Property Graph Schema Optimization for Domain-Specific Knowledge Graphs

Rana Alotaibi[1], Chuan Lei[2], Abdul Quamar[2], Vasilis Efthymiou[2], Fatma Özcan[2]
[1]University of California - San Diego, La Jolla, CA, USA
[2]IBM Research - Almaden, San Jose, CA, USA
[1]ralotaib@eng.ucsd.edu, [2]chuan.lei|vasilis.efthymiou@ibm.com, [2]ahquamar|fozcan@us.ibm.com

***Abstract*—Enterprises are creating domain-specific knowledge graphs by curating and integrating their business data from multiple sources. Ontologies provide a semantic abstraction for such knowledge graphs to describe their data in terms of the entities involved and their relationships. There has been a lot of effort to build systems that enable efficient querying over knowledge graphs, represented as property graphs. However the problem of schema optimization in the property graph setting has been largely ignored. In this work, we show that graph schema design has significant impact on query performance, and propose two algorithms to generate an optimized property graph schema from the domain ontology. To the best of our knowledge, we are the first to present an ontology-driven approach for property graph schema optimization. The rich semantic relationships in an ontology contain a variety of opportunities to reduce edge traversals and consequently improve the graph query performance. Our experimental study with two real-world knowledge graphs shows that our algorithms produce high-quality schemas, achieving up to 2 orders of magnitude speed-up compared to alternative schema designs.**

I. Introduction

Domain-specific knowledge graphs are playing an increasingly important role to derive business insights in many enterprise applications such as customer engagement, fraud detection, network management, etc. One distinct characteristic of these enterprise knowledge graphs, compared to the open-domain knowledge graphs like DBpedia [1] and Freebase [2], is their deep domain specialization. The domain specialization is typically captured by an ontology which provides a semantic abstraction to describe the entities and their relationships of the data in the knowledge graphs. A few widely used domain-specific ontologies include Unified Medical Language System (UMLS)[1] and SNOMED Clinical Terms[2] in the medical domain, Financial Industry Business Ontology (FIBO)[3] and Financial Report Ontology (FRO)[4] in the financial domain, and many more in various other domains[5]. These ontologies are often used to drive the creation of a knowledge graph by ingesting and transforming raw data from multiple sources into standard terminologies. The curated knowledge graphs allow users to express their queries in standard vocabularies, which promotes more interoperable and effective enterprise applications for specific domains [3].

There are two popular approaches to store and query knowledge graphs: RDF data model and SPARQL query language [4] or property graph model and graph query languages such as Gremlin [5] and Cypher [6]. An important difference between RDF and property graphs is that RDF regularizes the graph representation as a set of triples, which means that even literals are represented as graph vertices. Such artificial vertices make it hard to express graph queries in a natural way. The property graph model instead uses vertices to represent entities and edges to represent the relationships between them, with each specified using key-value properties pairs [7]. For this reason, property graph systems such as Neo4j[6], JanusGraph[7], Amazon Neptune[8], and Db2 Graph [8] are rapidly gaining popularity for graph storage and retrieval. Many graph applications (e.g., community detection, centrality analysis, and link prediction) heavily rely on the performance of graph queries over the property graph systems. Although many techniques have been proposed for optimizing query performance, system scalability, and transaction support for these systems [9]–[12], the problem of property graph schema [13] optimization has been largely ignored, which is also critical to graph query performance.

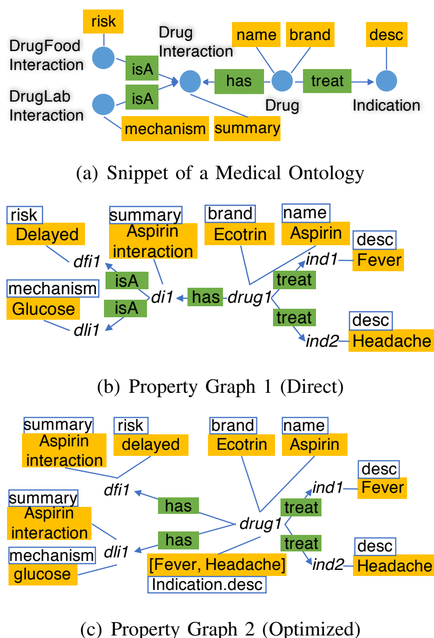
In this paper, we tackle the *property graph schema optimization problem* for domain-specific knowledge graphs. Our goal is to create an optimized schema[9] using a given ontology, such that the corresponding property graph can efficiently support various types of graph queries (e.g., pattern matching, path finding, or aggregation queries) with better query performance. The raw data is loaded directly as a property graph that conforms to the optimized schema. One straightforward way to create a property graph schema from an ontology is to directly map each ontology concept to a schema node, and to map each ontology relationship to a schema edge, analogous to ER diagram to relational schema mapping. However, we argue that the graph query performance varies vastly for different property graphs with the same data but corresponding to different schemas, and the rich semantic information in the ontology

[1]https://www.nlm.nih.gov/research/umls/index.html
[2]http://www.snomed.org/
[3]https://spec.edmcouncil.org/fibo/
[4]http://www.xbrlsite.com/2015/fro/us-gaap/xbrl/Ontology/Overview.html
[5]https://lod-cloud.net/
[6]Neo4j Graph Database - https://neo4j.com/
[7]JanusGraph - https://janusgraph.org/
[8]Amazon Neptune - https://aws.amazon.com/neptune/
[9]We use the terms property graph schema, graph schema, and schema interchangeably.

2375-026X/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00085

provides unique opportunities for schema optimization. We illustrate this using two examples from the medical domain.

Example 1 (Pattern matching query). Consider the ontology in Fig. 1(a), *summary* is a property of *DrugInteraction* concept, which is connected to *DrugFoodInteraction* and *DrugLabInteraction* concepts via inheritance (*isA*) relationships. Fig. 1(b) and Fig. 1(c) show two alternative property graphs conforming to two different schemas. In Fig. 1(b), the vertex *di1* (i.e., an instance of *DrugInteraction*) leads to both *dfi1* and *dli1*. In Fig. 1(c), *drug1* directly connects to *dfi1* and *dli1*. For any query that requires edge traversals from *drug1* to either *dfi1* or *dli1* or both, the property graph 2 requires less number of edge traversals. A pattern matching query interested in *Drug* and the associated *risk* of *DrugFoodInteraction* achieves 2 orders of magnitude performance gains on the optimized property graph (23ms) compared to the property graph 1 (3245ms).



(a) Snippet of a Medical Ontology

(b) Property Graph 1 (Direct)

(c) Property Graph 2 (Optimized)

Fig. 1: Motivating Example.

Example 2 (Aggregation query). In Fig. 1(a), *Drug* concept is also connected to *Indication* concept via a *treat* (1:M) relationship. In this case, we observe that if we replicate certain properties accessible via a *1:M* relationship, edge traversals can be avoided. Fig. 1(c) shows that the vertex *drug1* has an additional property, which is a list of descriptions replicated from the property *desc* of *ind1* and *ind2*. An aggregation query (COUNT) on the *desc* of *Indication* treated by *Drug* runs 8 times faster on this optimized property graph (78ms) than the property graph 1 (627ms). Hence, avoiding the edge traversals is extremely beneficial, especially when the number of edges between two types of vertices is large.

These two examples show that edge traversal is one of the dominant factors affecting graph query performance, and having an optimized schema can greatly improve query performance. We can reduce edge traversals by merging nodes or replicating data. However, this needs to be done carefully, as the resulting knowledge graph needs to preserve its semantics information. Fortunately, ontologies with their rich semantic information provide a variety of opportunities to reduce graph traversals. To generate an optimized graph schema, we need to identify these opportunities and design different techniques to exploit them in the ontology accordingly. Certain optimization techniques require data replication resulting in space overheads. Hence, the schema optimization also has to consider the trade-off between the query performance and the space consumption of the resulting property graph.

Our proposed approach. To the best of our knowledge, we are the first to address the problem of property graph schema optimization to improve graph query performance. In addition to the ontology, our approach also takes into account the space constraints, if any, and additional information such as data distribution and workload summaries[10]. We propose a set of rules that are designed to optimize the graph query performance with respect to different types of relationships in the ontology. When there is a space constraint, we estimate the cost-benefit of applying these rules to each individual relationship by leveraging the additional data distribution and workload information. We propose two algorithms, concept-centric and relation-centric, which incorporate the cost-benefit scores to produce an optimized property graph schema.

