Extracting Graphs Properties with Semantic Joins

Extracting Graphs Properties with Semantic Joins

Yang Cao[1], Wenfei Fan[1,2,4], Wenzhi Fu[1], Ruochun Jin[3], Weijie Ou[2], Wenliang Yi[2]
University of Edinburgh[1] Shenzhen Institute of Computing Sciences[2]
National University of Defense Technology[3] BDBC, Beihang University[4]
{yang.cao@, wenfei@inf., wenzhi.fu@}ed.ac.uk, jinrc@nudt.edu.cn, {ouweijie, yiwenliang}@sics.ac.cn

***Abstract*—This paper proposes an approach to querying a relational database $\mathcal{D}$ and a graph G taken together in SQL. We introduce a semantic extension of joins across $\mathcal{D}$ and G such that if a tuple t in $\mathcal{D}$ and a vertex v in G refer to the same real-world entity, then we join t and v to correlate their information and complement tuple t with additional properties of vertex v from the graph. Moreover, we extract hidden relationships between t and other entities by exploring paths from v. To support the semantic joins, we develop an extraction scheme based on LSTM, path clustering and ranking, to fetch important properties from graphs, and incrementally maintain the extracted data in response to updates. We also provide methods for implementing static joins when t is a tuple in $\mathcal{D}$, dynamic joins when t comes from the intermediate result of a sub-query, and heuristic joins to strike a balance between the complexity and accuracy. Using real-life data and queries, we experimentally verify the effectiveness, scalability and efficiency of the methods.**

I. Introduction

A question raised by our FinTech collaborators asks how they can write queries across relations and graphs, in SQL?

Example 1: Below are three queries taken from a FinTech company. The company maintains (1) a relational database $\mathcal{D}$ of customer and financial products, and (2) a graph G of (a) "knowledge" about products, brokers and customer, and (b) transactions, for links between customers and products, brokers and products. Such graphs are often used in product recommendation [1]. The two datasets are maintained by different departments, as commonly practiced in big companies. Fragments of $\mathcal{D}$ and G are shown in Figure 1.

(1) Complementing entities in relations (Q_1). A customer wants to find information (*e.g.,* risk) about investment product fd1 and its underlying securities (*e.g.,* whether it is based on a UK company). To answer Q_1, we need both data in $\mathcal{D}$ and data in G since while $\mathcal{D}$ has a record t_5 for the basics of fd1, we have to extract its *relevant* company information from G to "enrich" tuple t_5. This is nontrivial for SQL practitioners since they may not know the exact structure and vocabulary of graph G when G does not have a "schema".

(2) Deducing hidden links between entities (Q_2). To decide whether to recommend a financial plan fd2 to customer Bob (cid02), a broker checks whether (a) Bob has good credit, and (b) whether Ada (cid04), a customer who has already invested in fd2, has an investment pattern similar to Bob's, *e.g.,* whether Ada and Bob has bought stock of the same company. To answer query Q_2, one has to (a) inspect the information of Bob in $\mathcal{D}$, (b) enrich the records of Bob and Ada by extracting companies in which they invested from the graph, and (c) check whether the two invest in the same company.

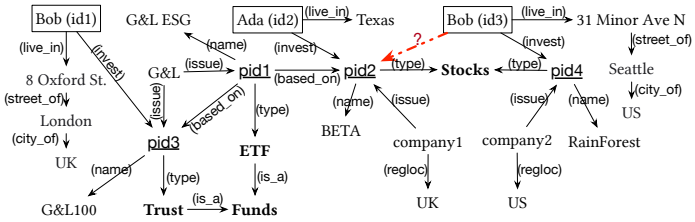
	cid	name	credit	bal	address
t_1	cid01	Bob	fair	$500k	8 Oxford St., London, UK
t_2	cid02	Bob	good	$110k	31 Minor Ave N, Seattle, US
t_3	cid03	Guy	good	$50k	10115 Berlin, Germany
t_4	cid04	Ada	fair	$100k	1200 Albert Ave, Texas, US

(a) customer relation

	pid	name	issuer	type	price	risk
t_5	fd1	G&L ESG	G&L	Funds	$90	medium
t_6	fd2	Beta	company1	Stocks	$120	high
t_7	fd3	G&L100	G&L	Funds	$100	low
t_8	fd4	RainForest	company2	Stocks	$80	medium

(b) product relation

(1) Relational database $\mathcal{D}$: customer and product



(2) Graph G for FinTech recommendation

Fig. 1: Relational database $\mathcal{D}$ and graph G in Example 1

(3) Extracting hidden links (Q_3). The third query is to find people with good credit and are within k hops of Bob (tuple cid02) in a social network G', to recommend them financial products in which Bob has invested. To answer Q_3, one has to check customer tuples in $\mathcal{D}$ for their credits and detect links between Bob and the customers by traversing paths in G'. □

The need for querying relations $\mathcal{D}$ and graphs G taken together has been increasingly evident in a variety of domains, not limited to FinTech. While business data often resides in relational database management systems (RDBMSs), graphs have found prevalent use in practice, *e.g.,* transaction graphs, knowledge bases, social networks and road networks. Gartner predicts that the practice of graph analytics will double annually (cf. [2]). With this comes the need for synthesizing data across $\mathcal{D}$ and G. Moreover, practitioners often want to write their queries in SQL after practicing SQL for decades.

To support SQL queries across $\mathcal{D}$ and G, we need to extract data from graph G and convert the data to relations. This is nontrivial. A common practice is to encode G as a "schema-less" edge relation. This, however, ignores the semantics and structure of G; it is hard for users to write sensible query, align entities in $\mathcal{D}$ and G, and correlate their information. Moreover, to access properties of a vertex v in G, one has to traverse a path in contrast to "local" attributes of a relational tuple, and path traversal requires costly joins in an RDBMS.

For instance, to sensibly answer query Q_1 in Example 1, one has to check whether fd1 in table product is pid1 in graph G. This requires us to check properties (name, issuer and type) of pid1 in G that may not be local, *e.g.,* Funds for pid1; similarly for checking whether cid02 in $\mathcal{D}$ and id3 in G are the same Bob. Moreover, to find UK-based company behind fd1 for Q_1, one has to examine a path (pid1, pid2, company1, UK). It is nontrivial to accurately extract properties without knowing the structure and labels of G. To retrieve UK from G as the country of company1, one need to select semantically close regloc instead of country, since country is not a label in G

Several questions have to be answered. How can we "join" a tuple t and a vertex v so that we can correlate their information and write sensible queries across $\mathcal{D}$ and G in SQL? What properties should we extract from graph G? How should we incrementally maintain the extracted data in response to updates? How can we answer queries across $\mathcal{D}$ and G efficiently without switching between RDBMS and graph systems at run time?

A semantic approach. To answer these questions, we propose a method that "joins" tuples in $\mathcal{D}$ and vertices in G if they semantically match, selectively extracts properties of vertices in G only if they match tuples in $\mathcal{D}$, and efficiently answer queries across $\mathcal{D}$ and G without accessing G online.

(1) Semantic joins. We propose a simple semantic extension to natural joins, referred to as *semantic joins*. Consider a tuple t in $\mathcal{D}$ and a vertex v in G. If t and v refer to the same real-world entity e, then we can "join" the two, extract relevant properties of v and enrich t with the additional "attributes". Moreover, we can identify hidden links between e and another entity represented by a tuple t', by checking paths between v and vertex v' that "matches" t'. Several methods are already in place for determining whether t and v match [3], [4], [5], [6], referred to as HER (Heterogeneous Entity Resolution).

With semantic joins, one can query $\mathcal{D}$ and G in SQL extended with only syntactic sugar; *e.g.,* Q_1 can be written as:

select risk, company
from product **e-join** G ⟨company, loc⟩ as T
where T.pid = fd1 and T.loc = UK

As will be seen in Section II, **e-join** denotes a semantic join. It correlates financial-plan entities in $\mathcal{D}$ and G. The query can be converted to an SQL query and is answered by RDBMS.

Graph property extraction. To support semantic joins, we propose REXT, a method for extracting relations from G. In contrast to core dumping the entire graph G into, *e.g.,* an edge relation, REXT *extracts* properties of vertices that match relational tuples, and *complement* the tuples with the properties.

More specifically, given a set S of tuples of schema R, REXT extracts a relation D_G of schema R_G from graph G such that each tuple t_G of D_G contains graph properties (attributes) of an entity encoded by a tuple t in S. The schema R_G and relation D_G are decided for S by selecting paths in G that (a) start from a vertex v that refers to the same entity as t, (b) lead to properties that differ from existing attributes in S, and match the interests of the users. REXT encodes paths of G via LSTM (long short-term memory) embedding, extracts attributes from paths via K-means clustering, and picks relevant ones with a ranking function. Users may opt to provide a set of keywords to specify their preference for attributes, and REXT extracts properties that semantically match the users' interest.

(3) Implementation. We implement semantic joins of a set S of tuples with graph G on top of existing RDBMSs. With REXT, we support (a) static joins, when S is a set of tuples in the input $\mathcal{D}$, (b) dynamic joins, when S is the intermediate result of a sub-query, and (c) heuristic joins that do not call HER and REXT online when graphs are "typed" and keywords are pre-determined, to strike a balance between the accuracy and efficiency. We also develop incremental methods to maintain the extracted data in response to updates to $\mathcal{D}$ and G.

These can be plugged into existing RDBMSs so that practitioners can query $\mathcal{D}$ and G in SQL, and hold on to the sophisticated query planers and optimizers of RDBMSs. It also sheds lights on data lakes [7] for (a) *query-driven data discovery* to find relevant graphs with vertices matching tuples in $Q(\mathcal{D})$ of a query Q; (b) *on-demand data integration* to augment tuples in $Q(\mathcal{D})$ with properties of matching vertices; and (c) *data extraction* to abstract schema/relations from raw data in graphs.

(4) Effectiveness. Using real-life data, we experimentally verify the following. (a) The extraction scheme (REXT) and semantic joins enable SQL queries across relations and graphs, which are difficult and costly if G is cast into vertex/edge relations. (b) REXT is accurate, with average F-measure 0.95. (c) REXT is efficient; it takes 230.4s on $\mathcal{D}$ with 3.4M tuples and G with 10.2M edges. (d) Atop PostgreSQL, our implementation of semantic joins is efficient, taking at most 9.2s across relations of 4.4M tuples and graphs of 31.1M edges when the query can be reduced to static joins; it is on average 114.9X faster than baselines. (e) On average, the incremental extraction method is 14.2X faster than REXT when updates $|\Delta G|$=5%$|G|$; it is still faster when $|\Delta G|$ is up to 45% of $|G|$.

Contributions & Organization. We deliver the following.

- A notion of semantic joins across $\mathcal{D}$ and G (Section II).
- Data and schema extraction methods (Section III).
- Implementation (static, dynamic and heuristic; Section IV).
- Evaluation for effectiveness and efficiency (Section V).

We discuss related and future work in Sections VI and VII.

II. Queries Across Relation and Graph

In this section, we define *semantic joins* and extend SQL accordingly for querying relations and graphs taken together.

A. Preliminary

We first review relational databases and graphs.

