The VLDB Journal
https://doi.org/10.1007/s00778-022-00756-y

REGULAR PAPER

HERMES: data placement and schema optimization for enterprise knowledge bases

Chuan Lei[1] · Abdul Quamar[2] · Vasilis Efthymiou[3] · Fatma Özcan[4] · Rana Alotaibi[5]

Received: 19 January 2021 / Revised: 11 March 2022 / Accepted: 8 June 2022

Abstract

Enterprises create domain-specific knowledge bases (KBs) by curating and integrating their business data from multiple sources. To support a variety of query types over domain-specific KBs, we propose HERMES, an ontology-based system that allows storing KB data in multiple backends, and querying them with different query languages. In this paper, we address two important challenges in realizing such a system: data placement and schema optimization. First, we identify the best data store for any query type and determine the subset of the KB that needs to be stored in this data store, while minimizing data replication. Second, we optimize how we organize the data for best query performance. To choose the best data stores, we partition the data described by the domain ontology into multiple overlapping subsets based on the operations performed in a given query workload, and place these subsets in appropriate data stores according to their capabilities. Then, we optimize the schema on each data store to enable efficient querying. In particular, we focus on the property graph schema optimization, which has been largely ignored in the literature. We propose two algorithms to generate an optimized schema from the domain ontology. We demonstrate the effectiveness of our data placement and schema optimization algorithms with two real-world KBs from the medical and financial domains. The results show that the proposed data placement algorithm generates near-optimal data placement plans with minimal data replication overhead, and the schema optimization algorithms produce high-quality schemas, achieving up to two orders of magnitude speed-up compared to alternative schema designs.

Keywords Knowledge base · Data placement · Schema optimization

1 Introduction

A growing number of enterprises are creating domain-specific knowledge bases (KBs) [24,27,58] by curating and integrating their business data, including structured, unstructured and semi-structured data. One distinct characteristic of these enterprise KBs, compared to cross-domain KBs like DBpedia [39] and YAGO 4 [60], is their deep domain specialization and understanding, which empowers many applications in various domains, such as healthcare and finance. These high-value KBs are used by various analysis applications that require different querying capabilities, such as informational search queries, precise structured queries, complex graph queries, in order to extract insights from various business entities and relationships.

In this paper, we propose a novel system, HERMES, that allows storing an enterprise KB in multiple backends using different data models and query languages to support a rich variety of query types over the KB. We argue that storing a KB in a polystore system enables a variety of queries and

Chuan Lei, Vasilis Efthymiou, Fatma Özcan, Rana Alotaibi: Work done while at IBM Research.

✉ Chuan Lei
chuan.lei@instacart.com

Abdul Quamar
ahquamar@us.ibm.com

Vasilis Efthymiou
vefthym@ics.forth.gr

Fatma Özcan
fozcan@google.com

Rana Alotaibi
ralotaib@eng.ucsd.edu

1 Instacart, San Francisco, USA

2 IBM Research - Almaden, San Jose, USA

3 FORTH - Institute of Computer Science, Heraklion, Greece

4 Google, Mountain View, USA

5 University of California - San Diego, San Diego, USA

Published online: 26 July 2022

applications that result in better business decisions. Following the standard definition [15], a KB consists of meta-level and instance-level data. To support deep domain specialization, we assume that the former is modeled in a domain ontology, while the latter can be stored in various data stores. The main benefit of using a domain ontology for data modeling is that it offers a rich and entity-centric view of the instance-level data stored in a KB, and the instance-level data can be stored in any backend data store. Currently, HERMES includes a relational database, a document store, and a property graph store. As a result, HERMES can answer SQL, document search, and graph queries.

Many open challenges are associated with building a system such as HERMES, ranging from query optimization, load balancing, data placement, data transformation and integration, to schema optimization [52,56]. In this paper, we address two of these challenges: data placement and schema optimization (for property graphs in particular).

Data Placement. Deciding how to model the data and which data store to use requires a deep understanding of the data, the expected workload, as well as the query processing capabilities of the different data stores. Structured data is best suited to be stored in relational databases and queried through SQL; text data is mostly indexed and retrieved through search indexes, like Elasticsearch [9] and Solr [8]; graph data is better analyzed using graph query languages (e.g., Gremlin [3] and Cypher [30]) in graph databases, like JanusGraph [4] and Neo4j [5]. There are two extreme solutions. On one end of the spectrum, one can model the data using a single data model, like the relational or graph model, and store the entire KB in the corresponding data store. However, one size does not fit all [56,57]. On the other end of the spectrum, one could model the entire KB data in multiple data models, and store the whole KB in multiple backends to support various query types. This would result in tremendous data replication. We argue that there exists an optimal solution between these two extremes, where we place different subsets of the KB in different data stores to support a rich variety of queries.

Schema Optimization. There have been many techniques proposed for optimizing data schema over relational and NoSQL stores [12,36,46], as different physical data organization results in varying performance. However, the problem of property graph schema optimization has been largely ignored, which is also critical to graph query performance. Edge traversal is one of the dominant factors affecting graph query performance, and having an optimized schema can greatly improve query performance. The rich semantic relationships in an ontology provide a variety of opportunities to connect or combine nodes together to reduce graph traversals. To generate an optimized graph schema, we need to identify and exploit these opportunities in the ontology, and design different techniques to utilize them accordingly.

Our Proposed Approach. In this paper, we propose HERMES, an ontology-driven polystore system for rich querying of enterprise KBs. We assume that HERMES has the ability to store KB data in any data store that provides the needed capability for the query types in the workload. With this flexibility, HERMES provides an off-line capability-based data placement to decide where to store KB data, as well as an ontology-driven schema optimization to decide how to store KB data conforming to a given ontology. Choosing the data store based on its querying capabilities and supported data model is consistent with the polystore approaches like BigDAWG [29] and more recent work in [1].

HERMES first aims to minimize the number of data stores involved in any given query, hence minimizing the cost of data movement and transformation across different stores incurred during a query execution. It utilizes off-line capability-based data placement algorithms for a given query workload, choosing the most appropriate data store based on its capabilities, and minimizing data movement costs. Subsequently, HERMES stores each subset of the KB data in an appropriate data store based on the generated data placement plan.

Then, HERMES optimizes the schema of the data on each data store to enable efficient querying. Given that there exist many techniques for relational schema optimization [12,36,46], in this paper, we focus on the property graph schema optimization to improve graph query performance. We first propose a set of rules that are designed to optimize the graph query performance with respect to different types of relationships in the ontology. We then introduce property graph schema optimization algorithms that leverage these rules to produce an optimized schema, taking into account space constraints, if any, and additional information such as data distribution and query workload.

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

- We propose a novel polystore system, HERMES, to support a rich variety of query types over enterprise KBs. We use domain ontologies to describe the data at a semantic level. Currently, HERMES includes a relational database, a document store, and a property graph store.
- We introduce an ontology-driven approach to tackle two critical challenges, data placement and schema optimization, to support a rich set of query types over domain-specific KBs.
- We propose capability-based data placement algorithms that use a given workload against the domain ontology to decide how to store the data, for each concept of the ontology, with minimum data replication while minimizing data movement.
- For graph schema optimization, we design a set of rules that reduce the edge traversals by exploiting semantic

relationships in the ontology, resulting in better graph query performance. We propose concept- and relation-centric algorithms that harness these rules to generate an optimized property graph schema from an ontology, under space constraints.
– Our experimental study shows that the proposed data placement algorithm generates near-optimal data placement plans with minimal data replication overhead, and the schema optimization algorithms effectively produce high-quality schemas for two real-world KBs from medical and financial domains. For query workloads over these two KBs, the HERMES system achieves up to 2 orders of magnitude speed-up compared to alternative schema designs.

The rest of the paper is organized as follows. Section 2 provides an overview of HERMES system and core concepts. Section 3 describes our data placement method and Sect. 4 explains the algorithms to produce optimized property graph schema. In Sect. 5, we provide our experimental results. Finally, we review related work in Sect. 6, and conclude in Sect. 7.

2 HERMES System

Figure 1 shows the overview of the HERMES system. The KB construction component is responsible for data enrichment/curation process, which consumes the heterogeneous data sources for information extraction, entity resolution, and data integration [16]. The data placement component decides where to place the data depending on the capability of data stores and the expected workload. The schema optimization component ensures the data is stored on each data store in an efficient way for the expected workload. Finally, the data loading component transforms the heterogeneous data based on the optimized schema and places the data in appropriate data stores according to the data placement plan.

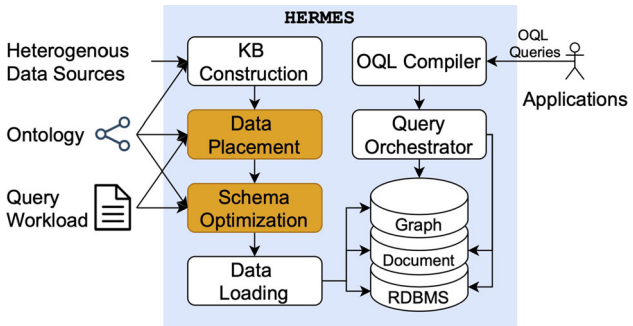


Fig. 1 HERMES system

The primary design goal of HERMES is to support rich queries that access data from diverse sources and under heterogeneous data models. For this purpose, HERMES uses a polystore architecture, with the ontology of the KB as the global schema. Currently, HERMES supports a relational database, a document store, and a property graph store. The main reason for choosing a property graph store (e.g., Neo4j or JanusGraph) over other graph stores (e.g., an RDF store) is the following. RDF represents a graph as a set of triples, in which even literals are represented as vertices. Those artificial vertices make it hard to naturally express graph queries in SPARQL. In contrast, property graph stores offer much more intuitive query languages such as Cypher and Gremlin, which require considerably less expertise to use compared to SPARQL. This is also the main reason why property graph stores have gained much popularity in recent years.

At runtime, users can query the KB using Ontology Query Language (OQL) (Sect. 2.2) against the domain ontology without knowing the complex data model and storage of the KB. Given an OQL query, the OQL compiler produces a logical representation in the form of a Query Graph Model (QGM) [49], which is essentially a DAG. The QGM captures the data flow and dependencies in a query using operator boxes, such as `SELECT`, `GROUP BY`, `SETOP`. Using QGM in HERMES allows us to orchestrate the data flow between different data stores during query execution, deferring the optimization of sub-queries to the data stores.

To support query routing to appropriate data stores, the query orchestrator takes the QGM representation of the input query, the data-to-store mappings generated by the data placement module, and the capability descriptions (Sect. 3.1) of each data store to produce a plan that routes the query to a single data store that has the relevant data and the capability to satisfy all the operations required by the query. In the event that the query cannot be answered by a single data store, the query orchestrator routes the query to a minimum number of data stores that can satisfy the data and operations required by the query. In HERMES, we choose to use a relational DBMS as the mediator that orchestrates the execution of the query, using UDFs that access the other stores [52]. For example, if a query involves operations from both a relational database and a document store, the relational database can invoke the search sub-query via a UDF-based mechanism [52], and integrate it into the rest of the query plan. Similarly, the relational engine can also act purely as a mediator to combine intermediate results from other data stores, if a query involves operations and data relevant to a search and graph store. We created a micro-benchmark (Sect. 5.3) to ascertain the efficacy of our query routing mechanism with an optimized query execution plan that leverages the capability-based data placement plan produced by HERMES.

In Sects. 3 and 4, we introduce the design of data placement and schema optimization (two yellow boxes highlighted

in Fig. 1) in more details. Note that we tackle these two problems independently since each of them by itself is NP-Hard. A holistic solution to collectively solve these two problems together, where a trade-off between data replication and query performance can be considered during the optimization, is left for future work. Next, we describe the domain ontology and the OQL language, both of which play a critical role in HERMES.

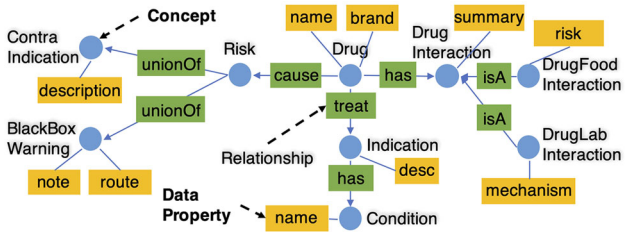
2.1 Domain ontology

A domain ontology describes a particular domain and offers a structured view of the data. Specifically, it provides a rich and expressive data model combined with a powerful object-oriented paradigm that captures a variety of real-world relationships between entities such as inheritance, union, functionality, etc.