Contributions. The contributions of this paper can be summarized as follows:

1. We propose an ontology-driven property graph schema optimization approach for domain-specific knowledge graphs.

2. We design a set of rules that reduce the edge traversals by exploiting the rich semantic relationships in the ontology, resulting in better graph query performance.

3. We propose concept-centric and relation-centric algorithms that harness the proposed rules to generate an optimized property graph schema from an ontology. The concept-centric algorithm utilizes the centrality analysis of concepts, and the relation-centric algorithm uses a cost-benefit model.

4. Our experiments show that our ontology-driven approach effectively produces optimized graph schemas for two real-world knowledge graphs from medical and financial domains. The queries over the optimized property graphs achieve up to 2 orders of magnitude performance gains compared to the graphs resulting from the baseline approach.

The rest of the paper is organized as follows. Section II introduces the basic concepts, formulates the problem, and provides an overview of our ontology-driven approach. Section III describes our optimization rules for different types of relationships in an ontology. Section IV explains the algorithms to produce optimized property graph schema. We provide our experimental results in Section V, review related work in Section VI, and finally conclude in Section VII.

[10]We refer to the access frequency of concepts, relationships and properties as workload summaries which will be formally defined later.

II. Preliminaries & Approach Overview

A. Preliminaries

An ontology describes a particular domain and provides a structured view of the data. Specifically, it provides an expressive data model for the concepts that are relevant to that domain, the properties associated with the concepts, and the relationships between concepts.

Definition 1 (Ontology ($\mathcal{O}$)). *An ontology $\mathcal{O}$ (C, R, P) contains a set of concepts $C = \{c_n | 1 \leq n \leq N\}$, a set of data properties $P = \{p_m | 1 \leq m \leq M\}$, and a set of relationships between the concepts $R = \{r_k | 1 \leq k \leq K\}$.*

An ontology is typically described in OWL [14], wherein a concept is defined as a *class*, a property associated with a concept is defined as a *DataProperty* and a relationship between a pair of concepts is defined as an *ObjectProperty*. Each DataProperty $p_i \in P_n$ represents a characteristic of a concept $c_n \in C$ and $P_n \subseteq P$ represents the set of DataProperties associated with the concept c_n. Each ObjectProperty $r_k = (c_s, c_d, t)$ is associated with a source concept $c_s \in C$, also referred to as the domain of the ObjectProperty, a destination concept $c_d \in C$, also referred to as the range of the ObjectProperty, and a type t. The type t can be either a functional (i.e., *1:1*, *1:M*, *M:N*), an inheritance (a.k.a *isA*) or a union/membership relationship[11]. In this paper, we use the ontology as a semantic data model of a knowledge graph.

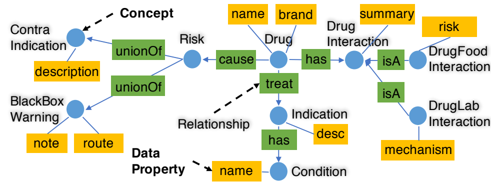


Fig. 2: Medical Ontology.

We adopt the widely used property graph model from [15].

Definition 2 (Property Graph ($\mathcal{PG}$)). *A property graph $\mathcal{PG}$ (V, E) is a directed multi-graph with vertex set V and edge set E, where each node $v \in V$ and each edge $e \in E$ has data properties consisting of multiple attribute-value pairs.*

Similar to a relational database schema that describes tables, columns, and relationships of a relational database, the property graph schema is critical and beneficial for creating high-quality domain-specific graphs. A property graph with an underlying schema enables graph query optimization, and allows definition of constraints as well as data exchange [13].

A property graph schema $\mathcal{PGS}$ can be specified in a data definition language such as Neo4j's Cypher [6], TigerGraph's GSQL [16], or GraphQL SDL [17]. They all define notions of node types and edge types, as well as property types that are associated with a node type or with an edge type. We adopt Cypher due to its popularity, but our proposed techniques are independent of the aforementioned languages. Table I provides the notations used in this paper.

[11]Even if inheritance and union are not ObjectProperties, we simplify the notation for presentation purposes.

TABLE I: Notations.

Notations	Definitions
$\mathcal{O}$	an ontology
c_i	$c_i \in C$: a concept in an ontology
r_i	$r_i \in R$: a relationship in an ontology
$c_i.P_i$	all data properties associated to c_i
$c_i.inE$	all incoming relationships of c_i
$c_i.outE$	all outgoing relationships of c_i
$c_i.R_i$	$c_i.R_i$ = $c_i.inE \cup c_i.outE$
$r_i.src$	the source concept of r_i
$r_i.dst$	the destination concept of r_i
$r_i.type$	the relationship type of r_i (i.e., *1:1*, *union*, *inheritance*, *1:M*, or *M:N*)
$\mathcal{PGS}$	a property graph schema
vs_i	a node schema defined in $\mathcal{PGS}$
$vs_i.P_i$	all data properties associated to vs_i
es_i	an edge schema defined in $\mathcal{PGS}$
$es_i.type$	the edge type of e_i
$\mathcal{PG}$	a property graph

B. Approach Overview

Given an ontology $\mathcal{O}$ providing a semantic abstraction of the input data, the problem of *property graph schema optimization* is to generate a property graph schema that produces the best query performance for various graph queries (e.g., pattern matching, path finding, or aggregation queries). Optimizing the property graph might entail data replication and hence increased memory footprint. In real knowledge graph applications, especially in a multi-tenant setting, there is a limit on the amount of memory that we can trade for query performance. Hence, we need to incorporate a space constraint while producing an optimized property graph schema.

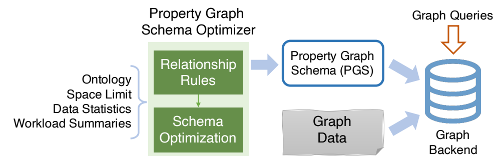


Fig. 3: Approach Overview.

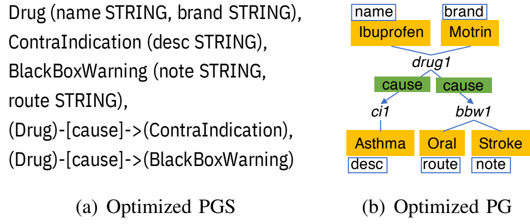
Fig. 3 provides an overview of our property graph schema optimization approach. The property graph schema optimizer takes as input an ontology and optionally a space limit, data statistics, as well as workload summaries[12]. It utilizes a set of rules designed for different types of relationships to produce an optimized property graph schema. The raw graph data is then loaded into a graph database (e.g., Neo4j or JanusGraph) conforming to the optimized schema. At query time, users can directly expresses graph queries against this instantiated property graph corresponding to the optimized schema.

[12]Access frequencies of concepts, relationships, and data properties in an ontology

III. Relationship Rules

Graph queries often involve multi-hop traversal or vertex attribute lookup/analytics on property graphs. As shown in the motivating examples, edge traversals over a graph are vital to the overall query performance. Hence, we focus on the rich semantic relationships in an ontology and propose a set of novel rules for different types of relationships, leading to significant performance improvements on the most common graph query patterns [18]. We briefly describe these rules below and the details can be found in [19].

Union Rule. In an ontology, a union relationship ($r_{un} = (c_i, c_j)$) contains a union concept (c_i) and a member concept (c_j). Each instance of a union concept is an instance of one of its member concepts, and vice versa. If we create a property graph directly from the ontology shown in Fig. 2, then the queries starting from any vertices of either *BlackBoxWarning* or *ContraIndication* concepts have to traverse through some vertex of *Risk* in order to reach the vertices of *Drug*. This leads to unnecessary edge traversal.



(a) Optimized PGS (b) Optimized PG

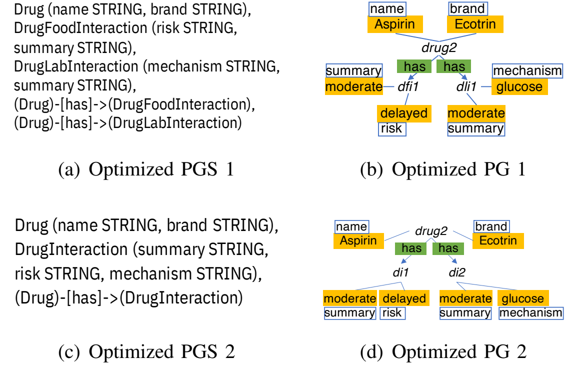
Fig. 4: Union Relationship.

