertex and its neighbors.

Theorem 2 $ReachGrid(u,k) = \bigcup_{u \rightsquigarrow v} ReachGrid(v, k-1)$

Proof 2 *ReachGrid of a vertex u can be computed by such equation,* $ReachGrid(u,k) = Grid(V_u^k)$*, where Grid represents the operation of finding all the grid cells that cover vertices in V_u^k. According to Theorem 1, we have $V_u^k = \bigcup_{u \rightsquigarrow v} V_v^{k-1}$. So* $ReachGrid(u,k) = Grid(\bigcup_{u \rightsquigarrow v} V_v^{k-1}) = \bigcup_{u \rightsquigarrow v} ReachGrid(v, k-1)$.

Theorem 3 *$RMBR(u,k)$ of u is equivalent to the minimum bounding rectangle of all $RMBR(v,k-1)$ of every neighboring vertex v of v. $RMBR(u,k) = MBR(RMBR(v,k-1)), u \rightsquigarrow v$.*

Proof 3 *RMBR of a vertex u can be computed by such equation, $RMBR(u,k) = MBR(V_u^k)$, where MBR represents the operation of computing the minimum bounding rectangle that covers vertices in V_u^k. According to Theorem 1, we have $V_u^k = \bigcup_{u \rightsquigarrow v} V_v^{k-1}$. So $RMBR(u,k) = MBR(\bigcup_{u \rightsquigarrow v} V_v^{k-1}) = MBR(RMBR(v,k-1))$.*

Theorem 4 *$GeoB(u,k)$ of a vertex u is equal to the disjunction of the $GeoB(v,k-1)$ values of all u's neighbors. $GeoB(u,k) = \bigvee_{u \rightsquigarrow v} GeoB(v,k-1)$*

Proof 4 *Similarly, according to Theorem 1, this theorem can be proved in a straightforward way.*

3.3.1 Initialization

The objective of the initialization process is to calculate the SIP for all vertexes in the graph where $k \leq B$. To achieve that, the initialization algorithm runs in three main steps. In the first two steps, the algorithm builds ReachGrid, RMBR and GeoB using the 1 to B hops neighborhood of each vertex in the graph. In the last step, the algorithm keeps only one type of SIP and drops the rest.

Algorithm 7 gives the pseudo code of the initialization process. In Phase I, the algorithm will generate 1-hop SIP for each vertex u by accessing its neighbors. For each vertex, its 1-hop SIP can be computed by accumulating the spatial location attributes of its neighbors. More specifically, ReachGrid represents the union of all

Algorithm 7 SIP Initialization

```
1: Function INITIALIZE(Graph G = {V, E})
2: /* PHASE I: One-Hop Initialization */
3: for each Vertex v ∈ V do
4:    for each vertex u ∈ Adj(u) do
5:       if u ∈ V_S then
6:          ReachGrid(v, 1) ← ReachGrid(v, 1) ∪ Grid(u)
7:          RMBR(v, 1) ← MBR(RMBR(v,1), v.loc)
8:          GeoB(v, 1) ← true
9:       end if
10:   end for
11: end for
12: /* PHASE II: 2 to B-hop Initialization (2 ≤ k ≤ B) */
13: for each k ∈ [2, B] do
14:   for each Vertex v ∈ V do
15:      for each vertex u ∈ Adja(u) do
16:         ReachGrid(v, k) ← ReachGrid(v, k) ∪ ReachGrid(u, k − 1)
17:         RMBR(v, k) ← MBR(RMBR(v,k), RMBR(u,k − 1))
18:         GeoB(v, k) ← GeoB(v, k) ∪ GeoB(u, k − 1)
19:      end for
20:   end for
21: end for
22: /* PHASE III: Unused SIP removal */
23: for each vertex v ∈ V do
24:   for each k ∈ [1, B] do
25:      if ReachGrid(v, k) coverage is larger than GRRatio then
26:         Remove ReachGrid
27:      end if
28:      if RMBR(v, k) coverage is larger than RBRatio then
29:         Remove RMBR
30:      end if
31:   end for
32: end for
```

grid cells that its neighbors are located in. RMBR is the bounding box for all its neighbors. Similarly, the algorithm computes the value of GeoB.

In phase II, the algorithm computes all three types of SIP for each vertex u where $2 \leq k \leq B$. The computation is performed starting from the smallest k value (i.e., $k = 2$) in an incremental fashion. In this phase, there is no need to access the spatial location attribute of each vertex. For each vertex u, the algorithm only accesses

every neighbor vertex v and accumulates the SIP of v for $k-1$ hops. In other words, to compute ReachGrid(u, k), RMBR(u, k) and GeoB(u, k), the algorithm follows Theorem 2, 3 and 4 respectively.

After the first two phases, each vertex possesses all three types of SIP for each hop. In the third phase, the algorithm decides what type of SIP needs to be maintained or discarded. The decision is made according to two system parameters, namely `GRRatio` and `RBRatio` (defined later in this section). For each vertex at each hop, the algorithm first decides whether to remove the ReachGrid. The spatial coverage ratio is considered to be crucial in making the decision. The total spatial coverage of RMBR is evaluated by using the total number of grid cells covered by the RMBR. Hence, the coverage ratio of ReachGrid is measured by using the following equation:

$$ratio = \frac{|ReachGrid|}{\#\ of\ grid\ cells\ covered\ by\ RMBR} \tag{3.3}$$

If the spatial coverage ratio of a ReachGrid is higher than `GRRatio`, then the ReachGrid property will be dropped. Replacing ReachGrid with RMBR will lead to loss of pruning power because the grid cells that do not exist in ReachGrid are still deemed reachable using the RMBR pruning method. These grid cells are false positives. Hence, the intuition is to drop the ReachGrid only when there are very few false positive grid cells.

If the ReachGrid property is kept, then RMBR and GeoB will be dropped and no further decision needs to be made. However, if ReachGrid is dropped, the algorithm will still decide whether to replace RMBR with GeoB. Similarly, it is determined by the spatial coverage ratio of RMBR, which can be measured as follows:

$$ratio = \frac{Area(RMBR)}{Area\ of\ the\ whole\ space} \tag{3.4}$$

In case the spatial coverage ratio of the RMBR is larger than `RBRatio`, the algorithm replaces RMBR with GeoB. Otherwise, it keeps RMBR and drops GeoB. The idea is quite similar to the previous rule.

3.4 Performance Evaluation

In this section, we present a comprehensive experimental evaluation of GeoExpand's effectiveness and performance. We compare the performance of the proposed graph traversal operator GeoExpand with the ordinary graph traversal operator Expand. We evaluate the performance of the following two approaches: (1) GeoExpand: The proposed GeoExpand approach with the following default parameters: (a) `GRRatio` set to 1 which means that there is no RMBR, and (b) `RBRatio` set to 1 which indicates that RMBRs are not degraded to GeoB and there is no GeoB. So all the SIPs are ReachGrids. (2) Expand: This method performs the naive graph traversal.

Experimental Environment. The input graph and index are stored in Neo4j graph database system. The source code for evaluating query response time is implemented in Java and compiled with java-7-openjdk-amd64. All evaluation experiments are run on a computer with 3.60 GHz 4 Cores CPU, 32GB 1600 MT/s RAM running Ubuntu 16.04 Linux OS.

Parameter Setting. Table 2 shows the parameter settings. Default grid resolution is 128×128. `GRRatio` is set to 1.0 which ensures that only ReachGrids are kept in SIP. `RBRatio` is set to 1.0 as well to ensure that no RMBR is replaced by GeoB. We set B=3 for all the datasets, which means the index contains 3-hop neighborhood information.

Table 2: Parameter Setting

Parameter	Default	Varying Range
selectivity	10^{-5}	10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}
Grid Resolution	128×128	32, 64, 96, 128
`GRRatio`	1.0	0, 0.01, 0.02
`RBRatio`	1.0	0, 0.5, 1
Query Length	3	1, 2, 3

3.4.1 Evaluation Metric

I will evaluate the indexing overhead and the query performance of GeoExpand. GeoExpand is evaluated by comparing its query time with that of Expand. When a graph query is issued to a database system, an optimized query plan will be generated. Such plan consists of traversal operators and other types of non-traversal operators . In the query performance evaluation, we evaluate the graph traversal query because such an operator is fundamental in all graph queries. The spatial range predicate at the end vertex is performed from 50 different graph vertices. We run the same traversal by using GeoExpand and Expand to search all the satisfying spatial vertices. The average time is recorded respectively.

3.4.2 Datasets

Three real datasets will be used in the evaluation. Detailed information of these datasets is listed in Table 3 where d_{avg} is the average degree of the graph. Three real Geo-Spatial datasets, including Yelp, Gowalla, and Foursquare, are used in the experimental evaluation. Yelp is a real Point-of-Interest recommendation dataset

extracted from *Yelp!*[1] as introduced in the beginning. It has vertices representing users and venues. Venues can have spatial location as their attributes. There are also two types of edges in the graph. One type is friendship, which exists between users and represents the two users are friends. Another one is review type, which means a user has reviewed a venue. Gowalla dataset is extracted from SNAP datasets[2]. Gowalla has the same types of vertices and edges. The graph consists of friendship relationship between users and their check-ins information as well. Foursquare dataset, which is extracted from Foursquare application through the public API[3], also consists of social connection (users' friendship) and check-ins information [43].

Table 3: Spatial Graph Datasets (K = 10^3)

Dataset	$\|V\|$	$\|V_S\|$	$\|E\|$	d_{avg}
Yelp	629K	77K	6033K	9.59
Gowalla	1477K	1280K	5881K	3.98
Foursquare	3296K	1143K	18723K	5.68

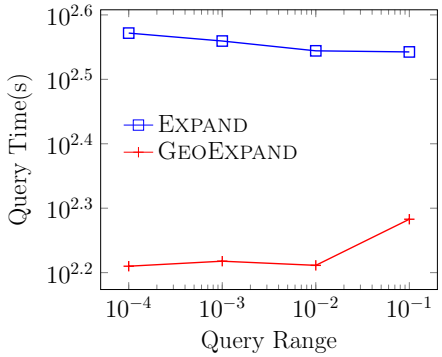
3.4.3 Query Response Time

In this section, we compare the query performance of EXPAND operator to our proposed GEOEXPAND operator. For each dataset, we change the spatial selectivity of the query rectangle and the length of the traversal. The spatial selectivity is evaluated by the ratio of spatial objects within the query rectangle to the total number of spatial

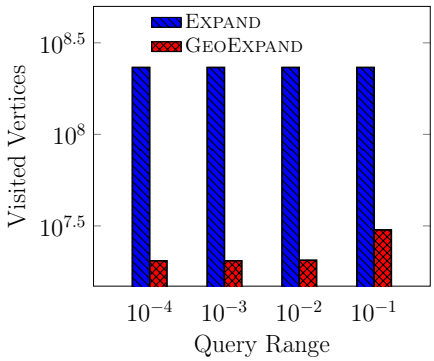
[1] https://www.yelp.com/dataset_challenge

[2] https://snap.stanford.edu/data/loc-gowalla.html

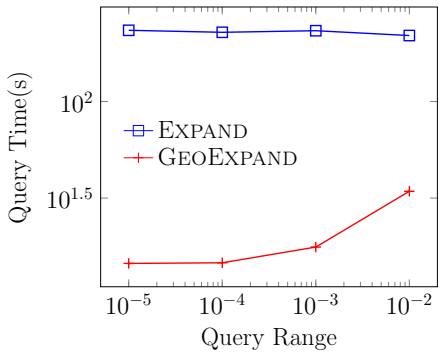
[3] https://archive.org/details/201309_foursquare_dataset_umn



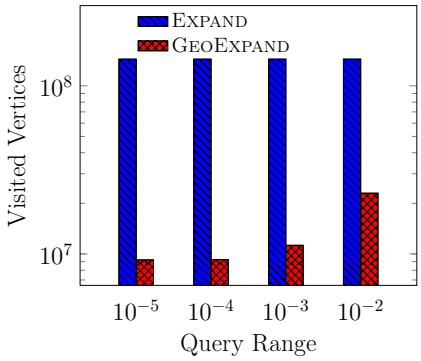
(a) Yelp query time



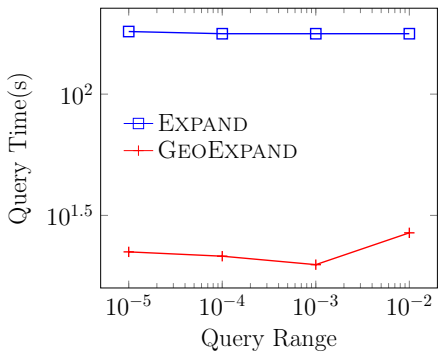
(b) Yelp visited vertices



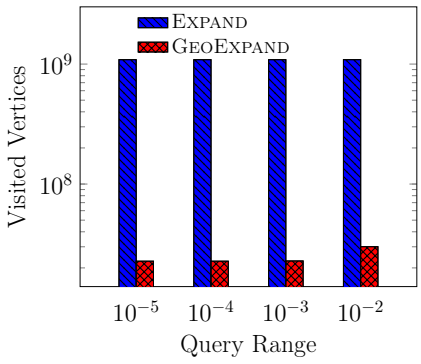
(c) Gowalla query time



(d) Gowalla visited vertices



(e) Foursquare query time



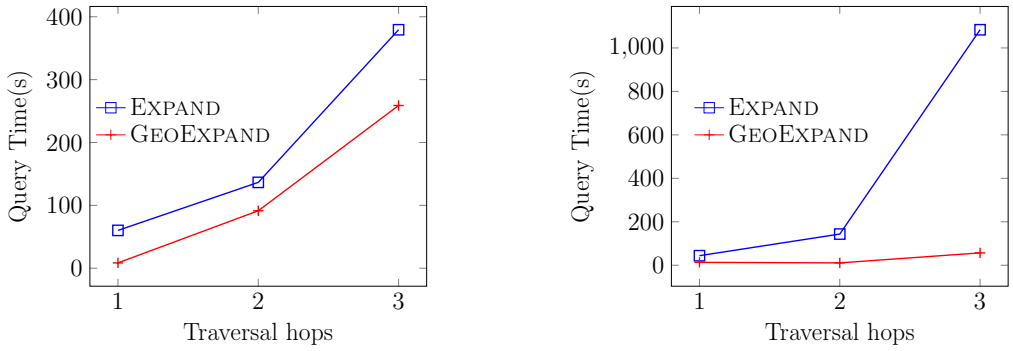
(f) Foursquare visited vertices

Figure 8: Response time and number of visited vertexes by varying the spatial query selectivity

objects. The length of the traversal varies between 1 and 3. We do not consider traversals longer than 3 hops. The reason is as follows. In a graph query, graph traversal operator will lead to a fast increasing of the result size. So, in the final query execution plan for a given graph query, long traversal will not appear for its poor efficiency. It will be replaced by several short traversal operators and join operators (such as hash join, etc.).

Spatial Selectivity. Figure 8 depicts the query response time and the number of visited vertexes for two traversal operators on all three real datasets. As is shown in Figure 8c, 8e, 8a, when the query spatial range size increases, the query time of EXPAND method does not vary because it will blindly traverse the graph until the final step and validate the spatial predicate. So it always accesses the same number of vertexes, which is shown in 8d, 8f, 8b. Using the EXPAND operator is extremely slow because it needs to expand all the vertices and check the location intensively. The figures show that it requires to visit almost billions of vertices. The query execution time of GEOEXPAND tends to increase when the query rectangle is less selective. That happens because when the query rectangle is larger (i.e., the spatial region is larger), the size of the result actually increases accordingly. Hence, it is inevitable that less vertexes can be pruned during the traversal by using SIP. However, as opposed to EXPAND, GEOEXPAND takes less traversal time in all datasets. The reason is that GEOEXPAND avoids accessing vertexes that cannot reach the target spatial region. So GEOEXPAND accesses 10x less vertexes than EXPAND.

Traversal Length. Figure 9 depicts the query response time for two different execution plans: The first plan uses the EXPAND operator and the other one employs the GEOEXPAND operators. In the experiments, we vary the length of the query traversal. The figure shows that the query response time of both methods will increase



(a) Gowalla (b) Foursquare

Figure 9: Query response time by varying the length of the traversal

rapidly when the length of the traversal increases. The traversal search space will increase exponentially with regard to increasing of number of hops in the traversal. That happens because of the inherent characteristic of the graph data structure. That means both methods will access more vertexes and edges when the traversal length is longer. However, GEOEXPAND can still outperform EXPAND for all query lengths, especially when the length is 3 which requires visiting more vertexes/edges and hence takes more time to execute.

3.4.4 Effect of GeoExpand Parameters

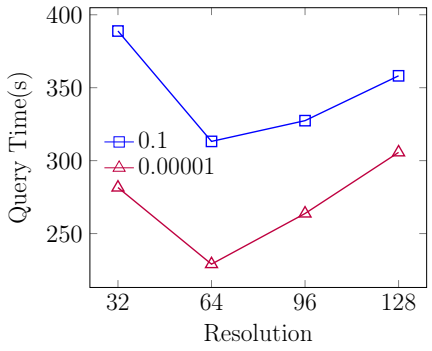
In this section, we evaluate the impact of various parameters on the storage overhead and query response time. These parameters include resolution of ReachGrid, `GRRatio` and `RBRatio`.

3.4.4.1 Grid Resolution

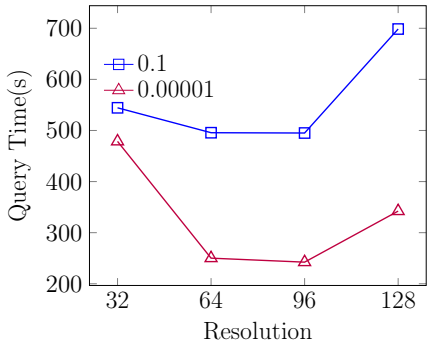
Figure 10 shows the query response time and number of visited vertexes for GeoExpand while varying the grid (i.e., ReachGrid) resolution. For each dataset, two spatial query selectivities are evaluated in the experiments. Figure 10a and 10b show that when the grid resolution increases (more grid cells), the query response time first decreases and then increases again. When the grid resolutions are 64×64 and 96×96 in Gowalla and Foursquare datasets respectively, the performance is the best. It reveals that increasing the resolution does not necessarily incur better performance in terms of query response time. When the grid resolution is higher, the SIP contains more information. Hence, it possesses stronger pruning power, which reduces the number of visited vertexes during the traversal. Figure 10c and 10d can support our explanation. However, higher grid resolution requires more pages to store the SIP. That means the cost to access the SIP also increases. And, since the query response time is influenced by both the number of visited vertexes and the cost to access each vertex, a very high grid resolution increases the overall query response time.

3.4.4.2 GRRatio

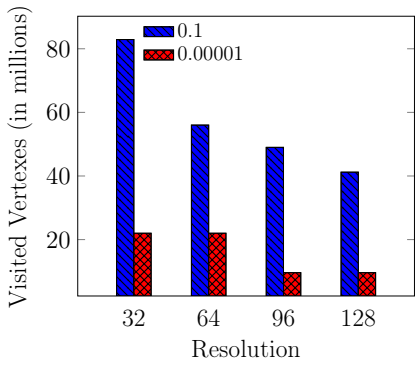
Table 4 depicts the query response time and storage overhead of the SIP with different values of the `GRRatio` parameter. The value of `GRRatio` varies from 0 to 0.02. We set `RBRatio` to 1 in order to exclude the influence of it. In consequence, SIP for all vertices are ReachGrid and RMBR. In Table 4, GeoExpand can perform better considering the query response time when `GRRatio` is larger. This is because the higher `GRRatio` is, the more ReachGrids are maintained, which are more powerful in



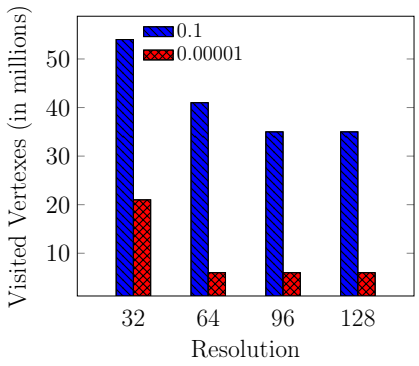
(a) Query time in Gowalla



(b) Query time in Foursquare



(c) Vertexes visited in Gowalla



(d) Vertexes visited in Foursquare

Figure 10: Response time and number of visited vertexes by varying the reach grid resolution

pruning sub-graphs than RMBRs. However, that comes at the cost of higher storage overhead according to Table 4.