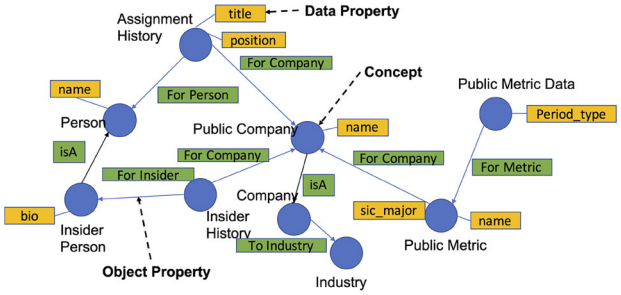
Definition 1 (Domain Ontology ($\mathcal{O}$)) A domain ontology $\mathcal{O}$ (C, R, P) contains a set of concepts $C = \{c_n | 1 \leq n \leq N\}$, a set of data properties $P = \{p_m | 1 \leq m \leq M\}$, and a set of relationships between the concepts $R = \{r_k | 1 \leq k \leq K\}$.

A domain ontology is typically described in OWL [6], 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*[1]. 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. Relation types t include functional (i.e., *1:1*, *1:M*), inheritance (a.k.a *isA*) and union/membership relations[2]. Next, we give two ontologies in medical and finance domains that describe the concepts and relationships in each of these domains.

HERMES uses domain ontologies to drive the curation and creation of the KB from heterogeneous data sources. The KB constructed based on the ontology can leverage its standard vocabularies/terminologies and semantically rich relationships, making it closer to real-world downstream applications (e.g., search and question answering). In many cases, domain ontologies are available or provided in a specific domain (e.g., FIBO ontology in finance, SNOMED in medicine, FoodOn in the food domain), describing the entities and their relationships of at a semantic level, irrespective of where and how the data is stored. In case where an ontology is not provided, users can either manually create one based on their domain knowledge or use bootstrapping techniques [32,40] to derive one from the data sources. A mapping captures the correspondence between the logical schema represented by the ontology and the physical schema of the underlying data stores.



(a) Medical Ontology



(b) Finance Ontology

Fig. 2 Ontology examples

2.2 Ontology query language

HERMES supports Ontology Query Language (OQL) [40,53], a query language expressed against the domain ontology as an abstraction to query the data without knowing how data is stored and indexed in multiple data stores. OQL supports all data types that are supported by OWL [6] as data property types, such as integer, boolean, string, etc. OQL can express sophisticated query operations that include aggregations, unions, and nested sub-queries among others. Additionally, OQL can express search operations such as full-text and fielded search over concept properties, as well query operations over indexed JSON documents, graph nodes and edges represented as a set of concepts and relations in the ontology. The constructs in OQL are inspired by SQL and look very much like SQL, but they represent operations over domain concepts and relations.

Figure 3 shows three OQL query examples against the financial ontology of Fig. 2b. For further details, we refer the reader to [53].

[1] The terms ObjectProperty and Relationship are used interchangeably in this paper.

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

```
Q1: Give me the revenue for Apple for each calendar year
OQL1: SELECT SUM(PMD.value), PMD.year_calendar, PC.name
FROM  PublicMetricData PMD, PublicMetric PM,
      PublicCompany PC
WHERE PM.name = 'REVENUE' AND
      PC.name = 'APPLE INC' AND
      PMD.period_type = 'YEARLY' AND
      PMD->forMetric = PM AND
      PMD->forPublicCompany = PC
GROUP BY PMD.year_calendar, PC.name

Q2: Give me the companies in the airline industry
OQL2:  SELECT  PC.name
FROM   PublicCompany PC, Industry IN
WHERE PC->is-a->toIndustry = IN AND
       IN.sic_major ='Transportation BY Air'

Q3: Show me company insiders from IVY League
OQL3: SELECT oCompany.name, oInsiderPerson.name
FROM  InsiderHistory oInsiderHistory, Company oCompany,
      InsiderPerson oInsiderPerson
WHERE oInsiderHistory->forCompany = oCompany AND
      oInsiderHistory->forInsider = oInsiderPerson AND
      oInsiderPerson.bio MATCH 'IVY League Graduate'
```

Fig. 3 OQL query examples

3 Data placement

As motivated in Sect. 1, enterprise applications may need to support a wide variety of query types, depending on their query workload. To support these different query types and achieve the best performance, HERMES stores and indexes the KB data across multiple data stores that provide the required capabilities.

A naïve solution to avoid data movement is to replicate the entire data across all data stores. However, this solution is an overkill and can lead to huge replication and storage space overheads. Moreover, not all stores provide all the necessary capabilities needed by the queries, so even full replication cannot eliminate data movement completely. For example, a query may require access to a document store for its search capability and a relational database for its join capability. The intermediate search results need to be moved from the document store to the relational database to generate the final results for this query. To minimize storage costs and unnecessary data movement and transformation, we propose capability-based data placement algorithms that assign data to data stores while taking into consideration both the expected workload and the capabilities of the underlying data stores in terms of the operations that they can perform. Next, we provide a formal definition of the data placement problem and describe mechanisms of expressing the capabilities of the data stores and query operations in the workload.

Definition 2 (Problem Definition - Data Placement) The problem of *data placement* is to assign subsets of a KB data to different data stores based on a given query workload and the query processing capabilities of the stores, while minimizing the amount of data replication such that the number of stores involved for any query in the workload is minimized[3].

3.1 Data store capability and query operation descriptions

We propose a declarative representation for the data store capabilities and query operation descriptions by using a set of *Representational constraints* and *Datalog* rules [11], respectively. A representational constraint is expressed using a fragment of the first-order logic, which is in the form of $\forall x_1, \ldots x_n \ \phi(x_1, \ldots x_n) \rightarrow \exists z_1, \ldots, z_k \ \psi(y_1, \ldots, y_m)$, where $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_m\}\{x_1, \ldots, x_n\}$. A Datalog rule is a first-order logic expression without negations and disjunctions that has the following form: $RL(\overline{X}) \leftarrow N_1(\overline{X_1})$, ..., $N_n(\overline{X_n})$, where RL, N_1, ..., N_n are predicate/relation names, $\overline{X_1}$, ..., $\overline{X_n}$ are tuples of constants and variables, and $N_n(\overline{X_n})$ is an atom. The body of a Datalog rule is defined as B=$\{N_1(\overline{X_1}), \ldots, N_n(\overline{X_n})\}$, while the head is defined as $RL(\overline{X})$. The variables that appear in the head of the query are called distinguished variables. Each variable in $\overline{X}$ must occur in at least one of $\overline{X_1}, \ldots, \overline{X_n}$. The rule is called *boolean* when it has an empty head.

3.1.1 Data store capability description

A key element in a knowledge base architecture is the primitives that are used to describe the capabilities of the underlying data stores. Several approaches for describing capabilities in heterogeneous databases have been proposed in the literature [22,42], which primarily focus on expressing the capabilities of a data store as views. They enumerate all possible queries (view definitions) that can be handled/answered by the data store. This approach is not scalable as the number of view definitions can be very large, potentially leading to query rewriting using infinite number of views.

Therefore, we propose to describe the capabilities of the data stores in terms of the operations that they support (e.g., join, group by, aggregation, fuzzy-text matching, path expressions, search, graph operations/primitives, etc.) rather than enumerating all possible queries that can be answered by the data stores. Additionally, for a finer grained description of each supported operation, we also provide a mechanism to express any associated limitations. For example, a `MAX` aggregation function might only be supported over numeric types.

[3] We make a distinction between stored data that is initially placed in the data stores and intermediate data that is generated during query execution.

Table 1 Notation

Notations	Definitions
L	Limitation id
D	Data type (e.g., numeric, text, graph, etc.)
F	Function type (e.g., search, graph, etc.)
OP	Operation type (e.g., JOIN, BFS, etc.)
S	Data store
CP	Capability name
$RL_{S_j}^{d_i}$	Description of representational constraint of a data store S_i in terms of its capability and limitation
H	Hypergraph of a workload
V	Set of concepts associated with a query q
E	Set of hyperedges each representing a summarized query

We define the following general form of a representational constraint RL_S^d to describe the capabilities of a data store S:

$$RL_S^d : \exists S\ \text{Store}(S, CP, L) \leftarrow \text{Capability}(CP) \wedge \text{Limitation}(L, D, F, OP)\ \forall CP\ \forall L\ \forall D\ \forall F\ \forall OP, \quad (1)$$

where Store, Capability and Limitation are expressed as relations in a Datalog rule.

More specifically, each *Capability* associated with a data store is defined in terms of an operation $OP \in$ {JOIN, AGG, ...} that the store can perform on a particular data item of a particular data type. Each capability is associated with a *Limitation* L_{id} that describes the constraints associated with the operation in terms of (1) the data type $D \in$ {NUMERIC, TEXT, ...} that the operation can consume, (2) the function type F that the operation is associated with (e.g., AGG $\in$ {MIN, MAX, AVG, COUNT}), and (3) any specific operation type OP it is associated with (e.g., JOIN $\in$ {INNER, OUTER, LEFT OUTER, ...}). Table 1 summarizes the notation used in this section.

Example 1 The description d_1 shows the capability of a relational store S_r in terms of a JOIN operation. The JOIN operation is further described by the limitation L_{id} that specifies the supported data type NUM (i.e., a numeric column) and the supported operation type INR (i.e., inner join).

$$RL_{S_r}^{d_1} : \text{Store}(S_r, \texttt{JOIN}, L_{id}) \leftarrow \text{Capability}(\texttt{JOIN}), \text{Limitation}(L_{id}, \texttt{NUM}, 0, \texttt{INR})$$

Example 2 The description d_2 shows the capability of a graph store S_g in terms of a graph operation GRAPHOP. The GRAPHOP operation is described by the limitation L_{id} that specifies the supported data type GRAPH, the supported graph function F associated with the operation (i.e., BFS) and the supported type of BFS operation (i.e., single-source BFS).

$$RL_{S_g}^{d_2} : \text{Store}(S_g, \texttt{GRAPHOP}, L_{id}) \leftarrow \text{Capability}(\texttt{GRAPHOP}), \text{Limitation}(L_{id}, \texttt{GRAPH}, \texttt{BFS}, \texttt{SSBFS})$$

Table 2 Sample operations descriptions

	Operation Descriptions
1	$OP_1^d()$ ←Capability(AGG), Limitation(L_i, NUM, SUM, 0)
2	$OP_2^d()$ ←Capability(EQ), Limitation(L_i, STRING, 0, 0)
3	$OP_3^d()$ ←Capability(JOIN), Limitation(L_i, INT, 0, INR)
4	$OP_4^d()$ ←Capability(FM), Limitation(L_i, STRING, 0, 0)
5	$OP_5^d()$ ←Capability(EQ), Limitation(L_i, CTYPE, 0, 0)
6	$OP_7^d()$ ←Capability(SRC), Limitation(L_i, TEXT, FS, 0)

3.1.2 Query operation description

We express the operations required by a given query as a set of *boolean* Datalog rules. Modeling the data store capabilities as well as the operations required by a query as Datalog rules, enables deductive reasoning to evaluate their compatibility and hence, facilitates the process of identifying appropriate stores for the required query operations. Each query operation is expressed as a rule that specifies the type of operation along with the limitations on the type of operands if any.

Example 3 The description OP_1^d shows an inner join query operation with a limitation that the join column can only be integer (INT).

$$OP_1^d() \leftarrow \text{Capability}(\texttt{JOIN}), \text{Limitation}(L_{id}, \texttt{INT}, 0, \texttt{INR}),$$

where the head of the rule is empty.

Table 2 provides a list of operation descriptions represented as Datalog rules for operations such as aggregations (AGG), joins (JOIN), fuzzy matching (FM), exact match or equality predicate (EQ), graph operations (GRAPHOP), search (SRC). Each operation is associated with limitations in terms of the data types that it operates on such as numeric (NUM), string (STRING), integer (INT) complex data types (CTYPE), graph (GRAPH), etc.

3.2 Data placement orchestrator

We design a data placement orchestrator that reasons about the data placement at the level of query operations over the

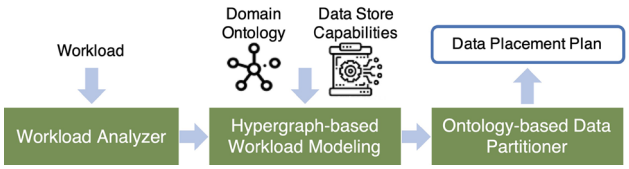


Fig. 4 Data placement orchestrator

domain ontology representing the schema of an enterprise KB. The data placement orchestrator identifies potentially overlapping subsets of the ontology based on the given workload against the KB and the capabilities of the underlying data stores. The orchestrator then outputs a mapping between the identified subsets and the target data stores where they best fit.

Figure 4 shows the major sub-components and workflow of our proposed data placement orchestrator design. The workload is represented as a set of queries expressed against the concepts in the domain ontology. As we are primarily interested in identifying the concepts in the ontology subject to the operations involved in a given workload, we introduce a workload analyzer (Sect. 3.2.1) that creates a summarized representation of the query workload. The summarized query workload is then modeled as a hypergraph where a query is represented by a hyperedge, and the nodes in each hyperedge represent the ontology concepts spanned by the query.

