Extended report. Language-aware Indexing for Conjunctive Path Queries

Yuya Sasaki
Osaka University, Japan
sasaki@ist.osaka-u.ac.jp

George Fletcher
Eindhoven University of Technology, Netherlands
g.h.l.fletcher@tue.nl

Onizuka Makoto
Osaka University, Japan
onizuka@ist.osaka-u.ac.jp

Abstract*—Conjunctive path queries (*CPQ*) are one of the most frequently used queries for complex graph analysis. However, current graph indexes are not tailored to fully support the power of query languages to express *CPQ*s. Consequently, current methods do not take advantage of significant pruning opportunities during CPQ evaluation, resulting in poor query processing performance. We propose the *CPQ*-aware path index CPQx, the first path index tailored to the expressivity of *CPQ*. CPQx is built on the partition of the set of source-target vertex pairs of paths in a graph based on the structural notion of *path-bisimulation.* Path-bisimulation is an equivalence relation on paths such that each partition block induced by the relation consists of paths in the graph indistinguishable with respect to *CPQ*s. This language-aware partitioning of the graph can significantly reduce the cost of query evaluation. We present methods to support the full index life cycle: index construction, maintenance, and query processing with our index. We also develop *interest-aware* CPQx to reduce index size and index construction overhead while accelerating query evaluation for queries of interest. We demonstrate through extensive experiments on 14 real graphs that our methods accelerate query processing by up to multiple orders of magnitude over the state-of-the-art methods, with smaller index sizes. Our complete C++ codebase is available as open source for further research.

***Index Terms*—Graph databases, Index, bisimulation**

I. Introduction

Graph data collections are increasingly ubiquitous in many application scenarios where the focus is on analysis of entities and the relationships between them [7], [34]. Example scenarios include knowledge graphs, social networks, biological and chemical databases, and bibliographical databases. Edge labels in these graphs indicate the semantics of relationships. For example, Figure 1 shows a social media network $\mathcal{G}_{ex}$ of twelve users (Sue, Tim, ...) and two blogs (123 and 987). Edges labeled "follows" (abbreviated as f) and "visits" (abbreviated as v) denote follows of people and visiting blogs, respectively.

Analytics on path and graph patterns is fundamental in applications of complex graphs. The *Conjunctive Path Queries* (*CPQ*) are a basic graph query language for finding source-target vertex pairs of paths on graphs, which supports path navigation patterns, cyclic path patterns, and conjunctions of patterns [7]. *CPQ* is defined by recursive expressions of edge labels, identity, join, and conjunction (see Sec. III for details). For example in Figure 1, the conjunction of ff and f^{-1} (where f^{-1} means navigating an edge labeled with f from its target to source) indicates a query to find people and their followers who are in a triad (i.e., a cycle of length three) [22]. The answer of the query is $\{(sue, zoe), (joe, sue), (zoe, joe)\}$.

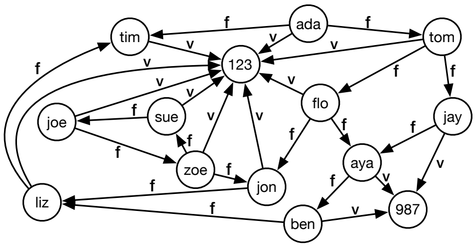


Fig. 1: A graph $\mathcal{G}_{ex}$ with edge labels $\mathcal{L} = \{\mathsf{f}, \mathsf{v}\}$

In recent analyses of real query logs it was discovered that *CPQ* covers more than 99% of query shapes appearing in practice [8], [9].[1] The *CPQ*s form a basic backbone of queries expressed in practical query languages such as SPARQL and Cypher used in contemporary graph analytics systems [7]. In general, many graph applications require subgraph structure matching, e.g., analytics on motifs [28], [41].

As graphs grow in size, graph database systems struggle to support efficient query evaluation [4], [34]. *Path indexing* is a general approach to accelerate graph queries, which essentially materializes the sets of source-target vertex pairs of paths in a graph associated with given label sequences [14], [15], [23], [29], [35]. We here revisit path indexing from the viewpoint of *language-awareness*, which targets a specific query language. Language-aware path indexes[2] support path navigation pattern evaluation while additionally leveraging language-specific structural filtering in index design to further accelerate a targeted query language. In particular, language-aware methods first partition the set of paths of a graph into *equivalence classes*, where the paths of the same class cannot be distinguished by any query in the given language, and then build an index on the set of equivalence classes. Query processing with language-aware indexes aggressively prunes out irrelevant paths, leading to significant (up to multiple orders of magnitude) speed-up in query evaluation over language-*unaware* indexes (i.e., path indexes which do not take advantage of the language-induced equivalence classes).

[1] *CPQ* can express all query pattern structures having treewidth no larger than 2 [8].