3.4.4.3 MBRatio

`RBRatio` varies between 0 and 1 in our experiment. `GRRatio` is fixed to 0. Hence, there is no ReachGrid in the index. All SIP are RMBRs and GeoBs but with different proportions. Table 4 shows that increasing `RBRatio` leads to better performance in terms of query response time. Since RMBR is a more precise representation of the spatial reachability information than GeoB, more RMBRs can reduce the overall query response time. When `RBRatio` increases, fewer RMBRs will be downgraded to GeoB.

The query response time of GeoExpand with different `RBRatio` value settings proves that RMBR is more powerful than GeoB in pruning vertexes/edges during the query execution. The trade-off between on-line query and storage overhead is unavoidable, which is depicted in Table 4.

GRRatio			
Setting Value	0.01	0.02	0.03
Index Size (MB)	445	461	493
Query Time (s)	444.885	305.681	266.718
MBRatio			
Setting Value	0	0.5	1
Index Size (MB)	131	156	173
Query Time (s)	591.823	479.322	376.821

Table 4: Tuning GeoExpand' s Parameters

Chapter 4

RISO-TREE: GRAPH-AWARE SPATIAL INDEX

In this chapter, we demonstrate Riso-Tree, an index that efficiently answers GraSp. The SpaIndex strategy utilizes the spatial index to filter the spatial objects that cannot satisfy the spatial predicate in GraSp. However, such a strategy searches the spatial index regardless of the graph information on the spatial objects. Riso-Tree augments the R-Tree index with neighboring graph information. Riso-Tree facilitates the graph-aware spatial index search by pruning sub-graphs that are guaranteed not to satisfy the spatial predicate.

4.1 Index Structure

Definition 1 *Given a graph database* $G = \{V, E, \varphi, \psi\}$ *and* $\mathcal{L}$ *where* $\mathcal{L}$ *is the set of all labels in the graph, a label path* p *can be defined as* $p = \mathcal{L}^*$.

Here p is a sequence of labels generated by Kleene star operation on $\mathcal{L}$. A label path does not include real vertex in the data graph. The length of p is the number of hops in p, denoted as $|p|$. It represents a path pattern. Use $p[i]$ to denote the label at a specific position. So $p[2n]$ and $p[2n+1]$ ($n = 0, 1, ..., |p|$) represent the labels of vertexes and edges in p respectively. If a real data path p' in the graph can satisfy $\varphi(p'[2n]) = p[2n]$ and $\psi(p'[2n+1]) = p[2n+1]$ for $n = 0, 1, ...,$, which means all labels of vertexes and edges on the data path p' can be mapped correctly from p; hence, we say p' can match p. For example, in Figure 3, $p = Group - [Member] - Person - [Visit] - Place$ is a label path with length of 2. $p[0]$ is $Group$ and $p[1]$ is $Member$. The data path

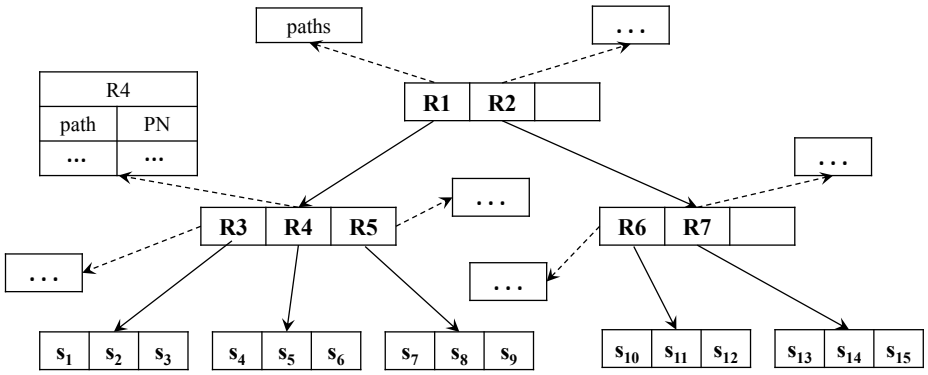


Figure 11: Riso-Tree Memory Structure

$p' = G1 - [Member] - Bob - [Visit] - s_{13}$ is a real path in the graph that can match p.

Given two vertexes u' and v', if there exists at least one path p' between them that can match p, we say that v' is reachable from u' through path p or u' can reach v' through p. It is denoted as $u' \xrightarrow{p} v'$. In the example graph, vertex *Bob* is reachable from s_{13} through a label path $p = Place - [Visit] - Person$ and it is denoted as $s_{13} \xrightarrow{p} Bob$. That said, Path Neighbor is defined as follows:

Definition 2 *Given a Riso-Tree node R and a label path p in a graph $G = \{V, G\}$, Path Neighbor of R with respect to p (PN_R^p) is the set of all vertexes in G that can be reached through the label path p from at least one spatial vertex that lies within R.*

$$PN_R^p = \bigcup_{u \sqsubset R} PN_u^p$$

Example. In Figure 3, given a label path p_1, Path Neighbor of s_{14} with respect to p_1 is $PN_{s_4}^{p_1} = \{\text{Bob, Carol}\}$. Another example, $PN_{R7}^{p_2} = \{G1\}$ since $G1$ is the only vertex connected to the spatial vertex enclosed by R7 through the label path p_2.

Basically, sub-graph entries stored in Riso-Tree nodes consist of label paths and Path Neighbors. Figure 11 shows the memory structure of the Riso-Tree. Similar

to the R-Tree, each Riso-Tree node is represented by an MBR that encloses a set of spatial objects (*i.e.,* spatial vertexes) within its boundary. In addition, each node also stores information about the sub-graph connected to the spatial objects enclosed by the node's MBR. Each leaf level node R stores a list of pairs $\langle p,\ PN_R^p \rangle$, where p is a label path and PN_R^p is the Path Neighbor for such node R with respect to p. For non-leaf nodes, only the label paths that exist from the spatial objects within R will be recorded. The length of the considered label paths (*i.e.,* number of hops) is bounded by the predefined parameter VisBound. Label Paths longer than B will not be taken into consideration by the Riso-Tree.

Non-selective Path Neighbor removal. In Riso-Tree, the Path Neighbors for a subset of Label Paths can occupy a tremendous amount of space. That happens because the selectivity of various graph labels can be skewed, and the graph connectivity to each spatial region may vary significantly. In this case, the non-selective candidate set may not lead to a considerable saving during the graph query processing step. Hence, it is deemed inefficient to maintain non-selective Path Neighbors in the index structure. Based on such an observation, we employ a strategy that reduces the overall storage overhead. The idea is to only keep the selective Path Neighbors. Since selectivity highly depends on the nature of each application, we defined a system parameter PN_{max} (supplied by the user), which determines the upper limit of Path Neighbors for each tree node. In other words, a Path Neighbor is considered selective and maintained only if its size does not exceed PN_{max}. For non-selective Path Neighbors, only their Label Paths are stored. In practice, an empty list will be stored with this Label Path. By doing so, the removed Path Neighbors are differentiated from the non-exist Path Neighbors. Non-exist Path Neighbor does not have Label Path while removed Path Neighbors have the Label Path but with an empty list.

R1	R3	R4	R5
Place - [*Visit*] - Person	Larry	Joe	Jane
Place - [*Visit*] - Person - [*Friend*] - Person	Dan, Jane	Alan, Bob	Larry, Carol, Kate
Place - [*Visit*] - Person - [*Visit*] - Place	s_1, s_2, s_3	s_4, s_5, s_6	s_7, s_8, s_9
Place - [*Visit*] - Person - [*Member*] - Group	G3	-	-

R2	R6	R7
Place - [*Visit*] - Person	Carol, Kate	Carol, Kate, Bob
Place - [*Visit*] - Person - [*Friend*] - Person	Jane	Joe, Kate, Jane, Carol
Place - [*Visit*] - Person - [*Visit*] - Place	s_{10}, s_{11}, s_{14}, s_{15}	{}
Place - [*Visit*] - Person - [*Member*] - Group	G1	G1

Figure 12: Riso-Tree Path Neighbor Information Example

Figure 12 depicts an example of a sub-graph information for Riso-Tree structure, which corresponds to the graph presented in Figure 3. In this case, VisBound is set to 2, which means only paths with the length shorter than or equal to 2 are considered. PN_{max} is set to 4. Non-leaf nodes, *e.g.,* R1 and R2, only store existing label paths. All their paths are listed in the corresponding tables. Both of them have 4 Label Paths. On the other hand, leaf nodes store Path Neighbors. For instance, the sub-graph information on R4 is organized by three Label Paths. Path Neighbor with respect to each Label Path is listed in the table as well. For R4 and R5, the dash symbol means they do not have that Label Path. So no information is stored. Different from R4 and R5, R7 has an empty Path Neighbor because it is $\{s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$ and its size exceeds PN_{max}.

Algorithm 8 Construction through insertion

```
1: INSERT(Node root, Vertex s)
2: Leaf Node R ← CHOOSELEAF(root, s)
3: Add s into R
4: ADJUSTTREE(R, true, true)
5: if R is full then
6:    NODESPLIT(R)
7: end if
```

4.1.1 Initialization

The first step of the initialization process is to lay out the Riso-Tree skeleton. Similar to the R-Tree, each spatial object will be inserted into a leaf node until all spatial objects are processed. Algorithm 8 shows the pseudo-code of the insertion operation. The algorithm calls CHOOSELEAF (Line 1) to determine the target leaf node R and insert s to R (Line 3). Then ADJUSTTREE is exploited to adjust MBR and Path Neighbor if necessary (Line 4). When one node is full, it is split into two new nodes using NODESPLIT (Line 6). In the following, we demonstrate each component in detail.

Choose Leaf. In the R-Tree construction algorithm, it chooses the target leaf for insertion which minimizes spatial area enlargement. By doing so, the dead space in each node is minimized. In Figure 13, R1 and R2 are two leaf nodes enclose $\{s_1, s_2, s_3\}$ and $\{s_4, s_5, s_6\}$, respectively. A spatial vertex s_7 is inserted and the algorithm needs to decide which leaf node is the target for the insertion. The spatial area enlargements are 0.015 and 0.01 respectively. s_7 will be inserted into R2 because the area enlargement of R1 is smaller in this case. As opposed to the R-Tree, however, spatial objects indexed by the Riso-Tree are part of the graph. Hence, the construction of Riso-Tree takes into account how such objects are connected to other vertexes

in the graph as well. A new distance function integrating both spatial and graph information is proposed for Riso-Tree construction. Similar to spatial area expansion, we define the graph Path Neighbor expansion (GPE) as follows:

$$GPE(R, s) = \frac{\sum_{|p|\leq 1} |PN_s^p - PN_R^p|}{|V|} \tag{4.1}$$

where R is a tree node in Riso-Tree and s is the spatial object to be inserted. GPE is defined on the Path Neighbor difference (expansion) if inserting s into R. The definition considers the paths whose length does not exceed 1. The reason is the size of the Path Neighbors can be huge if its length is larger than 1, which requires high computation cost. GPE is normalized by $|V|$, the total number of vertexes in the graph.

Then we define the normalized spatial area enlargement as follows:

$$SE = \frac{AreaEnlargement}{A_S} \tag{4.2}$$

where A_S is the area of region that covers all the spatial vertexes in the graph. Based on SE and GPE, we define a holistic enlargement SGE considering both spatial and graph aspects:

$$SGE = \alpha SE + (1 - \alpha)GPE \tag{4.3}$$

Here α is a user-defined parameter which controls the impact of SE and GPE.

Algorithm 9 depicts the pseudo-code of ChooseLeaf function. It returns the target leaf node in the Riso-Tree rooted at *root* for the insertion of a spatial vertex s. ChooseLeaf is called recursively to select the insertion location from the root node to the leaf node level (Line 6). At each tree level, the algorithm picks up the target node among all the children of the current root node. The child node with the smallest SGE is selected as the target node (Line 5).

Algorithm 9 Choose the leaf node for the insertion

```
1: ChooseLeaf(Node root, Vertex s)
2: if root is leaf then
3:     return root
4: end if
5: target ← argmin_R(SGD(R, s)), R ∈ root.children
6: return ChooseLeaf(target, s)
```

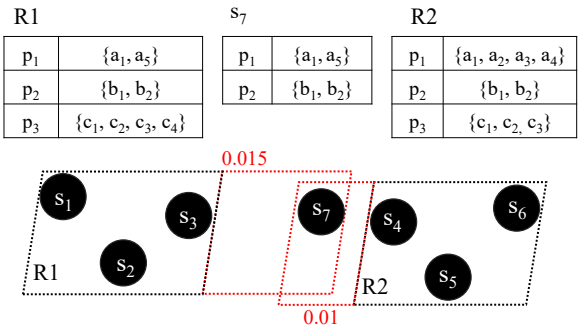


Figure 13: Insert a spatial vertex

Figure 13 depicts a real example of inserting a spatial vertex s_7 into the Riso-Tree. Assume the Path Neighbor of R1, s_7 and R2 are as shown in the table. R1 and R2 are two target Riso-Tree nodes for the insertion. SE(R1, s_7) = 0.015 and SE(R2, s_7) = 0.01 respectively. Considering only the spatial proximity in R-Tree, s_7 will be inserted into R2 because the area enlargement of R1 is smaller. Assume that there are 50 vertexes in the graph ($|V| = 50$) and $\alpha = 0.5$. If we consider graph proximity in Riso-Tree, GPE(R1, s_7) = 0 and GPE(R2, s_7) = $1/50 = 0.02$. So SGE(R1, s_7) = $0.5 \times 0.015 + 0.5 \times 0 = 0.0075$, SGE(R2, s_7) = $0.5 \times 0.01 + 0.5 \times 0.02 = 0.015$. In this case, s_7 is inserted into R1. So the target node is different from only considering SE in the previous example.

The motivation behind using GPE as the new metric is to keep the Path Neighbors as compact as possible, which in turn reduces the storage overhead. By inserting s_7 into $R2$ will make $PN^{p_1}_{R2}$ change to $\{a_1, a_2, a_3, a_4, a_5\}$. But inserting s_7 into $R1$ will

Algorithm 10 Adjust MBR and Path Neighbor of a Node

```
1: AdjustTree(Node R, Boolean MBRCheck, Boolean PNCheck)
2: if MBRCheck = true then
3:    Adjust R.MBR
4:    if MBR remains the same after adjustment then
5:       MBRCheck←false
6:    end if
7: end if
8: if PNCheck=true then
9:    Adjust PN_R
10:   if PN_R remains the same after adjustment then
11:      PNCheck←false
12:   end if
13: end if
14: if MBRCheck ∨ PNCheck then
15:   AdjustTree(R.parent, MBRCheck, PNCheck)
16: end if
```

not enlarge any Path Neighbor of $R1$. If s_7 is inserted into $R2$, a_5 will be stored in both nodes $R1$ and $R2$.

Adjust Tree Node. After a spatial vertex is inserted into the proper leaf node, the algorithm will update the MBR and Path Neighbor of the leaf node accordingly. Algorithm 10 shows the pseudo-code of the adjustment. The algorithm expands the MBR and Path Neighbor to cover the inserted spatial vertex. *MBRCheck* and *PNCheck* are the flags to indicate whether MBR and Path Neighbor are modified or not. If anyone is not modified, the algorithm set the corresponding flag to false (Line 5 and 11). If either flag is true, the algorithm updates the parent node recursively (Line 15). Otherwise, the algorithm terminates the adjustment.

Node Splitting: Algorithm 11 shows the pseudo-code of full node split. First, two objects within the full node will be selected as seeds. The two seeds are the first object of each new node. Then the remaining objects will be reassigned to each new node. In an R-Tree, both pick-up and re-assignment only consider spatial proximity.

Algorithm 11 Split a full node

```
1: NodeSplit(Node R)
2: O ← R.children
3: (o1, o2) ← argmax_(o1,o2) SGE(o1, o2), o1, o2 ∈ O
4: Initialize two nodes, r1 ← {o1}, r2 ← {o2}
5: O ← O − {o1, o2}
6: while O ≠ ∅ do
7:    o ← argmax_r |SGE(r1, o) − SGE(r2, o)|, o ∈ O
8:    Insert o into the node with smaller SGE
9:    AdjustTree(r, false, false)
10:   O ← O − {o}
11: end while
```

In the Riso-Tree full node splitting, the algorithm again considers both spatial and graph proximity. Equation 4.3 is exploited to evaluate the combined proximity. The pair of spatial objects with the maximum SGE will be selected as the new seeds (Line 3). Each seed will form a new node, denoted as r_1 and r_2 respectively. Each time, one remaining object with the largest SGE difference between r_1 and r_2 is processed (Line 7) and inserted into the node with smaller SGE (Line 8). Then AdjustTree is called to update the MBR and Path Neighbors (Line 9). The procedure will continue until all spatial objects are correctly assigned to the new nodes.

Path Neighbor Computation and Removal. After the tree skeleton is constructed, the algorithm computes the sub-graph entries for each node in the Riso-Tree. The sub-graph entries are constructed in a bottom-up manner. Starting from each leaf node in the Riso-Tree, the algorithm computes its Path Neighbors according to the following equation:

$$PN_R^p = \begin{cases} \bigcup\limits_{u \sqsubset R} v \in Adj(u), \varphi(v) = p[2], \psi(u,v) = p[1] & |p| = 1 \\ \bigcup\limits_{u \in PN_R^{p'}} v \in Adj(u), \varphi(v) = b, \psi(u,v) = a \qquad p = p' - [a] - b. \end{cases}$$

The algorithm first calculates the node's 1-hop Path Neighbors ($|p| = 1$) based on adjacent neighbors of the spatial objects enclosed by this Riso-Tree node. The returned

Path Neighbors are divided into different groups according to the different Label Paths. After the 1-hop Path Neighbors are constructed, the algorithm computes the 2, 3, ..., B-hop Path Neighbors one after another. Non-leaf nodes are processed from lower to higher levels in the Riso-Tree. Existing Label Paths for each non-leaf node will be accumulated from all its children nodes. During the Path Neighbor construction phase, the size of each PN is compared with PN_{max}. If $|PN| \leq PN_{max}$, then PN is maintained for this node. Otherwise, the algorithm removes it. A removed Path Neighbor is stored as an empty list. When computing k-hop Path Neighbors based on $k-1$-hop Path Neighbors, the algorithm may encounter an empty list in $k-1$-hop Path Neighbors. If this happens, the initialization algorithm will skip the empty list. So all the k-hop Path Neighbors that have the empty $k-1$-hop Path Neighbor as the prefix will not exist.

Example. Given the Riso-Tree in Figure 11 ($B = 2$, $PN_{max} = 4$), the algorithm computes 1-hop Path Neighbor for all leaf nodes, including $R3$, $R4$, $R5$, $R6$, $R7$. For instance, Path Neighbor of R7 with respect to the label path p_1 $PN^{p}_{R7} = \bigcup_{s \sqsubset R7} \{u' | s \xrightarrow{p_1} u'\}) = \{$Bob, Carol, Kate$\}$. At this hop, no Path Neighbor is removed because none of them exceeds PN_{max}. The algorithm then computes 2-hop Path Neighbor. For instance, Path Neighbor of R7 with respect to a label path p_2 $PN^{p_2}_{R7} = \bigcup_{v \in PN^{p_1}_{R7}} \{u' | v \rightarrow u', \varphi(u') =$`Group`$, \psi(v, u') =$`Member`$\} = \{$G1$\}$. Again, the algorithm checks the size of each Path Neighbor and $PN^{p_3}_{R7}$ is set with an empty list.