Relations. A database schema is $\mathcal{R} = (R_1, \ldots, R_n)$; each R_i is a relation schema $R_i(A_1, \ldots, A_{k_i})$, where A_i is an attribute. A *database* $\mathcal{D}$ *of* $\mathcal{R}$ is $(D_1, \ldots, D_n)$, where D_i is a *relation* of R_i. Following Codd [8], we consider relations in which each tuple t refers to a real-world entity. To simplify the discussion, we assume that t carries a tuple id (primary key).

Graphs. We consider *w.l.o.g. directed labeled graphs* $G = (V, E, L)$, where (a) V is a finite set of vertices, (b) $E \subseteq V \times V$ is a set of edges, and (c) L is a function such that for each vertex $v \in V$ (resp. edge $e \in E$), $L(v)$ (resp. $L(e)$) is a vertex (resp. edge) label. While edge labels typify predicates, vertex labels may carry values. Graph G may be a transaction graph, a social network, a knowledge base, etc.

A *path* ρ *from a vertex* v_0 in graph G is a list $\rho = (v_0, v_1, \ldots, v_l)$ of vertices such that (v_{i-1}, v_i) is an edge in E for $i \in [1, l]$. It is *undirected* if either (v_{i-1}, v_i) or (v_i, v_{i-1}) is an edge in E for $i \in [1, l]$, regardless of the orientation of the edge. The *length of* ρ is l, *i.e.,* the number of edges on ρ. A path is *simple* if a vertex appears on ρ at most once. We consider simple paths in the sequel, simply referred to as paths.

B. Semantic Joins

Semantic join is defined between a relational database $\mathcal{D}$ and a graph G. It assumes the availability of the following.

*Heterogeneous Entity Resolution (*HER*)* provides a function f that given a graph G and a set S of tuples, computes a set:

$$f(S, G) = \{(t, v) \mid t \in S, v \in V \text{ in } G, t \Rightarrow v\}.$$

Here $t \Rightarrow v$ denotes that tuple t and vertex v make a *match*, *i.e.,* t and v refer to the same entity. We refer to f as the HER *function* and $f(S, G)$ as the *match relation* of S and G.

We denote by $R_m(\mathsf{tid}, \mathsf{vid})$ *the schema of the match relation*, such that a tuple $(t.\mathsf{id}, v.\mathsf{id})$ of schema R_m denotes that (t, v) is a match in $f(S, G)$ for tuple t with $t.\mathsf{id}$ and vertex v with $v.\mathsf{id}$.

*Relation Extraction (*REXT*)*. Given a graph G and a relation schema R, the *relation extraction* (REXT) scheme deduces a schema R_G and a population function h, such that

(1) $R_G = (\mathsf{vid}, A_1, \ldots, A_m)$, where vid denotes a vertex v that matches tuples of R by HER, and A_i's are features of v; and

(2) h is a function that given a set S of tuples of R, returns an instance $h(S, G)$ of schema R_G by extracting corresponding properties of the vertices in $f(S, G)$ that match tuples in S.

As will be seen in Section III, R_G is composed of attributes $A_1, \ldots, A_m$ that (a) meet users' interests (see below), and (b) are absent from schema R, as additional and alternative attributes to complement R. Function h populates $h(S, G)$ by traversing paths in G from matching vertices in $f(S, G)$. We refer to R_G as the *extracted schema* for R from G, and to $h(S, G)$ as the *extracted relation* for S from G.

Remarks. There have been methods for HER: rule-based JedAI [3], parametric simulation [5], and ML models Silk [9], MAGNN [4] and EMBLOOKUP [6]. However, *no prior method* is in place for REXT. We will develop REXT in Section III.

Below we define semantic joins *w.r.t.* given HER function f and REXT scheme (*i.e.,* schema R_G and function h). Consider a graph G and a configurable parameter k as for REXT above.

There are two types of semantic joins: enrichment and link.

(I) Enrichment join. An *enrichment join* has the form $S \ltimes_{\mathcal{A}} G$, where S denotes a set of tuples of schema R that encode entities, and $\mathcal{A}$ is a set $\{A_1, \ldots, A_m\}$ of keywords that specifies an *extracted schema* $R_G(\mathsf{vid}, A_1, \ldots, A_m)$. Then $S \ltimes_{\mathcal{A}} G$ returns a relation of schema R' that consists of attributes of R and R_G. A tuple t is in $S \ltimes_{\mathcal{A}} G$ if

(1) $t[\mathsf{attr}(R)]$ is a tuple in S, where $\mathsf{attr}(R)$ denotes the set of attributes of R;

(2) $t[\mathsf{vid}]$ is the id of vertex v in G such that $t \Rightarrow v$ by HER f;

(3) for each $i \in [1, m]$, $t[A_i]$ is the property of v corresponding to A_i, extracted from G via REXT (see Section III).

Intuitively, for each tuple t in S, the join is to identify matching vertices v in G via HER and extend t with additional attributes $\mathcal{A}$ of v extracted from G via REXT. Here REXT takes keywords $\mathcal{A}$ and path bound k as parameters. Essentially, $S \ltimes_{\mathcal{A}} G$ is $S \bowtie f(S, G) \bowtie h(S, G)$ via the SQL join operator.

In relational algebra, one can write an enrichment join as either $R \ltimes_{\mathcal{A}} G$ for a relation schema R in $\mathcal{R}$, *i.e.,* when S is the input relation D of R in $\mathcal{D}$; or $Q \ltimes_{\mathcal{A}} G$ where Q is a sub-query over $\mathcal{D}$ and G, and S is the result relation $Q(\mathcal{D}, G)$.

Example 2: We can write Q_1 of Example 1 with an enrichment join $\pi_{\mathsf{risk},\mathtt{company}}\sigma_{\mathsf{pid}=\mathsf{fd1}\wedge\mathtt{loc}=\mathrm{UK}}\,\mathsf{product} \ltimes_{(\mathtt{company},\mathtt{loc})} G$. Here $\mathcal{A}$ = (company, loc) specifies the attributes to extract, and k=3. Assume that HER matches fd1 of the product table and vertex pid1 of graph G in Fig. 1 (as their name (G&L ESG), issuer (G&L) and type (Funds) match, among other things). We will see in Section III that REXT extracts company1 and UK from G. The enrichment join returns product tuple t_5 extended with company1 and UK; thus Q_1 returns (medium, company1). □

(II) Link join. A *link join* has the form $S_1 \bowtie_G S_2$, where S_1 (resp. S_2) is a set of tuples of schema R_1 (resp. R_2). It is to check, for each tuple $t \in S_1$ and $t' \in S_2$, whether vertices that match t and t' are within k hops of each other, and return $\{t_1\} \times \{t_2\}$ if so. That is, it "joins" tuples in S_1 and S_2, and returns a relation. The join condition is the k-hop connectivity between vertices in G that match tuples of S_1 and S_2 via HER.

In relational algebra, one can write a link join as $Q_1 \bowtie_G Q_2$ for two sub-queries Q_1 and Q_2 over $\mathcal{D}$ and G.

Example 3: Query Q_3 of Example 1 can use a link join over the customer relation and graph G': $\sigma_{\mathsf{cid}=\mathrm{cid02}}\mathsf{customer} \bowtie_{G'} \sigma_{\mathsf{credit}=\mathrm{good}}\mathsf{customer}'$, where customer′ renames customer. It finds customers within k hops of Bob in a social network. □

Remarks. (1) Unlike traditional joins, semantic joins operate on relations and graphs. Enrichment joins generate relation schemas with new attributes extracted from graph G, and link joins connect tuples subject to their links embedded in G.

(2) Users may specify interest with attribute names and values as $\mathcal{A}$ in an enrichment join $S \ltimes_{\mathcal{A}} G$. REXT extracts graph properties that semantically match $\mathcal{A}$ as additional attributes in the extracted schema (see Section III). This serves individual users' need but needs to run REXT online with different keywords $\mathcal{A}$. Users may provide not only potential attribute names but also values to exemplify the attributes of interest.

(3) Alternatively, users may opt to pick keywords $\mathcal{A}$ from reference lists. As will be shown in Section IV, REXT profiles

symbols	notations
$\mathcal{D}$, G	relational database, graph
$S \bowtie_{\mathcal{A}} G$, $S_1 \bowtie_G S_2$	enrichment join, link join
$f(S, G)$	HER match relation of schema $R_m(\mathsf{tid}, \mathsf{vid})$
$R_G(\mathsf{vid}, A_1, \ldots, A_m)$, $h(S, G)$	schema and relation extracted from G
R_τ, $g_\tau(G)$	schema and instance for τ entities in G

TABLE I: Notations

graph G and extracts frequent keywords and reference relation schemas in an offline preprocess, from query logs, user specifications, and selected vertex and edge labels in G, possibly with the help of expert users. Users may select relevant $\mathcal{A}$ from the reference lists, especially when they are not familiar with the vocabulary and structure of G. Moreover, semantic joins with such keywords do not need to run REext on-the-fly.

C. Extending SQL

We present gSQL (graph SQL), SQL with syntactic sugar for semantic joins. A gSQL query over database schema $\mathcal{R} = (R_1, \ldots, R_n)$ and graph G is of the form:

select $A_1, \ldots, A_h$
from $R_1, \ldots, R_n$,
S_1 **e-join** $G_1\langle\mathcal{A}_1\rangle, \ldots, S_m$ **e-join** $G_m\langle\mathcal{A}_m\rangle$,
T_1 **l-join** $\langle G_1\rangle T_1', \ldots, T_m$ **l-join** $\langle G_m\rangle T_m'$
where CONDITION-1 {**and** | **or** } ... {**and** | **or** } CONDITION-P

Here (a) G_i is a renaming of the graph G, S_i, T_i and T_i' are either relations in $\mathcal{R}$ or gSQL sub-queries over $\mathcal{R}$ and G; (b) $\mathcal{A}_i$ is a set of attribute names in schema R_G to be extracted from G (recall Section II-B); (c) **e-join** and **l-join** refer to enrichment join and link join, respectively; and (d) each CONDITION in the **where** clause is an SQL condition over $\mathcal{R}$ and the result relations of semantic joins in the **from** clause.

Like SQL, a gSQL query Q returns a relation, and its schema can be statically deduced from $\mathcal{R}$, Q and keywords $\mathcal{A}$. As will be seen in Section IV, Q can be rewritten into an SQL query, and hence gSQL extends SQL just with syntactic sugar.

Example 4: The query given in Section I for Q_1 is a gSQL query. One can also write Q_2 of Example 1 in gSQL, which is a traditional join on the results of two enrichment joins:

select * **from** customer **e-join** $G\langle$`stock, company`$\rangle$ as T_1,
customer **e-join** $G\langle$`stock, company`$\rangle$ as T_2
where T_1.cid = cid04 and T_2.cid = cid02 and T_2.credit = good
and T_1.company $= T_2$.company

It decides whether Ada(cid04) and Bob(cid02) can be joined by an attribute (company) not in $\mathcal{D}$ but extracted from G.

One can write query Q_3 in gSQL with a link join as:

select * **from** customer **l-join** $\langle G'\rangle$ customer as customer′
where customer.cid = cid02 and customer′.credit = good

Intuitively, Q_3 "extends" a self-join of customer such that the join condition requires to traverse G' to find customers who are connected to Bob (cid02) and have a good credit. □

The notations of the paper are summarized in Table I.

III. EXTRACTING RELATIONS FROM GRAPHS

A key step in semantic joins is to extract properties from graph G. We next present REext for extracting data/schema from G (Section III-A). We also develop IncExt, a method that incrementally maintains the extracted relations (Section III-B).