Hence we propose a union rule to alleviate this issue. The union rule first creates a union node vs_i (based on the corresponding c_i in $\mathcal{O}$) and its member node vs_j (based on the corresponding c_j in $\mathcal{O}$) in the property graph schema. Then the member node vs_j is connected to the other nodes that connect to the union node vs_i in the property graph schema. Fig. 4(a) and Fig. 4(b) show the property graph schema and the corresponding property graph after applying the union rule to the above example.

Inheritance Rule. An inheritance relationship ($r_{ih} = (c_i, c_j)$) contains a parent concept (c_i) and a child concept (c_j). Similar to the union rule, we create a parent node vs_i (corresponding to c_i) and its child node vs_j (corresponding to c_j) in the property graph schema. Unlike a union concept, a parent concept in the inheritance relationship may have instances that are not present in any of its children concepts. This leads to the following three cases.

1) Connect the child node directly to the nodes that are connected to its parent node, and attach all data properties $vs_i.P_i$ of vs_i to the child node vs_j in the schema;
2) Connect the parent node directly to the nodes that are connected to its child node, and attach all data properties $vs_j.P_j$ of vs_j to the parent node vs_i in the schema;
3) Or connect the parent vs_i and child vs_j nodes with an edge of type *isA*.

Fig. 5(a) and Fig. 5(b) demonstrate the first scenario where the data properties (*summary*) of the parent node *DrugInteraction* are directly attached to two children nodes *DrugFoodInteraction* and *DrugLabInteraction*. Fig. 5(c) and Fig. 5(d) depict the second case where the data properties *risk* and *mechanism* of two respective child nodes are now attached to the parent node *DrugInteraction*.



(a) Optimized PGS 1 (b) Optimized PG 1

(c) Optimized PGS 2 (d) Optimized PG 2

Fig. 5: Inheritance Relationship.

However, attaching data properties to a parent or child node incurs data replication. Hence we exploit the Jaccard similarity between $c_i.P_i$ and $c_j.P_j$ from the given ontology $\mathcal{O}$ to decide the best strategy for the inheritance relationship:

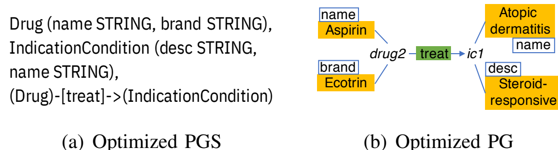
$$JS(c_i.P_i,\ c_j.P_j) = |c_i.P_i \cap c_j.P_j| \ / \ |c_i.P_i \cup c_j.P_j|. \quad (1)$$

If $JS(c_i.P_i, c_j.P_j) \geq \theta_1$, attaching the data properties from the child node to the parent node incurs less space overhead compared to the other way. If $JS(c_i.P_i, c_j.P_j) \leq \theta_2$ ($\theta_2 \leq \theta_1$), it is more cost effective to make the data properties of the parent node available at the child node. The inheritance rule avoids edge traversals in the resulting property graph.

One-to-one Rule. A *1:1* relationship ($r_{1:1} = (c_i, c_j), c_i{\neq}c_j$) indicates that an instance of c_i can only relate to one instance of c_j and vice versa. Two concepts (c_i and c_j) of a *1:1* relationship can be represented as one combined node vs_{ij} in the optimized schema, which is similar to joining two tables in relational databases where one row in one table is linked with only one row in another table and vice versa. If two tables are merged, then a join can be saved when two tables are queried together. The other tables can still join with the merged table through their respective relationships. Note that the two concepts that are merged to create a new combined node are not removed from the original graph. Namely, the *1:1* rule preserves the original semantics and does not lead to any information loss.

In Fig. 6(a), *IndicationCondition* is the node with two data properties, *name* and *note*, attached, representing *Indication* and *Condition*. Hence the edge traversal (e.g., from *Drug* to *Condition* in Fig. 2) is avoided and the number of instance vertices (i.e., space consumption) is reduced as well.

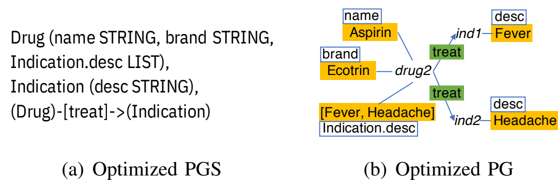
One-to-many Rule. A *1:M* relationship ($r_{1:M}$ = (c_i, c_j)) indicates that an instance of c_i can potentially refer to several



(a) Optimized PGS (b) Optimized PG

Fig. 6: 1:1 Relationship.

instances of c_j). However, an instance of c_j cannot have more than one corresponding instance of c_i. To better support the aggregation (e.g., COUNT, SUM, AVG, etc.) and neighborhood (1-hop) lookup functions in graph queries, we first create two nodes vs_i and vs_j corresponding to c_i and c_j in the optimized schema. Then we propagate each data property $vs_j.P_j$ of vs_j as a property of type *LIST* to the other node vs_i (Fig. 7(a)). This is similar to denormalization technique in relational databases where data replication is added to one or more tables in order to avoid costly joins.



(a) Optimized PGS (b) Optimized PG

Fig. 7: 1:M Relationship.

As depicted in Fig. 7(b), *Indication.desc* is a data property of *drug2* consisting of a list of descriptions (i.e., *[Fever, Headache]*) that saves the aggregation queries edge traversals to the other instance vertices (e.g., *ind1* and *ind2*). However, the newly introduced property of type *LIST* introduces additional space overheads. Therefore, choosing the appropriate set of data properties from each *1:M* relationship is critical to both query performance and space consumption.

Many-to-many Rule. An M:N relationship ($r_{M:N}$ = (c_i, c_j)) is essentially equivalent to two 1:M relationships, namely, $r_{1:M} = (c_i, c_j)$ and $r_{1:M} = (c_j, c_i)$. Therefore, the many-to-many rule is identical to the one-to-many rule, except that the property propagation is done for both directions. Hence applying the many-to-many rule leads to the same potential gains for queries with aggregate or neighborhood (1-hop) lookup functions at the cost of introducing space overhead.

In summary, all proposed rules reduce the number of edge traversals which improve graph query performance. Moreover, these rules can be utilized in graph systems using different storage backends. The potential benefits could be more significant when the storage backend changes from in-memory to disk as edge traversals may incur additional disk I/Os. However, these rules may incur space overheads. In Section IV, we describe our property graph schema optimization algorithms, trading off performance gain and space overhead.

IV. Property Graph Schema Optimization

In this section, we first introduce a property graph schema optimization algorithm in an ideal scenario (i.e., no space constraints). Then, we describe our concept-centric and relation-centric algorithms that harness the proposed rules and a cost-benefit model to generate an optimized property graph schema for a given space constraint.

A. Optimization Without Space Constraints

To produce an optimized property graph schema, we need to determine how to utilize the proposed rules described in Section III. A straightforward approach is to iteratively apply these rules in order and generate the property graph schema. We provide an overview of the schema optimization algorithm below and the details can be found in [19].

The proposed schema optimization algorithm takes as input an ontology $\mathcal{O}$ and first computes the Jaccard similarity scores for all inheritance relationships. Then, it iteratively applies the appropriate rule to each relationship in the ontology. At the end of each iteration, it checks if the ontology converges. Finally, when no more rule applies, a property graph schema is generated. In fact, these rules can be applied in any order, and the generated property graph schema is always the same.

Theorem 1. *Applying the union, inheritance, 1:M and M:N rules in any order produces a unique $\mathcal{PGS}$, if there is no space constraint.*

Proof. The proof can be found in the technical report [19]. □

B. Schema Optimization With Space Constraints

While the naïve approach harnesses all potential optimization opportunities aggressively, it incurs space overhead from *union*, *inheritance*, *1:M*, and *M:N* rules. This can be expensive, especially in a multi-tenant architecture, where many large-scale property graphs co-exist. Hence our goal is to produce an optimized property graph schema for a given space limit. To measure the quality and the space consumption of an optimized property graph schema, we leverage additional information such as data and workload characteristics.

Data characteristics contain the basic statistics about each concept, data property, and relationship specified in the given ontology. The statistics include the cardinality of data instances of each concept and relationship, as well as the data type of each data property. The data characteristics reflect the *cost* of applying the proposed rules to each relationship.