Riso-Tree Overhead Analysis. Two components contribute to the Riso-Tree storage cost (S_{riso}): (i) The size of the tree structure denoted as S_{tree} (ii) Path Neighbor information denoted as S_{PN}. Let the fanout of the tree be f and total number of spatial vertexes be $|V_S|$, then the number of levels of Riso-Tree is $log_f|V_S|$. So $S_{tree} = \sum_{i=0}^{log_f|V_S|} f^i$. S_{PN} consists of Path Neighbors and all the existing label

paths. Path Neighbors are computed based on k-hop neighbors for all spatial vertexes in the network for $0 < k \leq B$. Denote the average degree of the graph as d. The number of vertexes reachable by all spatial vertexes at specific hop k is $d^k|V_S|$. Denote the average number of labels each vertex possesses as l. The size of Path Neighbors is $\sum_{k=1}^{B} d^k l|V_S|$. For each node in the Riso-Tree, the maximum number of label paths it can have is determined by the total number of labels in the graph. Assume that the total number of labels is denoted by $|\mathcal{L}|$. Then, each Riso-Tree node can possess at most $\sum_{k=1}^{B} |\mathcal{L}|^k$ paths. In total, the number of Riso-Tree nodes is $\sum_{i=1}^{log_f|V_S|} f^i$. So the size of label paths is at most $(\sum_{k=1}^{B} |\mathcal{L}|^k)(\sum_{i=1}^{log_f|V_S|} f^i)$. To sum up, S_{riso} is as follows:

$$\begin{aligned} S_{riso} &= \sum_{i=0}^{log_f|V_S|} f^i + \sum_{k=1}^{B} d^k l|V_S| + (\sum_{k=1}^{B} |\mathcal{L}|^k)(\sum_{i=1}^{log_f|V_S|} f^i) \\ &\approx (d^B l + \frac{|\mathcal{L}|^B}{f} + 1)|V_S| \end{aligned}$$

The time for building the index (T_{riso}) also consists of two parts, the tree skeleton, and the Path Neighbor information, denoted as T_{tree} and T_{PN} respectively. The tree is constructed by keeping inserting new spatial objects. Each insertion needs to find the leaf node to store the inserted spatial object. It takes $log_f|V_S|$ time. As a result, T_{tree} = $|V_S|log_f|V_S|$. The construction time for Path Neighbor information also consists of two parts: (1) computing Path Neighbors for leaf nodes and (2) accumulating label paths for non-leaf nodes. The construction needs to access both Riso-Tree nodes and graph vertices. Similarly, the time to compute PN is $\sum_{k=1}^{B} d^k l|V_S|$ with respect to number of vertices being accessed. To perform such computation, the algorithm also needs to iterate through all the leaf nodes of Riso-Tree, which is $\frac{|V_S|}{f}$.

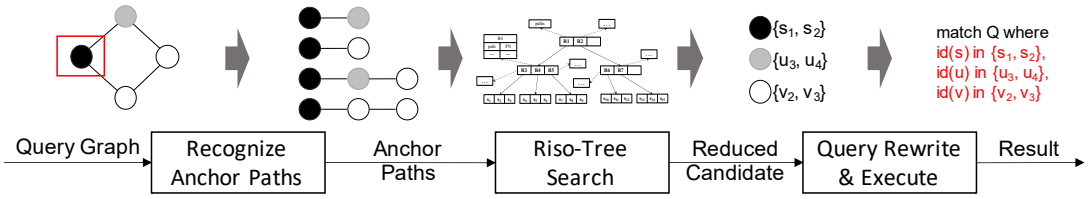


Figure 14: GraSp Query Solution Framework

Then accumulating label paths needs to access the whole Riso-Tree once, which takes $\sum_{i=0}^{log_f|V_S|} f^i$. In total, T_{riso} can be represented as follows:

$$\begin{aligned} T_{riso} &= |V_S|log_f|V_S| + \sum_{k=1}^{B} d^k l|V_S| + \frac{|V_S|}{f} + \sum_{i=0}^{log_f|V_S|} f^i \\ &\approx (log_f|V_S| + d^B l + \frac{2}{f})|V_S| \end{aligned}$$

4.2 Query Processing

GraSp queries, including GraSp-Range, GraSp-KNN and GraSp-Join can be solved by using the framework illustrated in Figure 14. It consists of three main steps:

1. Step I: Recognize Anchor Paths;
2. Step II: Riso-Tree Search;
3. Step III: Query Rewrite and Execute.

First, the algorithm recognizes anchor paths (RecognizePaths), which extracts the information of the query graph. All paths connected to the spatial query vertex are extracted and used as constraints for Riso-Tree search. Then, Riso-Tree is searched by following the corresponding algorithm for different types of GraSp queries. The

search space is pruned to reduced candidate sets. In the final step, the input graph query is rewritten on top of the reduced candidate sets. The new queries are executed by GDBMS to produce the query result. Step I is shared by all categories of GraSp queries. Step II and III are designed differently for each category. In the rest of this section, we explain details of RecognizePaths and a key component of searching Riso-Tree called CheckPaths which works for all types of GraSp queries. Details of Step II and III will be discussed in the solution of each type of GraSp query.

4.2.1 Recognize Anchor Paths

In the first step, the algorithm finds all the possible label paths in the query graph that are connected to the spatial query vertex belonging to a spatial predicate. We call such label path APath. It is formally defined as follows:

Definition 3 *An APath is a label path p that starts from the spatial query vertex with a spatial predicate and satisfies* $|p| \leq B$*, where* B *is the maximum length of label paths in the Riso-Tree. Such a spatial query vertex is called anchor vertex and the end vertex on the other side of the APath is called kite vertex.*

For the graph query given in Figure 3, a spatial range predicate is specified on the query vertex of type `Place`. In that case, `Place` is the anchor vertex for such a spatial predicate. RecognizePaths detects two APaths, $p_1 = Place - [Visit] - Person$ and $p_2 = Place - [Visit] - Person - [Member] - Group$. The kite vertexes for the two APaths are `Person` and `Group` respectively. Basically, the set of APaths of a given anchor vertex depicts the graph path information of the B-hop neighbors of this anchor vertex.

4.2.2 Check Paths In Riso-Tree

The APaths are generated in Step I can be utilized in search Riso-Tree for different GraSp query purposes. A key difference between Riso-Tree and existing spatial indexes is that it is facilitated with rich graph information. A component function called Check Paths (CHECKPATHS) will utilize the APaths to search Riso-Tree. It determines whether the current Riso-Tree node can satisfy the constraint of a given set of APaths.

Algorithm 12 demonstrates the pseudo-code of CHECKPATHS function. CHECKPATHS takes input as a set of APaths and a Riso-Tree node and determines whether this node contains all the APaths. If there is any path $p \in APaths$ that does not exist in $R.paths$, the algorithm will return false, which means that there is no spatial vertex within R that is connected to p. The algorithm will return true if and only if R owns all the label paths in APaths. Because some Path Neighbors are removed Riso-Tree, the existence validation for each Label Path needs to not only check the path itself (Line 3), but also its the prefix paths. If there exists a prefix path p' that $|PN_R^{p'}| = 0$ which means it is removed, p will also be treated as existing (line 6). The algorithm is correct because it only allows false positives. No qualifying Riso-Tree nodes are pruned wrongly.

For instance in Figure 3, if the leaf node R6 is checked whether it can satisfy the graph constraint with the APaths $\{p_1, p_2\}$, the function will verify $|PN_{R6}^{p_1}|$ and $|PN_{R6}^{p_2}|$. According to the stored Path Neighbor information shown in Figure 12, both p_1 and p_2 are included in R6. Then CHECKPATHS will return true in this case. If R4 is checked towards these two APaths, then it will return false because p_2 is not included by R4.

Algorithm 12 Check Paths Algorithm

```
1: CheckPaths(Anchor Paths APaths, Node R)
2: for each path p in APaths do
3:    if path ∈ R.paths then
4:       continue
5:    end if
6:    if exists a prefix path p' of p that |PN_R^{p'}| = 0 then
7:       continue
8:    end if
9:
10:   return false
11: end for
12:
13: return true
```

4.2.3 GraSp-Range

GraSp-Range query is a graph query with at least one spatial range predicate. Each spatial range predicate is represented as $\langle s, Q \rangle$, where s is a query vertex in the query graph and Q is a spatial rectangle. Recall that the main framework consists of three main steps, RecognizePaths, Riso-Tree search, and Query Rewrite and Execute. We will focus on Riso-Tree search and Query Rewrite and Execute in this section.

Once all the APaths are obtained, the algorithm searches the Riso-Tree to generate the reduced candidate sets for the input anchor vertex and its kite vertexes. That way, Riso-Tree can prune: (a) the spatial search space based on sub-graph information stored in each Riso-Tree node, and (b) the graph search space based on spatial indexing entries in the Riso-Tree. For each spatial predicate $\langle s, Q \rangle$, the algorithm searches the Riso-Tree starting from the root node. For each visited node, the algorithm prunes children nodes whose MBRs are not overlapping with Q. The algorithm also prunes the Riso-Tree nodes that do not contain any spatial vertex that can be matched to

the query graph. CheckPaths checks the graph constraint to prune unpromising branches in the Riso-Tree.

When the search reaches the leaf level, the algorithm takes advantage of Path Neighbor stored on this level to generate the reduced candidate sets for the anchor vertex and its kite vertexes. The reduced candidate sets for kite vertexes and anchor vertex are generated in different manners. For a kite vertex, its APath is a constraint on it because this kite vertex needs to be connected to the corresponding query range Q through its APath. The reduced candidate set for a kite vertex with respect to such APath can be computed by using the following equation:

$$PN_Q^p = \bigcup PN_{R_i}^p, R_i \cap Q \neq \emptyset \wedge \textsc{CheckPaths}(R_i, APaths) \tag{4.4}$$

where R_i is a leaf node of Riso-Tree that overlaps with Q and satisfies the graph constraint. Path Neighbors for Q with respect to p should be the union of Path Neighbors of these leaf nodes. Because of the existence of removed Path Neighbors, the equation needs some adjustment. If any $PN_{R_i}^p$ is an empty list, PN_Q^p will be set as the full set of the vertexes that can match kite vertex. It means that the candidate set of the kite vertex is not reduced. Recall that these removed Path Neighbors are not selective, so it does not influence the query performance much.

Example. Consider one of the APaths p_1 for the query in Figure 3 as an example. The leaf nodes that can satisfy both the spatial and graph constraints are $R6$ and $R7$. The reduced candidate set $PN_Q^{p_1} = PN_{R6}^{p_1} \cup PN_{R7}^{p_1}$ = {Bob, Carol, Kate} $\cup$ {Carol, Kate} = {Bob, Carol, Kate}. So the candidate set for `Person` is reduced from 8 to 3.

To obtain the reduced candidate set of an anchor vertex, the algorithm iterates through all the spatial objects that lie within the extents of each leaf node that survives both spatial and graph constraints. For each spatial vertex, the algorithm checks whether it is located within the query rectangle. Since branches that cannot

Algorithm 13 GraSp-Range Riso-Tree Search

```
1: Function SearchTree(Node root, Predicate ⟨s, Q⟩)
2: APaths ← RecognizePaths(⟨s, Q⟩)
3: VisitTreeNode(root, ⟨s, Q⟩, APaths)
4: return {⟨u, p, PN^p_Q⟩}

1: Function VisitTreeNode(node, ⟨s, Q⟩, APaths)
2: if node is a non-leaf node then
3:    for each ⟨R, child⟩ ∈ node do
4:       if R ∩ Q ≠ ∅ and CheckPaths(APaths, child) then
5:          VisitTreeNode(child, ⟨s, Q⟩, APaths)
6:       end if
7:    end for
8: end if
9: if node a leaf node then
10:   for each p ∈ APaths do
11:      PN^p_Q ← PN^p_Q ∪ PN^p_R
12:      for each spatial object o ∈ node do
13:         if o ⊏ Q then
14:            PN^s_Q ← PN^s_Q ∪ {o}
15:         end if
16:      end for
17:   end for
18: end if
```

satisfy the graph constraint have been pruned, the size of the candidate set will be smaller than using the traditional range query solution. For the query in Figure 3, the candidate set for s is $\{s_5, s_8, s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$ using the traditional approach. With the help of Riso-Tree, the search can prune R4 and R5 because they do not contain all the APaths. So s_5 and s_8 will not be in the candidate set for `Place`. The final candidate set will be $\{s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$.

Algorithm 13 shows the pseudo code of Riso-Tree search. The search algorithm takes as input the root node of the Riso-Tree *root* and a spatial range predicate $\langle s, Q\rangle$. The output of the algorithm is the reduced candidate sets for the kite vertexes and anchor vertex. The algorithm first detects all the APaths for the given anchor vertex

(Line 2). Then these APaths are used in the function VisitTreeNode (Line 3). Inside the VisitTreeNode function, the algorithm follows a recursive way to search the Riso-Tree. Riso-Tree nodes that cannot satisfy the spatial constraint or cannot pass the CheckPaths validation will be skipped during the search (Line 4). The candidate sets for kite vertexes and the anchor vertex are constructed at the leaf level (Line 14).

Notice that each candidate set is with respect to each APath. A kite vertex or anchor vertex can exist in different APaths. This can happen when a query vertex is connected to one range predicate through different paths or to different range predicates. As a result, before the Query Rewrite and Execute, the same query vertex belonging to different APaths requires a merging step. A set intersection operation will be executed on the candidate sets belonging to the same query vertex. This step will further reduce the candidate set size (search space). Based on the reduced candidate sets of kite vertexes and anchor vertexes, the algorithm performs the Query Rewrite and Execute step. The original query will be kept but augmented with more components generated from the new candidate sets.

Example. Consider the GraSp-Range in Figure 3 as a running example. There is only one spatial predicate $\langle Place, Q\rangle$ in the query. All the APaths are $\{p_1, p_2\}$. There are two kite vertexes, `Person` and `Group` and one anchor vertex, `Place`. The Riso-Tree search will start from the root node. Both $R1$ and $R2$ overlap with Q and include both APaths p_1 and p_2. Among all the children of R1, R3 cannot satisfy the spatial constraint. R4 and R5 cannot satisfy the graph constraint. So they are all pruned. For node R2, both its children R6 and R7 overlap with Q and pass the CheckPaths validation. Then the reduced candidate sets will be constructed based on leaf nodes $R6$ and $R7$. The new candidate sets for `Person`, `Group` and `Place` are constructed

as {G1}, {Bob, Carol, Kate} and {s_{10}, s_{11}, s_{13}, s_{14}, s_{15}} respectively. By calling the SearchTree algorithm, the reduced candidate sets for `Group`, `Person`, and `Place` are constructed as $\{G1\}$, $\{Bob, Carol, Kate\}$ and $\{s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$ respectively. Since each kite vertex and anchor vertex belong to only one APath, the candidate set merge step will do no computation. Finally, the algorithm rewrites the query based on the new candidate sets. The new query will be rewritten as follows:

Query 1 MATCH (G:Group)-[:Member]-(per:Person)-[:Visit]-(pl:Place) WHERE Within(pl, Q) and g in {G1} and per in {Bob, Carol, Kate} and pl in $\{s_{10}, s_{11}, s_{13}, s_{14}, s_{15}\}$.

When comparing the query with the one in Figure 3, Query 1 keeps the query graph main body. Besides that, more predicates regarding g, pl and s are added into the WHERE clause. These new predicates work for reducing the candidate sets of query vertexes in the query graph pattern. Finally, such an query will be executed by the GDBMS.

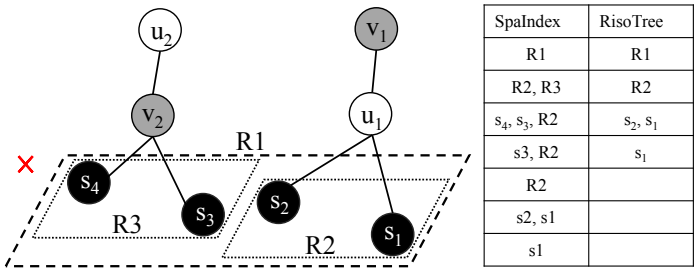
4.2.4 GraSp-KNN

GraSp-KNN integrates a KNN predicate into a graph query. The format of a KNN predicate is $\langle s, loc, K \rangle$, where s is the spatial query vertex in the graph and loc is the input searched location. The query will return top-K closest spatial vertexes from location loc that can satisfy the query graph. The SpaIndex strategy can be utilized to execute a GraSp-KNN query by extending the classic KNN query processing algorithm, as follows: It first finds the next closest spatial object by using the incremental KNN algorithm. For each spatial object, the algorithm then checks whether it can satisfy the query graph. If it can satisfy the graph constraint, add it to the result set. Otherwise,

Algorithm 14 Algorithm of GraSp-KNN

```
1: Function GraSp-KNN(G_q, SP_KNN = ⟨s, loc, K⟩)
2: result ← ∅
3: APaths ← RecognizePaths(G_q, SP_KNN)
4: q ←NewPriorityQueue()
5: if CheckPaths(root_s, APaths) = false then
6:    return result
7: else
8:    Enqueue(q, root_s, 0)
9: end if
10: while |result| ≤ K and q ≠ ∅ do
11:    e ← Dequeue(q)
12:    if e is a non-leaf node then
13:       for each child of e do
14:          if CheckPaths(child, APaths) = true then
15:             Enqueue(q, child, Dist(loc, child))
16:          end if
17:       end for
18:    else if e is a leaf node then
19:       for each child of e do
20:          Enqueue(q, child, Dist(loc, child))
21:       end for
22:    else
23:       Query Rewrite and Execute on e
24:       if At least one matching is found then
25:          result.Add(e)
26:       end if
27:    end if
28: end while
```

fetch the next closest spatial vertex and perform the same operation until K satisfying objects are found. Such an approach is still inefficient since so many spatial objects that cannot satisfy the graph constraint will still be accessed. Furthermore, the validation of the graph query constraint(s) is very time-consuming. To remedy that, our proposed GraSp-KNN query processing algorithm exploits the Riso-Tree to prune unnecessary tree branches as early as possible to reduce the amount of graph constraint validation.



SpaIndex	RisoTree
R1	R1
R2, R3	R2
s_4, s_3, R2	s_2, s_1
s3, R2	s_1
R2	
s2, s1	
s1	

Figure 15: Query Example for GraSp-KNN

Algorithm 14 shows the pseudo code of the GraSp-KNN query processing algorithm. The algorithm takes as input a query graph G_q and a KNN predicate $\langle s, loc, K \rangle$. The first step is still to recognize the APaths with respect to the anchor vertex s (Line 3). A priority queue q is used and the sorting key is the distance between an object (either a Riso-Tree node or a spatial object) and the query location loc. The root node $root_s$ of the Riso-Tree is accessed first. If all the APaths are included in the paths connected to $root_s$, then $root_s$ is enqueued to q (Line 8). Each time, the head element e will be dequeued (Line 11). Basically, e can be a non-leaf node, a leaf node or a spatial object. If e is a non-leaf node, for each child of e, it is checked towards all the APaths by using CheckPaths function. If the function returns true, such child node will be enqueued along with its distance to loc (Line 15). If e is a leaf node, each child of e which is a spatial object will be directly enqueued (Line 20). For the final case that e is a spatial object, the algorithm will rewrite the graph query that maps s to e, and execute the query (Line 23). If there is at least one qualifying map, e is added to the result. Such procedure continues until K objects are found or q is empty.

Example. Figure 15 shows an example of GraSp-KNN. The query location loc is drawn with a red cross in the figure. The query is $(v - u - s, \langle s, loc, 1 \rangle)$. In the GDBMS, the corresponding Cypher query is as follows:

Query 2 MATCH (v' : v) – (u' : u) – (s' : s) RETURN Dist(s', loc) AS dist ORDER BY dist LIMIT 1.