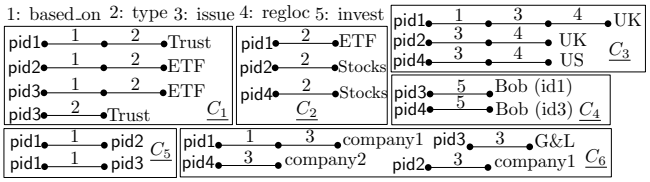


Fig. 2: Clustering result of paths in G.

We will use the following machine learning (ML) methods (embedding and clustering) and a notion of path patterns.

Sequence Embedding in Graphs. Sequence embedding represents token sequences as vectors such that the learned representations capture the sequential information. We adopt Long Short-Term Memory (LSTM) network since LSTM has verified effective and efficient in modeling the semantics of labels on paths in knowledge graphs [10], [11], [12], while others, such as BERT-based Transformer, suffer from heavier computation cost with little performance improvement.

K-means Clustering (KMC). KMC [13] aims to partition data points into clusters so that data points in the same cluster are much closer and more semantically similar than those in different ones. It can be efficiently parallelized [14] and often achieves excellent quality in practice [15].

Path Pattern and Matching. The *path pattern* of path $\rho = (v_0, v_1, \ldots, v_l)$ is a list $p_\rho = (L(v_0, v_1), \ldots, L(v_{l-1}, v_l))$ of edge labels on ρ. Two paths ρ_1 and ρ_2 are *of the same type* if $p_{\rho_1} = p_{\rho_2}$. Given a path ρ and a path pattern p, the *pattern matching* $M(\rho, p)$ between ρ and p returns true if $p_\rho = p$, and false otherwise. Matching $M(\rho, p)$ takes $O(min(\mathsf{len}(p_\rho), \mathsf{len}(p)))$ time, linear in the length of the shorter path pattern. We say a path ρ *conforms to pattern* p if $M(\rho, p) = \mathsf{true}$.

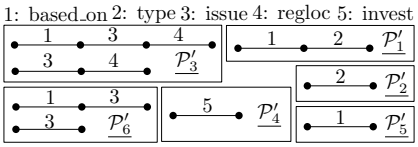
A. REext: A Relation Extraction Scheme

Given a graph G, a set S of tuples of a relation schema R, a set $\mathcal{A}$ of keywords, and a path length bound k, REext is to compute an extracted schema $R_G(\mathsf{vid}, A_1, \ldots, A_m)$ for S from G, and an instance $h(S, G)$ of R_G. Here $\mathcal{A}$ includes attribute names in R_G and attribute values that exemplify user interests. REext serves the needs of individual users by extracting R_G and $h(S, G)$ on-the-fly *w.r.t.* input S, $\mathcal{A}$ and k.

REext computes these in two phases: (I) discover a set $\mathbb{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ of clusters such that each $\mathcal{P}_i \in \mathbb{P}$ contains path patterns of similar semantic meanings, and extract the schema R_G from G; and (II) extract $h(S, G)$ based on the discovered $\mathbb{P}$ and R_G via path pattern matching, as follows.

I. Path Pattern Discovery. REext first discovers a set $\mathbb{P}$ of path pattern clusters and a schema $R_G(\mathsf{vid}, A_1, \ldots, A_m)$, where each $\mathcal{P}_i \in \mathbb{P}$ ($i \in [1, m]$) corresponds to attribute $A_i \in R_G$. More specifically, guided by an LSTM language model, REext first selects paths bounded by length k starting from each matching vertex v_i of G in $f(S, G)$, and stores the paths in a set P. Then it takes four steps to find the path patterns $\mathbb{P}$ and schema R_G via clustering, as follows.

(1) Path Selection. Viewing graph G as undirected, given an entity $v_i \in G$, for each edge $e = (v_i, l, v') \in G$, REext



(a) Clusters of path patterns

vid	loc	company
pid1	UK	company1
pid2	UK	company1
pid3	null	G&L
pid4	US	company2

(b) Extracted relation

Fig. 3: Clusters of path patterns based on $\mathbb{C}$.

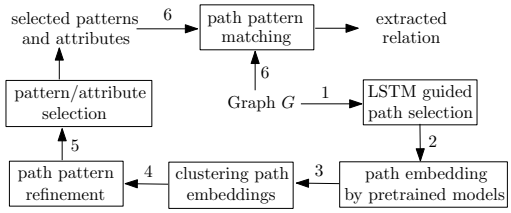


Fig. 4: Workflow diagram of relation extraction.

initiates a path $\rho_{v'} = (v_i, v')$, feeds the vertex label $L(v')$ to a model $\mathcal{M}_\rho$ and obtains a list $\mathcal{L}_1$ of edge labels along with their possibility of following the "word" $L(v')$. Here $\mathcal{M}_\rho$ is a typical language model that decides whether a sequence of vertex/edge labels is "interpretable", *i.e.,* statistically reasonable. Then among all outgoing edges from v', it chooses an edge (v', l', v'') where l' has the highest possibility in $\mathcal{L}_1$, appends vertex v'' to path $\rho_{v'}$, and feeds vertex label $L(v'')$ to $\mathcal{M}_\rho$ for getting the predicted list $\mathcal{L}_2$ in the next round. This iteration proceeds until (a) $\mathcal{M}_\rho$ returns a "stop signal", *i.e.,* the end of sentence tag "<eos>"; (b) there is no outgoing edge to choose; (c) the path length reaches limit k; or (d) the path forms a cycle (abandoned). All the paths $\rho_{v'}$ are included in P and returned.

Note that paths in knowledge graphs often have semantic meanings embedded in their labels, and LSTM has proven effective in capturing the semantics [10], [11], [12]. By taking LSTM as $\mathcal{M}_\rho$, the vertex/edge labels carried by selected paths from v_i often form interpretable "sentences". The method also avoids enumerating (exponentially many) paths in G.

To train $\mathcal{M}_\rho$, we conduct random walk in G and collect sequences of edge/vertex labels on random walk paths to build a training corpus. Taking the labels as sentences of words, we train $\mathcal{M}_\rho$ on the corpus driven by the "perplexity" loss [16]. The corpus construction and model training are unsupervised.

(2) Path Clustering. Given P, RExt groups all its paths into a set $\mathbb{C}$ of H clusters via vectorization and clustering, for a configurable parameter H. For each path $\rho_{ij} \in P$ where v_i (resp. v_{ij}) is the start (resp. end) vertex in ρ_{ij}, it employs a word embedding model $\mathcal{M}_e$ to extract a vector representation $x_{L(v_{ij})}$ of vertex label for each v_{ij}. Since most vertex labels are common in natural languages, we use the mean of GloVe embeddings [17] to represent $L(v_{ij})$, rather than training a new $\mathcal{M}_e$ from scratch. It also builds a vector representation $x_{\rho_{ij}}$ for each path ρ_{ij} via sequence embedding model $\mathcal{M}_\rho$, where ρ_{ij} is viewed as a sequence of edge labels. More specifically, RExt feeds edge labels on path ρ_{ij} in sequence to $\mathcal{M}_\rho$, and takes the network embedding output in the last step as $x_{\rho_{ij}}$, similar to [18]. By the sequence modeling capacity of LSTM, the embedding $x_{\rho_{ij}}$ can discern different orders of edge labels, and benefit the downstream clustering task. Concatenating $x_{L(v_{ij})}$ and $x_{\rho_{ij}}$ as x_{ij}, each vertex-path pair (v_{ij}, ρ_{ij}) is encoded by one feature vector and is collected in a set $\mathcal{X}$. For meaningless labels, *e.g.,* labels in Freebase, we use the mean of character GloVe embeddings as a trade-off between quality and efficiency. This is because (1) the clustering accuracy of path patterns is mainly decided by path embedding quality as different path patterns can be discerned by the sequential information of edge labels; and (2) it is costly to train and apply BERT-based transformers to embed meaningless labels.

Then RExt performs KMC on the set $\mathcal{X}$ with limited iterations to assign each vertex-path pair to one of H clusters. Each cluster $\mathcal{C}_l \in \mathbb{C}$ contains paths ρ_{ij} (represented by x_{ij}) from the set P of paths; paths put into the same cluster have similar semantic meanings, since the embeddings capture the semantics of vertex labels and path labels, and KMC groups semantically similar embeddings together. The clustering is unsupervised, without costly manual annotations.

Example 5: Continuing with Example 1, Figure 2 shows the clustering $\mathbb{C}$ of paths starting from each product in G. After path selection and embedding extraction, KMC groups paths of similar semantic meanings together. Note that path (pid3, type, Trust), which should be assigned to cluster C_2, is misclassified to C_1 as it is similar to (pid1, based_on, pid3, type, Trust). □

(3) Path Pattern Refinement. RExt then extracts path patterns from each $\mathcal{C}_l \in \mathbb{C}$, refines the patterns by "majority voting", and obtains a clustering $\mathbb{P}'$ of path patterns. More specifically, (1) for each cluster $\mathcal{C}_l \in \mathbb{C}$ ($l \in [1, H]$), it extracts the path pattern p_ρ of each path ρ in $\mathcal{C}_l$ and adds p_ρ to set $\mathcal{P}'_l$ (initially empty). RExt uses a counter for each path pattern p in $\mathcal{P}'_l$, which records the number of times that p is added to $\mathcal{P}'_l$. After this, $\mathbb{C}$ is converted to a path pattern clustering $\mathbb{P}' = \{\mathcal{P}'_1, \ldots, \mathcal{P}'_H\}$. (2) For each pattern p that appears in multiple clusters of $\mathbb{P}'$, RExt picks the cluster in which the counter for p is the largest. That is, we preserve pattern p in the cluster that contains the majority of paths conforming to p.

The refinement is needed since each path pattern carries a specific semantic meaning and should be in only one cluster that may correspond to an attribute in R_G. After this step, RExt obtains path pattern clustering $\mathbb{P}' = \{\mathcal{P}'_1, \ldots, \mathcal{P}'_{m'}\}$, where each $\mathcal{P}'_l$ is a candidate attribute for R_G (see below). Note that $m' \leq H$, since some clusters may vanish during refinement.