Access frequencies provide an abstraction of the workload in terms of how each concept, relationship, and data property accessed by each query in the workload. We use *AF*($c_i \xrightarrow{r_k} c_j.P_j$) to indicate the frequency of queries (the number of queries) that access a data property in $c_j.P_j$ from the concept c_i through the relationship r_k. The high frequency of a relationship indicates its relative importance among all relationships in the given ontology. Hence the access frequencies reveal the *benefit* of optimizing each relationship.

Definition 3 (Optimal Property Graph Schema)**.** *Let $\mathbb{PGS}$ be the set of all property graph schemas, such that $\forall \mathcal{PGS}' \in$*

$\mathbb{PGS}$ *we have* $Cost(\mathcal{PGS}') \leq S$, *where* S *is a given space constraint.* $\mathcal{PGS}_{opt} \in \mathbb{PGS}$ *is an optimal property graph schema if* $\nexists\ \mathcal{PGS}' \in \mathbb{PGS}$ *such that Benefit(*$\mathcal{PGS}'$*)* $>$ $Benefit(\mathcal{PGS}_{opt})$.

An optimal property graph schema dictates which subsets of relationships and their associated concepts are chosen to optimize for a given space constraint S. The search space thus consists of all possible combinations of relationship subsets, which is exponential in the number of relationships in a given ontology. As shown later in Proposition 1, finding an optimal property graph schema is NP-hard. Hence, we need to design efficient heuristics to produce a near-optimal property graph schema. To achieve this goal, we design a cost-benefit model (Eqs. 2-5) to capture the above described information and propose two property graph schema optimization algorithms driven by the cost-benefit model.

1) Concept-Centric Algorithm: As described in Section II, an ontology describes a particular domain and provides a concept-centric view over domain-specific data. Intuitively, some concepts are more critical to the domain, and have more relationships with the other concepts [20]. We expect these key concepts to be queried more frequently than other concepts, which is confirmed in [21]. This leads to our concept-centric algorithm that exploits the structural information in an ontology to identify key concepts which we believe are more likely to be accessed more often. Hence, this algorithm is useful when no workload summary is available.

To determine these key concepts, we utilize centrality analysis over the ontology to rank all concepts according to their respective centrality score. The centrality analysis is based on the commonly used PageRank algorithm [22] (a variant of Eigenvector centrality) as our intuition of key concepts is similar to the importance of website pages. Compared to PageRank, our *OntologyPR* (Algorithm 1) further introduces weights for both in and out degrees of concepts in determining their centrality scores.

Inheritance. We first remove inheritance relationships from the ontology while running the initial PageRank algorithm. This allows us to calculate the rank of a concept based on its relationships to non-hierarchical concepts. After computing the PageRank scores of all concepts, we re-attach the inheritance relationships and update the score of each concept by finding the parent with the highest score. The intuition is that a child concept inherits all its other properties from the same chain of concepts and hence would have a similar estimate of centrality.

Unions. For each incoming relationship to a union concept, we create new relationship between the source concept and each of the member concepts of the union. For each outgoing relationship, similarly, we create new relationships between the destination and each of the member concepts of the union. Thus the page rank mass is appropriately distributed to/from the member nodes of the union. Finally, the union node itself is removed from the graph as its contribution towards centrality analysis has already been accounted for by the new relationships to/from the member concepts of the union.

Out-degree of Concepts. In the default PageRank algorithm, the weight distribution is proportional to the in-degree of a node as it receives PageRank values from all its neighbors that point to it. However, for a domain ontology, we observe that both in-degree and out-degree are equally important in terms of the key concept. Hence, we introduce a reverse edge in the ontology, essentially making the graph equivalent to an undirected graph. Then, the $OntologyPR$ algorithm uses this modified ontology as an input to determine the centrality score of each concept.

Algorithm 1 Ontology PageRank Algorithm (OntologyPR)

```
Input: 𝒪 = (C, R, P)
Output: 𝒪 = (C, R, P)
 1: C_un ← empty set
 2: for each r ∈ R of type union do
 3:     c_i ← r.src                          // the union concept of r
 4:     c_j ← r.dst                          // the member concept of r
 5:     C_un.add(c_i)
 6:     c_j.R_j ← (c_j.R_j ∪ c_i.R_i)\r
 7: O.remove(C_un)
 8: for each r ∈ R do
 9:     if r is of type inheritance then
10:         R_ih.add(r)
11:         O.remove(r)
12:     else
13:         O.add(r')                        // add a reverse relation r'
14: pageRank(O)                  // PageRank on the modified ontology
15: O.add(R_ih)                  // add inheritance relationships back
16: updatePR(O) // update PageRank score for inheritance concepts
17: return O                     // O associated with PageRank scores
```

Using $OntologyPR$, we associate PageRank scores with each concept in the ontology. To accurately capture the relative importance of the concepts, we further leverage the *data characteristics* and *access frequency* information to rank all concepts. The ranking score for a concept is defined as follows.

$$Score(c_i) = \frac{c_i.pr \times AF(c_i)}{Size(c_i)}, \tag{2}$$

where $c_i.pr$ denotes the PageRank score of c_i, $AF(c_i)$ denotes the access frequency of c_i including accessing all data properties of c_i, and $Size(c_i)$ denotes the size of c_i including all data properties of c_i.

Algorithm 2 Concept-Centric Algorithm

```
Input: Ontology 𝒪 = (C, R, P), space limit S
Output: A property graph schema 𝒫𝒢𝒮
 1: O ← ontologyPR(O)
 2: C_srt ← sort(C)
 3: for each c ∈ C_srt do
 4:     for each r ∈ c.R do
 5:         S' ← S
 6:         O, S ← applyRules(r, S')
 7:         if S < 0 then
 8:             break
 9: PGS ← generatePGS(O)
10: return PGS
```

Based on Eq. 2, our concept-centric algorithm (Algorithm 2) first sorts all concepts in a descending order of their respective

scores (Lines 1-2). Then, it iterates through each concept c (Lines 3-8). For each concept, the algorithm utilizes the $applyRules$ procedure to apply all rules (Section III) to the relationships connecting to c. During this process, the algorithm updates the space limit as it is consumed by the rules. If the space is fully exhausted, the algorithm terminates and returns the optimized property graph schema (Lines 7-10).

Complexity Analysis. The *OntologyPR* is the dominant procedure in Algorithm 2, and its time complexity is $O((|R|+|C|)k)$, where $|R|$ is the number of relationships, $|C|$ is the number of concepts, and k is the maximum number of iterations. The time complexity of sorting concepts is $O(|C|log|C|)$. Finally, the time complexity of applying rules to the sorted concepts is $O(|R|)$. Thus, the overall time complexity of Algorithm 2 is $O((|R|+|C|)k)$.

2) Relation-Centric Algorithm: Intuitively, the concept-centric algorithm prioritizes the relationships of the key concepts in an ontology by leveraging information such as access frequency, data characteristics, and structural information from the ontology. However, the relationship selection is limited to each concept locally. Namely, the concept-centric algorithm does not have a global optimal ordering among all relationships in the ontology. To address this issue, we propose the relation-centric algorithm based on a cost-benefit model for each type of relationships described as follows.

Cost Benefit Models. The union rule, introduced in Section III, connects the member concept directly to all concepts that are connected to the union concept. Then, the benefit of applying this rule to a union relationship r is the access frequency of r, and the cost is the number of edges that we copy from the union concept to the member concept. The cost-benefit model is defined as follows.

$$\begin{array}{l} Benefit(r) = AF(c_i \xrightarrow{r} c_j) \\ Cost(r) = \sum_{r' \in (c_i.R_i \setminus R_{un})} |r'|, \end{array} \tag{3}$$

where c_i denotes the union concept and $|r'|$ denotes the number of edges between the instance vertices of c_i and the ones of a neighborhood concept[13] of c_i.