So the aim of this query is to find the closest s vertex from loc that can satisfy $v-u-s$. The table in Figure 15 depicts the elements in the queue for SpaIndex and Riso-Tree approach during the query processing. By searching the spatial index, SpaIndex first accesses the spatial objects s_4 and s_3 within R3 in sequence. The query for s_4 will be rewritten to

Query 3 *MATCH (v' : v) – (u' : u) – (s' : s) WHERE id(s') = s_4 RETURN s'.*

The query pattern is kept while the KNN predicate is reduced to only vertex s_4. However, such a query will return an empty set because the sub-graph at s_4 does not satisfy the query graph pattern. Similarly, the rewritten query for s_3 returns nothing. Then R2 is dequeued and its children are enqueued. Finally, s_2 is found as the final result. But, Riso-Tree can avoid accessing spatial objects within R3. The reason is that the existing paths for R3 are $v-u$ and v which does not include the APaths (u and $u-v$) of the query.

4.2.5 GraSp-Join

GraSp-Join query is a graph query with at least one spatial join predicate that joins two sets of spatial vertexes in the graph database. A spatial join predicate is commonly used in geospatial analytics. It can be any spatial relationship between spatial objects, such as overlap, enclosed by, within distance, etc. Without loss of generality, we present our work based on the `"Within distance"` spatial join operator, which means that two spatial objects need to be within a given distance from each

other. The format of the spatial join predicate is $SP_q = \langle s, t, dist \rangle$, where s and t are the spatial query vertexes included in the distance join predicate and $dist$ is the distance value. An example query is to find all the pairs of friend vertexes in a social graph who live within 1km from each other. The query can be written in Cypher as follows:

Query 4 MATCH (p_1:*Person*) – (p_2:*Person*) WHERE Dist(p_1, p_2) ≤ 1 RETURN p_1, p_2

The join predicate is represented by the Dist function. Similar to performing spatial join by using R-Tree, the algorithm (see Algorithm 15) keeps pairs of nodes in the Riso-Tree in a queue q. It starts by pushing the root nodes of Riso-Tree into q (Line 2) Each time, a pair of Riso-Tree nodes $\langle node_a, node_b \rangle$ is dequeued. For each child of $node_a$ and $node_b$, the algorithm checks both the join predicate and the graph constraint by using CheckPaths function. The pairs that can pass the validation are enqueued (Line 10). If the nodes are leaf nodes, each pair of spatial vertexes will be checked against the spatial join predicate. The pairs that can satisfy the predicate will be added to the candidate set (Line 17). The candidate set for the two anchor vertexes in each join predicate is much smaller as compared to the SpaIndex approach. This is because our algorithm filters out spatial vertexes that cannot satisfy the graph constraint without searching the graph. Once the candidate set for all anchor vertexes are generated, our approach rewrites the original query into a set of graph queries for each pair of spatial objects to search for the sub-graphs that can satisfy the query graph. Query 4 will be rewritten to:

Query 5 MATCH (p_1:*Person*) – (p_2:*Person*) WHERE id(p_1), id(p_2) in $\{(o_s, o_t)\}$ RETURN p_1, p_2

Algorithm 15 GraSp-Join Algorithm

Function GraSp-Join(G_q, $SP_{join} = \langle s, t, dist \rangle$)
$\langle APaths_s, APaths_t \rangle \leftarrow$ RecognizePaths(G_q, SP_{join})
Queue $q \leftarrow \emptyset$
$q.Enqueue(\langle root_s, root_t \rangle)$
while $q \neq \emptyset$ **do**
  $\langle node_s, node_t \rangle \leftarrow q.Dequeue()$
  if $node_s$ and $node_t$ are non-leaf nodes **then**
    for each $child_s$ of $node_s$ and $child_t$ of $node_t$ **do**
      if CheckConstraint($child_s$, $child_t$) = true **then**
        $q.Enqueue(\langle node_s, node_t \rangle)$
      end if
    end for
  else
    for each $o_s \sqsubset MBR(node_s)$ **do**
      for $o_t \sqsubset MBR(node_t)$ **do**
        if Distance(o_s, o_t)$\leq dist$ **then**
          Add $\langle o_s, o_t \rangle$ into the candidate set
        end if
      end for
    end for
  end if
end while
Rewrite & Execute Query

where $\{(o_s, o_t)\}$ represents the candidate set for the join predicate.

4.3 Experiments

In this section, we experimentally evaluate the proposed approach. We use four real datasets (Table 5): (1) Gowalla: geospatial graph dataset extracted from SNAP datasets[1] with 1477K vertexes, 9859K edges, and 1280K spatial vertexes. (2) Wikidata: A Knowledge graph with ≈47 Million vertexes (≈6 Million vertexes are spatial) and

[1] https://snap.stanford.edu/data/loc-gowalla.html

≈245 Million edges. (3) Foursquare: a geospatial graph dataset [43] with 3296K vertexes, 1143K spatial vertexes, and 19605K edges. (4) YELP[2]: An urban graph dataset (extracted from the Yelp Challenge dataset) with 629k vertexes (12% are spatial) and 8503K edges.

Table 5: Graph Datasets (K = 10^3)

Dataset	$\|V\|$	$\|V_S\|$	d_{avg}	$d_{avg}(V_S)$
Gowalla	1477K	87%	6.67	3.11
Wikidata	47116K	12%	5.20	6.10
Foursquare	3296K	35%	5.95	0.78
Yelp	630K	12%	13.5	31.9

We compare Riso-Tree to the SpaIndex, GraphTraverse and GEOENC approaches. GEOENC is the spatial encoding approach implemented in [29]. GEOENC was implemented for RDF stores and focuses on the entity encoding. From an execution strategy perspective, it takes advantage of the optimizer of the RDF engine. So basically it takes either GraphTraverse or SpaIndex strategy. There are two Riso-Tree approaches, denoted as Riso* and Riso. α equal to 1.0 for both approaches. B has different setups in different datasets. B is equal to 2 in Gowalla, Foursquare and Yelp datasets while is set with 1 in Wikidata dataset. Riso has PN_{max} of ∞, which means it does not take the Path Neighbor removal strategy. Riso* has PN_{max} of 80 and the Path Neighbors whose size larger than 80 are removed from the index.

Implementation of Riso-Tree in Neo4j GDBMS. Neo4j GDBMS provides full support for labeled property graph data. So the graph datasets are stored using Neo4j in a straightforward manner. Vertexes and edges are stored with their labels in the graph. The spatial location of a vertex is stored as a key-value pair using

[2]https://www.yelp.com/dataset_challenge

the property storage feature. The tree structure of Riso-Tree is stored as a graph in Neo4j. Each node in Riso-Tree is stored as a single vertex in Neo4j. The leaf node in Riso-Tree directly points to the spatial vertexes in the graph. The Label Paths of a Riso-Tree non-leaf node are stored as a list property on the node. The Path Neighbors of a Riso-Tree leaf node are stored a set of key-value pairs. Each Path Neighbor is a key-value pair of $(label\_path, neighbor\_list)$. The Riso-Tree index search is performed by using the Neo4j Java API [31]. By using the Neo4j Java API, our algorithm can access the children of a node in Riso-Tree. The spatial constraint and Label Path constraint can be checked by accessing the properties of Riso-Tree nodes. After the Riso-Tree search, the algorithm will form new Cypher queries based on the candidate sets. In the final step, our algorithm will call the Neo4j Java API which facilitates executing a given Cypher query and returns the result.

Query Generation. The query graphs are generated by exploiting a sub-graph generator similar to that in the existing paper [38]. The sub-graph generator randomly picks up a spatial vertex and a sub-graph around this spatial vertex is generated by using a random walk. It can ensure that the query graph will definitely have sub-graph matchings in the data graph. In the GraSp-Range, the spatial range selectivity is evaluated by the number of spatial objects in the query range to the total number of spatial objects in the graph.

A prototype of all approaches is implemented in Neo4j – a real open-source graph database management system. The source code for evaluating query response time is implemented in Java and compiled with java-8-openjdk-amd64. All experiments are run on a computer with a 3.00 GHz CPU, 128GB RAM running Ubuntu 18.04.

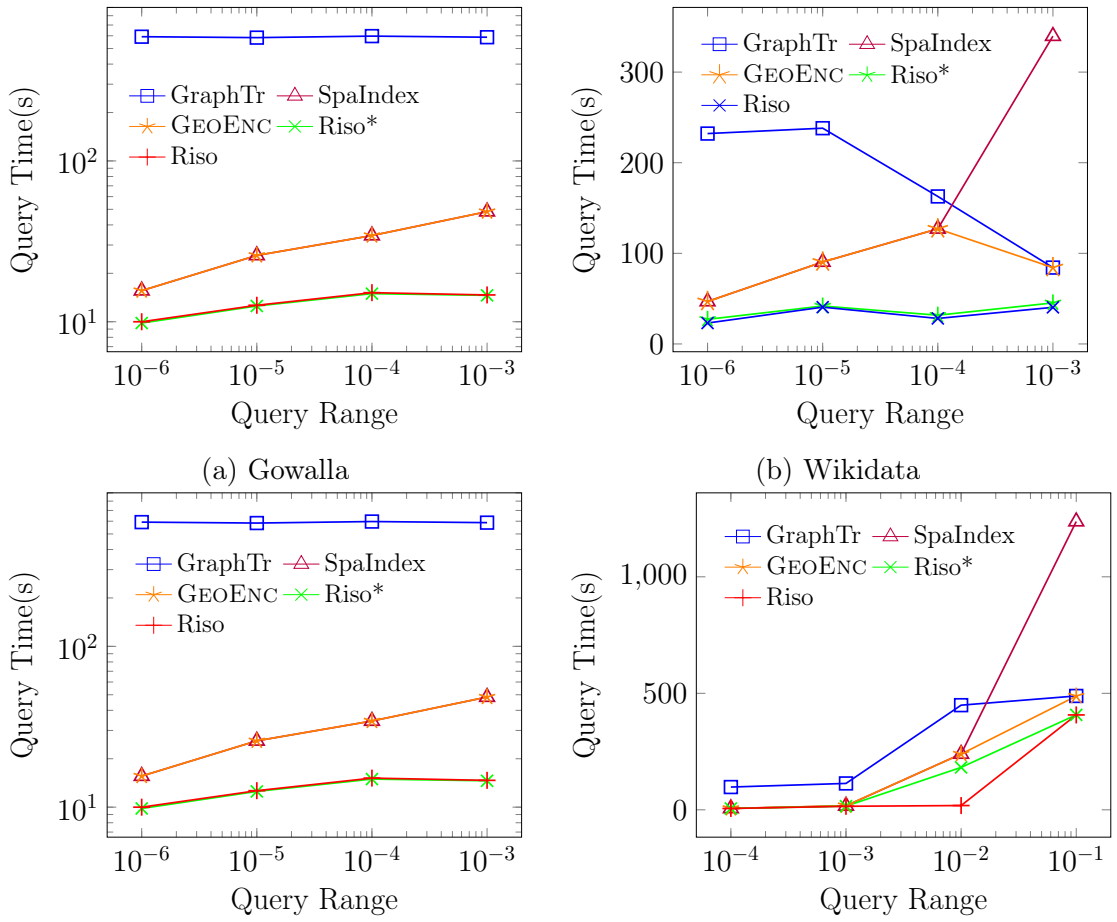
We evaluate the compared approaches using the following metrics: (i) Initialization time: the average time (in Seconds) each approach takes to construct all indexes and/or

auxiliary data structures necessary to answer a GraSp query, (ii) Storage overhead: the disk space (in Bytes) used to store the index and auxiliary data structures, (iii) Query response time: the average time the GDBMS takes to execute a GraSp query.

Impact of Spatial Selectivity on GraSp-Range. Spatial selectivities are varied in four different datasets. Query graphs used in this section have the size of 3 in Gowalla, Foursquare and Yelp and size of 2 in Wikidata. For each setting of selectivity, 50 queries are run and evaluated. The average running time is recorded and demonstrated. Figure 16 shows the query time of different approaches. GraphTraverse approach is the worst one most of the time, especially when the query is more selective in space. The reason is GraphTraverse does not utilize the spatial indexing technique to reduce the search space. SpaIndex can outperform GraphTraverse in all the datasets except for the highest spatial selectivity case in Wikidata and Yelp datasets. It is because R-Tree can reduce the size of the candidate set for the spatial query vertex with the help of the spatial index. The reduction factor is determined by the spatial selectivity. When the spatial predicate is less selective, more spatial vertexes lie within the query rectangle. Hence, the query performance of SpaIndex becomes worse at a high rate. Specifically, Yelp dataset has a high average degree for spatial vertexes (31.89). The increase in the number of qualified spatial vertexes has a high impact on the graph search performance. Similarly, Wikidata has a relatively higher average degree for spatial vertexes. So in these two datasets, SpaIndex becomes worse when the query is not selective in space. In all datasets, two Riso-Tree approaches can outperform SpaIndex, GraphTraverse and GeoEnc in most cases. The reason is two-folded. One is Riso-Tree not only exploits the power of spatial index but also provides more pruning capacity compared with SpaIndex by using the stored graph information. So it makes Riso* and Riso accesses less spatial index branches.

Meanwhile, candidate sets for query vertexes within B-hop distance from the spatial predicates can be reduced by orders of magnitude by using Riso-Tree. The query response time of Riso* and Riso tends to increase when there are more spatial vertexes in the query range since the effect of reducing the size of candidate sets diminishes in that case. But they are still better than their competitors. When comparing Riso* with Riso, Riso* can almost achieve equivalent performance as Riso, except that when selectivity is 0.001 in Gowalla dataset and selectivity is 0.01 in Yelp dataset. It can be noticed that the selectivity is with larger values in these cases. It is reasonable because with more spatial vertexes within the query rectangle, it has a higher probability to meet the removed Path Neighbor during the search. Then Riso can outperform Riso* because it keeps all pruning power of Riso-Tree. But considering the saving in storage overhead, Riso* is a more scalable approach compared with Riso.

Impact of K on GraSp-KNN. Figure 17 shows the query time of different methods by varying the value of K in the GraSp-KNN. The value of K are set as 1, 5, 25 and 125. The query graph used in this figure has the size of 3, 3,3 and 4 in four datasets. In each dataset, 50 different queries are run for each value of K. The average time of the 50 queries are demonstrated in the figure. As we can see, the query time of GraphTr does not change when varying the value of K. It is because GraphTr needs to enumerate all the matched subgraphs whatever the value of K is. Query time of SpaIndex and Riso increases as K increases. Both approaches will utilize the spatial index to find the closes spatial object and validate whether the spatial object can be matched to the query graph. When the value K increases, it takes more time to find more matched spatial objects. The query time of SpaIndex is even longer than GraphTr in Foursquare dataset when the K is between 5 and 25. Foursquare dataset has the smallest average degree for spatial vertexes (0.78). Spatial vertex have less

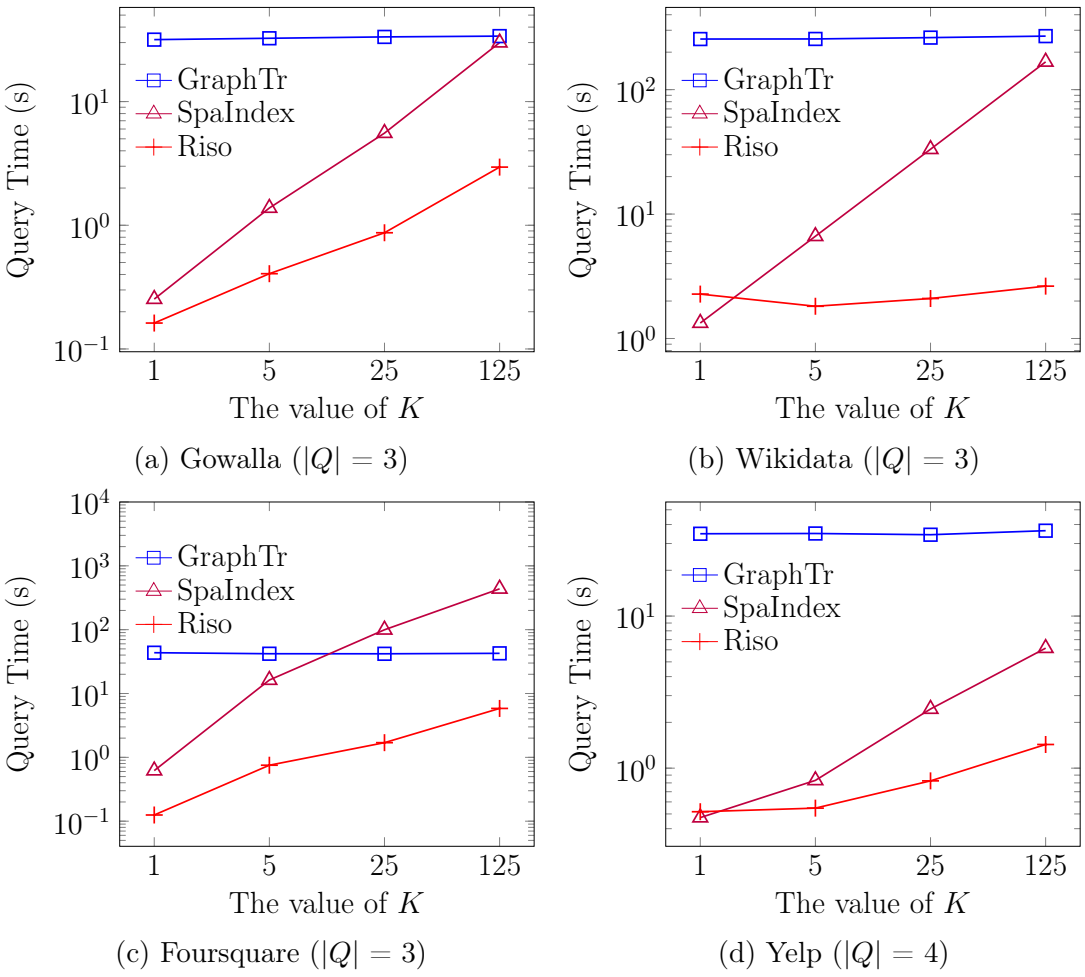


(a) Gowalla (b) Wikidata

(c) Foursquare (d) Yelp

Figure 16: GraSp-Range query response time

neighbors and it they have lower possibility to satisfy the graph pattern constraint. So it will take more time to find the qualifying spatial vertexes. Comparing Riso with SpaIndex, Riso can always outperform SpaIndex by up to 10 times. Riso can take advantage of the Riso-Tree to avoid going into some branches in the Riso-Tree that are not relevant to the query graph. So Riso have smaller search spatial index search space. Meanwhile, each spatial vertex will trigger a graph search to validate the query graph match. Riso checks much fewer spatial vertexes than SpaIndex approach. This is another reason why Riso outperform SpaIndex.



(a) Gowalla ($|Q| = 3$)

(b) Wikidata ($|Q| = 3$)

(c) Foursquare ($|Q| = 3$)

(d) Yelp ($|Q| = 4$)

Figure 17: GraSp-KNN (varying the value of K)

Impact of Query Graph Size on GraSp-KNN. Figure 18 depicts the query time of GraSp-KNN by varying the query graph size (number of vertexes in the query graph) in different datasets. The size of query graphs varies from 3 to 6. The value of K is set to 50. For each setup of query size, 50 queries are run and the average query time is recorded. For all the approaches, it tends to take longer processing time when the size of the query graph increases. When the size of the query graph increase, the complexity of the query graph leads to a higher cost to perform the graph search to find qualifying sub-graphs. Such increasing cost will take effect in all the

methods even for SpaIndex and Riso because a graph search phase occurs after every spatial object is fetched from the spatial index. But Riso still can always outperform SpaIndex because Riso performs much less query graph validations than SpaIndex in the graph search phase. Thanks to the pre-stored Path Neighbor information on each node in the Riso-Tree, Riso method can prune many unpromising branches in the spatial index. The pruned spatial vertexes will not appear in the graph constraint validation phase, which means the graph search does not happen from these pruned spatial vertexes. This is similar to what happens in the experiment of GraSp-KNN with different values of K.