Example 6: Figure 3(a) shows path pattern clustering $\mathbb{P}'$, which abstracts paths in $\mathbb{C}$ as patterns. Then the "majority voting" refinement removes pattern of misclassified path (pid3, type, Trust) from $\mathcal{P}'_1$ based on counters of path pattern. □

(4) Pattern/Attribute Selection. Given clustering $\mathbb{P}'$, RExt selects a set $\mathbb{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ of pattern clusters and builds schema $R_G(\mathsf{vid}, A_1, \ldots, A_m)$ based on a ranking function. For each cluster $\mathcal{P}'_i \in \mathbb{P}'$, it first finds matching paths from P of each pattern $p \in \mathcal{P}'_i$ and saves each matching vertex v_j along with $L(\rho.v_l)$ (v_l is the end vertex in a matching path ρ from v_j) in $\mathcal{W}_i$. This establishes one-to-one correspondence between $\mathcal{P}'_i \in \mathbb{P}'$ and $\mathcal{W}_i \in \mathbb{W}$ ($i \in [1, m']$). Such matching is efficient since the paths have been precomputed (in P) and each pattern

matching starts from vertex v_j. This combines the matching $(v_j, L(\rho.v_l))$, tuple $t_j \in f(S,G)$ and vector representations ($x_{L(v_l)}$ and x_ρ) in $\mathcal{X}$. Then, it applies the score function below to rank elements in both $\mathbb{W}$ and $\mathbb{P}'$ in descending order:

$$r(\mathcal{W}_i) = \frac{|\mathcal{W}_i|}{|P|} - \max_{\varphi \in [1,k_R]} \frac{\sum_{(v_j, L(\rho.v_l)) \in \mathcal{W}_i} \cos(x_{L(\rho.v_l)}, x_{t_j.A_\varphi})}{|\mathcal{W}_i|} + \max_{\varepsilon \in [1,m]} \frac{\sum_{(v_j, L(\rho.v_l)) \in \mathcal{W}_i} \cos(x_{L(\rho.v_l)}, x_{A_\varepsilon})}{|\mathcal{W}_i|},$$

where P is the precomputed set of selected paths, $\cos(\cdot,\cdot)$ returns the cosine similarity of two vectors, $x_{t_j.A_\varphi}$ (resp. x_{A_ε}) is the embedding of attribute $t_j.A_\varphi$ (resp. string $A_\varepsilon \in \mathcal{A}$) in the same way as $x_{L(v_{ij})}$, k_R is the arity of relation S, and m is the number of query-interest keywords in $\mathcal{A}$. Each $\mathcal{P}'_i \in \mathbb{P}'$ is ranked by the score of $\mathcal{W}_i$. For each $\mathcal{W}_i \in \mathbb{W}$ (resp. $\mathcal{P}'_i \in \mathbb{P}'$), the ranking function also finds a keyword $A_i \in \mathcal{A}$ that maximizes the third term as by-product, which serves as the attribute name in R_G for path pattern cluster $\mathcal{P}_i$.

Intuitively, the function gives higher scores to clusters that (1) match more paths in G (the first term), (2) are not similar to existing attributes in S (the second term), and (3) are semantically close to one of users interests in $\mathcal{A}$ (the third term). Thus, the extracted schema R_G tends to reduce the number of null values, add versatile information complementary to S, and meet the user's interests.

After this, starting from high-score pattern clusters, RExt may interact with the user by presenting matching result $(v_j, L(\rho.v_l))$ and pair $(t_j, v_j) \in f(S,G)$. If the user is satisfied with the presented matches in $\mathcal{W}'_i$, then $\mathcal{P}'_i$, renamed as $\mathcal{P}_i$, will be added to $\mathbb{P}$, and an attribute A_i (keyword that maximizes the third term in the ranking function for $\mathcal{W}'_i$) will be added to R_G.

Example 7: The ranking function sort $\mathbb{P}'$ as $[\mathcal{P}'_3, \mathcal{P}'_6, \mathcal{P}'_5, \mathcal{P}'_4, \mathcal{P}'_1, \mathcal{P}'_5]$. Here $\mathcal{P}'_1$ and $\mathcal{P}'_5$ rank the lowest as the information of their paths overlaps with **type** in product relation; and $\mathcal{P}'_3$ and $\mathcal{P}'_6$ rank top 2 since they match keywords in $\mathcal{A}$ = (loc, company). After seeing the path matches of each pattern cluster, $\mathcal{P}'_3$ and $\mathcal{P}'_6$ are renamed as $\mathbb{P} = \{\mathcal{P}_1, \mathcal{P}_2\}$, which yields schema R_G(vid, loc, company), for user inspection. □

Cost of Pattern Discovery. The model is trained *offline*. Path selection and embedding extraction take $O(N_e k d)$ time, where N_e is the number of entities in graph G, k is the path length bound, and d is the average degree of each entity. The KMC takes $O(|\mathcal{X}|H)$ time *w.r.t.* fixed vector length and iterations. The attribute ranking and selection take $O(kN_\rho(\sum_{j=1}^{m'} |\mathcal{P}'_j|) + m'k_R m|\mathcal{W}_i|)$ time, where N_ρ is the number of all paths in P, and $\sum_{j=1}^{m'} |\mathcal{P}'_j|$ is the number of patterns in $\mathbb{P}'$.

II. Relation Extraction. We next present Algorithm 1 for extracting a relation D_G (*i.e.,* $h(S,G)$) of schema R_G from G. It takes as input G, $f(S,G)$, $\mathbb{P}$, k, R_G, $\mathcal{M}_\rho$ and $\mathcal{A}$, as described in Algorithm 1. For each match (t, v_i) in $f(S,G)$, it extracts a tuple to complement t via path pattern matching.

More specifically, for each matching entity v_i, Algorithm 1 first finds paths starting from v_i in G and saves them in Π, guided by model $\mathcal{M}_\rho$ as in pattern discovery (SelectPath line 3). Then for each pattern cluster $\mathcal{P}_j$ that corresponds to attribute A_j in R_G, it performs path pattern matching between each path ρ in Π and each pattern p in $\mathcal{P}_j$, and assigns the label of the end vertices in the matching path, *i.e.,* $L(\rho.v_l)$, to θ_j as the attribute value (Extract line 4), where $L(\rho.v_l)$ maximizes value ranking function $\cos(x_{L(\rho.v_l)}, x_{A_j})$. If multiple paths are mapped to the same pattern cluster, the ranking function selects one that is semantically closest to keyword A_j. If there is no match for all path patterns in $\mathcal{P}_j$, a "null" value is assigned to θ_j. Finally, the extracted tuple consists of the vertex id of v_i and the attribute values of each θ_j (line 5).

Algorithm 1: Attribute extraction via pattern matching

Input: A set $\mathbb{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ of path pattern clusters, a path length bound k, a graph G, a set of vertex-tuple matches $f(S,G)$, extracted schema R_G(vid, $A_1, \ldots, A_m$), the model $\mathcal{M}_\rho$ for pattern discovery, user keywords $\mathcal{A}$.
Output: An extracted relation D_G of schema R_G.

1 $D_G \leftarrow \emptyset$;
2 **foreach** $(t_i, v_i) \in f(S,G)$ **do**
3 $\quad \Pi \leftarrow$ SelectPath $(G, v_i, k, \mathcal{M}_\rho, \mathcal{A})$;
4 $\quad (\theta_1, \ldots, \theta_m) \leftarrow$ Extract $(\mathbb{P}, \Pi, \mathcal{A})$;
5 $\quad D_G \leftarrow D_G \cup (v_i, \theta_1, \ldots, \theta_m)$;
6 **return** D_G;

Function Extract $(\mathbb{P}, \Pi, \mathcal{A})$:
1 $\quad$ **foreach** $\mathcal{P}_j \in \mathbb{P}$ **do**
2 $\quad\quad$ For each $p \in \mathcal{P}_j$ and each $\rho \in \Pi$, find the ρ that (1) makes $M(\rho, p) =$ true, and (2) maximizes the value ranking function for $A_j \in \mathcal{A}$.
3 $\quad\quad \theta_j \leftarrow L(\rho.v_l)$;
4 $\quad$ **return** $(\theta_1, \ldots, \theta_m)$

Example 8: Figure 3(b) shows the extracted relation D_G by matching path patterns of each cluster in $\mathbb{P}$, for user's interest $\mathcal{A}$. Note that the tuple of pid3 has "null" for location, as patterns in $\mathcal{P}_1$ find no matching paths starting from vertex pid3. □

Algorithm 1 takes $O(kN_\rho(\sum_{j=1}^{m} |\mathcal{P}_j|))$ time, where k and N_ρ are as given above; N is the number of pairs in $f(S,G)$; and $\sum_{j=1}^{m} |\mathcal{P}_j|$ is the number of all path patterns in $\mathbb{P}$. It caches and reuses the paths found during pattern discovery.

Extraction without reference tuples. RExt can also work without an input relation S as reference tuples, to extract relations from G as preprocessing, providing *reference schemas* of G for users to compose gSQL queries (recall Section II-B).

Consider graph G whose vertices can be classified as different types based on their labels. RExt identifies, for each type τ of vertices of G, a relation schema R_τ and an instance $g_\tau(G)$ of R_τ from G. It is a special case of attribute extraction since it has no reference tuples S. RExt extracts the relations in the same way as how it works when both G and a relation S are provided as input, except the following. (1) With G as the sole input, each time it only considers entity vertices of some type τ. (2) The second term of the attribute ranking function is empty since there is no S. It could also employ keywords $\mathcal{A}_\tau$ provided by users or extracted from query logs.

Figure 4 depicts the complete workflow of RExt.

B. IncExt: *Incremental Relation Extraction*

We next discuss how IncExt incrementally maintains relations extracted by RExt. There are two types of updates: (a) data updates, *i.e.,* updates to G and/or $\mathcal{D}$; and (b) user updates, *i.e.,* users' keywords $\mathcal{A}$ change when query interest shifts.

Data updates. For the lack of space, we focus on updates ΔG to graph G. Updates to $\mathcal{D}$ can be handled similarly.

Updates ΔG may change (a) attribute values previously extracted by RExt due to the change of paths in G; and (b) the match relation $f(S, G)$ computed by HER. For both cases, IncExt works by identifying the "affected" vertices of G that are in $f(S, G)$, and extracting the updated relations for them.

The logic of IncExt is simple: it collects a set V_Δ of vertices in G affected by Δ and relevant to $f(S, G)$, re-extracts their values (*i.e.,* tuples of R_G) via lines 3-4 of Algorithm 1, and commits the extracted values to D_G. IncExt collects V_Δ as follows: (a) for each new vertex v in $f(S, G)$, *i.e.,* vertices of G that are matched by some tuple in S via HER because of ΔG, add v to V_Δ; (a) for each vertex u in ΔG, add all old vertices v in $f(S, G)$ that are within k hops from u to V_Δ.

Intuitively, step (a) covers those newly matched vertices due to updates to $f(S, G)$ and ensures that IncExt extracts values for them. Step (b) assures that all entity vertices whose extracted values are likely affected by Δ will be re-examined by IncExt. Here by looking into k hops from vertices in ΔG, it covers all affected cases as RExt extracts values for each entity vertex v via paths from v that are of length at most k.

When ΔG is small, IncEXT is efficient since (1) only vertices that are within k hops of the graph updates are checked; (2) there is no need to rediscover path patterns; and (3) the cost for path pattern matching is small as described in the analysis of Algorithm 1. Moreover, there exists no accuracy loss in IncEXT compared with RExt starting from scratch, since pattern matching results of RExt and IncEXT are the same.

Keywords updates. When users' interest shifts and keywords in $\mathcal{A}$ change, the extracted schema R_G will likely change accordingly. To this end, instead of re-computing the schema R_G starting from scratch for the updated $\mathcal{A}$, IncExt only redoes the last phase (*i.e.,* step (4)) of path pattern discovery, to select attributes for R_G by using the ranking function $r()$ with updated keywords $\mathcal{A}$. After that, IncExt update the extracted relation of the new R_G for S. Without re-extracting the entire relation of R_G from scratch, IncExt only needs to extract values for those new attributes in R_G via Extract of Algorithm 1.

IV. Implementation of Semantic Joins

In this section we show how to implement semantic joins atop RDBMSs. We first develop exact methods for computing semantic joins, and identify gSQL that can be answered without invoking HER and RExt online (Section IV-A). We then present a heuristic method for generic queries (Section IV-B).

A. *Exact Methods of Semantic Joins*

We start with a conceptual-level exact semantic join method.

Baseline. The conceptual-level method computes semantic joins online in response to individual user's request (*e.g.,* keywords $\mathcal{A}$). It calls external HER and RExt at query time.