The benefit of applying the inheritance rule to an inheritance relationship is the access frequency of that relationship multiplied by the Jaccard similarity between $c_i.P_i$ and $c_j.P_j$. Depending on that similarity, the cost of inheritance rule can be either the number of new edges attached to the parent, or the number of new edges attached to the child. Formally:

$$\begin{array}{l} Benefit(r) = AF(c_i \xrightarrow{r} c_j.P_j) \times JS(c_i, c_j) \\ Cost(r) = \begin{cases} \sum_{p \in c_j.P_j} |c_j| \times p.type + \sum_{r \in (c_j.R_j \setminus R_{ih})} |r|, \\ \qquad \text{if } \theta_1 < JS(c_i, c_j) \\ \sum_{p \in c_i.P_i} |c_i| \times p.type + \sum_{r \in (c_i.R_i \setminus R_{ih})} |r|, \\ \qquad \text{if } JS(c_i, c_j) < \theta_2, \end{cases} \end{array} \tag{4}$$

where $JS(c_i, c_j)$ denotes the Jaccard similarity between $c_i.P_i$ and $c_j.P_j$, $p.type$ indicates the data type size of p (e.g., the size of `INT`, `DOUBLE`, `STRING`, etc.), $\sum_{p \in c_j.P_j} |c_j| \times p.type$ ($\sum_{p \in c_i.P_i} |c_i| \times p.type$) denotes the space overheads incurred by propagating $c_j.P_j$ ($c_i.P_i$) to c_i (c_j), and $\sum_{r \in (c_i.R_i \setminus R_{ih})} |r|$ ($\sum_{r \in (c_j.R_j \setminus R_{ih})} |r|$) denotes the space overhead incurred by connecting the neighbors of c_i (c_j) to c_j (c_i).

Similarly, the cost-benefit model for one-to-many rule, leveraging both data characteristics and access frequency information, is described as:

$$\begin{array}{l} Benefit(r) = AF(c_i \xrightarrow{r} c_j.p) \\ Cost(r) = |r| \times p.type, \end{array} \tag{5}$$

where $|r| \times p.type$ denotes the space overhead incurred by replicating p as a data property of type $LIST$ to c_i.

As described in Section III, each *M:N* relationship is equivalent to two *1:M* relationships. Thus, we first convert each *M:N* relationship into two *1:M* relationships, and then use Eq. 5 to decide the cost-benefit for each of them. Potentially some of the original *M:N* relationships could be optimized for only one direction. This increases the flexibility of applying many-to-many rule such that more frequently accessed data properties can be propagated to the other end of the relationship.

With the cost and benefit scores, our goal is to select a subset of relationships in the ontology that maximize the total benefit within the given space limit. We map our relationship selection problem to the NP-hard 0/1 Knapsack Problem [23].

Proposition 1 (Reduction)**.** *If both benefit and cost of a relationship are positive, then every instance of the relationship selection problem can be reduced to a valid instance of the 0/1 Knapsack problem.*

Proof. The proof can be found in the technical report [19]. □

We adopt the fully polynomial time approximation scheme (FPTAS) [23] for our relation selection problem. It guarantees that the benefit of the optimized property graph schema $Benefit(\mathcal{PGS})$ is within 1-ϵ ($\epsilon \in (0,1]$) bound to the benefit of the optimal property graph schema $Benefit(\mathcal{PGS}_{opt})$.

Algorithm 3 Relation-Centric Algorithm

```
Input: O = (C, R, P), space limit S
Output: A property graph schema PGS
    // Compute Jaccard similarity for each inheritance relationship
    and get all parent concepts
 1: for each r ∈ R of type inheritance do
 2:     r.js ← computeJS(r)
 3: Benefit, Cost ← ∅
 4: for each r_i ∈ R do
 5:     Benefit[i] ← Benefit(r_i)
 6:     Cost[i] ← Cost(r_i)
 7: R_opt ← knapsack(R, Benefit, Cost, S)
 8: for each r_i ∈ R_opt do
 9:     O ← applyRules(r_i)
10: PGS ← generatePGS(O)
11: return PGS
```

Algorithm 3 takes as inputs an ontology and the space limit. It computes the Jaccard similarity scores for all inheritance relationships (Lines 1-2). Then it computes the cost and benefit for each relationship in the ontology $\mathcal{O}$ using Eqs. 3-5 (Lines

[13]The neighborhood concepts do not include the member concepts of c_i.

3-6). Next, the FPTAS algorithm is used to select the near-optimal subset of relationships R_{opt} with the given space limit S (Line 7). In *applyRules* procedure, the algorithm applies the corresponding rules; $r \in R_{opt}$ (Lines 8-9). Lastly, an optimized property graph schema is generated (Lines 10-11).

Complexity Analysis. The FPTAS *knapsack* is the dominant procedure in Algorithm 3, and its time complexity is $O(|R|^2 \lfloor |R|/\epsilon \rfloor)$ [23], where $|R|$ is the number of relationships and $\epsilon \in (0, 1]$. The rest of Algorithm 3 is linear to $|R|$. Thus, the time complexity of Algorithm 3 is $O(|R|^3)$.

V. Experimental Evaluation

A. Experimental Setup

Infrastructure. We implemented our approach in Java with JDK 1.8.0 running on Ubuntu 14.04 with 16-core 3.4 GHz CPU and 128 GB of RAM. We choose two popular graph database systems, Neo4j and JanusGraph, as our graph backends. We executed each experiment ten times and here we report their average.

Data sets. To evaluate the effectiveness of our system on different application domains, we use the following two data sets and their corresponding ontologies.

1. Medical data set (*MED*) contains medical knowledge that is used to support evidence-based clinical decision and patient education. The total size of this data set is around 12 GB. The corresponding medical ontology consists of 43 concepts, 78 properties, and 58 relationships (11 inheritance, 5 one-to-one, 30 one-to-many, and 12 many-to-many relationships).

2. Financial data set (*FIN*) [24] includes data from two main sources: Securities and Exchange Commission (SEC)[14] and Federal Deposit Insurance Corporation (FDIC)[15]. The size of the data set is approximately 53 GB. The corresponding financial ontology contains 90 concepts, 96 properties, and 103 relationships (4 union, 69 inheritance, and 30 one-to-many relationships).

Methodology and metrics. To evaluate the quality of the property graph schemas produced by our algorithms, we vary the space limit and the Jaccard similarity thresholds for inheritance relationships with two different workload summaries (uniform and Zipf). Specifically, we show how effectively *PGSG* leverages the given space limit, how robust *PGSG* is to various workloads, and how sensitive *PGSG* is to different similarity thresholds. *PGSG* chooses the property graph schema with a higher total benefit score from relation-centric (*RC*) and concept-centric (*CC*) algorithms. We measure the quality of a property graph schema by $BR = \frac{B_{SC}}{B_{NSC}}$, where B_{NSC} is the total benefit score of the property graph schema generated by the schema optimization algorithm without any space constraint, and B_{SC} indicates the total benefit score achieved by either RC or CC algorithm.

To verify the graph query performance, we express most graph queries in both Cypher [6] and Gremlin [25], including path, reachability, and graph analytical queries. Among these query types, we construct a variety of query workloads conforming to different workload distributions over both *FIN* and *MED*. The details of these query workloads are described in Section V-C. We use latency as the metric to measure these graph queries. Latency is measured in milliseconds as the total time of all queries in a workload executed in a sequential order. We also use the number of edge traversals required in a query as the second metric. It directly reveals the computational savings achieved by our optimized property graph schema.

B. Property Graph Schema Quality

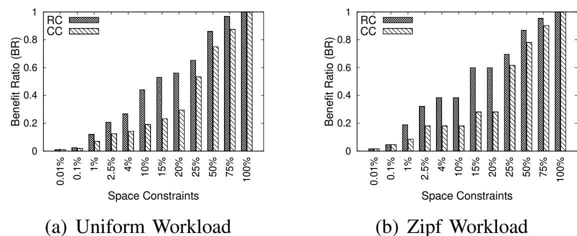
1) Varying Space Constraint: In Fig. 8 and Fig. 9, we focus on the quality of the property graph schema produced by our concept-centric (*CC*) and relation-centric (*RC*) algorithms compared to our method without space constraints (*NSC*). We choose two commonly seen workload summaries, uniform and Zipf distributions. Namely, the access frequencies of concepts in the ontology follow either uniform or Zipf distribution. And the skew factor of Zipf distribution is set to 1. We first use *NSC* to produce an optimal property graph schema PGS_{NSC} without any space constraint, and then compute the total benefit score B_{NSC} achieved by PGS_{NSC}. The space used by *NSC* is approximately 29GB for *MED* and 106GB for *FIN*, respectively. The total amount of space needed by the direct mapping algorithm S_{DIR} is 12GB for *MED* and 53GB for *FIN*, respectively. Then we vary the space constraint from S_{DIR} to S_{NSC}, such that the range of the Y-axis in Fig. 8 and Fig. 9 is from 0 to 1.