GraSp-Join Performance. To analyze the influence of spatial join distance, we vary its value in the GraSp-Join. Figure 19 shows the query time all the datasets under different join distances. The join distance varies between 0.0001, 0.0002, 0.0003 and 0.0004. The spatial data in the experiment is in longitude and latitude format. Here the unit of the distance value is the degree. When the join distance increases, the query time of both SpaIndex and Riso approaches increase because more qualifying pairs of spatial vertexes are within the join distance. The query time of GraphTr increases a much slower ratio when the join distance increases. The query graph complexity has more influence on the the query time of GraphTr than the spatial selectivity (join distance). The figures also show that Riso can achieve up to 10 times better performance than SpaIndex approach. The performance of Riso will eventually be worse than GraphTr when the join distance is large enough. This trend is obvious in the Foursquare dataset that the query time GraphTr and Riso are almost equivalent.

Studying the indexing overhead. Table 6 shows the indexing overhead of SpaIndex and two Riso-Tree approaches. Index overheads of Riso* and Riso are higher

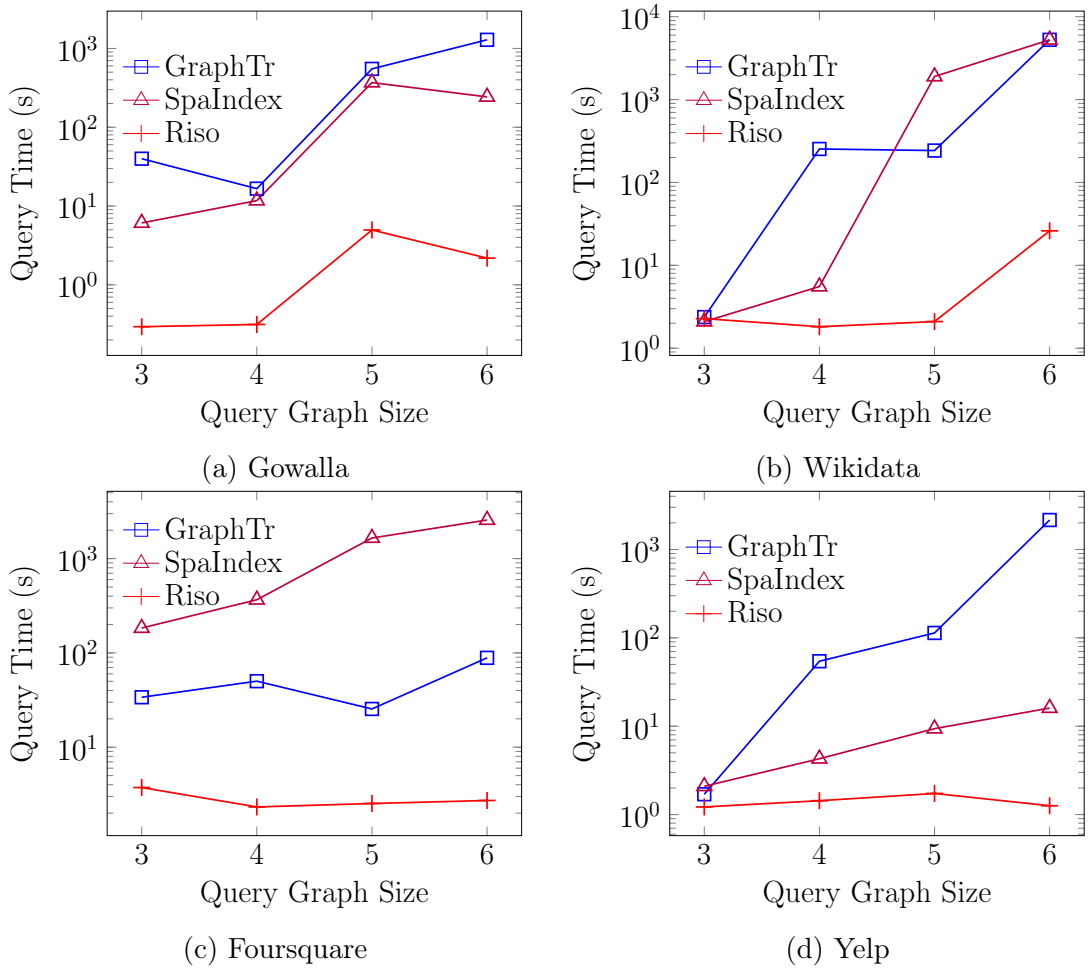
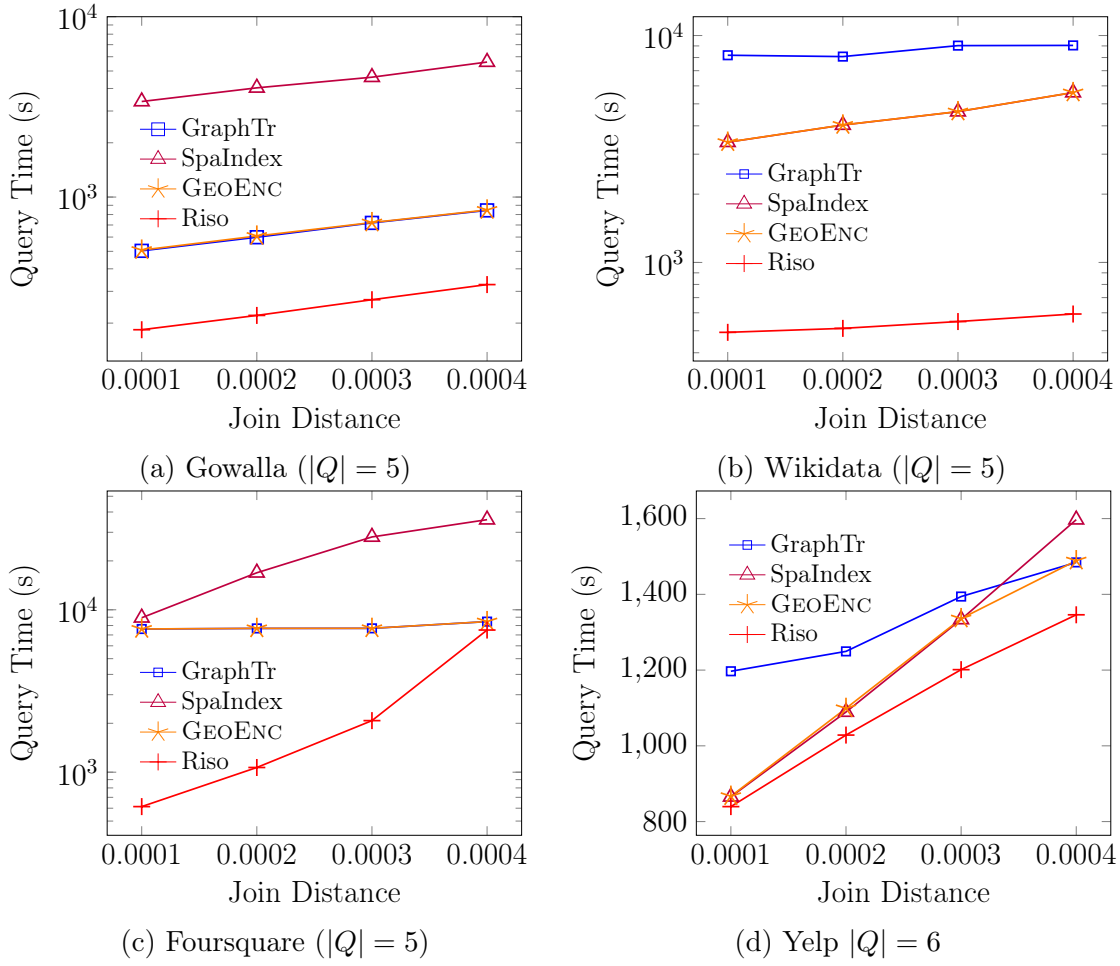


Figure 18: GraSp-KNN, varying the query graph size ($K = 50$)

than SpaIndex. It is because Riso-Tree will need to compute and store the Path Neighbor graph information besides the spatial index (R-Tree). The difference between SpaIndex and Riso-Tree approaches are related to the characteristics of the datasets. For Yelp dataset, construction time of Riso* is four times longer than SpaIndex and the index size is 100x larger than SpaIndex. It is because Yelp dataset has a high average degree of spatial vertexes. This makes the size of Path Neighbors big when the computation goes to 2-hop neighbors. But in Foursquare dataset, index size of Riso* is only 4 times larger than SpaIndex. The reason is that Foursquare has a small



(a) Gowalla ($|Q| = 5$)

(b) Wikidata ($|Q| = 5$)

(c) Foursquare ($|Q| = 5$)

(d) Yelp $|Q| = 6$

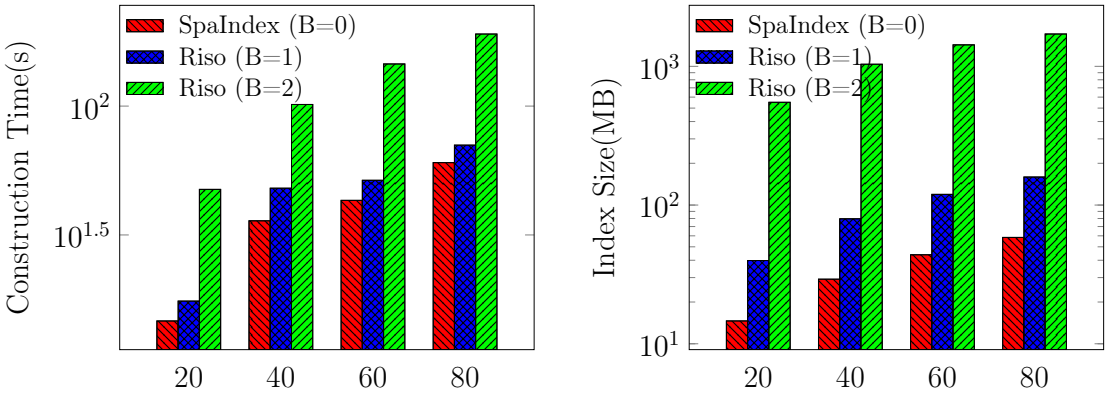
Figure 19: GraSp-Join by varying join distance

average degree of spatial vertexes. Wikidata is denser than Foursquare but the index size of Riso* in Wikidata is only twice higher than SpaIndex. It is because B is set to 1 in Wikidata. 1-hop Path Neighbor is a relatively light overhead. Index overhead of Riso is higher than Riso* because Riso* has a smaller value PN_{max}. Table 6 shows that index size of Riso* can be much smaller than Riso. The reduced ratio of storage overhead in Gowalla, Wikidata and Yelp are 72%, 49% and 52% respectively. The storage saving in Foursquare is only 12% because the majority of Path Neighbors in

Table 6: Indexing Overhead

Gowalla	Build Time	Index Size	**Foursquare**	Build Time	Index Size
SpaIndex	304.69 s	57.08 MB	SpaIndex	222.43 s	44.19 MB
Riso*	423.93 s	695.60 MB	Riso*	240.88 s	161.28 MB
Riso	461.68 s	2530.27 MB	Riso	244.39 s	183.49 MB
Wikidata	Build Time	Index Size	**Yelp**	Build Time	Index Size
SpaIndex	1470.8 s	222.43 MB	SpaIndex	12.70 s	2.99 MB
Riso*	1619.6 s	536.32 MB	Riso*	58.77 s	454.58 MB
Riso	1658.7 s	1067.36 MB	Riso	74.15 s	950.45 MB

Foursquare have small size due to its sparsity. So the removal strategy only applies to a few Path Neighbors.



(a) Init. Time (Spatial Ratios) (b) Index Size (Spatial Ratios)

Figure 20: Indexing Overhead

Figures 20a and 20b show the indexing overhead for a synthetic dataset that simulates various spatial to non-spatial vertexes ratios. The synthetic dataset consists of 3775K vertexes and 16519K edges. We randomly pick up 20%, 40%, 60% and 80% vertexes as spatial objects. Their location is randomly selected in a [0, 1000] $\times$ [0, 1000] 2D space. When the ratio of spatial vertexes increases, both the initialization time and storage size of SpaIndex (B=0) increase. The reason is the size of the tree skeleton is determined by the number of spatial objects. More spatial vertexes will

lead to more nodes at each tree level. Since more spatial vertexes and more tree nodes exist, it takes a longer time and more space to construct and store Path Neighbors. That also explains why the overhead of Riso (B=1) and Riso (B=2) is higher when the number of spatial vertexes increases. When comparing the initialization time and index size of Riso (B=1) and Riso (B=2), there is a huge difference because the number of neighbors expands exponentially when the number of hops increases.

Impact of PN_{max}. Figure 21 shows the impact of PN_{max} on the index size in each dataset. We change PN_{max} among 10, 40, 160, 640. PN_{max} controls the maximum size of every Path Neighbor in Riso-Tree. So when PN_{max} is set to 10, it means no Path Neighbor stored in Riso-Tree has size larger than 10. Riso does not remove any Path Neighbors. So its index size is always the same for all PN_{max} values. Riso does not have the limitation on the size of Path Neighbor or it can be treated as $PN_{max} = \infty$. So Riso is used as the baseline method because it sets the upper-bound for the index size of Riso*. When PN_{max} becomes larger, index size increases towards that of Riso in all four datasets. The reason is straightforward that there are fewer Path Neighbors removed because of a higher threshold for the Path Neighbor removal.

For Riso* approach, it removes some Path Neighbors whose sizes exceed PN_{max}. So all the Path Neighbors in Riso are divided into two parts. One is the remaining Path Neighbors and the other is the removed Path Neighbors. When a PN_{max} is selected, Riso* only keeps the qualifying Path Neighbors. The PN count ratio of Riso* represents how many Label Paths of Riso are covered by the Path Neighbors of Riso*. Its range is from 0% to 100%. A larger PN count ratio means less Path Neighbors removed and the graph information is more accurate. So PN count ratio becomes larger when PN_{max} increases. Table 7 demonstrates the PN count ratio of Riso* under different value settings of PN_{max}. In Gowalla and Wikidata datasets

when PN_{max} is 10, more than 80% Path Neighbors can be covered with around 20% storage overhead compared with Riso (Figure 21a, 21b). Since the majority of Label Paths are covered, the paths in the query graph have a high probability are covered by the Path Neighbors of Riso*. So the saving of storage overhead is considerable while accuracy is not harmed too much. The results in Foursquare and Yelp datasets show different trends. In Foursquare dataset, the saving of storage is only around 50% when PN_{max} is 10. The reason is that its average degree is 5.94 and the average degree of spatial vertexes is only 0.775. The spatial vertexes in Foursquare have much fewer neighbors. So Foursquare does not have many big-sized Path Neighbors and the removal strategy has less benefit. In Yelp dataset, PN count ratio has a sharp increase between 10 and 40. It indicates that many Path Neighbors have sizes within this range. It is because Yelp dataset has a high average degree of all vertexes (13.5) and spatial vertexes (31.89). There are many Path Neighbors whose sizes are within the range from 10 to 40. When PN_{max} changes from 10 to 40, these Path Neighbors are brought back to the index.

Table 7: PN Count Ratio to Riso (%)

PN_{max}	10	40	160	640
Gowalla	93.4	97.0	98.5	99.6
Wikidata	90.3	98.1	99.7	99.8
Foursquare	78.3	92.0	99.2	100.0
Yelp	22.0	90.0	98.6	99.5

Figure 22 demonstrates the query time for different settings of PN_{max}. For each dataset, we show its performance with two different spatial selectivities. It can be observed that the query performance is better when the value of PN_{max} is larger. The reason is straightforward that less Path Neighbors are removed which makes the

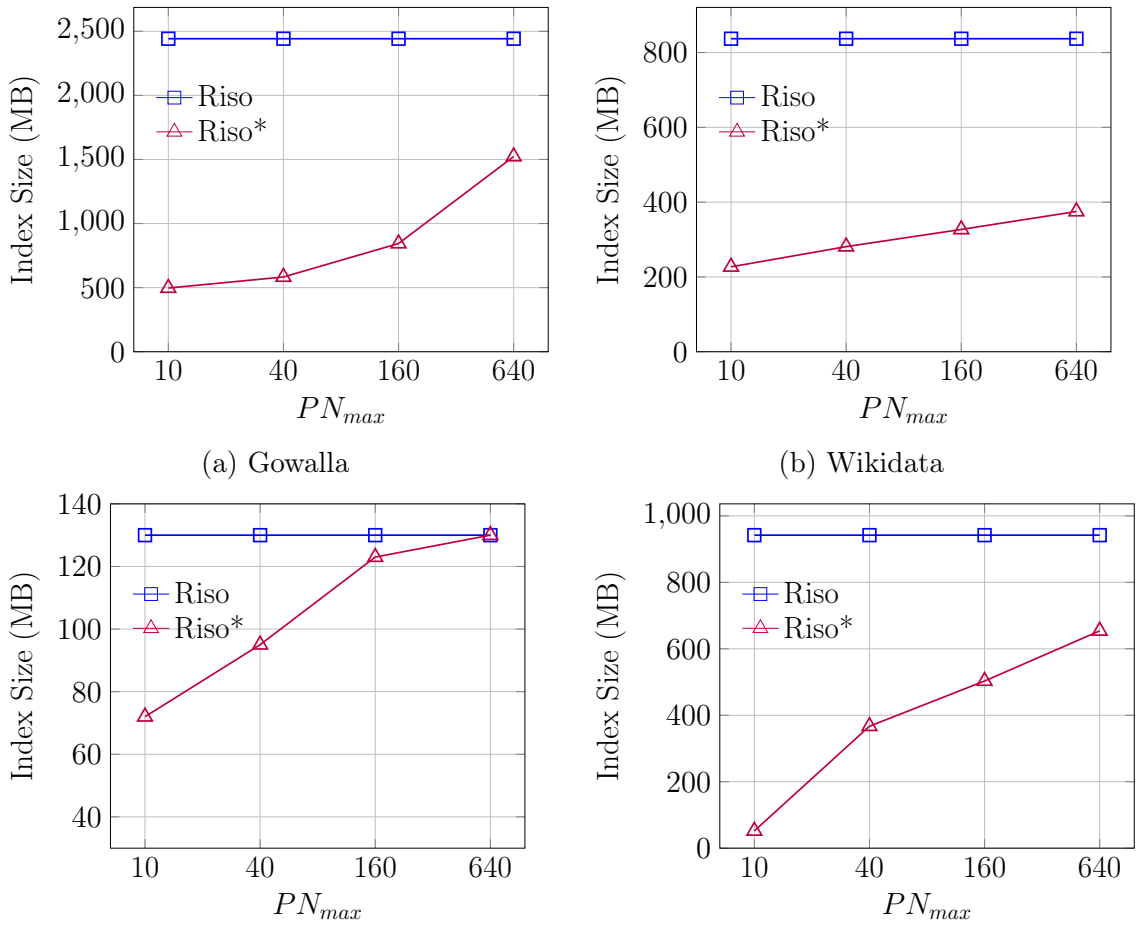
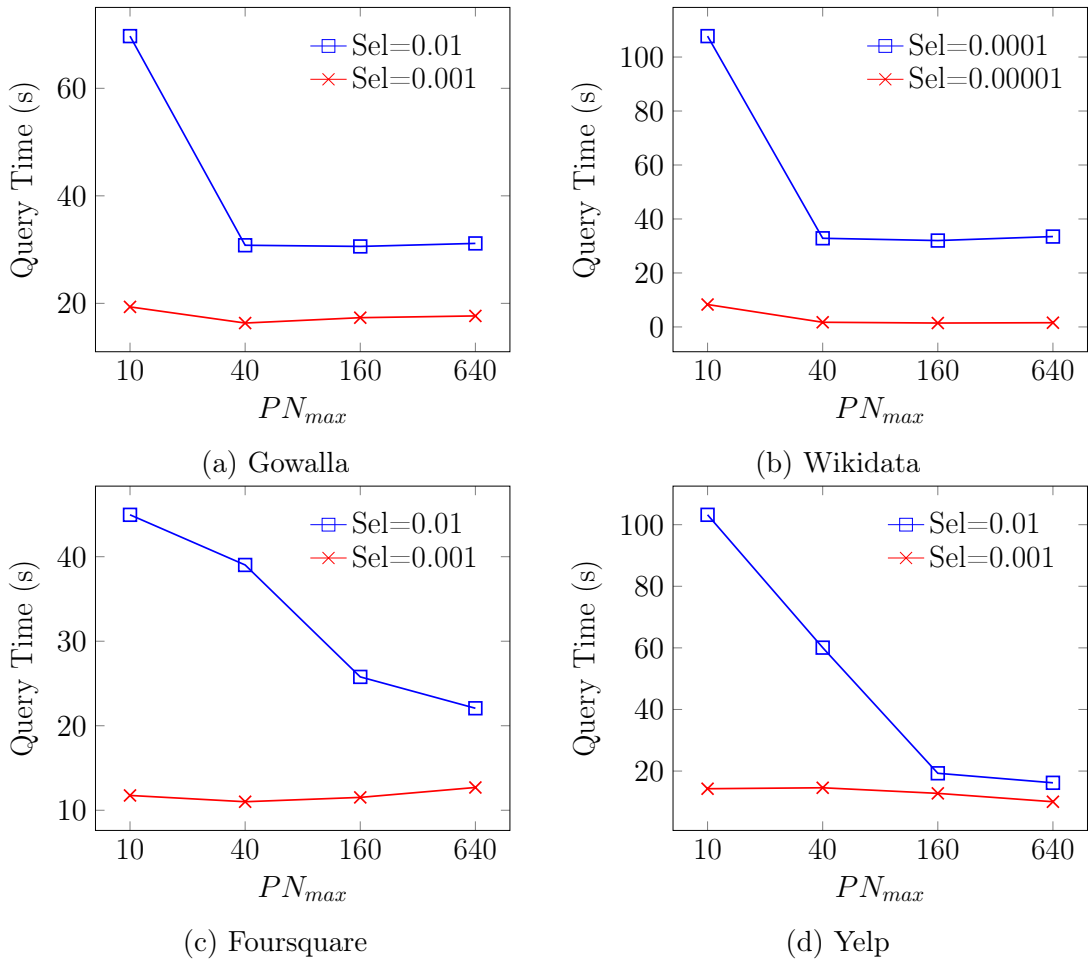