The ontology-based data partitioner (Sect. 3.2.2) groups the concepts and relations in the hypergraph into potentially overlapping subsets based on the query operations. The data corresponding to these subsets is then placed on individual data stores based on their supported operations. All data placement decisions are made at the granularity of the identified ontology subsets, placing all the data of any ontology concept in its entirety. In other words, we do not horizontally partition concepts across different stores.

3.2.1 Workload analyzer & hypergraph-based workload modeling

Workload Analyzer. The workload analyzer takes as input the expected query workload (provided by users or learnt from query logs), and for each query, it creates two sets. The first is a set of domain ontology concepts, that the query accesses. The second is a set of operations (e.g., join, aggregation) that the query performs over those concepts. To generate a summarized representation of the given queries, the workload analyzer groups the queries that access the same set of concepts into a group and then creates a set that combines each query's associated operations in the group. The workload analyzer generates an operational description for each query operation in the combined set.

Table 3 Summarized representation of queries

	Operation Descriptions	Concepts
Q_1	$OP_1^d()$ ←Capability(AGG), Limitation(L_i,NUM,SUM,0)	PublicMetric
	$OP_2^d()$ ←Capability(EQ), Limitation(L_i,STRING,0,0)	PublicMetricData
	$OP_3^d()$ ←Capability(JOIN), Limitation(L_i, INT,0,INR)	PublicCompany
Q_2	$OP_2^d()$ ←Capability(EQ), Limitation(L_i,STRING,0,0)	PublicCompany
	$OP_3^d()$ ←Capability(JOIN), Limitation(L_i,INT,0,INR)	Industry
	$OP_4^d()$ ←Capability(FM), Limitation(L_i,STRING,0,0)	
Q_3	$OP_4^d()$ ←Capability(FM), Limitation(L_i,STRING,0,0)	Company
	$OP_3^d()$ ←Capability(JOIN), Limitation(L_i,INT,0,INR)	InsiderHistory InsiderPerson

Example 4 In Fig. 3, we list three OQL queries (Q_1–Q_3) issued against the concepts in the finance ontology (Fig. 2b). Each query consists of multiple types of operations and not all operations can achieve their best performance on a relational store, such as fuzzy matching. Therefore, we utilize the description of query operations to capture the type of operation along with its limitations to generate the summarized representation of the workload. This way, the representation is not tied to any specific data stores.

Table 3 shows the corresponding summarized representation of the above queries. The table summarizes the operation descriptions, as well as the concepts in the finance ontology (Fig. 2b) that are subjected to these operations for a given workload. The summarized workload expressed over the domain ontology is modeled as a hypergraph, which allows us to use graph analysis techniques to cluster or group concepts together based on the operations performed on them. Then, we make data placement decisions for data corresponding to each clustered group of concepts, based on the operational capabilities of the underlying data stores.

Hypergraph-based Workload Modeling. The workload is modeled as a hypergraph H, which is defined as a triplet $H = (V, E, OP^d)$. A hyperedge $e_i \in E$ represents a summarized query and the set of nodes $V_{e_i} \subseteq V$ spanned by the hyperedge represent the set of concepts accessed by the query. Each hyperedge e_i is also associated with the set of operation descriptions $OP_{e_i}^d \subseteq OP^d$ that are performed by the query.

Example 5 Figure 5 shows a corresponding hypergraph of the summarized representation in Table 3, wherein the hyper-

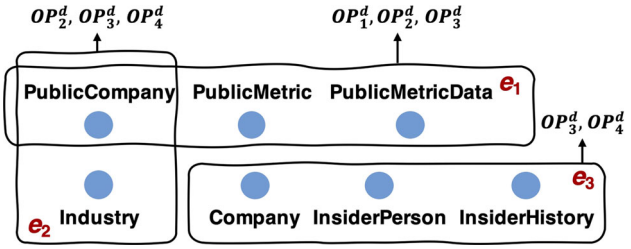


Fig. 5 Concept-level hypergraph

edge e_1 spans over three nodes (concepts) *PublicMetric*, *PublicMetricData*, and *PublicCompany*. The edge e_1 is also associated with three operation descriptions OP_1^d, OP_2^d, and OP_3^d.

3.2.2 Ontology-based data partitioner

The ontology-based data partitioner follows a two-step approach for data placement. First, it runs graph analysis algorithms that we will describe shortly over the hypergraph H representing the summarized workload, to group the concepts in the domain ontology $\mathcal{O}$ based on the similarity of the operations that are performed on these concepts. Second, the data corresponding to these identified subsets of the ontology is then mapped to underlying data stores based on their capabilities RL_S^d, while minimizing the amount of replication required.

The resulting capability-based data placement minimizes data movement and transformation for a given workload at query processing time, and greatly enhances the efficiency of query processing in HERMES. The final output of the data partitioner is a concept-to-store mapping M that maps the ontology concepts to the appropriate data stores. Next, we describe our proposed graph analysis algorithms for deciding data placement.

Note here, that our data placement approach does not explicitly consider any load balancing across different data stores in HERMES. We focus mainly on the capability of the underlying data stores and place data based on the expected workload against the knowledge base. In doing so, we assume that each underlying data store has the capability to transparently take care of system issues such as scalability, load balancing, and handling workload skew.

Operation-based Clustering (*OC*)**Algorithm.** The *OC* algorithm (Algorithm 1) groups concepts based on the operations that they are subject to. For each operation description $OP_i^d \in OP^d$ in the hypergraph H, the algorithm creates a cluster $CL_{OP_i^d}$.The algorithm then iterates over the set of operation descriptions associated with each hyperedge e_i, and for each operation description OP_i^d, it assigns all the concepts spanned by e_i to the cluster $CL_{OP_i^d}$.

Once the concepts have been clustered together, the partitioner assigns each concept cluster $CL_{OP_i^d}$ to a set of data stores S, such that each $S_i \in S$ has a capability description that matches the operation description OP_i^d of the cluster. Finally, the algorithm generates a mapping M that maps each concept in each cluster $CL_{OP_i^d}$ to the set of data stores in S.

Algorithm 1 Operator-based clustering (OC) algorithm

Input: $H = (V, E, OP^d)$, RL_S^d
Output: M
1: Initialize $CL = \phi$
2: **for** each $OP_i^d \in OP^d$ **do**
3: **for** each $e_j \in E$ **do**
4: **if** $OP_i^d \in OP_{e_j}^d$ **then**
5: $CL_{OP_i^d}$.add(e_j)
6: **end if**
7: **end for**
8: CL.add($CL_{OP_i^d}$)
9: **end for**
10: Initialize $M = \phi$
11: **for** each $CL_{OP_i^d} \in CL$ **do**
12: **for** each $RL_{S_i}^{d_i} \in RL_S^d$ **do**
13: **if** OP_i^d.getBody() == $RL_{S_i}^{d_i}$.getBody() **then**
14: **for** each concept $c \in CL_{OP_i^d}$ **do**
15: M.add(c, S_i) // add $\langle c, S_i \rangle$ to a map M
16: **end for**
17: **end if**
18: **end for**
19: **end for**
20: **return** M

Example 6 Given a hypergraph H shown in Fig. 5, we have three data stores S_1, S_2 and S_3. The capability description of S_1 satisfies OP_1^d, OP_2^d and OP_3^d operations' descriptions listed in Table 3, S_2 supports OP_2^d and OP_4^d, and S_3 supports only OP_2^d. Our operator-based clustering algorithm creates four clusters: $CL_{OP_1^d}$, $CL_{OP_2^d}$, $CL_{OP_3^d}$, and $CL_{OP_4^d}$. It then places all concepts that are spanned by hyperedge e_1, namely, *PublicCompany*, *PublicMetric*, and *PublicMetricData* into $CL_{OP_1^d}$, $CL_{OP_2^d}$, and $CL_{OP_3^d}$ clusters. Concepts spanned by hyperedge e_2, i.e., *PublicCompany* and *Industry* are placed into $CL_{OP_2^d}$, $CL_{OP_3^d}$ and $CL_{OP_4^d}$ clusters. Finally, it places the concepts *Company*, *InsiderPerson*, and *InsiderHistorty* in clusters $CL_{OP_3^d}$ and $CL_{OP_4^d}$ clusters.

Once the concepts are clustered by operations, the data placement algorithm maps each concept in the cluster to the set of data stores that support the corresponding operations. The final *concept-store* mapping has *PublicCompany*, *PublicMetric*, *PublicMetricData* concepts mapped to S_1, S_2 and S_3. However, *Industry*, *Company*, *InsiderPerson*, and *InsiderHistorty* concepts are only mapped to S_1 and S_2.

Although the operator-based clustering (*OC*) algorithm described above minimizes data movement at query pro-

cessing time by placing data into stores supporting the corresponding operations, it introduces some replication overheads as the same concept cluster may be placed at multiple stores if their capabilities match the cluster's operations. To further minimize the replication overhead, we propose a more refined Min-Cover (*MC*) algorithm.

Min-Cover (*MC*)**Algorithm.** The Min-Cover algorithm improves the operator-based clustering algorithm by further minimizing the amount of data replication, while still minimizing the data movement at query processing time. Algorithm 2 leverages the minimum set-cover algorithm to find the minimum number of data stores to support the complete set of operations required by each hyperedge in the query workload hypergraph. In fact, it minimizes the span of each hyperedge across the set of data stores that satisfy the set of operations required by the hyperedge.

Algorithm 2 Min-Cover (MC) Algorithm

Input: $H = (V, E, OP^d), C_S^d$
Output: M
1: Initialize $S_{e_i} = \phi$
2: **for** $e_i \in E$ **do**
3: Initialize $I = \phi$
4: **while** $U_{e_i} \neq 0$ **do**
5: $S_i^{OP^d} = argMax_{e_i}(U_{e_i}, RL_S^d)$
6: I.add(S_i)
7: U_{e_i}.remove($S_i^{OP^d}$)
8: **end while**
9: **for** each $S_i \in I$ **do**
10: **for** each concept $c \in V_{e_i}$ **do**
11: M.add(c, S_i)
12: **end for**
13: **end for**
14: **end for**
15: **return** M

For each hyperedge e_i in the hypergraph H, a universe of the set of operation descriptions $U_{e_i} = \{OP_i^d, ..., OP_n^d\}$, where OP_i^d is an operation description associated with the hyperedge e_i, a set of data stores $S = \{S_i, ..., S_n\}$, where each data store S_i has a specific set of capability descriptions $RL_{S_i}^d = \{RL_{S_i}^{d_i}, ..., RL_{S_i}^{d_n}\}$, the algorithm finds the minimum number of data stores $I \subseteq S$ that cover the universe U_{e_i}. Each concept in the hyperedge e_i is then mapped to a store $S_i \in I$. Once the algorithm iterates through all hyperedges, it produces the final concept-to-store mapping M.

Example 7 Continuing with our example in Fig. 5, the Min-Cover algorithm produces a *concept-store* mapping M, where *PublicCompany*, *Industry*, *Company*, *InsiderPerson*, and *InsiderHistory* concepts are mapped to S_1 and S_2. *PublicMetricData* and *PublicMetric* concepts are mapped to S_1. No concepts are mapped to S_3, as S_1 and S_2 cover all required operations for the hypergraph.

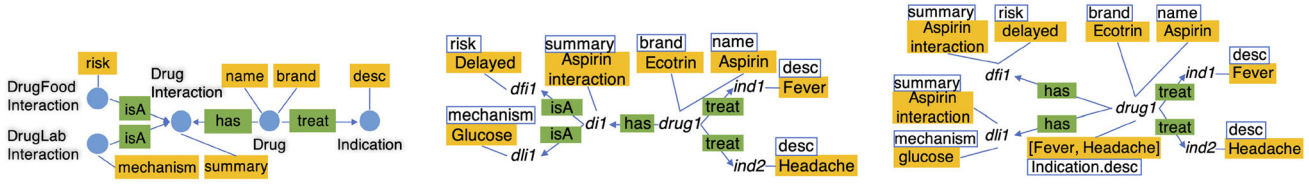
Discussion. The workload-aware data placement in HERMES may become sub-optimal as the query workloads can be uncertain or change overtime in real-world applications. This is due to the fact that HERMES assumes a static workload and produces its data placement plan based on the summarized representation of this workload. However, the Min-Cover algorithm works very well in practice and is robust to small changes in the workload, such as selection columns, join, and filter predicates, as these operations essentially require the same capabilities from the underlying data stores and the data placement remains optimal without any re-optimization.