In Fig. 8, we observe that *RC* consistently outperforms *CC* with both uniform and Zipf workloads. The reason is that *RC* has a global ordering of all relationships, and the global ordering is near-optimal with respect to the given space constraint due to the adopted approximate Knapsack algorithm. On the contrary, *CC* suffers from a rather local optimal ordering with respect to each concept. Hence, it misses the opportunity to utilize the space for more beneficial relationships. Moreover, we observe that with approximately 20% of the maximum space constraint, *RC* is able to produce high-quality property graph schemas which achieve above 50% of the total benefit. In other words, both algorithms can effectively utilize the rather limited space. Lastly, both *RC* and *CC* produce the same property graph schema as PGS_{NSC} when the space constraint reaches 100%, which substantiates Theorem 1.

Similarly, *RC* outperforms *CC* In Fig. 9, as *CC* utilizes the space for one concept at a time, missing the opportunities for more beneficial relationships in the ontology. We also observe that both algorithms, with uniform and Zipf workloads, have a couple of drops when the space constraint increases. The reason is primarily due to the complexity of *FIN* ontology. Given that the inheritance relationships are more dominant in *FIN*, the given space may be exhausted quickly by certain inheritance relationships. Again, *RC* and *CC* produce the same schema as PGS_{NSC} with 100% space constraint.

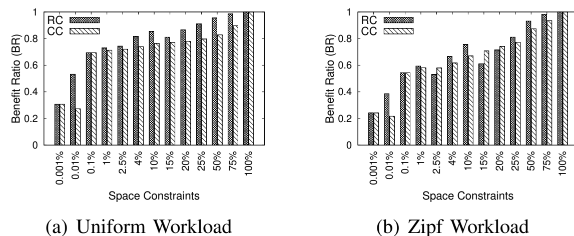
2) Varying Jaccard Similarity: In Fig. 10, we show the sensitivity of both *CC* and *RC* with respect to the Jaccard similarity thresholds (θ_1 and θ_2). In this experiment, we choose

[14]https://www.sec.gov/dera/data/financial-statement-data-sets.htm
[15]https://www.fdic.gov/regulations/resources/call/index.html



(a) Uniform Workload (b) Zipf Workload

Fig. 8: Varying Space Constraints (MED).

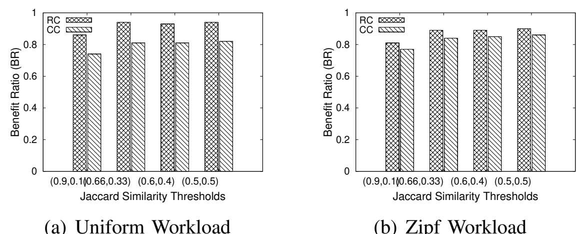


(a) Uniform Workload (b) Zipf Workload

Fig. 9: Varying Space Constraints (FIN).

FIN ontology because it consists of multiple inheritance relationships. We also choose the same uniform and Zipf workloads used in Section V-B1. Note that the space constraint in this experiment is set to (S_{NSC}-S_{DIR})/2 under each specific Jaccard similarity threshold. The reason is that the cost (space overhead) of the same inheritance relationship can vary (Eq. 4) depending on the similarity threshold. Consequently, the space consumption of the optimal property graph changes under different thresholds.

As shown in Fig. 10, both *CC* and *RC* are robust under different similarity thresholds. They achieve more than 70% of the maximum benefit score with only 50% space constraint. This shows that when the cost-benefit of an inheritance relationship changes due to a different threshold, both *CC* and *RC* can adjust accordingly by choosing different and more beneficial relationships to optimize. Hence, the total benefit scores achieved by both algorithms are relatively stable.



(a) Uniform Workload (b) Zipf Workload

Fig. 10: Varying Jaccard Thresholds (FIN).

In summary, *CC* and *RC* produce high quality property graph schemas under various settings. They work effectively with any given space constraints. Moreover, *RC* always produces a near-optimal property graph schema and outperforms *CC* in most cases. Our property graph schema generator still leverages both algorithms to choose the property graph schema with the highest benefit score under any space constraints.

C. Graph Query Execution

In this section, we focus on the graph query execution performance over the property graphs created by our ontology-driven approach. We use both *MED* and *FIN* data sets to conduct our experiments. First, we create a micro benchmark to empirically examine whether the property graph schema from our approach can actually benefit a set of graph primitives including simple pattern matching, vertex property lookup, and aggregation on vertices. Second, we study the overall execution time for a given graph query workload by mixing the above graph primitives. We run the graph queries, expressed in Cypher and Gremlin, on Neo4j and JanusGraph, respectively. Here our goal is not to compare the performance between two systems, rather to show that our schema optimization results in query performance improvements irrespective of the backend.

1) Microbenchmark Using Graph Primitives: With both *MED* and *FIN* data sets, we compare the query performance of the property graph created by the optimized graph schema (*OPT*) to the baseline property graph created by a direct mapping of the ontology (*DIR*). *OPT* is produced by the our method without space constraints (*NSC*) and the Jaccard similarity thresholds are $\theta_1 = 66\%$, $\theta_2 = 33\%$. All queries (Q_1-Q_{12}) are first expressed against *DIR* and then rewritten into the semantically equivalent queries over *OPT*. These queries are constructed according to the query patterns in [26]. We list several representative queries below.

```
Q1: MATCH (d:Drug)-[p:cause]->(r:Risk)<-
[p2:unionOf]-(ci:ContraIndication)
RETURN d.name
Q3: MATCH (aa:AutonomousAgent)<-[r1:isA]-
(p:Person)<-[r2:isA]-(cp:ContractParty)
RETURN aa
Q5: MATCH (dl:DrugLabInteraction)-[r:isA]->
(di:DrugInteraction)
RETURN di.summary
Q7: MATCH (n:Corporation)
RETURN n.hasLegalName
Q9: MATCH p=(d:Drug)-[r:hasDrugRoute]->
(dr:DrugRoute)
RETURN dr.drugRouteId, size(COLLECT(
d.brand)) AS numberOfDrugBrands
Q11: MATCH p=(con:Contract)-[r:isManagedBy]->
(corp:Corporation)
RETURN size(COLLECT(con.hasEffectiveDate)) AS
numberOfEffectiveDates
```

As shown in Fig. 11 , the results are unequivocal. The optimized schema has significant advantages over the direct mapping schema for all types of queries. The graph pattern matching queries (Q_1-Q_4) report all matches of a sub-graph with 3 vertices and 2 edges in the property graph. Query execution times with our approach are at least 2.4 times faster than the direct mapping schema. The number of edge traversals on *DIR* is always 2 as the query is specified with 2 edges connecting 3 vertices. On the other hand, our property graph only requires at most 1 edge traversal as some of the neighbor vertices have been already merged with the starting vertices.

Q_5-Q_8 are vertex property lookup queries. Both Q_5 and Q_8 are interested in a property of a vertex of a parent concept,

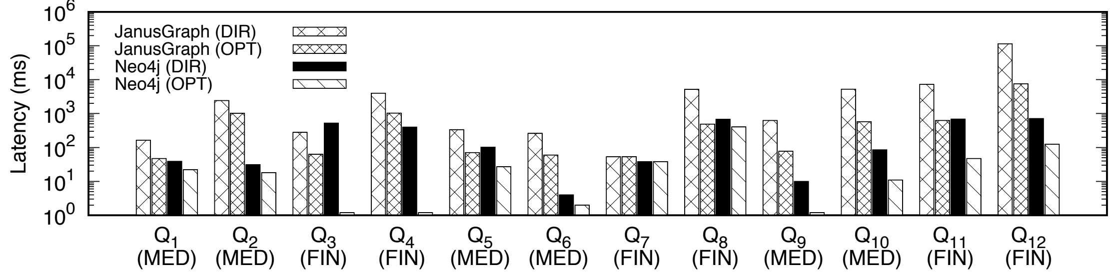


Fig. 11: Microbenchmark - Pattern Matching (Q_1-Q_4), Property Lookup (Q_5-Q_8), Aggregation (Q_9-Q_{12}).

and the starting vertex is a vertex of a child concept. Q_6 starts from a vertex and looks for a property of its neighbor vertex. *OPT* has the property of type $List$ with the starting vertex, and is able to return the result without any edge traversal. Q_7 looks for a property of the starting vertex. In this case, *OPT* and *DIR* have identical query performance as no edge traversal is required. Hence *OPT* takes advantage of having the property of the parent concept available at the starting vertex, and consequently returns the result without any edge traversals. Therefore, the query runs more than an order of magnitude slower on the property graph of *DIR* than the one on *OPT* in the worst case.