Figure 21: Impact of PN_{max}

Riso-Tree pruning more powerful. When the PN_{max} exceeds a specific value, increasing PN_{max} will not improve the performance. In Gowalla and Wikidata, this specific value for PN_{max} is 40 while in Yelp it is 160. In Foursquare, the value is larger than 640. The set of experimental queries need to access partial Path Neighbors. When the accessed Path Neighbor are all covered by the Riso-Tree with the specific value of PN_{max}, increasing PN_{max} will not lead to more accurate candidate sets for query vertexes. Another observation is that the impact of PN_{max} differs for queries with different selectivities. For queries with higher selectivity, the impact is not obvious. It



(a) Gowalla (b) Wikidata

(c) Foursquare (d) Yelp

Figure 22: Query Time By Varying PN_{max}

is because the query region is large and only overlaps with large number of Riso-Tree nodes. Overlapping with more nodes will lead to a higher probability to access the leaf nodes which contain the removed Path Neighbors. So the query performance has a higher frequency to be influenced by PN_{max}.

Impact of α. The value of α determines the influence of spatial and graph information in the tree construction phase. In order to analyze the impact of α, Wikidata is used as our experimental dataset. B is fixed to 1 and PN_{max} is fixed to infinite, which means no constraint on the size of Path Neighbor. Table 8 shows the

index size, query time, overlap leaf count and page access when the value of α changes. The selectivity of the query rectangle is 0.001. The index size only includes the Path Neighbor without considering tree size. Table 8 clearly shows that when α increases, the index size tends to increase. When α increases, the graph information has a smaller influence in the construction, including the target leaf node selection and tree node split. The constructed Riso-Tree tends to achieve better space partition (less dead space and overlapped area) while causing more separation and duplicates in graph information which will increase the index size. Regarding the query performance, it achieves better query performance when α increases. This is because when α increases, Riso-Tree has less dead space and fewer overlapped areas between tree nodes. So the query rectangle will overlap with fewer nodes. With less overlapped nodes, the candidate sets formed by these leaf nodes have a smaller size. This will in turn reduce the query response time because fewer vertexes will be accessed during graph search as well. Table 8 shows such a trend that when the value of α increases, fewer leaf nodes overlap with the query rectangle and the number of database page access decreases.

Table 8: Impact of α in Wikidata

α	Index Size	Query Time	Leaf Hit	Page Hit
0.0	727 MB	452 s	51	3188K
0.25	755 MB	300 s	18	2220K
0.5	763 MB	341 s	21	1801K
0.75	768 MB	225 s	19	1967K
1.0	805 MB	152 s	10	771K

Parameter Setup. Several parameters can be tuned for Riso-Tree, including B, PN_{max} and α. We discuss how to pick up the default values for these parameters to help users start using the system. Parameter B has a huge impact over the performance on the Riso-Tree. When increasing the value of B, it can improve the

query performance while at the cost of exponentially increasing the construction time and index size. But when the value B is large, increasing the value of B brings less improvement on query performance. The number of neighbors for a vertex will increase exponentially when the path hop number increases. As a result, the size of Path Neighbor will become large quickly when B increases. The Path Neighbor cannot reduce the search space due to its low selectivity. To sum up, the feasible setup of B should be less than 3. But it can be larger if the graph is sparse. PN_{max} controls the maximum size of each Path Neighbor. Because it removes the Path Neighbor list whose size is larger than PN_{max}, it also ensures the contribution of the Path Neighbor to reducing the search space. Based on the experiment, a value around 100 is a good choice. α determines the weight of the graph and spatial influence when constructing the tree structure. Increasing α reduces the dead space in the Riso-Tree index while increasing the storage overhead. It is fine to use the $\alpha = 1$ because the other two parameters have some control over the indexing overhead. Another benefit of using $\alpha = 1$ is that it does not need to compute the graph path expansion GPE, which can accelerates the Riso-Tree construction.

Chapter 5

INDEX MAINTENANCE

5.1 SPA-Graph Maintenance

In this section, we explain how the system maintains SIP in response to updates in the graph database that include edge/vertex insertion/deletion. Updating an attribute (i.e., vertex spatial property) can be using a combination of graph structure updates. When $v.loc$ is modified, it can be represented as deleting all connected edges of v and re-adding all the edges on the vertex with a new spatial location. Having said that, we consider the following updates:

5.1.1 Adding an edge

When an edge (u, v) is added to the graph, the system does not need to update all SIP. The update scope is bounded by B. More specifically, the vertexes influenced by inserting an edge are less than B-1 hops from either u or v. The reason is that SIP only consider neighbors within B hops. u and v will impact vertexes that are within B-hops from u and/or v. The main idea is to ensure that Theorems 2, 3 and 4 still hold for all vertexes. To achieve that, the update algorithm runs in two main phases, Local Update and Neighbor Transfer. Figure 23 depicts the main steps.

Local Update. The direct impact of inserting an edge is on u and v themselves. In Local Update, the algorithm updates SIP of u and v. Without loss of generality, we focus on the SIP updates of v from that of u. Maintaining u' s SIP by exploiting v'

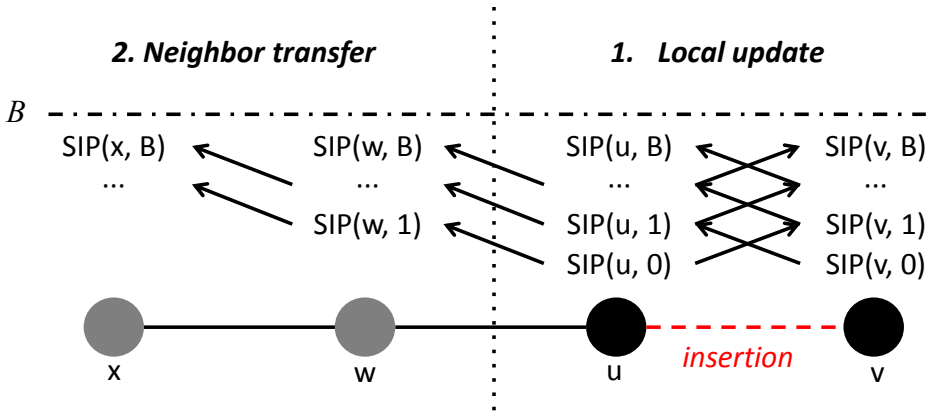


Figure 23: Adding an edge

s SIP is similar to the SIP initialization algorithm. Generally, it follows this equation:

$$SIP(u, k+1) = SIP(u, k+1) \cup SIP(v, k), k \in [0, B-1] \tag{5.1}$$

The algorithm updates the $k+1$-hop SIP of u by merging the k-hop SIP of v. Here SIP(v, 0) represents $v.loc$.

Neighbor Transfer. After the Local Update phase, the spatial indexing properties (SIP) of u' s neighbors might need to be maintained as well. This is determined by whether SIP(u, k) is modified. One possibility is that SIP of u remains the same if SIP(u, k) has already enclosed the spatial range of SIP(v, $k-1$). In this case, SIP(u, k) is actually not modified. If SIP(u, k) is modified, then the algorithm will update SIP(w, $k+1$) where w is a neighbor of u. Such update will be performed recursively on each neighboring of w by following Equation 5.1. If SIP(u, k) is not modified after the update, there is no need to update SIP(w, $k+1$). In Figure 23, the value of B limits the upper bound of the number of hops which in turn limits the number of vertices to be accessed during the update. In other words, each accessed vertex is at most (B - 1) hops from u or v.

Algorithm 16 shows the pseudo code for inserting an edge between u and v. Once an insertion happens, the algorithm calls the UPDATE function to maintain the SIP

Algorithm 16 Edge Insertion Update

```
1: Function INSERT(Vertex u, Vertex v)
2: Create edge (u, v)
3: for each k ∈ [0, B-1] do
4:     UPDATE(u, v, k)
5:     UPDATE(v, u, k)
6: end for
```

Algorithm 17 Update SIP of a specific hop

```
1: Function UPDATE(Vertex u, Vertex v, Hop k)
2: if k ≥B then
3:     return
4: end if
5: SIP(u, k + 1) ← SIP(u, k + 1) ∪ SIP(v, k)
6: if SIP(u, k + 1) is modified then
7:     for each w ∈ Adja(u) do
8:         UPDATE(w, u, k + 1)
9:     end for
10: end if
```

of both u and v for hops within the [0, B - 1] range. UPDATE takes Hop k as its input and updates SIP(u, $k+1$) by using SIP(v, k). If SIP(u, $k+1$) is modified, the UPDATE function will be called recursively on each neighbor vertex of u to update its SIP(u, $k+2$). The algorithm terminates when $k \geq B$.

5.1.2 Update Cases

Algorithm 16 gives the pseudo code and the general rule (Equation 5.1) used to update SIP when an edge is inserted. However, since each spatial indexing property (i.e., SIP) can be of different categories (e.g., ReachGrid, RMBR), maintaining SIP(u, $k+1$) accounts for each of these categories. To achieve that, the algorithm leverages the rule in Table 9. The cases marked with ✓ means that such validation can be handled in a straightforward way. $v.loc$ only exists in SIP(v, 0). The update from

Algorithm 18 Update ReachGrid

```
1: Function UpdateReachGrid(Vertex u, Vertex v, Hop k)
2: switch (type of SIP(v, k))
3: case ReachGrid:
4:    ReachGrid(u, k + 1) ← ReachGrid(u, k + 1) ∪ ReachGrid(v, k)
5:    if ReachGrid(u, k + 1) is modified then
6:       return true
7:    end if
8:    if ReachGrid(v, k) contains boundary cell of ReachGrid(u, k + 1) then
9:       return true
10:   end if
11: case RMBR:
12:   Construct ReachGrid(v, k) by using RMBR(v, k)
13:   Perform the steps in case ReachGrid
14: case GeoB:
15:   SIP(u, k + 1) ← true
16:   return true
17: return false
```

Algorithm 19 Update RMBR

```
1: Function UpdateRMBR(Vertex u, Vertex v, Hop k)
2: switch (type of SIP(v, k))
3: case RMBR:
4:    RMBR(u, k + 1) ← MBR(RMBR(u, k + 1), RMBR(v, k))
5:    if RMBR(u, k + 1) is modified then
6:       return true
7:    end if
8: case ReachGrid:
9:    Construct RMBR(v, k) by using ReachGrid(v, k)
10:   Perform the steps in case RMBR
11: case GeoB:
12:   SIP(u, k + 1) ← true
13:   return true
14: return false
```

$v.loc$ is trivial. So, the column of $v.loc$ are all marked with ✓. In case $v.loc$ exists for vertex v, GeoB will be set to *true*, RMBR(u, 1) will be MBR(RMBR(u, 1), $v.loc$), and ReachGrid(u, 1) will be ReachGrid(u, 1) ∪ $Grid(v.loc)$, respectively. The updates between two identical types of spatial indexing properties are similar to that presented in the initialization algorithm. For the inter-type updates, less accurate SIP can be

Algorithm 20 Update GeoB

```
1: Function UpdateGeoB(Vertex u, Vertex v, Hop k)
2: if GeoB(u, k + 1) is true then
3:     return false
4: else
5:     if SIP(v, k) is false then
6:         return false
7:     else
8:         SIP(u, k + 1) ← SIP(v, k)
9:         return false
10:    end if
11: end if
12: return false
```

updated easily by more accurate SIP but not the reverse way. GeoB can be upgraded to RMBR and ReachGrid as follows: In case GeoB is true, it will not be modified by either RMBR or ReachGrid. Otherwise, the algorithm switches it to RMBR or ReachGrid. RMBR can be updated by ReachGrid by creating the MBR for ReachGrid. For the remaining update cases (with cross marks), directly updating SIP can later lead to inaccurate query results. For instance, if SIP(u, $k + 1$) is ReachGrid type and SIP(v, k) is RMBR type, it is not straightforward to update the ReachGrid to RMBR. There are two strategies to handle such cases. They are lightweight and Reconstruct strategies.

Table 9: Updating Cases of SIP

To / From	GeoB	RMBR	ReachGrid	$v.loc$
GeoB	✓	✓	✓	✓
RMBR	×	✓	✓	✓
ReachGrid	×	×	✓	✓

5.1.3 Maintenance Strategies

When the user inserts a new edge between u and v, the system follows one of two maintenance strategies, described as follows:

Lightweight Strategy. The Lightweight strategy is to take advantage of the existing SIP information stored in v. To update a ReachGrid with an RMBR, all grid cells covered by the RMBR will be added into the ReachGrid. When either the RMBR or ReachGrid is updated by GeoB with true value, the Lightweight strategy will directly replace the RMBR or ReachGrid with true-valued GeoB. In case the GeoB value is false, no modification is needed.

Reconstruct Strategy. This strategy reconstructs the ReachGrid(v, $k-1$) for the cases marked with a cross in Table 9. Then, GeoExpand on u will be updated accordingly. The system performs the Reconstruct strategy by accumulating all the ReachGrid(w, $k-2$) where w is the neighbor of v. When SIP(w, $k-2$) is not a ReachGrid, then the $k-3$ hops ReachGrids of all neighbors of w will be obtained to reconstruct ReachGrid(w, $k-2$).

The Lightweight strategy does not need to traverse the graph to update the SIP. Hence, the Lightweight strategy is faster compared with the Reconstruct strategy. Nonetheless, the Lightweight strategy may lead to less pruning power, which in turn increases the overall query response time.

Until here, we discuss how to update u and v, which are connected by the newly-added edge. The SIP of vertexes that are connected to u or v within less than B hops may also need to be updated accordingly. If SIP(u, k) is modified after being updated by SIP(v, $k-1$), then for 1-hop neighbors of u, their $(k+1)$ hops SIP should be updated. Similarly, for the 2-hops neighbors of u, their $(k+2)$ hops SIP will be

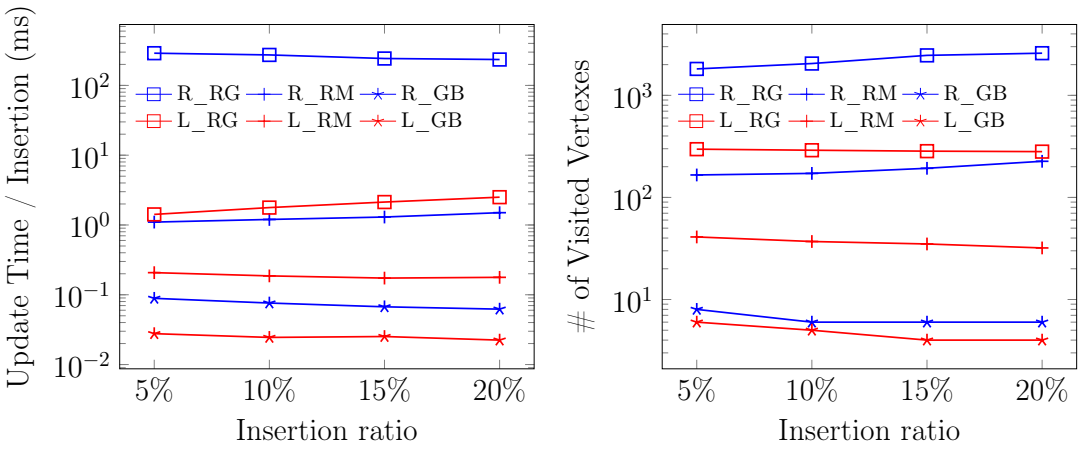
updated. Such update will be performed in a recursive manner; Hence, the influenced vertices are bounded by $B-k$ hops. Sometimes, when a SIP is updated by another, it is not really modified after the update. For instance, if RMBR(u, k) is updated using RMBR(v, $k-1$) and RMBR(u, k) totally encloses RMBR(v, $k-1$), RMBR(u, k) will have the same spatial boundaries; In this case, it is not modified. We differentiate the two cases after one update because if SIP(u, k) is not modified, there is no need to update SIP(w, $k+1$). Such a rule also holds true during the recursive update procedure.

5.1.4 Deleting an edge

When deleting an edge (u, v) , SIP of u and v with all numbers of hops may be impacted. Without loss of generality, we still focus on one side of u. The strategy to maintain the correctness of the SIP stored in u is to reconstruct each k-hop SIP. Therefore, the system compares the new version of SIP(u, k) with the old one. If it remains the same, that means some SIP of other vertexes that are computed based on it do not need to be updated. In case it is modified, all vertexes within $B-k$ hops from u need to be updated. If a vertex w is t-hops reachable from u ($w \in V_u^t$), then SIP(w, $k+t$) will be updated by accumulating information from its 1-hop neighbors again. Such reconstruction is executed from 1 to ($B-k$) hops neighbors for u.