In case of a significant workload shift, that subjects the ontology concepts to a different set of operations, the data placement would become sub-optimal and then, HERMES would require to re-optimize the data placement plan accordingly. Consequently, the data would need to be re-partitioned among the data stores, which can be expensive. Incremental re-partitioning mechanisms such as [50] can be leveraged for reducing the cost of such data migration. Further investigation is left as future work.

4 Schema optimization

Given a data placement plan, the goal of HERMES's schema optimization is to optimize the data schema on each data store for high-performance query and analysis. As mentioned in Sect. 2, schema optimization in relational and NoSQL stores has been extensively studied [12,36,46]. In this paper, we adopt the correlation-aware approach [36] for the relational store, and exploit NoSE [46] for our NoSQL store. In the rest of this section, we focus on schema optimization for the property graph store. We argue that the graph query performance varies vastly for different property graphs with the same data but corresponding to different schemas. We illustrate this using two examples from the medical domain.

Example 8 (Pattern Matching Query) Consider the ontology in Fig. 6a, *summary* is a property of *DrugInteraction* concept, which is connected to *DrugFoodInteraction* and *DrugLabInteraction* concepts via inheritance (*isA*) relationships. Figure 6b and c shows two alternative property graphs conforming to two different schemas with several vertices and edges. In Fig. 6b, the vertex *di1* (i.e., an instance of *DrugInteraction*) leads to both *dfi1* and *dli1*. In Fig. 6c, *drug1* directly connects to *dfi1* and *dli1* vertices. For any query that requires edge traversals from *drug1* to either *dfi1* or *dli1* or both, the property graph 2 clearly 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. 6 Motivating example

Example 9 (Aggregation Query) In Fig. 6a, *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. Figure 6c 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). In this case, avoiding the edge traversals is extremely beneficial, especially when the number of edges between these 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. Moreover, the rich semantic relationships in an ontology provide a variety of opportunities to reduce graph traversals. To generate an optimized graph schema, we need to identify and exploit these opportunities in the ontology, and design different techniques to utilize them accordingly.

Definition 3 (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.

A property graph schema $\mathcal{PGS}$ can be specified in a data definition language such as Neo4j's Cypher [30], TigerGraph's GSQL [26], etc. 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 4 provides the notations used in the property graph schema optimization.

Definition 4 (Problem Definition - Schema Optimization) 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, any practical solution needs incorporate a space constraint while producing an optimized property graph schema.

Table 4 Notation

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
$\mathcal{PGS}$	a property graph schema
vs_i	$vs_i \in VS$: a schema vertex
$vs_i.PS_i$	all property schema of 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

Figure 7 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[4]. 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 data store (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.

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

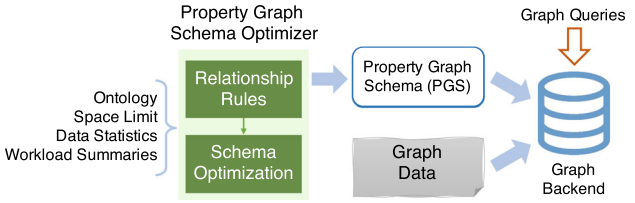
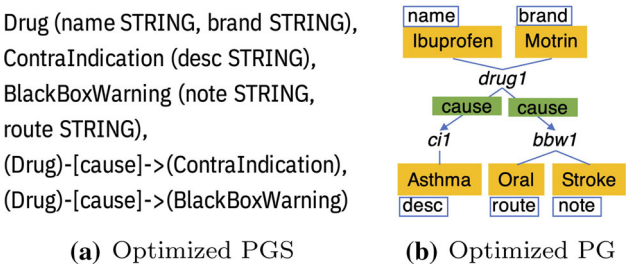


Fig. 7 Schema optimization overview



(a) Optimized PGS **(b)** Optimized PG

Fig. 8 Union relationship

4.1 Relationship rules

Graph queries often involve multi-hop traversal or vertex attribute lookup/analytics on property graphs. As shown in Fig. 6, 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. These rules minimize edge traversals and consequently improve graph query performance.

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 each instance of a member concept is also an instance of the union concept. For example, *BlackBoxWarning* and *ContraIndication* are two member concepts of a union concept *Risk*. A graph query accessing an instance of *Risk* is equivalent to accessing the instances of either *BlackBoxWarning*, or *ContraIndication*, or both. A query 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.

Algorithm 3 Union Rule (union)

Input: A union relationships r_{un}
1: $vs_i \leftarrow r_{un}.src$ // the union concept of r_{un}
2: $vs_j \leftarrow r_{un}.dst$ // the member concept of r_{un}
3: **for each** $r \in vs_i.ES_i$ **do**
4: **if** $\neg$(r of type $union$) **then**
5: $vs_j.ES_j \leftarrow vs_j.ES_j \cup r$
6: **end if**
7: **end for**
8: **return** vs_i, vs_j

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 (Algorithm 3). Figure 8a and b shows 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 design the inheritance rule to optimize the schema to be more compact and precise regarding the concepts associated with inheritance relationships. 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 possible scenarios.

1. Connect the child node vs_j directly to the nodes that are connected to its parent node vs_i, 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*.

In the first two cases, edge traversals can be avoided in the property graph conforming to the property graph schema. Figure 2a shows that *DrugFoodInteraction* and *DrugLabInteraction* are two children concepts of *DrugInteraction*. Applying the inheritance rule to these concepts can lead to two alternative optimized property graph schemas shown in Fig. 9.

However, attaching the data properties ($c_i.P_i$) from the parent concept to the child concept incurs data replication as $c_i.P_i$ is shared among all children concepts (Fig. 9b). If the number of data properties shared by the children concepts is large, the data replication can introduce significant space overhead. On the other hand, when the data properties ($c_j.P_j$) from the children concepts are replicated to their parent concept (c_i), c_i may end up with a large number of data properties (Fig. 9d). However, these data properties may not exist in many instance vertices of c_i. Consequently, the instance vertices of c_i may consume unnecessary space. To remedy the above two issues, we propose to exploit the Jaccard similarity [41] between $c_i.P_i$ and $c_j.P_j$ to decide the best strategy for the inheritance relationship:

$$JS(c_i.P_i,\ c_j.P_j) = |c_i.P_i \cap c_j.P_j| \ / \ |c_i.P_i \cup c_j.P_j|. \quad (2)$$

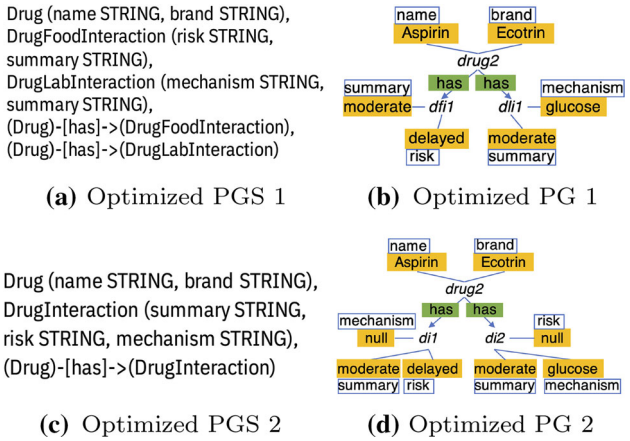


Fig. 9 Inheritance relationship

As described in Algorithm 4, if $JS(c_i.P_i, c_j.P_j)$ is greater than a threshold θ_1, it indicates that the child concept c_j shares a lot of data properties with its parent concept c_i. In this case, moving $c_j.P_j$ from the child concept to c_i incurs less space overhead compared to the other way. Similarly, if $JS(c_i.P_i, c_j.P_j)$ is less than a threshold θ_2 ($\theta_2 \leq \theta_1$), the child concept c_j has little in common with its parent c_i. Therefore, it is more cost effective to make the data properties of the parent concept $c_i.P_i$ available at c_j. In either case, the inheritance rule avoids edge traversals in the resulting property graph. The Jaccard similarity is computed based on the original ontology, as it represents the semantic similarity between two concepts with an inheritance relationship.

Algorithm 4 Inheritance Rule (inheritance)

Input: An inheritance relationship r_{ih}
1: $vs_i \leftarrow r_{ih}.src$ // Parent concept
2: $vs_j \leftarrow r_{ih}.dst$ // Child concept
3: $jsim \leftarrow JS(vs_i.PS_i, vs_j.PS_j)$ // Jaccard similarity of r_{ih}
4: **if** $jsim > \theta_1$ **then**
5: $vs_i.P_i \leftarrow vs_i.PS_i \cup vs_j.PS_j$
// ES_{ih} is the set of inheritance relationships
6: $vs_i.ES_i \leftarrow (vs_i.ES_i \cup vs_j.ES_j) \backslash r_{ih}$
7: **else if** $jsim < \theta_2$ **then**
8: $vs_j.PS_j \leftarrow vs_j.PS_j \cup vs_i.PS_i$
9: $vs_j.ES_j \leftarrow (vs_j.ES_j \cup vs_i.ES_i) \backslash r_{ih}$
10: **end if**
11: **return** vs_i, vs_j

One-to-one Rule. A *1:1* relationship ($r_{1:1} = (c_i, c_j)$) indicates that an instance of c_i can only relate to one instance of c_j and vice versa (e.g., *Indication* and *Condition* in Fig. 2a). 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 (Algorithm 5), 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

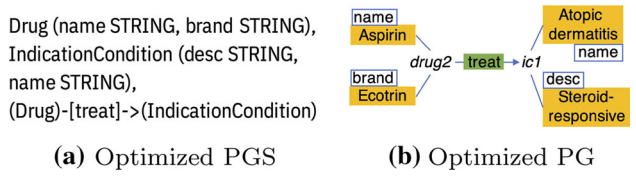


Fig. 10 1:1 relationship

are merged, a join can be saved when two tables are queried together.

Algorithm 5 1:1 Rule (oneToOne)

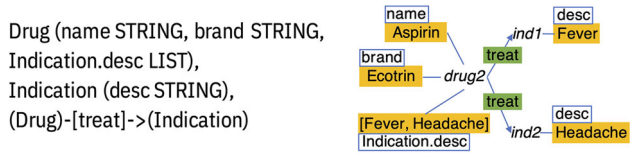
Input: A 1:1 relationship $r_{1:1}$
1: $vs_i \leftarrow r_{1:1}.src$
2: $vs_j \leftarrow r_{1:1}.dst$
3: $vs_{i,j} \leftarrow \emptyset$
4: $vs_{i,j}.ES_{i,j} \leftarrow (vs_i.ES_i \cup vs_j.ES_j) \backslash r_{1:1}$
5: $vs_{i,j}.PS_{i,j} \leftarrow vs_i.PS_i \cup vs_j.PS_j$
6: **return** $vs_{i,j}$

In Fig. 10a, *IndicationCondition* is the merged concept with two data properties, *name* and *note*, attached. Hence, the edge traversal from *Drug* to *Condition* in Fig. 2a is avoided and the number of instance vertices (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 instances of c_j). In other words, in a *1:M* relationship, an instance of c_i allows zero, one, or many corresponding instances of c_j. However, an instance of c_j cannot have more than one 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. 11a). The aggregation and neighborhood lookup functions can directly leverage these localized list properties instead of traversing through the edges of the *1:M* relationships. 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. As depicted in Fig. 11b, *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*). The potential savings can be substantial when there are many edges between instance vertices of two concepts such as *Drug* and *Indication*.

However, the newly introduced property of type *LIST* introduces additional space overheads, which can be expensive depending on the data distribution. Therefore, choosing



(a) Optimized PGS **(b)** Optimized PG

Fig. 11 1:M relationship

the appropriate set of data properties from each *1:M* relationship to propagate is critical with respect to both query performance and space consumption. Algorithm 6 corresponds to the one-to-many rule.

Algorithm 6 1:M Rule (oneToMany)

Input: A 1:M relationship $r_{1:M}$
1: $vs_i \leftarrow r_{1:M}.src$
2: $vs_j \leftarrow r_{1:M}.dst$
3: **for each** $p \in vs_j.PS_j$ **do**
4: $\quad vs_i.PS_i$.addAsList(p)
5: **end for**
6: **return** vs_i, vs_j

Many-to-many Rule. An *M:N* relationship ($r_{M:N} = (c_i, c_j)$) indicates that an instance of c_i can have several corresponding instances of c_j, and vice versa. An M:N relationship 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. Namely, in the optimized schema, a data property of the node vs_i corresponding to c_i in $\mathcal{O}$ is propagated as a property of type *LIST* to the node vs_j corresponding to c_j in $\mathcal{O}$, and vice versa. 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 additional space consumption.

In summary, all proposed rules reduce the number of edge traversals which improve graph query performance. The *one-to-one* rule simply combines nodes together to avoid edge traversals while reducing the number of nodes in the graph. Both *union* and *inheritance* rules introduce new edges to bring nodes closer. Both *one-to-many* and *many-to-many* rules replicate data properties between nodes to improve the aggregation and 1-hop lookup functions in graph queries. Hence, *union*, *inheritance*, *one-to-many*, and *many-to-many* rules incur space overheads. In Sect. 4.2, we introduce our property graph schema optimization, trading off performance gain and space overhead.