Q_9-Q_{12} are graph aggregation queries that involve traversal from one vertex to the other. They count the number of neighbors of the starting vertex. On average, the query execution time is an order of magnitude faster for *OPT* approach compared to *DIR*. Again, the reason is that the aggregation on the neighbor vertices can be instantaneously returned from the starting vertex. The above results suggest that using the proposed ontology-driven approach can bring significant benefits to a variety of graph queries.

Lastly, we observe that the performance gain on Neo4j is more substantial compared to JanusGraph (e.g., Q_3, Q_4, Q_9, etc.). This shows that disk-based graph systems (e.g., Neo4j) benefits much more from our techniques, as the optimized schema requires significantly less disk I/O. Namely, the graph system loads less number of vertices and edges into memory. We expect such benefit to become even greater when the size of the property graph increases.

TABLE II: Microbenchmark - Number of Edge Traversals.

Graph Queries	*# Edge Traversals*		*Graph Queries*	*# Edge Traversals*	
	DIR	*OPT*		*DIR*	*OPT*
Q_1	21,608	6,072	Q_7	0	0
Q_2	288,142	115,014	Q_8	493,588	0
Q_3	36,272	0	Q_9	67,397	0
Q_4	510,460	97,614	Q_{10}	429,636	15,327
Q_5	38,768	0	Q_{11}	524,265	0
Q_6	32,586	0	Q_{12}	110,4756	548,262

In addition, Table II reveals that *OPT* substantially reduces the number of edge traversals required in most queries, which leads to significant computational savings and performance gains. In several cases (e.g., Q_3, Q_6), edge traversals can be completely avoided as the queried information is available locally within the starting vertices. On the other hand, the performance gains of certain queries (e.g., Q_5, Q_8, Q_{12}) are not as significant as others, even though the number of edge traversals with *OPT* is much smaller than the one with *DIR*. The reason is that the costs of lookup and return operations are non-trivial in both *DIR* and *OPT*, which can be observed from the latency of these queries in Fig. 11 as well.

2) Graph Query Workload Performance: To evaluate the runtime performance of the property graph schema generated by our approach, we first generate a set of query workloads, including both uniform and Zipf distributions in terms of the access frequency of the concepts in the ontology. We vary the Zipf's skew factor from 0 (i.e., uniform distribution) to 2 (highly skewed). All query workloads consist of 15 queries of mixed types (i.e., pattern matching, lookup, and aggregation), similar to the ones used in the microbenchmark. The space limit is set to 20% of the space consumed by *NSC* (i.e., 15.4GB for *MED* and 80GB for *FIN*). The Jaccard similarity thresholds are $\theta_1 = 66\%$ and $\theta_2 = 33\%$. The optimized schemas (*OPT*$_{MED}$ and *OPT*$_{FIN}$) are produced by the best performing algorithm of *RC* and *CC*.

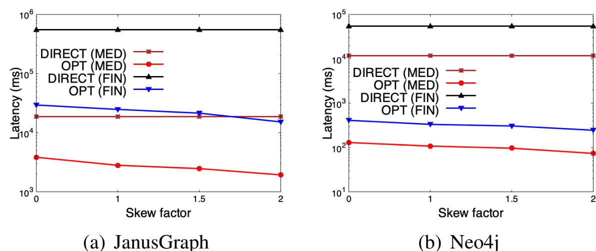
TABLE III: Benefit Ratio w.r.t B_{NSC}.

Skew Factor	*MED*				*FIN*			
	0	1	1.5	2	0	1	1.5	2
RC	56%	59%	62%	71%	67%	71%	74%	88%
CC	30%	43%	50%	63%	65%	74%	80%	88%

Table III shows the quality of the property graph schema produced by *RC* and *CC* compared to the one without space constraints *NSC*. We observe that both *RC* and *CC* correctly prioritize the most cost-effective relationships when the workloads are highly skewed. *RC* performs better than *CC* over *MED*, because *MED* has more data properties per concepts and *RC* makes more flexible decisions in terms of which relationships to optimize. On the other hand, *CC* performs better than *RC* over *FIN* as it successfully selects few concepts that are frequently accessed by the highly skewed workloads.

We compare our optimized schemas to the direct mapping schemas (*DIRECT*$_{MED}$, *DIRECT*$_{FIN}$) on both JanusGraph and Neo4j. The total query latency measures the performance on these property graphs corresponding to different schemas.

Fig. 12 shows the total query latency in log scale. Both OPT_{MED} and OPT_{FIN} offer significant performance boosts to the graph query workloads on both JanusGraph and Neo4j. In Fig. 12(a), we observe that the total query latency on the optimized schema, on average, is around 7 and 26 times faster than the direct mapping one over *MED* and *FIN*, respectively. The winning margin is substantially bigger (i.e., 129 and 176 times faster) on Neo4j (Fig. 12(b)). The total query latency on both optimized schema is approximately 2 orders of magnitude faster than the direct mapping. Moreover, we also observe that the total query latency decreases with increasing skew factor. Both OPT_{MED} and OPT_{FIN} achieve the lowest latency when the workload distributions are highly skewed. This indicates that the most frequently accessed concepts and relationships in the workloads are chosen to be optimized given the space limit. Based on these results, we verify that the designed rules for different types of relationships in the ontology are effective in terms of reducing edge traversals and consequently improving the graph query performance. Furthermore, we demonstrate that our approach can effectively utilize the given space constraint by leveraging data distribution and workload summaries.



(a) JanusGraph

(b) Neo4j

Fig. 12: Total Query Latency (best viewed in color).

D. *Efficiency of Property Graph Schema Algorithms*

We also study the efficiency of our concept-centric and relation-centric algorithms. The execution times to produce an optimized property graph schema with different space constraints range from 23 to 192 ms in *CC*, and from 34 to 373 ms in *RC*. We also observe that neither algorithm is sensitive to the space constraint, since both algorithms have a polynomial time complexity with respect to the number of concepts and relationships in the given ontology. Due to space constraints, the detailed analysis can be found in [19].

VI. Related Work

Schema optimization for improving query performance has been studied in the database community for decades [27]–[30]. In recent years, the emergence of many large-scale knowledge graphs has drawn attention for schema optimization. In this section, we present important works in this field, highlighting the main differences to our approach.

Schema Optimization in RDBMS/NoSQL. Schema design in relational database systems has been extensively studied [28], [29], [31]–[34]. RDBMSs provide a clean separation between logical and physical schemas. The logical schema includes a set of table definitions and determines a physical schema consisting of a set of base tables [28], [29], [31]. The physical layout of these base tables is then optimized with auxiliary data structures such as indexes and materialized views for the expected workload [31], [33]. Typically, the physical design often involves identifying candidate physical structures and selects a good subset of these candidates [34]. NoSE [30] is introduced to recommend schemas for NoSQL applications. Its cost-based approach utilizes an integer programming formulation to generate a schema based on the conceptual data model from the application.

In principle, our approach is similar to the logical schema design in RDBMSs, which defers the physical design to the underlying graph systems. Other than that, our approach is different from the above methods since the data modeling for graphs is inherently different from the relational data model. Specifically, the graph structure results in more expressive data models than those produced using relational databases, allowing the formation of graph queries (e.g., reachability, path finding, pattern matching) in a very intuitive fashion. Moreover, our approach exploits the rich semantic information available in an ontology to drive the schema optimization, which is not considered by any of the previous works.

Schema Optimization in Knowledge Graphs. In the last few years, RDF has been growing significantly for expressing graph data. A variety of schemas have been proposed for physically storing graph data in both centralized and distributed settings [9], [10], [12], [35]–[40]. Some of these works focus on optimizing RDF data storage and SPARQL queries based on either workload statistics [9], [10], [36], [37] or heuristics [11]. A fundamental difference to those works is that we neither re-load the data to follow a new schema, nor build new indices, nor optimize the queries on-the-fly (e.g., join reordering). Instead, inspired by database literature as stated above, we provide an optimized property graph schema design before loading the data, and directly instantiate a property graph conforming to this schema on a graph database. Graph queries are then executed on the graph database, where graph query optimization techniques can be further utilized.