5.1.5 SPA-Graph Maintenance Evaluation

To evaluate the performance of the SPA-Graph update algorithm, we randomly insert 5%, 10%, 15% and 20% edges into the Yelp graph dataset. Here 5% indicates the

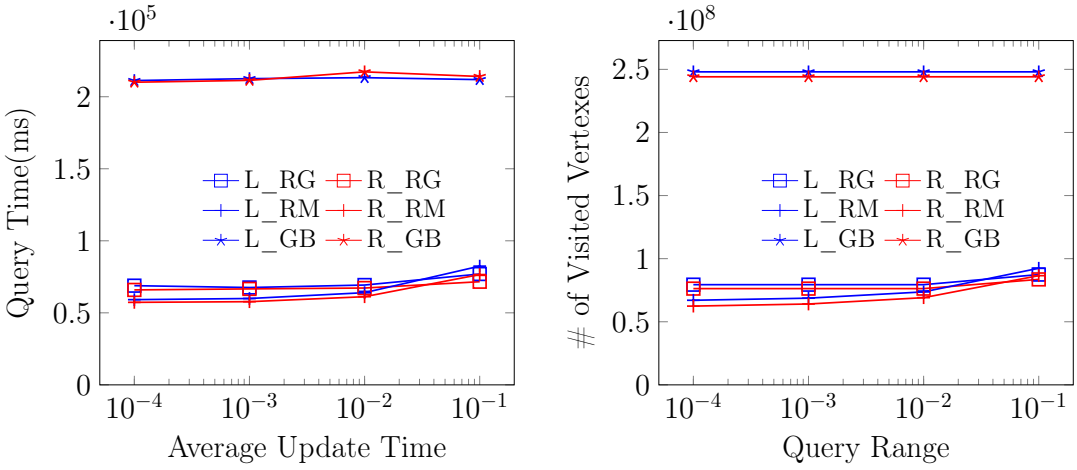


(a) Average insertion time

(b) The number of visited vertices

Figure 24: Inserting different number of edges to Yelp dataset

ratio to the total number of edges in the graph. We consider three parameter setups of SIP. The setups make the SIP be all ReachGrid (abbr. RG), all RMBR (abbr. RM) and all GeoB (abbr. GB). For each index setup, we test the two update strategies, i.e., Lightweight (abbr. L) and Reconstruct (abbr. R) strategies. For instance, LRG indicates the index with all ReachGrid using the Lightweight update strategy. Figure 24a and 24b shows the average time and the number of visited vertexes for each edge insertion. It can be observed that SIP with all ReachGrid requires the longest update time while GeoB requires the least regardless of the update strategies. It is because each ReachGrid consists of a list of ids. If a ReachGrid of a vertex contains many ids and an edge is inserted between such vertex and another, it is time-consuming to update the ReachGrid. As a result, the complexity of the computation on ReachGrid leads to the cost to update ReachGrid being far higher than the other two types of indexes. The update of RMBR is just to compute all the coordinates. It has equal complexity for all the vertexes and it is much less complex than that for ReachGrid. But its complexity is higher than updating GeoB because updating GeoB only needs the bitwise operation. By comparing the Lightweight with the Reconstruct strategies



(a) Response time after edges insertion

(b) The number of visited vertices

Figure 25: Query performance after inserting 10% edges

for all three index setups, the Lightweight update time is always shorter than the Reconstruct. This makes sense because the Lightweight strategy does not need to reconstruct the index by traversing the graph. Figure 24b also proves our explanation because it shows the Lightweight strategy requires to visit less vertexes than the Reconstruct strategy for the same index setup.

Figure 25 shows the query response time and the number of visited vertexes after inserting 10% edges into the Yelp graph dataset. The purpose is to evaluate the influence of maintenance strategies on the query performance. We can observe from Figure 25a that the Lightweight and the Reconstruct strategy share similar query time. The number of visited vertexes is also similar for two maintenance strategies. It reveals that the Lightweight can achieve the similar performance compared with the Reconstruct strategy even it loses some accuracy in the reachable region representation. Combined with the update speed, we could say that the Lightweight strategy can be a better one because it can achieve similar query performance with a higher update speed.

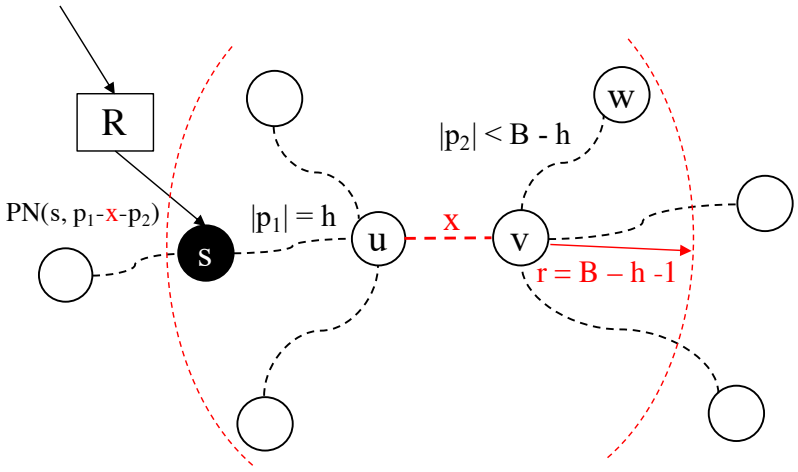


Figure 26: Riso-Tree Maintenance

5.2 Riso-Tree Maintenance

Inserting/deleting an edge. Algorithm 21 shows how to update Riso-Tree when an edge is inserted. When an edge of label x between u and v is inserted into the graph, the algorithm first reconstructs Path Neighbors of length from 1 to $B-1$ hops for u and v based on the original graph where the inserted edge is not considered (Line 2 and 4). The algorithm scans each vertex w in PN_u^p and PN_v^p where $|p| \in [1, B\text{-}1]$. Foe each spatial vertex s, the algorithm updates Path Neighbor information of the leaf node that contains s (Line 9). Besides, all the ancestor nodes of this leaf node will also be updated (Line 11). Let us use a generic example to demonstrate such a procedure. In Figure 26, a new edge with label x is inserted between u and v. There exists one spatial vertex s that is h-hop distant from u. The Riso-Tree node that contains s is denoted as R. The path from s to u is p_1. In order to reflect the edge insertion, vertex v should be added into $PN_R^{p_1 x \varphi(v)}$ where $p_1 x \varphi(v)$ is a label path that concatenates p_1, x and the label of v. Besides that, Path Neighbors of v, PN_v^p where $|p| < B - h$, will also be added into Path Neighbors of R. For a vertex w which

Algorithm 21 Maintenance algorithm for adding an edge

```
1: AddEdge(Vertex u, Vertex v, EdgeLabel x)
2: for |p| ∈ [1, B − 1] do
3:    Compute PN_u^p and PN_v^p
4: end for
5: for each vertex w ∈ PN_u^p do
6:    if w ∈ V_S then
7:       PN_R^{pxφ(v)} ← PN_R^{pxφ(v)} ∪ {v} (R ∋ s)
8:       for each vertex t ∈ PN_v^{p_2} (|p_2| < B − 1) do
9:          PN_R^{p_1xp_2} ← PN_R^{p_1xp_2} ∪ t
10:      end for
11:      Update ancestor nodes of R recursively if necessary
12:   end if
13: end for
14: for each vertex w ∈ PN_v^p do
15:   Repeat the same steps for u
16: end for
```

is connected from v through path p_2 and $|p_2| < B - h$, w will be added into $PN_R^{p_1xp_2}$. Besides updating Riso-Tree node R, ancestor nodes of R might need to be updated as well. The updating rule is as follows: Existing paths of R's ancestor nodes need to be updated if the updated path does not exist in R before the insertion. It means that if p_1xp_2 does not exist in R before the updating, then the algorithm adds p_1xp_2 to the ancestors of R recursively.

When an edge between u and v is deleted from the graph, the impacted extend is similar to inserting an edge. Only the vertexes within $B - 1$ hops from u or v will be impacted. So the first step is still to compute the Path Neighbors within $B-1$ hops from u and v. For each spatial vertex s in PN_u^p and PN_v^p where $1 \leq |p| < B$, the corresponding Riso-Tree node R that contains s needs to be updated. The way to update the Path Neighbors of R is to recalculate them. Even though such an update is costly, a lazy-update strategy can be taken to avoid frequent and instant re-computation for the deletion accordingly. Riso-Tree is actually filtering-based index

structure. So keeping a superset of the the "accurate" Path Neighbor will not impact the correctness of the query algorithm. This is similar to that enlarging the area of MBR in R-Tree does not influence the correctness of the spatial query. As a result, the system just needs to executed the re-computation periodically, e.g. every month. Or the re-computation is executed when a certain number of deletions happen on the graph.

Skip Irrelevant Vertexes Strategy. When looking into the the algorithm of the edge insertion and deletion, it can be discovered that not all the edge updates will result in Path Neighbor updates. The edge updates that happen between some vertexes do not affect the sub-graph information stored in Riso-Tree. These vertexes are called 'irrelevant vertexes'. It can be defined as follows:

Definition 4 *A vertex v is a irrelevant vertex iff (1) $v \notin V_S$, (2) $\forall w \in PN_v^p$ where $|p| < B$, $w \notin V_S$.*

A vertex is irrelevant if there is no spatial vertex within its (B-1)-hop neighbors. We can have the following theorem for irrelevant vertexes:

Theorem 5 *If u and v are irrelevant vertexes, adding or deleting an edge between u and v will not cause updates of Path Neighbor in Riso-Tree.*

The correctness of Theorem 5 can be proved based on Algorithm 21. If both u and v are irrelevant, no spatial vertex exists within (B-1)-hop neighbors of u or v. If so, no spatial vertex is influenced by the edge between u and v. By taking advantage of this characteristic, we keep a set of all the irrelevant vertexes. Any time an insertion or deletion happens between two vertexes u and v, the algorithm first checks whether u and v are in the set. If both vertexes are irrelevant, the algorithm can directly

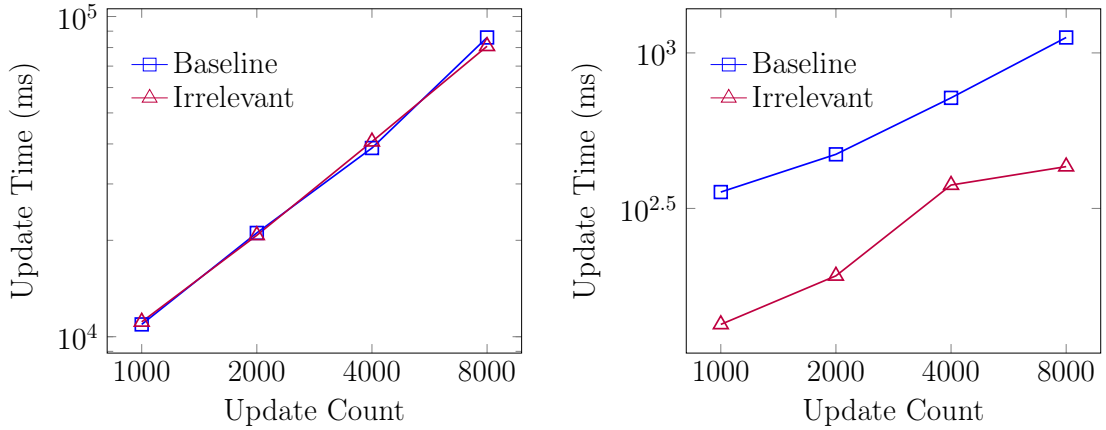
skip the rest operations because Path Neighbor will not change. Otherwise, it follows Algorithm 21.

Inserting/deleting a vertex: Inserting a new vertex with edges attached to it can be decomposed as inserting a solitary vertex and inserting the connected edges. Since the edge insertion has been discussed, we focus on inserting a solitary vertex in this paragraph. If the inserted vertex is not spatial, the algorithm will not change the structure of Riso-Tree. If the vertex is a spatial one, the algorithm will pick up the leaf node to insert that vertex by following the rule of minimizing SGE. MBR of the leaf node and the ancestor nodes will be updated accordingly. When a non-spatial solitary vertex is deleted from the graph, the algorithm does not need to modify the Riso-Tree. When a solitary spatial vertex s is deleted from a leaf node R, the system updates the Riso-Tree in a similar way to the R-Tree. If the number of entries in R is still valid, the algorithm will just update the MBR of R and its ancestors if necessary. Otherwise, all the entries in R will be reinserted into the Riso-Tree.

5.2.1 Riso-Tree Maintenance Evaluation

Figure 27 shows the update performance for two different strategies in Foursquare dataset. Baseline is the strategy without using 'irrelevant vertex' and Irrelevant method is the one that takes the 'irrelevant vertex' strategy. We insert different numbers of edges into the graph, ranging from 1000 to 8000. Figure 27a shows the updating time when random edges are inserted. It can be observed that when the number of edges increases, the overall update time increases linearly. The average time is only around 10 ms for each insertion. It also reveals that the two strategies almost have the same performance for random edges. It is because very few edges

start and end in irrelevant vertexes. As a result, we also test the 'irrelevant edges' to evaluate the performance of the 'irrelevant vertex' strategy. Here 'irrelevant edges' means the edges that start and end in irrelevant vertexes. Figure 27a shows the update time for irrelevant edges. In this case, Irrelevant approach can achieve much better performance than Baseline approach.



(a) Random Edges

(b) Irrelevant Edges

Figure 27: Update performance on Foursquare

Chapter 6

A GEOSPATIAL KNOWLEDGE MANAGEMENT SYSTEM

In this chapter, 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 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.

6.1 System Overview

Figure 28 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.

6.1.1 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, Wikidata, etc. The graph data is managed by the graph database.

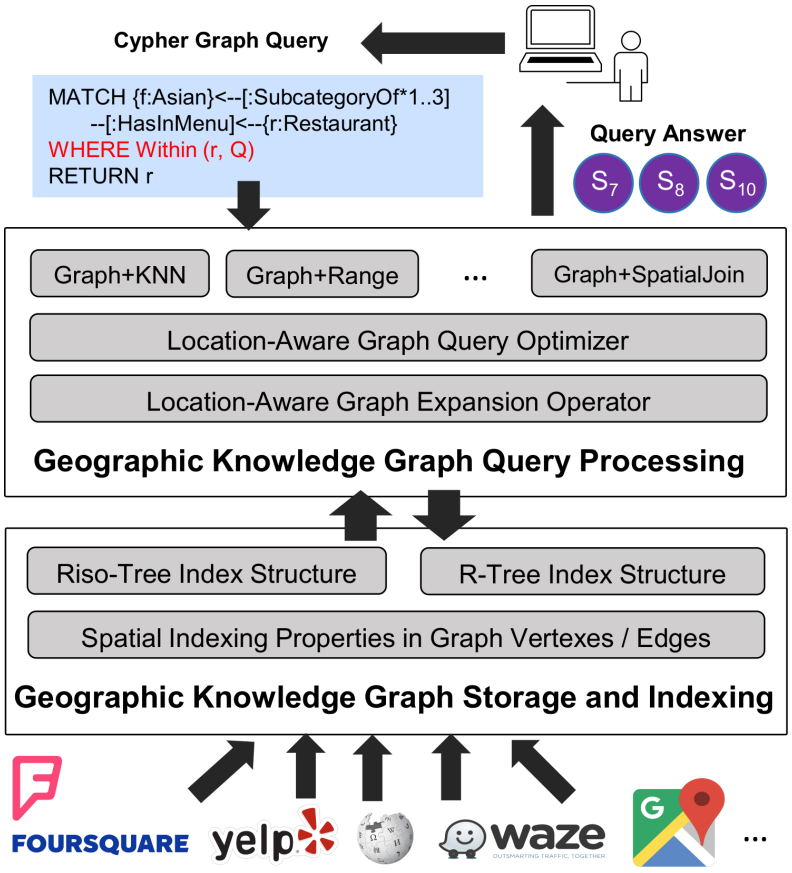


Figure 28: System architecture overview

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).

The spatial objects in the graph data are indexed by Riso-Tree index structure. Riso-Tree is a graph-aware spatial index. 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. The details of Riso-Tree are demonstrated in Chapter 4.

6.1.2 Query Processing Coordinator

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 a 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 a menu in Asian food and are close to a resort (e.g., the distance less than 1 km).

6.2 Scenario

In this section, we demonstrate real user scenarios by using Spindra. The Wikidata graph dataset [55] 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

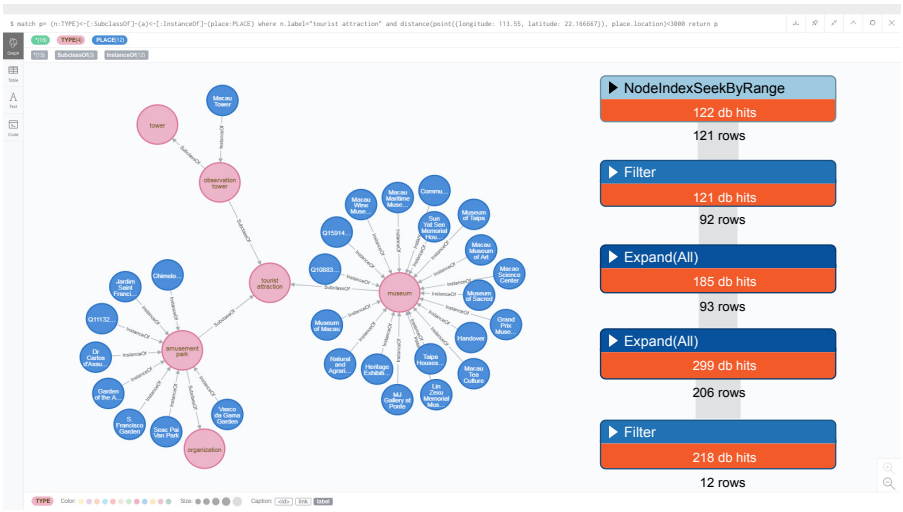


Figure 29: Backend Data View Interface

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 29 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 *SubclassOf* other categories. Normally, the query should only search for museums. However, there could be no museum nearby. In this scenario, the places whose categories are related to the 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 29 also shows the execution plan of the query by directly running a Cypher query in Neo4j. Because of the existence of the spatial index, spatial filtering will take be performed. The first two boxes in the plan reflect such a step. Such a 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 30 shows a web interface that 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 center by dragging the red marker

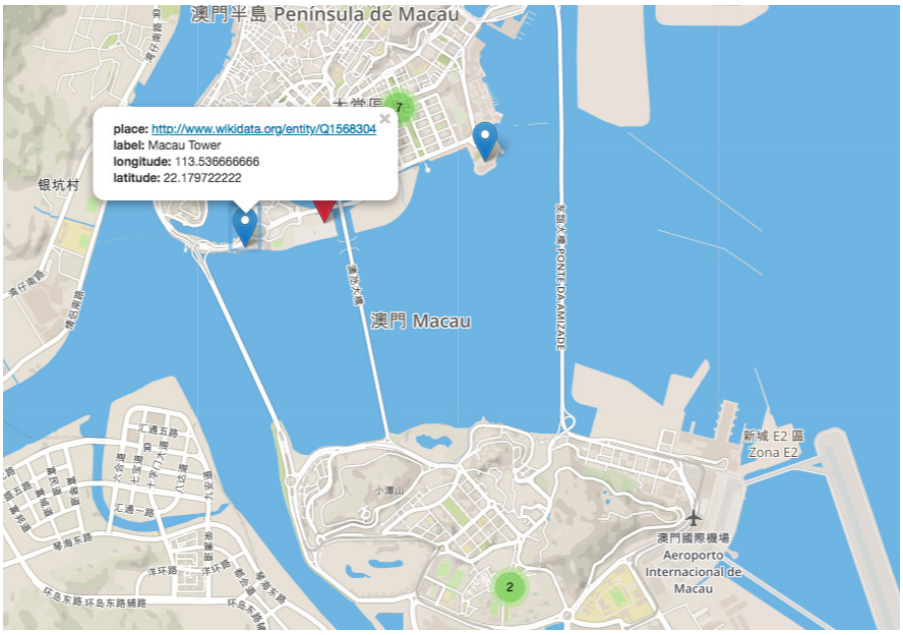


Figure 30: Query Result Map View

and change the search radius. The user can also change the search topic to Hotel, University, etc.

Chapter 7

CONCLUSION

In this thesis, we discuss geospatial graph queries which combine the subgraph isomorphism with three basic spatial predicates, including spatial range, spatial join, and spatial KNN predicates. Due to the limit of the existing strategies, two data structures, including the augmented graph data structure (SPA-Graph) and the graph-aware spatial index (Riso-Tree) are proposed to break the isolation between the graph database and spatial database. We also demonstrate the efficient query algorithm for accessing SPA-Graph and Riso-Tree respectively. Last but not least, different techniques are applied to the maintenance algorithm to provide support for graph updates.