4.2 Property graph schema optimization

To produce an optimized property graph schema, we need to determine how to utilize the proposed rules described in Sect. 4.1. A straightforward approach is to iteratively apply these rules in order and generate the property graph schema.

Specifically, Algorithm 7 takes as input an ontology O and first computes the Jaccard similarity scores for all inheritance relationships (Lines 1-2). Then, it iteratively applies the appropriate rule to each relationship in the ontology (Lines 3-16). At the end of each iteration, it checks if the ontology converges (Line 17). Finally when no more rule applies, a property graph schema is generated (Lines 18-19). In fact, these rules can be applied in any order, and the generated property graph schema is always the same.

Algorithm 7 Ontology to PGS without Space Limits

Input: Ontology $\mathcal{O} = (C, R, P)$
Output: A property graph schema $\mathcal{PGS}$
1: **for each** $r \in R$ of type *inheritance* **do**
2: $\quad r.js \leftarrow$ computeJS(r)
3: **end for**
4: $PGS \leftarrow \emptyset$
5: **repeat**
6: $\quad PGS_{prev} \leftarrow PGS$
7: $\quad$ **for each** $r \in R$ **do**
8: $\quad\quad$ **switch** $r.type$ **do**
9: $\quad\quad\quad$ **case** *1:1*
10: $\quad\quad\quad\quad PGS \leftarrow PGS \cup$ oneToOne(O, r)
11: $\quad\quad\quad$ **case** *1:M*
12: $\quad\quad\quad\quad PGS \leftarrow PGS \cup$ oneToMany(O, r)
13: $\quad\quad\quad$ **case** *M:N*
14: $\quad\quad\quad\quad PGS \leftarrow PGS \cup$ manyToMany(O, r)
15: $\quad\quad\quad$ **case** *union*
16: $\quad\quad\quad\quad PGS \leftarrow PGS \cup$ union(O, r)
17: $\quad\quad\quad$ **case** *inheritance*
18: $\quad\quad\quad\quad PGS \leftarrow PGS \cup$ inheritance(O, r)
19: $\quad$ **end for**
20: **until** $PGS = PGS_{prev}$
21: **return** PGS

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 Let $O = (C, R, P)$ be an ontology given as input to Algorithm 7, and let $O_{out} = (C_{out}, R_{out}, P_{out})$ be the resulting ontology, which is used in Line 18 to produce the output $\mathcal{PGS}$. Proving Theorem 1 is equivalent to proving that applying the rules for any $R' \subseteq R$ in any order will yield the same result O_{out}. The theorem trivially holds when $|R'| = 0$ ($O_{out} = O$), and when $|R'| = 1$ (only one rule can be triggered).

Base case. $|R'| = 2$, i.e., for any two relationships, applying the rules in any order yields the same result. Since we

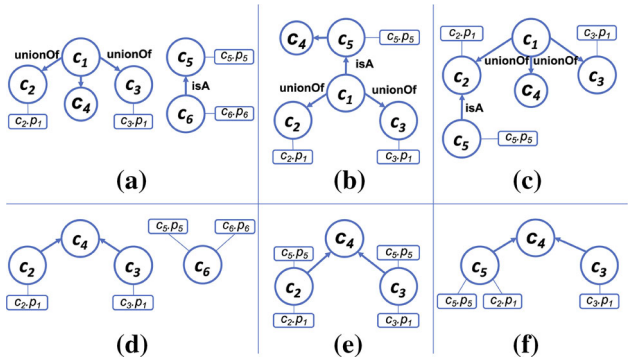


Fig. 12 Union and inheritance rules independence

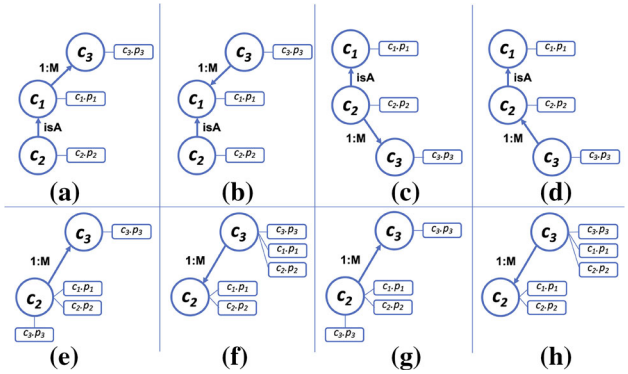


Fig. 13 Inheritance and 1:*M* rules independence

only have two relationships, only two rules will be triggered if the relationships are of different types, or one rule will be triggered twice if the two relationships are of the same type.

(i) Union and Inheritance. To prove that union and inheritance rules are order-independent, we examine all the cases in which those two rules may be triggered in the same graph. We assume that the Jaccard similarity between the two concepts connected with an inheritance relationship is less than θ_2 (Algorithm 4), so the inheritance rule is triggered and the properties of the parent concept are copied to the child concept. It is straightforward to apply the following observations to the case in which the Jaccard similarity is greater than θ_1 as well. Figure 12 contains more than two relationships, but only two relationships are sufficient to prove the case. The additional relationships shown are for illustration purpose only.

In the trivial case of Fig. 12a, the source and destination concepts of the union and inheritance relationships are not inter-connected. If we apply the union rule first, we will end up with the left part of Fig. 12d, leaving the right part of Fig. 12a unchanged, and if we apply the inheritance rule first, we end up with the right part of Fig. 12d, leaving the left part of Fig. 12a unchanged. In both cases, applying the second rule generates the graph of Fig. 12d.

The case shown in Fig. 12b is more complex, where the same concept (c_1) corresponds to a union concept and a child concept. Applying the union rule first, we remove c_1 and connect its member concepts c_2 and c_3 to c_5 through inheritance relationships. Then, the inheritance rule is triggered, removing c_5, copying its properties to its new children c_2 and c_3, and connecting them to c_4, as shown in Fig. 12e. If we apply inheritance first, instead of union, then we first remove c_5, copy its properties to c_1 and connect c_1 to c_4. Then, applying the union rule, we remove c_1 and connect the member concepts c_2 and c_3 to c_4, again resulting in the graph of Fig. 12e.

In a similar way, we can show that union and inheritance rules are order-independent in the case of Fig. 12c, in which the same concept (c_2) corresponds to a member concept and a parent concept. If we apply the union rule first, we remove c_1 and connect the member concepts c_2 and c_3 to c_4. Then, applying the inheritance rule, we remove c_2, copy its properties to c_5, and connect c_4 to c_5, resulting in the graph of Fig. 12f. If we apply the inheritance rule first, we remove c_2, copy its properties to c_5, and connect c_1 to c_5 through a union relationship. Finally, we apply the union rule and remove c_1, connecting c_4 to c_5 and c_3, also resulting in the graph of Fig. 12f.

(ii) Inheritance and 1:M. We follow a similar strategy to prove that inheritance and 1:M rules are order-independent, enumerating all possible cases in which those two rules may be triggered in the same graph. Again, we assume that the Jaccard similarity between the two concepts connected with an inheritance relationship is less than θ_2, so the inheritance rule is triggered and the properties of the parent concept are copied to the child concept.

We skip the trivial case in which the inheritance and 1:M relationships are not related, and start with the case depicted in Fig. 13a, where the parent concept c_1 is also the source concept of an 1:M relationship. If we apply inheritance first, then we copy the properties of c_1 to c_2, remove c_1 and connect c_2 to c_3 through a 1:M relationship. Then, we apply the 1:M rule and copy c_3's properties to c_2, resulting in the graph of Fig. 13e. If we apply the 1:M rule first, then we first copy the properties of c_3 to c_1 and then we apply inheritance to copy the properties of c_1 (also including the properties of c_3) to c_2, remove c_1 and connect c_2 to c_3 through a 1:M relationship, resulting again in the graph of Fig. 13e.

In the case of Fig. 13b, the parent concept (c_1) is now also the destination of an 1:M relationship. If we apply inheritance first, then we copy the properties of c_1 to c_2, remove c_1 and connect c_3 to c_2 through a 1:M relationship. Then, we apply the 1:M rule and copy c_2's properties to c_3, resulting in the graph of Fig. 13f. If we apply the 1:M rule first, then we first copy the properties of c_1 to c_3 and then we apply inheritance to copy the properties of c_1 to c_2, remove c_1 and connect c_3 to c_2 through a 1:M relationship. Finally, we apply 1:M rule

again and copy the properties of v_2 to v_3, again resulting in the graph of Fig. 13f.

In Fig. 13c, c_2 is a child and a source concept of a 1:M relationship. In short, if we apply inheritance first, we remove c_1 and copy its properties to c_2 and then we apply 1:M and also copy the properties of c_3 to c_1, resulting in Fig. 13g. If we apply 1:M first, we copy the properties of c_3 to c_2 and then we apply inheritance to copy the properties of c_1 to c_2 and remove c_1, again resulting in Fig. 13g.

Finally, in Fig. 13d, c_2 is a child and a destination concept of a 1:M relationship. If we apply inheritance first, we remove c_1 and copy its properties to c_2 and then we apply 1:M and copy the properties of c_2 (including the properties of c_1) to c_3, resulting in the graph of Fig. 13h. If we apply 1:M first, we copy the properties of c_2 to c_3 and then we apply inheritance to copy the properties of c_1 to c_2 and remove c_1. Again, we need to trigger the 1:M rule once more to copy the properties of c_2, now also including the properties of c_1, to c_3 and get the graph of Fig. 13h. For the remaining pairs of rules (iii) – (vi), we can follow the same strategy and prove that they are order-independent for all possible cases.

Induction hypothesis. Applying the rules in any order for any $R' \subseteq R$, where $|R'|$=n, always results in the same O'. Then, applying the rules in any order for any $R'' \subseteq R$, such that $|R''| = n$+1 and $R' \subset R''$, will always result in the same O'', since there is only one additional relationship in R'' compared to R', and only one possible rule corresponding to this new relationship can be triggered. □

While the naïve approach harnesses all potential optimization opportunities aggressively, it incurs space overheads from *union*, *inheritance*, *1:M*, and *M:N* rules. In cases where the number of such relationships is large in the ontology, this can be expensive with respect to the space consumption, especially in a cluster setting, where many large-scale property graphs co-exist. Hence our goal is to produce an optimized property graph schema for a given space limit. The quality and the space consumption of an optimized property graph schema are measured based on the total benefit and cost (i.e., space consumed) by applying the rules (given by Eqs. 4-6 in Sect. 4.2.2).

Definition 5 (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 budget. $\mathcal{PGS}_{opt} \in \mathbb{PGS}$ is an optimal property graph schema if $\nexists\ \mathcal{PGS}' \in \mathbb{PGS}$ such that Benefit($\mathcal{PGS}'$) > $Benefit(\mathcal{PGS}_{opt})$.

Finding an optimal property graph schema is exponential in the number of concepts and relationships in the ontology, which is practically infeasible. Hence, we need to design efficient heuristics to produce a near-optimal property graph schema. To achieve this goal, we propose two property graph schema optimization algorithms that 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 allow us to identify and prioritize the more beneficial relationships when applying *union, inheritance, one-to-many* and *many-to-many* rules, such that the space can be used more efficiently.

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 it is imperative to apply the above rules to these relationships with high frequency.

In case of no prior knowledge about access frequency, we assume that it follows a uniform distribution. Our approach can also handle updates (i.e., insert, delete, and modify) to the property graph if they do not incur any schema changes. If the accumulated updates change the data distributions, then we can apply the rules locally to the affected part of the ontology. Note that data statistics changes can invalidate certain rule applied earlier, or can trigger new rules, especially inheritance and union rules. We can make local adjustments to accommodate these changes. Minimizing such transformation overheads is left as future work.

4.2.1 Concept-centric algorithm

As described in Sect. 2.1, 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 [51]. We expect these key concepts to be queried more frequently than others. This leads to our concept-centric algorithm that exploits the structural information in an ontology to identify key concepts and thus provides an estimation of the expected workload over the ontology.

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 [19] as its underlying assumption, more important websites likely to receive more links from other websites, is similar to our intuition of key concepts. In this work, we utilized a modified PageRank algorithm, called *OntologyPR* [13], to determine the centrality score of each concept in an ontology. Com-

pared to PageRank, the *OntologyPR* is customized to several unique features specific to ontologies such as inheritance and unions. Below we describe these designs and the *OntologyPR* algorithm can be found in [13].