For enrichment join $S \bowtie_{\mathcal{A}} G$, we first use HER to compute the match relation $M = f(S, G)$ between tuples of S and vertices of G, with which we then extract relation $h(S, G)$ from G for S and $\mathcal{A}$, by invoking RExt (Section III). Thus $S \bowtie_{\mathcal{A}} G$ simply reduces to a three-way natural join $S \bowtie M \bowtie h(S, G)$.

For link join $S_1 \bowtie_G S_2$, we invoke HER to identify the matching vertices to tuples of S_1 and S_2 in G, check their pairwise distance via a bi-directional BFS search, and return the join of any $t_1 \in S_1$ and $t_2 \in S_2$ if there exist matching vertices to t_1 and t_2 within k hops for a pre-defined bound k.

HER and RExt can be encapsulated as SQL UDFs, stored procedures, or a combination of SQL and external scripts [19].

An efficient method. The above implementation requires to invoke HER and RExt at run time, which could be costly. To this end, we identify a large class of gSQL queries which we can answer without calling HER and RExt at query time.

The idea is to profile graph G and extract a collection $\mathcal{D}_G$ of relations from G for all entities (tuples) in database $\mathcal{D}$ beforehand offline, by invoking RExt as offline preprocessing with G (recall Section III-A). In doing so, gSQL queries that are issued online can then be answered by using $\mathcal{D}$ and $\mathcal{D}_G$ only, without invoking HER and RExt on-the-fly.

More specifically, by taking G as the only input, RExt pre-computes and maintains the following: (1) for each input relation D, the set $f(D, G)$ of HER matches; (2) for each schema R in $\mathcal{R}$, a set $\mathcal{A}_R$ of frequent keywords; and (3) the extracted schema R_G and relation $h(D, G)$ for D and $\mathcal{A}_R$. The relations are materialized in RDBMS and incrementally maintained by IncExt in response to updates (Section III-B).

To speed up link joins, we also pre-compute connectivity relations g_L for vertices of G that match selected tuples in D. It is specified by two sets of predicates P and P' over D such that if v and v' match tuples t and t' of D that satisfy some predicates in P and P', respectively; then (v, v') is in g_L iff v and v' are connected in G within k hops. Here P and P' are the selection conditions of sub-queries Q_1 and Q_2 in a link join $Q_1 \bowtie_G Q_2$. We keep those g_L for recent queries as a cache.

Well-behaved queries. Given $\mathcal{D}_G$, we identify gSQL queries that can be answered without external calls of HER and RExt. We say that an enrichment join $Q \bowtie_{\mathcal{A}} G$ is *well-behaved* if (1) $\mathcal{A} \subseteq \mathcal{A}_R$, *i.e.,* the keywords of $\mathcal{A}$ are covered by the pre-selected $\mathcal{A}_R$; and (2) either (a) the output schema R_Q of Q contains one and only one tuple id of some base relation R of $\mathcal{D}$ or (b) R_Q contains attributes from only one base relation R of $\mathcal{D}$. Intuitively, condition (1) ensures that well-behaved queries reference the pre-extracted schemas, and condition (2) ensures that each answer to Q refers to an entity in some base relation of $\mathcal{D}$ that can be deduced via R_Q. A link join $Q_1 \bowtie_G Q_2$ is well-behaved if both Q_1 and Q_2 are well-behaved.

A gSQL query Q is *well-behaved* if every semantic join in Q is well-behaved. It takes linear time to decide whether a gSQL Q is well-behaved, via a bottom-up scan of the syntax tree of Q, to check whether each of its semantic joins is well-behaved.

Example 9: Assume that $\mathcal{A}_{\mathsf{customer}} = \{\mathtt{stock}, \mathtt{company}\}$ and $\mathcal{A}_{\mathsf{product}} = \{\mathtt{company}, \mathtt{loc}\}$. Then by definition, Q_1, Q_2 and Q_3 of Example 4 are all well-behaved gSQL queries. □

With $\mathcal{D}_G$, well-behaved gSQL can be converted to SQL and answered without calling HER and REXt online. We discuss the implementation for two classes of semantic joins below.

(a) Static joins. A *static enrichment join* has the form $Q_e = R \bowtie_{\mathcal{A}} G$ where R is a schema in the input database schema $\mathcal{R}$. For a well-behaved Q_e, *i.e.,* when $\mathcal{A} \subseteq \mathcal{A}_R$, We convert Q_e into a three-way natural join $Q = R \bowtie f(D, G) \bowtie h(D, G)$ and directly answer it via the underlying RDBMS without invoking HER and REXt at runtime. One can readily verify that $Q_e(\mathcal{D}, G) = Q(\mathcal{D}, \mathcal{D}_G)$, *i.e.,* the answers to query Q_e computed in this way are exact. Similarly, a *static link join* has the form $Q_l = R_1 \bowtie_G R_2$ for input schemas R_1 and R_2 in $\mathcal{R}$. We convert Q_l simply to $g_L(D_1, D_2, G)$. Again Q_l can be exactly answered by the underlying RDBMS using the pre-computed $\mathcal{D}_G$, without calling HER and REXt.

(b) Dynamic joins are of the form $Q = Q_1 \bowtie_{\mathcal{A}} G$ or $Q = Q_1 \bowtie_G Q_2$, where Q_1 and Q_2 are queries instead of base relations.

When Q is well-behaved, we can convert Q to an SQL query along the same lines as well-behaved static joins, and computes its exact answers by using the underlying RDBMS. For instance, if each tuple in $Q_1(\mathcal{D}, G)$ refers to a tuple in relation D of schema R and $\mathcal{A} \subseteq \mathcal{A}_R$, then enrichment join $Q_1 \bowtie_{\mathcal{A}} G$ can be rewritten into an equivalent SQL query $Q_1 \bowtie R \bowtie f(D, G) \bowtie h(D, G)$, where $Q_1(\mathcal{D}, G)$ identifies a subset of D, and Q_1 is inductively converted into an SQL query. We can answer this query just like static joins, by RDBMS without calling HER and REXt at query time; similarly for link joins.

B. From Exact Joins to Heuristic Joins

There are gSQL queries that are not well-behaved.

Example 10: Recall $\mathcal{D}$ and G from Example 1. Assume that Bob (cid02) wants to invest in medium-risk stocks for at least 1000 shares. To do this, a broker uses an enrichment join $Q_4 = Q' \bowtie_{(\mathtt{company})} G$ to find suitable companies for Bob, where Q' identifies all medium-risk stocks that Bob is able to buy in 1000 shares, via a join $\sigma_{\mathsf{cid}=\mathsf{cid02}}\mathsf{customer} \bowtie_{\mathsf{bal}\geq 1000\times\mathsf{price}} \sigma_{\mathsf{risk}=\mathsf{medium}}\mathsf{product}$. Then Q_4 is not well-behaved since Q' fetches the id attributes of both customer and product. □

We next propose a heuristic method to answer such queries without calling HER and REXt, to strike a balance between the cost of query evaluation and the accuracy of query answers. The assumption is that graph G is "typed", *i.e.,* the types of its entities can be determined by their labels, and that the default bound k and reference keywords $\mathcal{A}_R$ are decided as above.

Heuristic joins. Recall that REXt computes schema R_τ and relation $g_\tau(G)$ for entities of each type τ when taking G as input alone (Section III-A), using default k and $\mathcal{A}_R$. With these, we develop an approximate method for semantic joins.

Consider enrichment join $Q_e = Q \bowtie_{\mathcal{A}} G$. Denote the output schema of Q by R_Q, and $Q(\mathcal{D}, G)$ by S. We answer it in three steps; the case for link joins is similar.

(1) We first identify what types τ of vertices in G may match and enrich tuples in S. This is conducted by matching attributes of R_Q and R_τ via schema-level matching [20], [21]. We mark a relation $g_\tau(G)$ as relevant to Q if R_τ and R_Q share the most common attributes among all the relations $g_{\tau'}(G)$ that are extracted without reference tuples (recall Section III-A). We approximate $Q \bowtie_{\mathcal{A}} G$ by joining Q with such $g_\tau(G)$.

(2) We match S and $g_\tau(G)$ by either (a) pairwise tuple comparison based ER method or (b) end-to-end ER that takes entire S and $g_\tau(G)$ as input and computes the match relation all at once. Here (a) can be implemented as a simple UDF as the join condition between S and $g_\tau(G)$ to check whether $t \in S$ and $t' \in g_\tau(G)$ make a match [22]. The implementation of (b) requires to take S and $g_\tau(G)$ as input and encapsulates a complete ER solution, *e.g.,* JedAI [23], in the UDF.

(3) We convert Q_e to a join between S and $g_\tau(G)$ with ER matching as the join condition. This is conducted in RDBMS via stored procedures or UDF, without calling HER and REXt.

Example 11: We can answer Q_4 of Example 10 with heuristic join. Assume that we have extracted a relation $g_{\mathsf{product}}(\mathsf{pid}, \mathsf{company})$ for $\mathsf{pid}'s$ (products) from G. We can deduce that g_{product} matches $R_{Q'}$ since they share the same pid attribute and the extracted attribute company of Q_4 is also in g_{product}. Hence, it links answer tuples to Q_5 (*e.g.,* $\{t_2\} \bowtie \{t_8\}$ in $\mathcal{D}$ of Fig. 1) with tuples of g_{product} (*e.g.,* (pid4, company2)), by identifying pid4 and fd4 via ER (*e.g.,* [23]). By using heuristic joins, Q_4 is answered without calling REXt and HER online. □

V. Experimental Study

Using real-life and synthetic data, we conducted four sets of experiments to evaluate (1) the need for semantic joins when querying relations and graphs taken together, (2) the quality of semantic join results, and the efficiency of (3) semantic join implementations and (4) incremental relation extraction.

Experimental setting. We start with the experimental settings.

Datasets. We used 6 collections of real-life datasets (see Table II); each is a pair of a graph and a relational dataset. (1) Drugs includes drug relation [24] of drug products, interact relation [25] of drug interactions, and knowledge graph drugKG [26] about drugs with their efficacies and diseases with descriptions about their symptoms. (2) FakeNews is composed of relation fakenews [27] about news sources and topicKG graph [28] for categories and themes of new reports, with the domains of the keywords extracted from the headline. (3) Movie collects movies, directors, actors, etc., in relations of IMDB [29] and LinkedMDB graph [30]. (4) MovKB includes IMDB relations and YAGO3 graph [31]. (5) Paper collects publications and authors in relations from DBLP [32] and a graph of RKBExplorer [33], [34]. (6) Celebrity collects athletes and politicians in DBpedia relations [35], [36], [37] and YAGO3 graph [31]. We linked entities across graphs and their corresponding relations. The relations and graphs of Movie, Paper and Celebrity are from closely related data sources, and those of Drugs, FakeNews and MovKB are from

Data coll.	Relations	Graphs
Drugs	drug/interact: 15K/192K tuples	drugKG: 264K vertices, 445K edges
FakeNews	fakenews: 2.1M tuples	topicKG: 1.3M vertices, 4M edges
Movie	IMDB: 39.2M tuples	LinkedMDB: 2.3M vertices, 5.4M edges
MovKB	IMDB: 39.2M tuples	YAGO3: 3.4M vertices, 10.2M edges
Paper	DBLP: 4.4M tuples	RKBExplorer: 15.9M vertices, 31.1M edges
Celebrity	DBpedia: 372K tuples	YAGO3: 3.4M vertices, 10.2M edges

TABLE II: Dataset collections

independent data sources but share overlapped information.