There are several directions for future work. First, more workload of the experiment for edge insertion will be considered. Currently, the queries for evaluating the query time are generated randomly. It may not access the graph vertexes related to the edge insertion. The new query workload will specifically access the vertexes impacted by the edge insertion. Second, the cardinality estimation plays an important role in determining the optimized execution plan. In geospatial graph data, the cardinality estimation of either graph query or spatial query is not trivial. Moreover, the correlation between the spatial data and the graph data has an impact on the cardinality estimation. The existing graph database systems, like Neo4j, assume that there is no correlation for simplicity, which can lead to huge errors in estimating cardinality. The last direction is to consider the geospatial graph query in a distributed system. In the thesis, we only propose the techniques for a centralized system. How

to extend the proposed techniques to facilitate the new system architecture requires solving some new problems, like how to partition the geospatial graph data and how to maintain the index structure, etc.

REFERENCES

[1] Ablimit Aji et al. "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce". In: *Proc. VLDB Endow.* 6.11 (2013), pp. 1009–1020. URL: http://www.vldb.org/pvldb/vol6/p1009-aji.pdf.

[2] *ArcGIS Utility Network*. https://www.esri.com/en-us/arcgis/products/arcgis-utility-network-management/overview.

[3] Nikos Armenatzoglou, Stavros Papadopoulos, and Dimitris Papadias. "A general framework for geo-social query processing". In: *PVLDB* 6.10 (2013), pp. 913–924.

[4] Petko Bakalov, Erik G. Hoel, and Sangho Kim. "A Network Model for the Utility Domain". In: *SIGSPATIAL/GIS*. ACM, 2017, 32:1–32:10.

[5] Alexandru T. Balaban. "Applications of graph theory in chemistry". In: *Journal of Chemical Information and Computer Sciences* 25.3 (1985), pp. 334–343. URL: https://doi.org/10.1021/ci00047a033.

[6] Jie Bao, Mohamed F Mokbel, and Chi-Yin Chow. "Geofeed: A location aware news feed system". In: *ICDE*. IEEE. 2012, pp. 54–65.

[7] Norbert Beckmann et al. "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles". In: *SIGMOD*. 1990, pp. 322–331.

[8] Lu Chen et al. "Maximum Co-located Community Search in Large Scale Social Networks". In: *PVLDB* 11.10 (2018), pp. 1233–1246. URL: http://www.vldb.org/pvldb/vol11/p1233-chen.pdf.

[9] *Cypher Language*. https://neo4j.com/developer/cypher-query-language/.

[10] *DBpedia*. https://wiki.dbpedia.org/.

[11] Yerach Doytsher, Ben Galon, and Yaron Kanza. "Querying geo-social data by bridging spatial networks and social networks". In: *ACM LBSN*. 2010, pp. 39–46. URL: http://doi.acm.org/10.1145/1867699.1867707.

[12] Yerach Doytsher, Ben Galon, and Yaron Kanza. "Querying socio-spatial networks on the world-wide web". In: *WWW*. ACM. 2012, pp. 329–332.

[13] Yerach Doytsher, Ben Galon, and Yaron Kanza. "Storing routes in socio-spatial networks and supporting social-based route recommendation". In: *Proceedings*

of the 3rd ACM SIGSPATIAL International Workshop on Location-Based Social Networks. 2011, pp. 49–56.

[14] Ahmed Eldawy and Mohamed F. Mokbel. "SpatialHadoop: A MapReduce framework for spatial data". In: *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. Ed. by Johannes Gehrke et al. IEEE Computer Society, 2015, pp. 1352–1363. URL: https://doi.org/10.1109/ICDE.2015.7113382.

[15] Wenfei Fan. "Graph pattern matching revised for social network analysis". In: *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*. 2012, pp. 8–21. URL: https://doi.org/10.1145/2274576.2274578.

[16] Yixiang Fang et al. "Effective Community Search over Large Spatial Graphs". In: *PVLDB* 10.6 (2017), pp. 709–720. URL: http://www.vldb.org/pvldb/vol10/p709-fang.pdf.

[17] Yixiang Fang et al. "On Spatial-Aware Community Search". In: *IEEE Trans. Knowl. Data Eng.* 31.4 (2019), pp. 783–798. URL: https://doi.org/10.1109/TKDE.2018.2845414.

[18] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. "Keyword Search on Spatial Databases". In: *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*. Ed. by Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen. IEEE Computer Society, 2008, pp. 656–665. URL: https://doi.org/10.1109/ICDE.2008.4497474.

[19] *Freebase*. http://www.freebase.com.

[20] *GeoSPARQL*. http://www.opengeospatial.org/standards/geosparql.

[21] *Google's Knowledge Graph*. http://www.techwyse.com/blog/search-engine-optimization/seo-efforts-to-get-listed-in-google-knowledge-graph/.

[22] *GraphDB*. http://graphdb.ontotext.com.

[23] Helen M Grindley et al. "Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm". In: *Journal of molecular biology* 229.3 (1993), pp. 707–721.

[24] Antonin Guttman. "R-Trees: A Dynamic Index Structure For Spatial Searching". In: *SIGMOD*. 1984.

[25] Ramaswamy Hariharan et al. “Processing Spatial-Keyword (SK) Queries in Geographic Information Retrieval (GIR) Systems”. In: *19th International Conference on Scientific and Statistical Database Management, SSDBM 2007, 9-11 July 2007, Banff, Canada, Proceedings*. IEEE Computer Society, 2007, p. 16. URL: https://doi.org/10.1109/SSDBM.2007.22.

[26] Erik G. Hoel and Hanan Samet. “Benchmarking Spatial Join Operations with Spatial Output”. In: *VLDB*. 9, pp. 606–618.

[27] Jinsoo Lee et al. “An In-depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases”. In: *PVLDB* 6.2 (2012), pp. 133–144. URL: http://www.vldb.org/pvldb/vol6/p133-han.pdf.

[28] Zhisheng Li et al. “IR-Tree: An Efficient Index for Geographic Document Search”. In: *IEEE Trans. Knowl. Data Eng.* 23.4 (2011), pp. 585–599. URL: https://doi.org/10.1109/TKDE.2010.149.

[29] John Liagouris et al. “An Effective Encoding Scheme for Spatial RDF Data”. In: *PVLDB* 7.12 (2014), pp. 1271–1282.

[30] Kyriakos Mouratidis et al. “Joint search by social and spatial proximity”. In: *TKDE* 27.3 (2015), pp. 781–793.

[31] Inc. Neo4j. *Neo4j Java API*. https://neo4j.com/developer/java/. 2020.

[32] *Neo4j Graph Database*. https://neo4j.com/.

[33] Miles Ohlrich et al. “SubGemini: Identifying SubCircuits using a Fast Subgraph Isomorphism Algorithm”. In: *Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993*. 1993, pp. 31–37. URL: https://doi.org/10.1145/157485.164556.

[34] *OpenCyc*. http://sw.opencyc.org/.

[35] *Oracle Spatial and Graph*. https://www.oracle.com/database/technologies/spatialandgraph.html.

[36] Barak Pat, Yaron Kanza, and Mor Naaman. “Geosocial Search: Finding Places based on Geotagged Social-Media Posts”. In: *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015 - Companion Volume*. Ed. by Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi. ACM, 2015, pp. 231–234. URL: https://doi.org/10.1145/2740908.2742847.

[37] John W. Raymond and Peter Willett. "Maximum common subgraph isomorphism algorithms for the matching of chemical structures". In: *Journal of Computer-Aided Molecular Design* 16.7 (2002), pp. 521–533. URL: https://doi.org/10.1023/A:1021271615909.

[38] Xuguang Ren and Junhu Wang. "Multi-query Optimization for Subgraph Isomorphism Search". In: *Proc. VLDB Endow.* 10.3 (Nov. 2016), pp. 121–132. URL: https://doi.org/10.14778/3021924.3021929.

[39] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[40] Alberto Sanfeliu and King-Sun Fu. "A distance measure between attributed relational graphs for pattern recognition". In: *IEEE Trans. Systems, Man, and Cybernetics* 13.3 (1983), pp. 353–362. URL: https://doi.org/10.1109/TSMC.1983.6313167.

[41] Mohamed Sarwat and Yuhan Sun. "Answering Location-Aware Graph Reachability Queries on GeoSocial Data". In: *ICDE*. 2017.

[42] Mohamed Sarwat et al. "Context Awareness in Mobile Systems". In: *Data Management in Pervasive Systems*. 2015, pp. 257–287.

[43] Mohamed Sarwat et al. "LARS*: An Efficient and Scalable Location-Aware Recommender System". In: *TKDE* 26.6 (2014), pp. 1384–1399.

[44] Jieming Shi et al. "Density-based place clustering in geo-social networks". In: *SIGMOD*. ACM. 2014, pp. 99–110.

[45] Rohit Singh, Jinbo Xu, and Bonnie Berger. "Global alignment of multiple protein interaction networks with application to functional orthology detection". In: *Proceedings of the National Academy of Sciences* 105.35 (2008), pp. 12763–12768.

[46] Rachna Somkunwar and Vinod Moreshwar Vaze. "A Comparative Study of Graph Isomorphism Applications". In: *International Journal of Computer Applications* 162.7 (2017).

[47] Yuhan Sun, Nitin Pasumarthy, and Mohamed Sarwat. "On Evaluating Social Proximity-Aware Spatial Range Queries". In: *MDM*. 2017.

[48] Yuhan Sun and Mohamed Sarwat. "A generic database indexing framework for large-scale geographic knowledge graphs". In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information*

Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018. Ed. by Farnoush Banaei Kashani et al. ACM, 2018, pp. 289–298. URL: https://doi.org/10.1145/3274895.3274966.

[49] Yuhan Sun and Mohamed Sarwat. "A spatially-pruned vertex expansion operator in the Neo4j graph database system". In: *GeoInformatica* 23.3 (2019), pp. 397–423. URL: https://doi.org/10.1007/s10707-019-00361-2.

[50] Yuhan Sun, Jia Yu, and Mohamed Sarwat. "Demonstrating Spindra: A Geographic Knowledge Graph Management System". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. 2019, pp. 2044–2047. URL: https://doi.org/10.1109/ICDE.2019.00235.

[51] Konstantinos Theocharidis et al. "SRX: efficient management of spatial RDF data". In: *VLDB J.* 28.5 (2019), pp. 703–733. URL: https://doi.org/10.1007/s00778-019-00554-z.

[52] *Titan Distributed Graph Database*. http://titan.thinkaurelius.com/.

[53] *Virtuoso*. http://virtuoso.openlinksw.com.

[54] Akrivi Vlachou et al. "Efficient spatio-temporal RDF query processing in large dynamic knowledge bases". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. 2019, pp. 439–447. URL: https://doi.org/10.1145/3297280.3299732.

[55] Denny Vrandečić and Markus Krötzsch. "Wikidata: A Free Collaborative Knowledgebase". In: *Commun. ACM* 57.10 (Sept. 2014), pp. 78–85. URL: http://doi.acm.org/10.1145/2629489.

[56] Junhu Wang et al., eds. *Databases Theory and Applications - 29th Australasian Database Conference, ADC 2018, Gold Coast, QLD, Australia, May 24-27, 2018, Proceedings*. Vol. 10837. Lecture Notes in Computer Science. Springer, 2018. URL: https://doi.org/10.1007/978-3-319-92013-9.

[57] *WikiData*. https://www.wikidata.org/wiki/Wikidata:Main_Page.

[58] *YAGO*. https://yago-knowledge.org/.

[59] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. "Spatial data management in apache spark: the GeoSpark perspective and beyond". In: *GeoInformatica* 23.1 (2019), pp. 37–78. URL: https://doi.org/10.1007/s10707-018-0330-9.

[60] Yinghua Zhou et al. “Hybrid index structures for location-based web search”. In: *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*. Ed. by Otthein Herzog et al. ACM, 2005, pp. 155–162. URL: https://doi.org/10.1145/1099554.1099584.

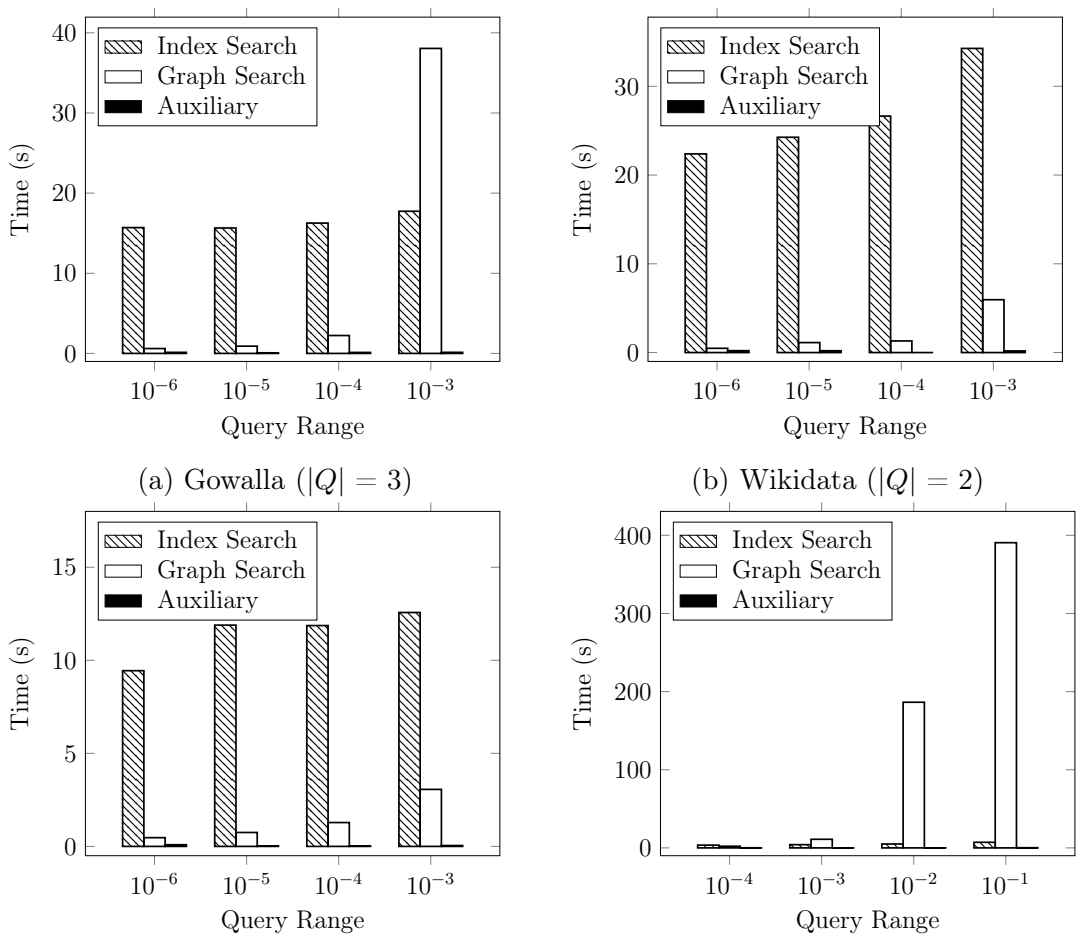
APPENDIX A

A.1 Composition of Riso query time in GraSp-Range

Figure 31 demonstrates the composition of Riso query time in GraSp-Range (Figure 16). The query time of Riso mainly consists of the time of Riso-Tree search, graph search and auxiliary operations, like recognize APaths and query rewrite. The time of auxiliary operations is negligible in all cases. The query time of Riso are determined by the time of Riso-Tree search and graph search. The time of Riso-Tree search is influenced by the spatial selectivity. When the query range contains more spatial vertexes, the search time tends to increase. The time of graph search has similar trend. But they increase at a different speed. The time of graph search is also influenced by the complexity of the data graph and query graph. The query graphs used in this experiment have similar complexity (query graph size). The difference lies in the complexity of each graph dataset. Yelp dataset has the highest average degree for whole dataset and spatial vertexes. So the time of graph search in Yelp dataset increase fast and it dominates the query time of Riso. In the rest datasets, the Riso-Tree search time is more influential in most of the scenarios.

A.2 Example Queries in Experiments

Figure 32, 33, 34 show the example queries for GraSp-Range, GraSp-KNN and GraSp-KNN respectively. Different colors are are used to identify the labels of query vertexes. The query vertexes in the spatial predicate are colored with black. Figure 32 shows two queries of GraSp-Range of size 3. The query pattern is after the "MATCH" keyword. In the query on the left side, three query vertexes a_0, a_1 and a_2 have the labels of "high school", "island nation" and "human" respectively. This query searches the qualified subgraphs that the vertex with label "high school" is located within the region (134.94, 34.96, 134.97, 34.98). In the query on the right side, the spatial range predicate is with the center vertex a_0. Figure 33 demonstrates two queries of GraSp-KNN of size 3. The left-side query searches the K subgraphs that the "mountain" vertex is closest to the location (69.46, 32.65). The right-side query has the same number of query vertexes but the spatial predicate is on the center vertex a_0. Figure 34 demonstrates a query of GraSp-Join of size 5. The join predicate is on vertex a_0 and a_3 that the distance between a_0 and a_3 is less than 0.0001 (degree).



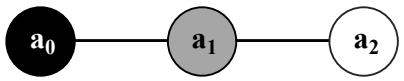
(a) Gowalla ($|Q| = 3$)

(b) Wikidata ($|Q| = 2$)

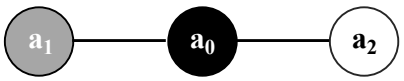
(c) Foursquare ($|Q| = 3$)

(d) Yelp ($|Q| = 3$)

Figure 31: Composition of GraSp-Range query response time

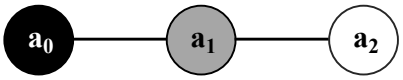



```
MATCH (a0:`high school`),
   (a1:`island nation`), (a2:`human`),
   (a0)--(a1), (a1)--(a2)
WHERE 134.94 <= a0.lon <= 134.97
   and 34.96 <= a0.lat <= 34.98
RETURN id(a0), id(a1), id(a2)
```




```
MATCH (a0:`building`),
   (a1:`old town`), (a2:`country`),
   (a0)--(a1), (a0)--(a2)
WHERE 8.800756 <= a0.lon <= 8.802922
   and 53.077614 <= a0.lat <= 53.079780
RETURN id(a0), id(a1), id(a2)
```

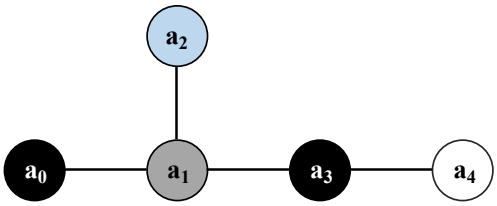
Figure 32: Example query for GraSp-Range




```
MATCH (a0:`mountain`),
   (a1:`country`), (a2:`river`),
   (a0)--(a1), (a1)--(a2)
RETURN id(a0), id(a1), id(a2)
ORDER BY dist (a0, (69.46, 32.65))
LIMIT K
```


```
MATCH (a0:`hamlet`), (a2:`great power`),
   (a1:`rural settlement of Russia`),
   (a0)--(a1), (a0)--(a2)
RETURN id(a0), id(a1), id(a2)
ORDER BY dist (a0, (35.09, 62.29))
LIMIT K
```

Figure 33: Example query for GraSp-KNN




```
MATCH (a0:`street`),
   (a1:`country of the Kingdom of the Netherlands`),
   (a2:`place with town rights and privileges`),
   (a3:`street`), (a4:`human settlement`),
   (a0)--(a1), (a0)--(a2), (a1)--(a3), (a1)--(a4)
WHERE dist (a0, a3) < 0.0001
RETURN id(a0), id(a1), id(a2), id(a3), id(a4)
```

Figure 34: Example query for GraSp-Join