Other works [12], [38]–[40] attempt to transform RDF data into relational data and provide SPARQL views over relational schemas, leveraging the many years of experience in RDBMS schema optimization. Unlike those approaches, we do not create views over a property graph. Instead, we directly express property graph queries in Cypher or Gremlin over optimized property graphs. Whether a graph database internally uses views or not for query optimization is orthogonal to this work. Thakkar [41] introduces Gremlinator, a SPARQL-to-Gremlin translator, to enable users to query property graph systems using SPARQL. However, it does not optimize the graph stored in the graph database. We see Gremlinator as a complementary work that could potentially exploit our work to provide a SPARQL interface over an optimized property graph schema.

Recently, works such as [5], [8], [42], [43] address a similar problem in the context of property graphs. GRFu-

sion [43] focuses on filling the gap between the relational and the graph models rather than optimizing the graph schema to achieve better query performance. Szárnyas et al. [42] propose to use incremental view maintenance for property graph queries. However, their approach can only support a subset of property graph queries by using nested relational algebra. SQLGraph [5] and Db2 Graph [8] introduce a physical schema design that combines relational storage for adjacency information with JSON storage for vertex and edge attributes. However SQLGraph and Db2 Graph also focus on physical schema design which only targets on the relational databases. Materialized views [44], [45] are also introduced to answer graph pattern queries. Views are either given as inputs or generated based on query workloads. Then a subset of views are chosen to answer a query. The optimized schema generated from our approach can be considered as a view on the original property graph, which can be consumed by their technique.

VII. Conclusions

To the best of our knowledge, our ontology-driven approach is the first to address the property graph schema optimization problem for domain-specific knowledge graphs. Our approach takes advantages of the rich semantic information in an ontology to drive the property graph schema optimization. The produced schemas gain up to 3 orders of magnitude graph query performance speed-up compared to a direct mapping approach in two real-world knowledge graphs.

References

[1] J. Lehmann, R. Isele, M. Jakob *et al.*, "Dbpedia-A large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, 2015.

[2] K. D. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *ACM SIGMOD*, 2008, pp. 1247–1250.

[3] X. L. Dong and D. Srivastava, *Big Data Integration*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2015.

[4] S. Sakr and G. Al-Naymat, "Relational processing of RDF queries: a survey," *SIGMOD Record*, vol. 38, no. 4, pp. 23–28, 2009.

[5] W. Sun, A. Fokoue, K. Srinivas *et al.*, "SQLGraph: An efficient relational-based property graph store," in *ACM SIGMOD*, 2015, pp. 1887–1901.

[6] N. Francis, A. Green, P. Guagliardo *et al.*, "Cypher: An evolving query language for property graphs," in *ACM SIGMOD*, 2018, pp. 1433–1445.

[7] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "PGQL: a property graph query language," in *Graph Data-management Experiences and Systems*, 2016, p. 7.

[8] Y. Tian, E. L. Xu, W. Zhao *et al.*, "IBM db2 graph: Supporting synergistic and retrofittable graph queries inside IBM db2," in *ACM SIGMOD*, 2020, pp. 345–359.

[9] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins," in *ICDE*, 2011, pp. 984–994.

[10] M. Meimaris, G. Papastefanatos, N. Mamoulis, and I. Anagnostopoulos, "Extended characteristic sets: Graph indexing for SPARQL query optimization," in *ICDE*, 2017, pp. 497–508.

[11] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz, "Heuristics-based query optimisation for SPARQL," in *EDBT*, 2012, pp. 324–335.

[12] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee, "Building an efficient RDF store over a relational database," in *ACM SIGMOD*, 2013, pp. 121–132.

[13] "GQL standard," https://www.gqlstandards.org/, June 2020.

[14] "OWL 2 web ontology language document overview," https://www.w3.org/TR/owl2-overview/, March 2020.

[15] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*. O'Reilly Media, Inc., 2013.

[16] A. Deutsch, Y. Xu, M. Wu, and V. Lee, "Tigergraph: A native MPP graph database," *CoRR*, vol. abs/1901.08248, 2019.

[17] O. Hartig and J. Hidders, "Defining schemas for property graphs by using the graphql schema definition language," in *GRADES-NDA*, 2019, pp. 6:1–6:11.

[18] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *VLDB J.*, vol. 29, no. 2, pp. 595–618, 2020.

[19] C. Lei, R. Alotaibi, A. Quamar, V. Efthymiou, and F. Özcan, "Property graph schema optimization for domain-specific knowledge graphs," *CoRR*, vol. abs/2003.11580, 2020.

[20] A. Quamar, F. Özcan, and K. Xirogiannopoulos, "Discovery and creation of rich entities for knowledge bases," in *ExploreDB*, 2018.

[21] A. Quamar, C. Lei, D. Miller *et al.*, "An ontology-based conversation system for knowledge bases," in *ACM SIGMOD*, 2020, pp. 361–376.

[22] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *WWW*, 1998, pp. 107–117.

[23] V. V. Vazirani, *Approximation Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2001.

[24] J. Sen, C. Lei, A. Quamar *et al.*, "ATHENA++: Natural language querying for complex nested sql queries," *PVLDB*, vol. 13, no. 11, pp. 2747–2759, 2020.

[25] "Gremlin query language," https://tinkerpop.apache.org/gremlin.html, March 2020.

[26] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large SPARQL query logs," *PVLDB*, vol. 11, no. 2, pp. 149–161, 2017.

[27] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[28] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 91–128, 1988.

[29] D. C. Zilio, J. Rao, S. Lightstone *et al.*, "Db2 design advisor: Integrated automatic physical database design," in *VLDB*, 2004, pp. 1087–1097.

[30] M. J. Mior, K. Salem, A. Aboulnaga, and R. Liu, "Nose: Schema design for nosql applications," in *ICDE*, May 2016, pp. 181–192.

[31] S. Agrawal, S. Chaudhuri, and V. R. Narasayya, "Automated selection of materialized views and indexes in sql databases," in *VLDB*, 2000, pp. 496–505.

[32] N. Bruno and S. Chaudhuri, "Automatic physical database tuning: A relaxation-based approach," in *ACM SIGMOD*, 2005, pp. 227–238.

[33] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik, "Coradd: Correlation aware database designer for materialized views and indexes," *PVLDB*, vol. 3, no. 1-2, pp. 1103–1113, 2010.

[34] D. Dash, N. Polyzotis, and A. Ailamaki, "Cophy: A scalable, portable, and interactive index advisor for large workloads," *PVLDB*, vol. 4, no. 6, pp. 362–372, 2011.

[35] J. Huang, D. J. Abadi, and K. Ren, "Scalable sparql querying of large rdf graphs," *PVLDB*, vol. 4, pp. 1123–1134, 2011.

[36] A. Maduko, K. Anyanwu, A. P. Sheth, and P. Schliekelman, "Estimating the cardinality of RDF graph patterns," in *WWW*, 2007, pp. 1233–1234.

[37] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[38] S. Harris and N. Shadbolt, "SPARQL query processing with conventional relational database systems," in *WISE*, 2005, pp. 235–244.

[39] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach, "Sw-store: a vertically partitioned DBMS for semantic web data management," *VLDB J.*, vol. 18, no. 2, pp. 385–406, 2009.

[40] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan, "An efficient sql-based RDF querying scheme," in *VLDB*, 2005, pp. 1216–1227.

[41] H. Thakkar, D. Punjani, J. Lehmann, and S. Auer, "Two for one: querying property graph databases using SPARQL via gremlinator," in *GRADES-NDA*, 2018, pp. 12:1–12:5.

[42] G. Szárnyas, "Incremental view maintenance for property graph queries," *arXiv preprint arXiv:1712.04108*, 2017.

[43] M. S. Hassan, T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi, "Extending in-memory relational database engines with native graph support," in *EDBT*, 2018, pp. 25–36.

[44] W. Fan, X. Wang, and Y. Wu, "Answering pattern queries using views," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 2, pp. 326–341, 2016.

[45] J. M. F. da Trindade, K. Karanasos, C. Curino, S. Madden, and J. Shun, "Kaskade: Graph views for efficient graph analytics," in *ICDE*, 2020, pp. 193–204.