Queries. We evaluated 36 queries across relations and graphs of the six collections, 6 queries for each. Among them, 32 involve enrichment joins, 4 need link joins, 4 are dynamic, 10 contain more than one semantic joins, 17 have negation, and 4 have aggregation. All queries require to check both data in relations and data in graphs. For each $R \bowtie_{\mathcal{A}} G$ in the queries, keywords in $\mathcal{A}$ are the attribute types to be extracted from G.

Implementation. We implemented both RExt and IncExt in Python and C (see Section III). We also implemented all the semantic join methods in C++ (Section IV) and deployed them atop PostgreSQL; we remark that our methods are database agnostic and can be deployed over other RDBMS.

For RExt, we parallelized KMC [38] and attribute ranking. We adopted pre-trained 100-dimensional GloVe word embedding [17] and LSTM networks [16] for vectorizing vertex and path labels, respectively. The LSTM model is trained with default configurations in [16] on graph G of each dataset. We performed L2 normalization before vector concatenation; each vertex-path pair was represented by a 200-dimension vector.

As baselines, we also implemented (a) RExtBertEmb, RExt using Bert as word embedding, (b) RExtShortEmb, RExt using shorter 50-dimensional GloVe as word embedding, (c) RExtBertSeq, RExt using Bert for sequence embedding, (d) RExtShortSeq, RExt using narrower 50-wide hidden layer LSTM for sequence embedding, and (e) RndPath, RExt taking random paths pertaining to each entity for relation extraction. We find no prior method that can do the job of RExt.

Configuration. We ran the experiments on a cluster of 10 linux machines, each with 2 Intel Xeon 2.2 GHz CPUs and 64 GB memory. We used all 10 machines for RExt, and used a PostgreSQL (v13.5) server powered by 3.2GHz CPU and 16 GB memory, which also served HER. When RExt profiles graphs G for offline preprocessing, we configured RExt with all the keywords and attribute names referenced by the gSQL queries we tested. We used JedAI [3] as both HER for semantic joins and the ER step of heuristic joins, with configurations picked for each data collection based on its characteristics following [3]. For heuristic joins, we extracted 2, 2, 2, 4, 1 and 1 relations for Movie, MovKB, Paper, Celebrity, Drugs and FakeNews, respectively which account for 15.7%, 0.9%, 0.07%, 0.03%, 39.5% and 7.7% of the size of the raw graphs. Each experiment was run 5 times; the average is reported here.

Experimental results. We next report our findings.

Exp-1: Case study. We first validated the need for semantic joins. Recall query Q_1 from Example 1. As shown in Example 4, Q_1 requires both data in relations and data in graphs. Without RExt, the closest that an RDBMS can do is to import and

join type	all	non-well-behaved	enrichment	link
F-measure	0.88	0.81	0.89	0.81

data col.	Drugs	FakeNews	Movie	MovKB	Paper	Celebrity
F-measure	0.95	0.82	0.84	0.89	0.88	0.90

TABLE III: Relative accuracy of heuristic joins

encode the graph G in Example 1 as vertex and edge relations, and then approximate Q_1 in SQL over $\mathcal{D}$ and the cast relations. However, this is neither accurate nor practical as one does not know how many joins are required to retrieve attributes from G, not to mention its prohibitive computational cost.

We further demonstrate its needs via public datasets with two real-life tasks from biomedical and media applications.

Drug interactions (q_1): "find drugs that are for the same disease but in conflict with each other". It is over Drugs with (simplified) drug(CAS, name, class) and interact(CAS_1, CAS_2, type) relations, and graph drugKG for drugs, efficacy, symptoms and diseases. Intuitively, q_1 first retrieves disease data for drug and then checks whether their type is -1 (in conflict).

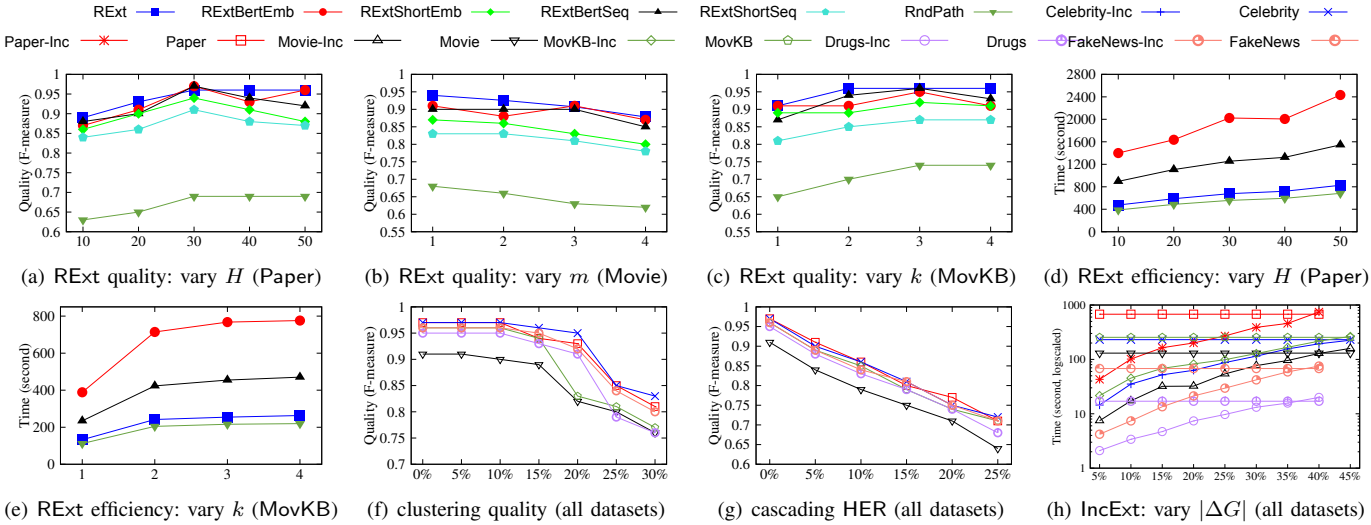
This is done with semantic joins as: $\sigma_{T_1.\text{disease}=T_2.\text{disease}}(T_1 \bowtie_{T_1.\text{CAS}=\text{CAS}_1} \sigma_{\text{type}=-1}\text{interact} \bowtie_{T_2.\text{CAS}=\text{CAS}_2} T_2)$, where T_1 and T_2 are renamings of semantic join drug $\bowtie_{\text{disease}}$ drugKG. It retrieves typical targeted diseases and symptoms for drugs by joining drug with graph drugKG, and checks interactions with interact for drugs that may be used for the same disease.

This cannot be simply done in RDBMS by converting drugKG to an edge relation and joining it with drug since a drug may be connected to a disease via a path of related symptoms and diseases, and relational joins are not able to distinguish strongly relevant paths from irrelevant ones, even when we know how many joins are needed. For instance, while drug Spinosad is often used for disease Pediculosis as confirmed by its connection to Pediculosis via a path pattern (drug → efficacy → symptom ← disease), drug Dimenhydrinate is not for Intestinal fluke infection although they are also connected via the same path pattern. Hence, RDMBS would not distinguish the case of Spinosad for Pediculosis from Dimenhydrinate for Intestinal fluke infection, while RExt can tell the difference due to its ML-based path matching.

Fake news authors (q_2): "find domain keywords used by fake news authors". It is over the FakeNews with relation fakenews(author, country, lanugage) and topicKG, a knowledge graph of news category, with keywords extracted from the headline. Similar to q_1 above, q_2 needs to thematize each author in fakenews by extracting the best topic from graph topicKG that covers keywords in the title's of news published by the author. By the model-guided path pattern discovery and selection, RExt is able to extract the most suitable topic that covers words from news titles for each author.

Exp-2: Quality of semantic joins. We next tested (a) the effectiveness of RExt and (b) the accuracy of heuristic joins.

(I) Effectiveness of RExt. For each relation schema R, we first picked and dropped m attributes (columns) from R, yielding relation schema R'. We then tested the ability of RExt to



(a) REXt quality: vary H (Paper) (b) REXt quality: vary m (Movie) (c) REXt quality: vary k (MovKB) (d) REXt efficiency: vary H (Paper)

(e) REXt efficiency: vary k (MovKB) (f) clustering quality (all datasets) (g) cascading HER (all datasets) (h) IncExt: vary $|\Delta G|$ (all datasets)

Fig. 5: Performance of REXt and IncExt

recover the dropped values (columns) in R from graph G.

For instance, we dropped columns volume and affiliation from the DBLP relation of Paper, yielding a relation DBLP$'$. We then used REXt to "recover" the dropped columns by extracting from graph RKBExplorer of Paper via semantic join DBLP$' \bowtie_{\mathcal{A}}$ RKBExplorer, where $\mathcal{A}$ is a set of keywords that include 'volume' and 'affiliation'. When varying the size $|\mathcal{A}|$ of $\mathcal{A}$ to evaluate the impact of keywords, we expanded $\mathcal{A}$ with randomly picked volume and affiliation values from the dropped columns (*e.g.,* 'vol. 41' and 'NASA') as additional keywords. We calculated the accuracy (F-measure) of join results by taking the original DBLP relation as the ground truth.

(a) Impact of parameters. Using this method, we evaluated the F-measure of REXt with varying configurations. We varied path length bound k, the number H of clusters, the number $|\mathcal{A}|$ of keywords, and the number m of (dropped) attributes, and tested the accuracy of REXt over all 6 data collections. When varying a parameter, all other parameters are set to the default (30, 3, 4 and 3 for H, m, $|\mathcal{A}|$ and k, respectively). Key results are shown in Figures 5(a), 5(b) and 5(c).

(1) When varying H from 10 to 50, the F-measure of REXt on all the datasets first increases and then remains the highest. This is because KMC better discerns different vertex-path-pairs with larger H. However, if H keeps growing, noisy and small clusters are introduced. Such clusters can be removed by path pattern refinement (Section III-A). This avoids false patterns and thus the accuracy remains the highest with larger H.

(2) Varying m from 1 to 4, the accuracy of REXt decreases, *e.g.,* from 0.94 to 0.88 over Movie (Fig. 5(b)). This is because with larger m, REXt has to extract more attributes, and with this come larger uncertainty and slightly lower F-measure.

(3) Varying k from 1 to 4, the F-measure of REXt increases, *e.g.,* from 0.91 to 0.96 on MovKB, as longer paths can capture more candidate attributes. However, the quality plateaus when k keeps growing, *e.g.,* k from 3 to 4, since attributes extracted by longer paths have weaker associations and are less useful.

(4) Varying $|\mathcal{A}|$ from 3 to 6, the F-measure of REXt fluctuates a bit but is consistently above 0.89. This shows that REXt is robust in accuracy, regardless of how users specify the keywords.

(b) Ablation study. We next study the contribution of each ML component to the accuracy of REXt (recall Section III-A).

(1) LSTM for $\mathcal{M}_\rho$. We first examined the impact of LSTM for sequence embedding by comparing the accuracy of REXt with RExtBertSeq and RExtShortSeq. The results with varying parameters are shown in Figures 5(a)-5(c). REXt is robust with varied $\mathcal{M}_\rho$: the accuracy of the relations extracted by REXt is consistently better but rather close to that of RExtBertSeq in all cases; the gap between REXt and RExtShortSeq is slightly larger than RExtBertSeq since the accuracy of 50-wide LSTM is much worse than Bert and LSTM. This also confirms the choice of LSTM in REXt since Bert is computationally costly and narrower LSTM is less competitive in accuracy.