Inheritance. To cater for inheritance relationships, we remove these relationships from the ontology while running the initial PageRank algorithm. This allows us to calculate the page ranks of a concept based on the links from other concepts that are not children of the same concept. The reason is that a parent concept would accumulate a significant amount of weight from its children and grandchildren, which does not truly reflect the importance of the parent concept. On the other hand, a child concept would also inherit its parent weight, which introduces noise into the centrality estimation. After computing the page rank values of all concepts, we reattach these relationships and update the page ranks of each concept by doing a depth-first traversal over its inheritance relationships to find the parent with the highest page rank. If this value is higher than the current page rank of the concept, we use this value as the new page rank of the concept. This enables a child concept to inherit the page rank of its parent.

Unions. The union concept in an ontology represents a logical membership of two or more concepts. Any incoming edge to a union concept can therefore be considered as pointing to at least one of the member concepts of the union. Similarly each outgoing edge can be considered as emanating from at least one of the member concepts. To handle union concepts, the *OntologyPR* algorithm iterates over all incoming and outgoing edges to/from the union concept. For each incoming edge to the union concept, we create new edges between the source concept and each of the member concepts of the union. For each outgoing edge, similarly, we create new edges 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 toward centrality analysis has already been accounted for by the new edges to/from the member concepts of the union.

Out-degree of Concepts. In the default PageRank algorithm, the weight distribution of the page rank is proportional to the in-degree of a node as it receives page rank values from all its neighbors that point to it. In other words nodes with a high in-degree would tend to have a higher page rank than nodes with a low in-degree. 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.

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.

$$Score(c_i) = \frac{c_i.pr \times AF(c_i)}{Size(c_i)} \tag{3}$$

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 8 Concept-Centric Algorithm

Input: Ontology $\mathcal{O} = (C, R, P)$, space limit S
Output: A property graph schema $\mathcal{PGS}$
1: $O \leftarrow$ ontologyPR(O)
2: $C_{srt} \leftarrow$ sort(C)
3: **for each** $c \in C_{srt}$ **do**
4: **for each** $r \in c.R$ **do**
5: $S' \leftarrow S$
6: $O, S \leftarrow$ applyRules(r, S')
7: **if** $S < 0$ **then**
8: **break**
9: **end if**
10: **end for**
11: **end for**
12: $PGS \leftarrow$ generatePGS(O)
13: **return** PGS

Based on Eq. 3, our concept-centric algorithm (Algorithm 8) 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 (Sect. 4.1) to the relationships connecting to c. During this process, the algorithm updates the space limit as it is consumed by the rules. Once the space is fully exhausted, the algorithm terminates (Lines 7-8) and returns the optimized property graph schema (Line 10).

4.2.2 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. To address this issue, we propose the relation-centric algorithm based on a cost-benefit model for each type of relationships.

Cost Benefit Models. The union rule, introduced in Sect. 4.1, 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. Formally:

$$\begin{aligned} Benefit(r) &= AF(c_i \xrightarrow{r} c_j) \\ Cost(r) &= \textstyle\sum_{r' \in (c_i.R_i \setminus R_{un})} |r'|, \end{aligned} \tag{4}$$

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[5] 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:

$$\begin{aligned} Benefit(r) &= AF(c_i \xrightarrow{r} c_j.P_j) \times JS(c_i, c_j) \\ Cost(r) &= \begin{cases} \sum_{p \in c_j.P_j} |c_j| \times p.type + \sum_{r \in (c_j.R_j \setminus R_{ih})} |r| \\ \qquad \text{if } \theta_1 < JS(c_i, c_j), \\ \sum_{p \in c_i.P_i} |c_i| \times p.type + \sum_{r \in (c_i.R_i \setminus R_{ih})} |r| \\ \qquad \text{if } JS(c_i, c_j) < \theta_2, \end{cases} \end{aligned} \tag{5}$$

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, 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 defined as:

$$\begin{aligned} Benefit(r) &= AF(c_i \xrightarrow{r} c_j.p) \\ Cost(r) &= |r| \times p.type, \end{aligned} \tag{6}$$

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 Sect. 4.1, each *M:N* relationship is equivalent to two *1:M* relationships. Thus, we first convert each *M:N* relationship in the ontology into two *1:M* relationships, and then use Eq. 6 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.

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

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 0/1 Knapsack Problem, which is NP-hard [63].

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 [13]. □

Here, we adopt the fully polynomial time approximation scheme (FPTAS) [63] for our relation selection problem, which guarantees that the benefit of the optimized property graph schema $Benefit(\mathcal{PGS})$ is within 1-ϵ ($\epsilon > 0$) bound to the benefit of the optimal property graph schema $Benefit(\mathcal{PGS}_{opt})$.

Algorithm 9 takes as inputs an ontology and the space limit. Similar to Algorithm 7, it computes the Jaccard similarity scores for all inheritance relationships (Lines 1-3). Then it computes the cost and benefit for each relationship in the ontology $\mathcal{O}$ using Eqs. 4, 5, and 6 (Lines 4-8). Next, the FPTAS algorithm is used to select the near-optimal subset of relationships R_{opt} with the given space limit S (Line 9). In *applyRules* procedure, the algorithm applies the corresponding rules; $r \in R_{opt}$ (Lines 10-12). Lastly, an optimized property graph schema is generated (Lines 13-14).

Algorithm 9 Relation-Centric Algorithm

Input: $\mathcal{O} = (C, R, P)$, space limit S
Output: A property graph schema $\mathcal{PGS}$
1: **for each** $r \in R$ of type $inheritance$ **do**
2: $r.js \leftarrow$ computeJS(r)
3: **end for**
4: $Benefit, Cost \leftarrow \emptyset$
5: **for each** $r_i \in R$ **do**
6: $Benefit[i] \leftarrow$ Benefit(r_i)
7: $Cost[i] \leftarrow$ Cost(r_i)
8: **end for**
9: $R_{opt} \leftarrow knapsack(R, Benefit, Cost, S)$
10: **for each** $r_i \in R_{opt}$ **do**
11: $O \leftarrow$ applyRules(r_i)
12: **end for**
13: $PGS \leftarrow$ generatePGS(O)
14: **return** PGS

5 Experimental study

5.1 Experimental setup

Infrastructure. We instantiated HERMES with three different types of backend data management systems, including a rela-

tional DBMS (Db2[6]), a document store (Watson Discovery Services (WDS) built on top of Elasticsearch), and a graph store (either Neo4j [5] or JanusGraph [4]). Each provides different query processing capabilities. For example, WDS is a document-oriented data store for complex search queries. However, it lacks support for full SQL-style joins as opposed to Db2. Neo4j and JanusGraph are the graph database platforms specialized in various complex graph processing tasks.

Data Sets. We use the following two data sets and their corresponding ontologies.

1. Financial data set (*FIN*) [54] includes data from two main sources: Securities and Exchange Commission (SEC) [7] and Federal Deposit Insurance Corporation [2]. The size of the data set is approximately 53 GB. The corresponding financial ontology contains 28 concepts, 96 properties, and 138 relationships.
2. 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.

Methodology and Metrics. To evaluate the effectiveness of the data placement algorithms, we choose query workloads over *FIN* and *MED* based on the most commonly seen operations in their respective application. Specifically, our workloads consist of a variety of select-project-join (SPJ) and aggregation queries similar to Q_1–Q_3 shown in Fig. 3. Fuzzy-text matching, top-k operation, range predicates, and graph operations (i.e., graph pattern matching, vertex property lookup, graph analytical) are also involved in both workloads on *FIN* and *MED* knowledge bases.

To evaluate the quality of the property graph schema 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 Benefit Ration $BR = \frac{B_{SC}}{B_{NSC}}$, where B_{NSC} is the total benefit score of the property graph schema generated by Algorithm 7 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 [30] and Gremlin [3], 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 financial and medical data sets. The details of these query workloads are described in Sect. 5.5. 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 sequential order. We also use the number of edge traversals required in a query as the second metric.

5.2 Effectiveness of data placement algorithms

In this experiment, we evaluate the effectiveness of our data placement algorithms and compare them with alternative approaches for different query workloads. Each of our capability-based data placement algorithms stores data in one or more data stores based on the operations in a given workload and the capability of the data store. We therefore evaluate the effectiveness of the data placement algorithms in terms of the amount of replication needed to guarantee no stored[7] data movement at query processing time for a given workload.

We compare our two data placement algorithms, *MC* and *OC*, with two alternative techniques, including full replication (*FULL*), which replicates each data item across all data stores, and ideal data placement with minimal data replication (*IDEAL*). IDEAL is computed by an exhaustive search among all possible data placement plans. For each candidate data placement plan, we compute its replication ratio and choose the one with the minimal ratio. We study the data replication overhead of these algorithms by varying the replication ratio from the query workloads over both *FIN* and *MED* knowledge bases. The replication ratio is defined as $RR = \sum R_{c_i}/|C|$, $\forall\ c_i$ in C, where R_{c_i} is the number of stores that are required to support the operations on each concept c_i accessed in the query workload, and C is the total number of concepts in the ontology. The replication overhead is defined as $Overhead = (RR_{act} - RR)/RR$, where RR_{act} is the actual replication ratio required by the above methods. Intuitively, the closer the actual data replication is to RR, the more effective the data placement algorithm will be. Hence the replication overhead of the *IDEAL* method is always 0, which is not shown in Fig. 14.

As depicted in Fig. 14, all algorithms generate identical data placement plan without any overhead in the extreme case ($RR = 3$). In other cases, *MC* algorithm consistently outperforms the alternative approaches with minimal data replication overheads. The reason is twofold. First, we exploit the minimum set cover algorithm that is able to identify the optimal solution in most cases [55]. Second, the number of backend data stores used in the experiments is 3. Hence, the

[6] Db2 is a registered trademark of IBM Corporation

[7] We make a distinction between stored data that is initially placed in the data stores and intermediate data that is generated during a query execution.

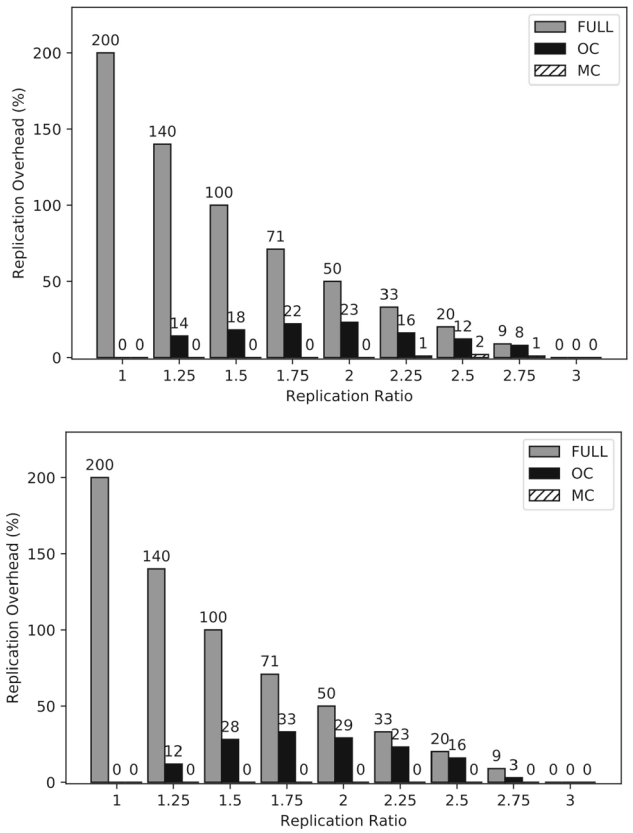


Fig. 14 Data replication overhead (top: MED, bottom: FIN)

```
Q1: SELECT SUM (PMD.metric_value) AS total, COMPINFO.id,
     PMD.year_calendar
FROM PublicMetricData PMD, PublicMetric PM,
     PublicCompany PC, Document DOC,
     Companyinfo COMPINFO
WHERE DOC.Self MATCH ('FISCAL STRESS') AND
      PC->hasFinancialReport= DOC AND
      DOC->Document_companyinfo=COMPINFO AND
      PM.metric_name = 'REVENUES' AND
      PMD.period_type = 'yearly' AND
      PMD->forMetric = oPublicMetric AND
      PMD->forPublicCompany = PC AND
      PMD.year_calendar >= '2012' AND
      PMD.year_calendar <= '2017'
GROUP BY PMD.year_calendar, COMPINFO.id
HAVING total > 1000000

Q2: SELECT DR.drugName, CI.contraindicationText
FROM  Drug DR, Contraindication CI, AgeRange AR, Indication IND,
      Recommendation REC, RecommendationRating RR, Document DOC
WHERE DOC.treatment MATCH ('Peptic ulcer disease') AND
      DR->HasDocumnet = DOC AND
      CI->toDrug = DR AND
      IND->ForDrug = DR AND
      REC->ToIndication = IND AND
      REC>ToRecommendationRating = RR AND
      REC->ToAgeRange = AR AND
      RR.recommendationRatingName = 'Class IIb' AND
      AR.age = 'Adult'
ORDER BY CI.contraindicationOrder
FETCH FIRST 5 ROWS ONLY

Q3: SELECT oPatient.Patient_ID
FROM Patient P, Observe O, SNOMED S, GRAPH G
WHERE P->hasObservation = O AND
      O->hasCondition = S AND
      G.Self MATCH ('S.condition ~ 381') AND
      G->hasSNOMEDDisease = S
```

Fig. 15 Micro-benchmark OQL queries

possibility of placing data to an unnecessary data store is relatively low. Our experience with various uses cases suggest that a small number of stores are often sufficient to support a variety of query types over domain-specific enterprise knowledge bases.

On the contrary, *OC* algorithm can only produce close to optimal data placement plan when the replication ratio is very low (i.e., $RR = 1.25$). It is not as robust as *MC* algorithm when the complexity of the given workload increases. Comparing *FIN* and *MED*, we observe that the quality of the data placement plan produced by *OC* algorithm over *MED* is slightly better than the one produced over *FIN*. The reason is that our *MED* knowledge base consists of fewer concepts compared to *FIN*. Thus, the replication overhead incurred by *OC* is relatively low.

5.3 Impact of data placement on query execution

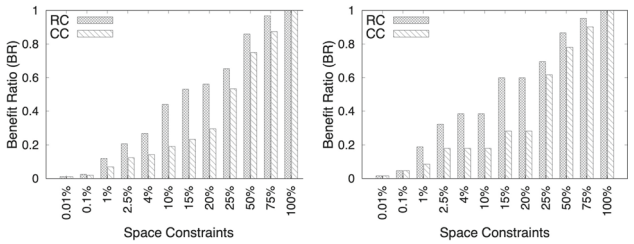
We evaluate the impact of our data placement algorithms on the performance of query execution with a micro-benchmark, consisting of a few representative queries selected from the workloads over FIN and MED (Fig. 15). Queries Q_1 and Q_2 require operations supported by a relational database (Db2) and a document store (WDS), while query Q_3 requires operations supported by a relational database (Db2) and a graph store (Neo4j). We use *latency* in seconds as the metric (Table 5), including the execution time on different data stores, and the time spent on data materialization, transformation and transmission, represented as data movement time.

HERMES runtime exploits the capability-based data placement and executes Q_1 using Db2 as the mediator since it supports a richer set of operations (e.g., `JOIN`, `AGGREGATION`, etc.). Db2 first executes the fuzzy search predicate on WDS as a UDF and then retrieves its results. The rest of the query is executed on Db2. The alternative plan is generated and executed against a data placement that has not been optimized based on the data store capabilities. The plan first executes a portion of the query on Db2 and generates the intermediate results that are required to be moved to WDS for the fuzzy search. In this process, the intermediate results from Db2 need to be first ingested and indexed in WDS, which leads to significant overheads. As shown in Table 5, the total average latency of Q_1 by HERMES is 2.54 seconds, which is 2 orders of magnitude (170x) faster compared to the alternative plan. Looking more closely, we observe that the alternative plan incurs a high overhead for data movement due to sub-optimal data placement.

Q_2 is executed on the MED knowledge base. It contains operations similar to Q_1 with an additional top-k operation involved. As shown in Table 5, the total average latency of HERMES's plan is 1.11 seconds, which is 13x faster than the alternative plan based on a sub-optimal data placement. It leads to unnecessary data transformation and movement of approximately 17,000 records from Db2 to WDS.

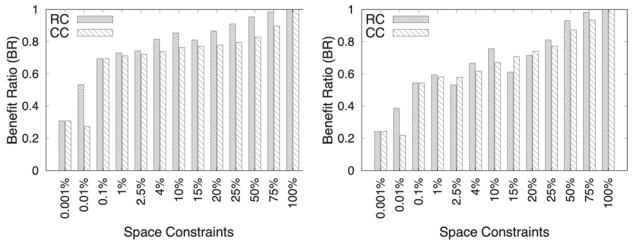
Table 5 Micro-benchmark query execution time (seconds)

Queries	Breakdown	HERMES	Alternative
Q_1 (FIN)	Execution	2.01	110.2
	Movement	0.53	231.2
Q_2 (MED)	Execution	1.11	4.36
	Movement	0.17	16.17
Q_3 (MED)	Execution	2,883	6,530
	Movement	581	0



(a) Uniform workload **(b)** Zipf workload

Fig. 16 Varying space constraints (MED)



(a) Uniform workload **(b)** Zipf workload

Fig. 17 Varying space constraints (FIN)

Q_3 finds patients with a particular condition (381) and other similar conditions (SNOMED CT diseases), which requires a graph reachability sub-query. Thanks to the capability-based data placement, HERMES's execution plan executes the reachability sub-query to find similar disease on the graph backend that stores the SNOMED ontology. It then combines the results with the rest of the operations on Db2. The alternative plan executes on a sub-optimal data placement that places all the data on Db2. As shown in Table 5, HERMES's plan executes efficiently in 3.4 seconds, whereas the alternate plan takes 6.5 seconds due to the expensive self-join involved to evaluate the reachability query. This problem can be further exacerbated if a query contains multiple reachability sub-queries.

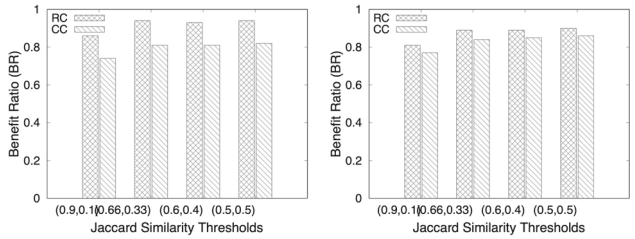
5.4 Property graph schema quality

Varying Space Constraint. In Figs. 16 and 17, 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 (Algorithm 7). 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. The Zipf workload gives more access to the key concepts in the ontology. 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} as well as the total amount of space S_{NSC} needed by PGS_{NSC}. We also compute the total amount of space S_{DIR} needed by the direct mapping algorithm from the given ontology (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. We, then, vary the space constraint from S_{DIR} to S_{NSC}, such that the range of the Y-axis in Figs. 16 and 17 is from 0 to 1. Figs. 16 and 17 show results from *MED* and *FIN* data sets, respectively.

In Fig. 16, we observe that RC consistently outperforms CC with both uniform and Zipf workloads. The reason is that RC creates 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, both algorithms are 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. 17, 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 property graph schema as PGS_{NSC} with 100% space constraint.

Varying Jaccard Similarity. In Fig. 18, 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 *FIN* ontology because it consists of multiple inheritance relationships. Uniform and Zipf workload distributions are used to examine the robustness of our CC and RC algorithms. Note that the space constraint in this experiment is



(a) Uniform workload **(b)** Zipf workload

Fig. 18 Varying Jaccard thresholds (FIN)

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. 5) depending on the similarity threshold. Consequently, the space consumption of the optimal property graph changes under different thresholds. As shown in Fig. 18, both *CC* and *RC* are robust under different similarity thresholds. In the worst case, they achieve more than 70% of the maximum benefit score under 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 other more beneficial relationships to optimize. Hence, the total benefit scores achieved by both algorithms are relatively stable.

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

5.5 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. Note that 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.

Microbenchmark with Graph Primitives. Using 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*). The following parameter settings are used to produce OPT: Jaccard similarity thresholds $\theta_1 = 66\%$, $\theta_2 = 33\%$, and space constraint 0.5 ($S_{NSC} - S_{DIR}$). 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 [17]. 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. 19, 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, 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

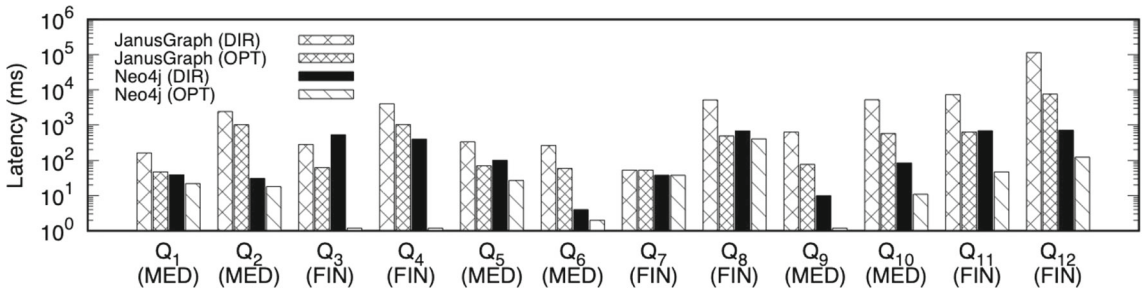


Fig. 19 Microbenchmark - pattern matching (Q_1-Q_4), property lookup (Q_5-Q_8), aggregation (Q_9-Q_{12})

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 is more substantial on Neo4j compared to JanusGraph (Q_3, Q_4, Q_9, etc.). Note that this comparison is not about Neo4j vs JanusGraph, but rather using them as examples to show that disk-based graph systems (e.g., Neo4j) benefit 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. In addition, Table 6 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. 19.

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

Table 6 Microbenchmark - number of edge traversals

	# Edge Traversals			*# 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

Table 7 Benefit ratio w.r.t B_{NSC}

Skew	*MED*			(%)	*FIN*			(%)
Factor	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

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 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 7 shows the quality of the property graph schema produced by *RC* and *CC* compared to the one without space constraints *NSC*. The benefit ratio (BR) is defined as $BR = B_{SC}/B_{NSC}$, where B_{SC} is the total benefit score achieved by either *RC* or *CC* algorithm, and B_{NSC} is the benefit score of the property graph schema generated by Algorithm 7 without any space constraint. 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

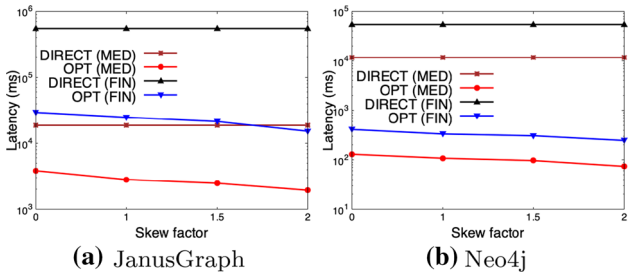


Fig. 20 Total query latency (MED & FIN)

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 is used to measure the performance on these property graphs corresponding to different schemas.

Figure 20 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. 20a, we observe that the total query latency on the optimized schema is around 7 and 22 times faster than the direct mapping one over *MED* and *FIN*, respectively. The winning margin is even bigger on Neo4j (Fig. 20b). 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 information such as data distribution and workload summaries.

5.6 Efficiency of property graph schema algorithms

Finally, we study the execution time of our concept-centric and relation-centric algorithms (Table 8). First, we observe that both *CC* and *RC* produce an optimized property graph schema in less than one second with different space constraints (shown in Table 8 as percentages of the space consumed by Algorithm 7). The optimization time of both algorithms is negligible compared to an exhaustive search approach, which even failed to produce an optimal schema for *MED* after 3 hours. Second, neither of the algorithms 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. Third, *RC* is consistently faster than *CC*, and the performance difference is more significant in *FIN*. This is due to the cost of *OntologyPR* procedure being dominant in *CC*. It usually takes more iterations to converge when the ontology (i.e., *FIN*) is more complex.

Table 8 Efficiency of *RC* & *CC* (time in *ms*)

Space	*MED*			*FIN*		
Constraint	*25%*	*50%*	*75%*	*25%*	*50%*	*75%*
RC	23	23	26	192	188	193
CC	34	36	36	373	344	372

6 Related work

Our work is related to knowledge bases [43], ontology-based access systems [64], as well as poly stores [1,21,29,31,35,37,38]. In this section, we briefly discuss these systems, and then focus on important works in the areas of data placement and schema optimization, highlighting the main differences to our approach.

The emergence of many large scale knowledge bases (KBs), such as DBpedia [39] and YAGO 4 [60], provide a new opportunity to represent the knowledge of the objective world. These KBs are open-domain and often stored in a single data store with a unified schema. Standard query languages such as SPARQL have been used to access the KBs. However, it remains tedious and difficult for end users to query such KBs because of the complexity of the query languages and the KB schema. Instead of forcing the users to express all their query needs in a single query language like SPARQL, we propose HERMES which uses multiple backends with different query languages to provide a rich variety of query types over the KB. We optimize data placement and physical data organization to minimize redundancy, while providing high performance.

More recently, polystore systems [1,21,29,31,35,37,38] have been developed to address the above-mentioned limitations. These systems do not hide the heterogeneity of the data stores. Instead, they provide an integrated, single point of access to several data stores; through one or more query languages without a notion of a global schema. The user queries can contain various sub-queries; each is expressed in its own data model and query language, and executed by a respective backend storage engine. Moreover, these systems

provide a runtime environment (thin middleware) to coordinate and combine query execution across distinct data stores. In this paper, we adapt a similar architecture to support a rich variety of queries over KBs.