(2) GloVe for $\mathcal{M}_e$. Similarly, we tested the robustness of REXt *w.r.t.* the word embedding model $\mathcal{M}_e$, by comparing the accuracy of REXt with RExtBertEmb and RExtShortEmb. As shown in Figures 5(a)-5(c), the accuracy of extracted relations is relatively stable. Although REXt is consistently the best, its gap with RExtBertEmb and RExtShortEmb is moderate.

(3) $\mathcal{M}_\rho$ and $\mathcal{M}_e$ taken together. Although REXt is robust with different models for $\mathcal{M}_\rho$ and $\mathcal{M}_e$, its accuracy dramatically decreases if we drop both $\mathcal{M}_\rho$ and $\mathcal{M}_e$. For instance, the accuracy of REXt is consistently 21% higher than that of RndPath, which uses random paths instead of the ML guided method in REXt (Figures 5(a)-5(c)). This validates the effectiveness of the ML-based ($\mathcal{M}_e$ and $\mathcal{M}_\rho$) path selection in REXt.

(4) Clustering. Finally, we randomly injected noisy labels to the path clustering produced by KMC, and evaluated the impact of KMC quality on the accuracy of REXt. As shown in Fig. 5(f), the accuracy of REXt does not significantly drop until 20% noisy labels are injected. This is because the pattern refinement step is robust against clustering errors by picking the majority of the patterns, making REXt robust with KMC.

(c) Impact of HER. We have evaluated the impact of the accuracy of ML components in RExt on the extracted relations in (b) above. Now we further examine the impact of cascading error from HER, by testing the accuracy of RExt with HER matches $f(S, G)$ of varying accuracy. Specifically, we follow the setting of Exp-2(a) but randomly injected mismatches. Denote by η the percentage of mismatches in the HER match relation. As shown in Fig. 5(g), the accuracy of RExt deteriorates, as expected, when η increases. However, the impact of η is relatively stable as the accuracy of RExt decreases almost proportionally to η. This is because mismatches only cause RExt to extract properties for the "wrong" target tuple, without affecting the extraction for other correctly matched tuples.

(II) Accuracy of heuristic joins. We also evaluated the accuracy of heuristic joins over all 6 data collections, by assuming that HER and RExt are accurate. We first examined and found that 32 out of 36 of gSQL queries are well-behaved, and hence can be exactly answered via the optimized semantic join implementation instead of the conceptual-level baseline.

We then tested the accuracy of heuristic joins *relative to* HER and RExt. We "enforced" heuristic joins on all the gSQL queries, including well-behaved ones. By treating exact join results as the ground truth, we tested the F-measure of heuristic joins. As shown in Table III, on average the F-measure of heuristic join results is 0.88 for all types of joins over all datasets, and is 0.81 for non-well behaved joins. This shows that heuristic join is accurate in approximating semantic joins.

Exp-3: Efficiency. We next evaluated the efficiency of (1) semantic joins in answering gSQL queries and (2) RExt in extracting data from graphs, over all 6 data collections. Using the same setting of Exp-2 for parameters k, H and m, we tested (a) the end-to-end evaluation time of all 36 queries; (b) the efficiency of RExt and the baselines; and (c) the effectiveness of heuristic joins for queries that are not well behaved.

(I) *Offline preprocessing*. We start with the offline preprocessing involved, *i.e.,* (a) model training cost for extraction and (b) pre-computation used by well-behaved and link joins.

(a) Training. It takes 64s, 76s, 220s, 32s and 43s to train the LTSM model for RExt over LinkedMDB, YAGO3, RKBExplorer, drugKG and topicKG graphs, respectively. The training time of baselines RExtBertEmb and RExtBertSeq is much longer than that of RExt, *e.g.,* 542s and 674s over YAGO3, respectively, while baselines RExtShortEmb and RExtShortSeq are comparable to RExt.

(b) Pre-computation. Recall that well-behaved and heuristic joins use pre-computed relations. We report their costs below.

(1) For well-behaved enrichment joins, it takes RExt 130s, 254s, 677s, 230s, 17s and 68s to extract relations $h(D, G)$ for Movie, MovKB, Paper, Celebrity, Drugs and FakeNews, respectively, taking 15.7%, 0.9%, 0.07%, 0.03%, 39.5% and 7.7% of the size of the raw data collection for materialization.

(2) For link joins, it takes 0.01% of the size of the graph on average to cache all link joins in the queries via g_L ($k = 3$). This is due to the fact that the number of tuples participated in the link joins is typically not large, leading to limited pairs of vertices in the connectivity relation (cache, Section IV).

We next move on to online computations. We assume a cold start for link joins, *i.e.,* no connectivity cache g_L by default.

(II) *End-to-end performance*. We evaluated the (online) evaluation time of gSQL queries by the conceptual-level baseline, optimized method for well-behaved joins, and heuristic joins. With the relations $h(D, G)$ pre-computed in (I), we have 32 well-behaved gSQL queries out of the total 36 queries.

(1) Both well-behaved and heuristic joins scale well with big relations and graphs. In particular, for well-behaved joins, with pre-extracted relations that account for 10.7% of the size of graphs, it returns query answers within 9.2s across relations of up to 4.4 million tuples and graphs with 31.1 million edges.

(2) When using the optimized join implementation for well-behaved queries and heuristic join for non-well-behaved ones, on average it is 114.9X faster than the conceptual-level baseline. This is because a large percentage (88.9%) of test queries benefit from the processing of well-behaved queries, which is substantially faster by avoiding invoking HER and RExt at query time. This said, the conceptual-level baseline does not take very long despite running HER and RExt online, *e.g.,* it takes 47.5s on average to answer a query.

(3) While heuristic join is slower than the optimized join implementation, it is still 8.19X faster than the baseline on average, up to 27.9X. This, together with Exp-2, shows that heuristic joins are an effective approximation to semantic joins.

(4) In particular, for those well-behaved queries with link joins, the optimized method is still effective with g_L disabled, on average 6.13X faster than the baseline. This is because even without g_L, well-behaved link joins could still benefit from pre-extracted relations by RExt and HER. Furthermore, when g_L is enabled and link joins are a cache hit, the speedup goes up to 23.8X because of no need for traversing G on-the-fly.

(III) *Performance of* RExt. We also tested the scalability of RExt and its baselines for extracting relations $h(S, G)$ from graph G when S is an entire input relation. Varying H, m and k as in Exp-2, we tested the time for extracting attributes. Key results are reported in Figures 5(d)-5(e).

(1) RExt is faster than its Bert variants, *e.g.,* on average 3.03X and 1.78X faster than RExtBertEmb and RExtBertSeq over MovKB, respectively. Unsurprisingly, RndPath is the fastest of all methods due to its simpler design but lower accuracy.

(2) As expected, RExt take longer to extract attributes with larger k, *e.g.,* it takes RExt from 132.42s to 263.28s when k is increased from 1 to 4 over MovKB. This is because with larger k, RExt has to examine more paths when extracting values. Similarly, KMC and attribute ranking take longer with larger H. In contrast, the runtime of RExt is not very sensitive to m and $|\mathcal{A}|$ since they only affect the final selection of the attributes and values, and incur a much smaller cost compared to the time for path exploration, clustering and attribute ranking.

We remark that that the extraction time here is much longer than it takes to answer online gSQL queries (even via the conceptual-level baseline). This is because when answering gSQL queries, RExt only needs to extract data for the tuples selected by the queries, which are often only a small fraction of the full relations. Moreover, well-behaved queries reuse the results extracted by the preprocessing and does not invoke RExt at run time; hence query evaluation is even faster.

Exp-4: Incremental maintenance. We next evaluated the performance of IncExt for handling updates, using the same workload and configurations as in Exp-2 above. We focus on updates to graphs G; the case for updating $\mathcal{D}$ is similar.

We generated random updates ΔG consisting of the same number of insertions and deletions, so that the size of the graph remains unchanged. We compared the runtime of IncExt against RExt that re-computes HER matches and extracted data in the updated G. We varied $|\Delta G|$ from 5% to 45% of $|G|$.

As shown in Figure 5(h), when $|\Delta G| = 5\%|G|$, IncEXT beats RExt by 15.9X, 15.7X, 17.5X, 11.8X, 8.1X and 16.2X on Celebrity, Paper, Movie, MovKB, Drugs and FakeNews respectively. It is still faster than RExt when $|\Delta G|$ is up to 45%, 35%, 40%, 45%, 35% and 35% of $|G|$, respectively.

Summary. We find the following. (1) Relation extraction and semantic joins enable us to express real-life queries across relations and graphs in SQL, which is not possible when G is simply cast as vertex/edge relations. (2) By means of semantic joins, RExt is able to extract high-quality relations from graphs with average F-measure above 0.95. (3) RExt scales well with large relations and graphs, *e.g.,* it extracts attributes in 230.4s on relations with 3.4M tuples and graphs with 10.2M edges. (4) Our implementation of semantic joins answers online gSQL queries efficiently, taking at most 9.2s when queries are well-behaved, 114.9X faster than the baseline. (5) Heuristic joins speed up semantic joins that are not well-behaved by 3.1X on average, with F-measure above 0.81. (6) IncExt handles updates efficiently. It updates the extracted relations 14.2X faster than RExt when $|\Delta G|$ accounts for 5% of $|G|$, and is still faster when $|\Delta G|$ is up to 45% of $|G|$.

VI. Related Work

We categorize the related work as follows.

Schema/data extraction. There has been research on summarizing semistructured data in terms of tree and graph patterns with descriptions [39], [40]. There has also been work on relation-schema extraction by navigating extracted data [41], querying nested key-value data [42], developing schema [43] and integrating XML data with relations [44]. Related is also feature selection in ML (see [45], [46], [47] for surveys). The idea is to filter out attributes that are irrelevant to the tasks, by ranking features using hand-crafted criteria [45], [48] or by assessing the closeness to a manually labeled training set [46], [47]. The ML models target images, texts and tableaux data.

RExt differs from the prior work in the following. (1) It extracts relations from graphs to enrich an *existing relation schema*, not to abstract the topological structure of the entire graph. (2) It extracts attributes and their values based on users' interest for queries, while no prior methods are query-driven. (3) As opposed to feature selection techniques [45], [46] that focus on relations, text or images, we tackle graphs that model data in topological structures; moreover, different from feature selection via supervised training [46], [47], we extract relations from graphs by sequence embedding and clustering, which are unsupervised and data-driven, reducing manual labeling cost.

Multi-model systems. A host of system designs have been studied to support datasets in multiple data models, often dubbed as polyglot systems [49], [50], [51], [52], [53], [54], [55], [56], multistores [57], [58], [59], or polystores [60], [61], [62].

Our work differs from these approaches in the following.

(1) Instead of developing new systems, this work aims to equip existing RDBMSs with a convenient capacity of querying relations and graphs, retaining the ease and composability of SQL.

(2) While HER, RExt and semantic joins can be plugged into polyglot systems and support queries across relations and graphs. Currently polyglot systems consider neither linking entities with heterogeneous structures nor graph property extraction. In particular, relation-based systems rely on costly SQL joins as graphs are encoded as "schemaless" edge relations.