Several works such as [1,35,45] attempt to enable access to data stored across multiple data backends with a single interface and point of access. However, these systems take the existing data placement as given and only aim to route queries across different stores based on data locality. They do not consider the data placement according to the query workloads nor the capabilities of each underlying data backend. Hence they are likely to suffer from significant data movement penalties at query time. Du et al. [28] introduce a workload-driven data placement approach to support (R/W) ETL (OLAP) workloads over a streaming engine and an OLAP engine. In particular, it focuses on balancing ingestion and query analysis performance as both stores can perform similar operations but have different storage capacity and access time trade-offs.

Extensive work is available for the schema design problem in relational database and NoSQL systems [12,20,25,36,46,65]. Relational database systems 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 [12,65]. The physical layout of these base tables is then optimized with auxiliary data structures such as indexes and materialized views for the expected workload [12,36]. Typically, the physical design often involves identifying candidate physical structures and selects a good subset of these candidates [25]. NoSE [46] is introduced to recommend schemas for NoSQL applications. Its cost-based approach utilizes a binary integer programming formulation to generate a schema based on the conceptual data model from the application.

In recent 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 [10,18,23,33,44,47,48]. Some of these works focus on optimizing RDF data storage and SPARQL queries based on either workload statistics [44,47,48] or heuristics [62]. Other works [10,18,23,33] 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. Angles et al. [14] introduce direct mappings for transforming an RDF into a property graph, including data and schema.

Similar approaches [34,59] are introduce to address the problem in the context of property graphs. GRFusion [34] focuses on filling the gap between the relational and the graph models rather than optimizing the graph schema to achieve better query performance. SQLGraph [59] and Db2 Graph [61] introduce a physical schema design that combines relational storage for adjacency information with JSON storage for vertex and edge attributes. They translate Gremlin queries into SQL queries to leverage relational query optimizers.

Our ontology-driven approach is different for the following reasons. First, our approach produces a high-quality schema applicable to any graph system compatible with property graph model and Gremlin or Cypher queries. Second, we exploit the rich semantic information in an ontology to guide the schema design. Last but not least, our approach can further leverage these techniques to decide how the property graph should be stored on different storage backends.

7 Conclusion

In this paper, we introduce an ontology-driven polystore system, HERMES, for querying domain-specific enterprise knowledge bases. We tackle two critical design challenges in polystores: data placement and schema optimization. We proposed data placement algorithms that partition the domain ontology into overlapping subsets and store the corresponding data in different data stores depending on their capabilities, as well as the operations performed on the data in a given workload. We also leverage the rich semantic information in a domain ontology to drive the property graph schema optimization for high query performance. Our experimental evaluation uses two real-world KBs to demonstrate the effectiveness of our data placement and schema optimization techniques on our HERMES with a relational store, a document store, and a graph store. The results show that the data placement method generates the near-optimal plan with minimal data replication overhead. The schema optimization algorithms produce high-quality schemas, achieving up to 2 orders of magnitude speed-up compared to alternative schema designs.

References

1. VLDB Workshop: Poly'20. https://sites.google.com/view/poly20/program
2. Federal deposit insurance corporation. https://www.fdic.gov/regulations/resources/call/index.html (2019)
3. Gremlin query language. https://tinkerpop.apache.org/gremlin.html (2019)
4. Janusgraph: Distributed graph database. http://janusgraph.org/ (2019)
5. The neo4j graph platform. https://neo4j.com/ (2019)
6. Owl 2 web ontology language document overview. https://www.w3.org/TR/owl2-overview/ (2019)
7. Securities and exchange commission. https://www.sec.gov/dera/data/financial-statement-data-sets.html (2019)
8. Apache solr. https://lucene.apache.org/solr/ (2020)
9. Elasticsearch: Open source search & analytics. https://www.elastic.co/ (2020)

10. Abadi, D.J., Marcus, A., Madden, S., Hollenbach, K.: SW-Store: a vertically partitioned DBMS for semantic web data management. VLDB J. **18**(2), 385–406 (2009)
11. Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
12. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in sql databases. VLDB **2000**, 496–505 (2000)
13. Alotaibi, R., Lei, C., Quamar, A., Efthymiou, V., Özcan, F.: Property graph schema optimization for domain-specific knowledge graphs. In: ICDE, pp. 924–935 (2021)
14. Angles, R., Thakkar, H., Tomaszuk, D.: Mapping rdf databases to property graph databases. IEEE Access **8**, 86091–86110 (2020)
15. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The description logic handbook: theory, implementation, and applications. Cambridge University Press, Cambridge (2003)
16. Bharadwaj, S., Chiticariu, L., Danilevsky, M., et al.: Creation and interaction with large-scale domain-specific knowledge bases. PVLDB **10**(12), 1965–1968 (2017)
17. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. PVLDB **11**(2), 149–161 (2017)
18. Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient RDF store over a relational database. In: SIGMOD, pp. 121–132 (2013)
19. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. In: WWW, pp. 107–117 (1998)
20. Bruno, N., Chaudhuri, S.: Automatic physical database tuning: A relaxation-based approach. In: SIGMOD, pp. 227–238 (2005)
21. Bugiotti, F., Bursztyn, D., Deutsch, A., I, I., I, M.: Invisible glue: Scalable Self-Tuning Multi-Stores. In: CIDR (2015)
22. Chawathe, S.S., Garcia-Molina, H., Hammer, J., et al.: The TSIMMIS project: integration of heterogeneous information sources. In: Proceedings of the 10th Meeting of the Information Processing Society of Japan, pp. 7–18 (1994)
23. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient sql-based RDF querying scheme. In: VLDB, pp. 1216–1227 (2005)
24. Christophides, V., Efthymiou, V., Stefanidis, K.: Entity Resolution in the Web of Data. Theory and Technology. Morgan & Claypool Publishers, Synthesis Lectures on the Semantic Web (2015)
25. Dash, D., Polyzotis, N., Ailamaki, A.: Cophy: a scalable, portable, and interactive index advisor for large workloads. PVLDB **4**(6), 362–372 (2011)
26. Deutsch, A., Xu, Y., Wu, M., Lee, V.: Tigergraph: a native MPP graph database. CoRR **abs/1901.08248** (2019)
27. Dong, X.L., Srivastava, D.: Big data integration. Synthesis lectures on data management. Morgan & Claypool Publishers, San Rafael (2015)
28. Du, J., Meehan, J., Tatbul, N., Zdonik, S.: Towards dynamic data placement for polystore ingestion. In: BIRTE, pp. 2:1–2:8 (2017)
29. Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., et al.: The BigDAWG polystore system. SIGMOD Record **44**(2), 11–16 (2015)
30. Francis, N., Green, A., Guagliardo, P., et al.: Cypher: an evolving query language for property graphs. In: SIGMOD, pp. 1433–1445 (2018)
31. Gog, I., Schwarzkopf, M., Crooks, N., et al.: Musketeer: all for one, one for all in data processing systems. In: Proceedings of the Tenth European Conference on Computer Systems, p. 2 (2015)
32. Han, X., Hu, L., Sen, J., Dang, Y., Gao, B., Isahagian, V., Lei, C., et al.: Bootstrapping natural language querying on process automation data. In: IEEE SCC, pp. 170–177. IEEE (2020)
33. Harris, S., Shadbolt, N.: SPARQL query processing with conventional relational database systems. In: WISE, pp. 235–244 (2005)
34. Hassan, M.S., Kuznetsova, T., Jeong, H.C., Aref, W.G., Sadoghi, M.: Extending in-memory relational database engines with native graph support. In: EDBT, pp. 25–36 (2018)
35. Kharlamov, E., Mailis, T., Bereta, K., et al.: A semantic approach to polystores. In: IEEE Big Data, pp. 2565–2573 (2016)
36. Kimura, H., Huo, G., Rasin, A., Madden, S., Zdonik, S.B.: Coradd: correlation aware database designer for materialized views and indexes. PVLDB **3**(1–2), 1103–1113 (2010)
37. Kolev, B., Bondiombouy, C., Valduriez, P., et al.: The cloudmdsql multistore system. In: SIGMOD, pp. 2113–2116 (2016)
38. LeFevre, J., Sankaranarayanan, J., Hacigumus, H., et al.: Miso: souping up big data query processing with a multistore system. In: SIGMOD, pp. 1591–1602 (2014)
39. Lehmann, J., Isele, R., Jakob, M., et al.: Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web (2015)
40. Lei, C., Özcan, F., Quamar, A., Mittal, A.R., Sen, J., Saha, D., Sankaranarayanan, K.: Ontology-based natural language query interfaces for data exploration. IEEE Data Eng. Bull. **41**(3), 52–63 (2018)
41. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd edn. Cambridge University Press, New York, NY, USA (2014)
42. Levy, A., Rajaraman, A., Ordille, J.: Querying heterogeneous information sources using source descriptions. Tech. rep, Stanford InfoLab (1996)
43. Lu, J., Holubová, I., Cautis, B.: Multi-model databases and tightly integrated polystores: Current practices, comparisons, and open challenges. In: CIKM, p. 2301–2302 (2018)
44. Maduko, A., Anyanwu, K., Sheth, A.P., Schliekelman, P.: Estimating the cardinality of RDF graph patterns. In: WWW, pp. 1233–1234 (2007)
45. McHugh, J., Cuddihy, P.E., Williams, J.W., et al.: Integrated access to big data polystores through a knowledge-driven framework. In: IEEE Big Data (2017)
46. Mior, M.J., Salem, K., Aboulnaga, A., Liu, R.: Nose: schema design for nosql applications. In: ICDE, pp. 181–192 (2016)
47. Neumann, T., Moerkotte, G.: Characteristic sets: accurate cardinality estimation for RDF queries with multiple joins. In: ICDE, pp. 984–994 (2011)
48. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
49. Pirahesh, H., Hellerstein, J.M., Hasan, W.: Extensible/rule based query rewrite optimization in starburst. In: SIGMOD, pp. 39–48 (1992)
50. Quamar, A., Kumar, K.A., Deshpande, A.: SWORD: scalable workload-aware data placement for transactional workloads. In: EDBT, pp. 430–441 (2013)
51. Quamar, A., Özcan, F., Xirogiannopoulos, K.: Discovery and creation of rich entities for knowledge bases. In: ExploreDB (2018)
52. Quamar, A., Straube, J., Tian, Y.: Enabling rich queries over heterogeneous data from diverse sources in healthcare. In: CIDR (2020)
53. Saha, D., Floratou, A., Sankaranarayanan, K., et al.: Athena: an ontology-driven system for natural language querying over relational data stores. PVLDB **9**(12), 1209–1220 (2016)
54. Sen, J., Ozcan, F., Quamar, A., Stager, G., Mittal, A.R., Jammi, M., Lei, C., Saha, D., Sankaranarayanan, K.: Natural language querying of complex business intelligence queries. In: SIGMOD, pp. 1997–2000 (2019)
55. Slavík, P.: A tight analysis of the greedy algorithm for set cover. In: STOC '96 (1996)
56. Stonebraker, M.: The case for polystores. https://wp.sigmod.org/?p=1629 (2015)
57. Stonebraker, M., Cetintemel, U.: "one size fits all": an idea whose time has come and gone. In: ICDE, p. 2–11 (2005)

58. Suchanek, F.M., Weikum, G.: Knowledge harvesting in the big-data era. In: SIGMOD, pp. 933–938 (2013)
59. Sun, W., Fokoue, A., Srinivas, K., Kementsietsidis, A., Hu, G., Xie, G.T.: Sqlgraph: an efficient relational-based property graph store. In: SIGMOD, pp. 1887–1901 (2015)
60. Tanon, T.P., Weikum, G., Suchanek, F.M.: YAGO 4: A reason-able knowledge base. In: ESWC, pp. 583–596 (2020)
61. Tian, Y., Xu, E.L., Zhao, W., et al.: IBM db2 graph: supporting synergistic and retrofittable graph queries inside IBM db2. In: SIGMOD, pp. 345–359 (2020)
62. Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., et al.: Heuristics-based query optimisation for SPARQL. In: EDBT, pp. 324–335 (2012)
63. Vazirani, V.V.: Approximation Algorithms. Springer-Verlag, Berlin, Heidelberg (2001)
64. Xiao, G., Calvanese, D., Kontchakov, R., Lembo, D., Poggi, A., Rosati, R., Zakharyaschev, M.: Ontology-based data access: a survey. In: IJCAI, p. 5511–5519 (2018)
65. Zilio, D.C., Rao, J., Lightstone, S., et al.: Db2 design advisor: integrated automatic physical database design. In: VLDB, pp. 1087–1097 (2004)