(3) Multistores and polystores do not yet support graphs (storage and queries). Instead, semantic joins and relation extraction are system agnostic and can be incorporated into existing RDBMSs to support SQL across relations and graphs.

Entity linking in query processing. There has also been work on entity-aware query processing (EQP) [63] and query-aware entity resolution (QER) [64], [65], [66]. Both aim to embed online entity linking into query processing, to clean data on-the-fly. They target relational data and queries, not graphs.

In contrast, semantic joins and graph property extraction aim to correlate entities in different models (relations and graphs), synthesize the data and sensibly answer queries. We also provide a method to incrementally maintain extracted relations.

VII. Conclusion

The novelty of the work consists of (1) a notion of semantic joins; (2) an ML-based method for extracting relations from graphs; (3) an incremental method for maintaining extracted graph properties in response to updates; and (4) efficient implementations of semantic joins. These provide RDBMSs with a capacity to querying relations and graphs in SQL. We have experimentally verified that the approach is promising.

One topic for future work is to cope with $\mathcal{D}$ and G when values are missing from their key attributes; one way to impute missing values is via semantic joins with knowledge bases. Another topic is to extend the extraction method with graph embeddings and capture the vicinity of given graph nodes.

Acknowledgments. Cao is supported by Royal Academy of Engineering Research Fellowship RF\201920\19\319 and Edinburgh-Huawei Joint Lab. Fan is supported in part by Royal Society Wolfson Research Merit Award WRM/R1/180014.

REFERENCES

[1] "Neo4j use cases," *https://neo4j.com/use-cases/*.

[2] Business of Data, "How graph databases are transforming advanced analytics," *https://www.business-of-data.com/articles/graph-databases*, 2020.

[3] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis, "Three-dimensional entity resolution with jedai," *Information Systems*, vol. 93, p. 101565, 2020.

[4] X. Fu, J. Zhang, Z. Meng, and I. King, "MAGNN: Metapath aggregated graph neural network for heterogeneous graph embedding," in *Proceedings of The Web Conference 2020*, 2020, pp. 2331–2341.

[5] W. Fan, L. Geng, R. Jin, P. Lu, R. Tugey, and W. Yu, "Linking entities across relations and graphs," in *ICDE*. IEEE, 2022, pp. 634–647.

[6] G. Abuoda, S. Thirumuruganathan, and A. Aboulnaga, "Accelerating entity lookups in knowledge graphs through embeddings," in *ICDE*. IEEE, 2022, pp. 1111–1123.

[7] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, and P. C. Arocena, "Data lake management: Challenges and opportunities," *PVLDB*, vol. 12, no. 12, pp. 1986–1989, 2019.

[8] E. F. Codd, "Relational completeness of data base sublanguages," *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[9] R. Isele, A. Jentzsch, and C. Bizer, "Silk server-adding missing links while consuming linked data," in *COLD*, 2010, pp. 85–96.

[10] M. Li, Q. Zeng, Y. Lin, K. Cho, H. Ji, J. May, N. Chambers, and C. Voss, "Connecting the dots: Event graph schema induction with path language modeling," in *EMNLP*, 2020, pp. 684–695.

[11] X. V. Lin, R. Socher, and C. Xiong, "Multi-hop knowledge graph reasoning with reward shaping," in *EMNLP*, 2018.

[12] Y. Lin, Z. Liu, H. Luan, M. Sun, S. Rao, and S. Liu, "Modeling relation paths for representation learning of knowledge bases," *arXiv preprint arXiv:1506.00379*, 2015.

[13] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA, 1967, pp. 281–297.

[14] X. Li and Z. Fang, "Parallel clustering algorithms," *Parallel Computing*, vol. 11, no. 3, pp. 275–290, 1989.

[15] A. Fahad, N. Alshatri, Z. Tari, A. Alamri, I. Khalil, A. Y. Zomaya, S. Foufou, and A. Bouras, "A survey of clustering algorithms for big data: Taxonomy and empirical analysis," *IEEE transactions on emerging topics in computing*, vol. 2, no. 3, pp. 267–279, 2014.

[16] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," *arXiv preprint arXiv:1708.02182*, 2017.

[17] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[18] M. Ebraheem, S. Thirumuruganathan, S. Joty, M. Ouzzani, and N. Tang, "Distributed representations of tuples for entity resolution," *PVLDB*, vol. 11, no. 11, pp. 1454–1467, 2018.

[19] Microsoft, "sp_execute_external_script (transact-sql)," https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-execute-external-script-transact-sql?view=sql-server-ver16, 2022.

[20] D. Beneventano, S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, and M. Vincini, "Information integration: The MOMIS project demonstration," in *VLDB*, 2000, pp. 611–614.

[21] E. Rahm and P. A. Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[22] G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas, "Blocking and filtering techniques for entity resolution: A survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–42, 2020.

[23] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis, "The return of JedAI: End-to-end entity resolution for structured and semi-structured data," *PVLDB*, vol. 11, no. 12, pp. 1950–1953, 2018.

[24] "Drugbank release," https://go.drugbank.com/releases/latest#open-data.

[25] J. Y. Ryu, H. U. Kim, and S. Y. Lee, "Deep learning improves prediction of drug–drug and drug–food interactions," *Proceedings of the National Academy of Sciences*, vol. 115, no. 18, pp. E4304–E4311, 2018. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.1803294115

[26] "Kegg medicus," https://www.kegg.jp/kegg/medicus.html.

[27] "Getting real about fake news," https://www.kaggle.com/datasets/mrisdal/fake-news.

[28] "News category dataset," https://www.kaggle.com/datasets/rmisra/news-category-dataset.

[29] "IMDB," *https://www.imdb.com/interfaces*.

[30] O. Hassanzadeh and M. P. Consens, "Linked movie data base," in *LDOW*, 2009.

[31] *https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/downloads*.

[32] DBLP, "Relational DBLP data," *https://dblp.uni-trier.de/xml/*.

[33] ——, "RDF DBLP data," *http://dblp.rkbexplorer.com*.

[34] H. Glaser, I. C. Millard, and A. Jaffri, "Rkbexplorer. com: a knowledge driven infrastructure for linked data providers," in *European Semantic Web Conference*. Springer, 2008, pp. 797–801.

[35] *https://github.com/dbpedia/dbpedia/tree/master/tools/DBpediaAsTables*.

[36] "DBpedia as tables, athelete," *http://web.informatik.uni-mannheim.de/DBpediaAsTables/DBpedia-en-2016-04/csv/Athlete.csv.gz*.

[37] "DBpedia as tables, politician," *http://web.informatik.uni-mannheim.de/DBpediaAsTables/DBpedia-en-2016-04/csv/Politician.csv.gz*.

[38] "Parallel k-means data clustering," http://users.eecs.northwestern.edu/~wkliao/Kmeans/.

[39] K. Christodoulou, N. W. Paton, and A. A. Fernandes, "Structure inference for linked data sources using clustering," in *Transactions on Large-Scale Data and Knowledge-Centered Systems XIX*. Springer, 2015, pp. 1–25.

[40] K. Kellou-Menouer and Z. Kedad, "Schema discovery in RDF data sources," in *ER*. Springer, 2015, pp. 481–495.

[41] M. J. Cafarella, D. Suciu, and O. Etzioni, "Navigating extracted data with schema discovery." in *WebDB*, 2007, pp. 1–6.

[42] M. DiScala and D. J. Abadi, "Automatic generation of normalized relational schemas from nested key-value data," in *SIGMOD*, 2016, pp. 295–310.

[43] W. Spoth, B. S. Arab, E. S. Chan, D. Gawlick, A. Ghoneimy, B. Glavic, B. C. Hammerschmidt, O. Kennedy, S. Lee, Z. H. Liu, X. Niu, and Y. Yang, "Adaptive schema databases," in *CIDR*, 2017.

[44] P. Janga and K. C. Davis, "Mapping heterogeneous XML document collections to relational databases," in *ER*. Springer, 2014, pp. 86–99.

[45] M. A. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *TKDE*, vol. 15, no. 6, pp. 1437–1447, 2003.

[46] R. Sheikhpour, M. A. Sarram, S. Gharaghani, and M. A. Z. Chahooki, "A survey on semi-supervised feature selection methods," *Pattern Recognit.*, vol. 64, pp. 141–158, 2017.

[47] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[48] M. Gütlein, E. Frank, M. A. Hall, and A. Karwath, "Large-scale attribute selection using wrappers," in *CIDM*. IEEE, 2009, pp. 332–339.

[49] A. Deshpande, "In situ graph querying and analytics with graphgen: Extended abstract," in *GRADES*, 2018, pp. 2:1–2:2.

[50] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia, "GraphFrames: an integrated API for mixing graph and relational queries," in *GRADES*, 2016, p. 2.

[51] A. Quamar and A. Deshpande, "Nscalespark: Subgraph-centric graph analytics on apache spark," in *NDA*. ACM, 2016.

[52] A. Jindal, S. Madden, M. Castellanos, and M. Hsu, "Graph analytics using vertica relational database," in *BigData*. IEEE Computer Society, 2015, pp. 1191–1200.

[53] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker, "VERTEXICA: your relational friend for graph analytics!" *PVLDB*, vol. 7, no. 13, pp. 1669–1672, 2014.

[54] B. A. Steer, A. Alnaimi, M. A. B. F. G. Lotz, F. Cuadrado, L. M. Vaquero, and J. Varvenne, "Cytosm: Declarative property graph queries without data migration," in *GRADES*, 2017, pp. 4:1–4:6.

[55] J. Fan, A. G. S. Raj, and J. M. Patel, "The case against specialized graph analytics engines," in *CIDR*, 2015.

[56] K. Zhao and J. X. Yu, "All-in-one: Graph processing in RDBMSs revisited," in *SIGMOD*, 2017, pp. 1165–1180.

[57] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "HadoopDB: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[58] M. Zhu and T. Risch, "Querying combined cloud-based and relational databases," in *CSC*, 2011, pp. 330–335.

[59] J. Kepner, W. Arcand, W. Bergeron, N. T. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (D4M) database and computation system," in *ICASSP*. IEEE, 2012, pp. 5349–5352.

[60] J. Duggan, A. J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. B. Zdonik, "The BigDAWG polystore system," *SIGMOD Rec.*, vol. 44, no. 2, pp. 11–16, 2015.

[61] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu, "Demonstration of the Myria big data management service," in *SIGMOD*, 2014, pp. 881–884.

[62] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jiménez-Peris, R. Pau, and J. Pereira, "The CloudMdsQL, multistore system," in *SIGMOD*. ACM, 2016, pp. 2113–2116.

[63] E. Ioannou, W. Nejdl, C. Niederée, and Y. Velegrakis, "On-the-fly entity-aware query processing in the presence of linkage," *PVLDB*, vol. 3, no. 1, pp. 429–438, 2010.

[64] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra, "QDA: A query-driven approach to entity resolution," *TKDE*, vol. 29, no. 2, p. 402–417, Feb. 2017.

[65] L. Zhu, S. Fan, Q. Ma, W. Meng, and H. Liu, "Top-N query processing with real-time entity resolution," in *EECS*, 2017, pp. 236–241.

[66] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov, "QuERy: A framework for integrating entity resolution with query processing," *PVLDB*, vol. 9, no. 3, pp. 120–131, 2015.