[2] Previous studies (e.g., [7]) refer to language-aware indexes as "structural indexes." However, other studies (e.g., [20]) used this terminology differently, which may lead to confusion. Hence, we do not use this terminology here.

Example (*impact of language-awareness*). To illustrate the potential benefits of a language-aware path index, we give an example where such an index would provide an order of magnitude decrease in the cost of query evaluation. Consider the conjunction of $\mathtt{ff}$ and $\mathtt{f}^{-1}$ on the graph $\mathcal{G}_{ex}$ in Figure 1. The sets of source-target vertex pairs that are connected at most 2 length are disjointly partitioned into 30 classes (see Figure 3). Among the classes, only a single class corresponds to the result of conjunction of $\mathtt{ff}$ and $\mathtt{f}^{-1}$ (the class with $c = 7$ in Figure 3). Language-aware indexing has a potential to significantly reduce the search spaces if we can search for such classes efficiently. Indeed, as we will see in our experimental study, *CPQ*-aware indexing accelerates query processing time by up to multiple orders of magnitude.

Research challenges. As a backbone of graph queries in practice, *CPQ* is an excellent candidate language for language-aware indexing for accelerating evaluation of contemporary graph queries. Unfortunately, prior language-aware path indexing methods do not support correct full processing of *CPQ*; see the detailed discussion in Sec. II. Consequently, *there are no known CPQ-aware path indexes.*

How do we practically realize language-aware indexing for *CPQ*? It has been shown that the structural notion of *path-bisimulation* is tightly coupled to the expressive power of *CPQ*. Path-bisimulation is an equivalence relation on paths in a graph, defined completely in terms of the structure of the graph. Critically important for index design, for each equivalence class induced by path-bisimulation, for every query $q \in CPQ$, either all paths or no paths in the equivalence class appear in the evaluation of q on the graph [13]. To date, however, *there has been no study of the practical usability of path-bisimulation in the design of indexing for CPQ.*

It is not immediately evident that path-bisimulation can even be used for *CPQ*-aware indexing in practice. Indeed, path-bisimulation has never been applied to graph indexing. This raises the main research question we investigate in this paper: *Can indexing based on path-bisimulation help to practically accelerate the evaluation of CPQ?*

Towards addressing our research question, several challenges must be overcome. The formal characterization of the expressive power of *CPQ* must be put into practice with new index structures and query processing algorithms, which are non-trivial design challenges. It is not immediately evident that indexing on path-bisimulation can be succinctly represented, maintained, and effectively indexed for real graphs. The naive index design based on path-bismulation is that the index stores all correspondences from any *CPQ*s to equivalence classes, which is not practical in terms of the index size and maintainability. Furthermore, there are no off-the-shelf algorithms to efficiently compute path-bisimulation, as current practical methods are designed for partitioning the vertex set of a graph, e.g., [1], [27]. Since prior studies were either theoretical or not applicable to our problem, we must build *new bridges between theory and practice* to realize practical path indexing methods for *CPQ*.

Our contributions. In this paper, we build these bridges with the introduction of the first language-aware path index for *CPQ*, through four contributions.

(1) Index based on path-bisimulation (Sec. IV-B). We propose CPQx, the first *CPQ-aware index*. We design CPQx to find source-target pairs from label sequences involved in given *CPQ*s through the *CPQ*-equivalence classes. Our index is essentially an inverted index with two data structures; one is for finding *CPQ*-equivalence classes from label sequences and the other is for finding source-target pairs from the *CPQ*-equivalence classes. Our design reduces the index size and supports maintainability because CPQx stores correspondences from label sequences to source-target pairs. Our simple and effective index design makes practical the theoretical notion of path-bisimulation. We formally establish that CPQx is never larger than the state-of-the-art language-unaware index [14].

(2) Algorithms for index life cycle (Secs. IV-C–IV-E). We support the full index life cycle by introducing algorithms for efficient index *construction*, efficient index *maintenance*, and for accelerated *query processing* with the index. In particular, our index construction algorithm efficiently computes the *CPQ*-equivalence classes by pruning the computation of path-bisimilar source-target vertex pairs that do not affect to compute the *CPQ*-equivalence classes. We theoretically show that time complexity of index construction and maintenance algorithms are polynomial. Our query processing with CPQx significantly accelerates the evaluation of *CPQ*s by effectively using *CPQ*-equivalence classes. These algorithms contribute to the practicality of our *CPQ*-aware index.

(3) Interest-aware CPQx (Sec. V). As many practical application scenarios are interest-driven (i.e., users have specific navigation patterns for analysis), we develop an *interest-aware* CPQx, called iaCPQx. iaCPQx is built over a new notion *interest-aware path-equivalence*, which more succinctly represents the set of source-target vertex pairs than path-bisimulation. iaCPQx can support to evaluate arbitrary *CPQ*s and accelerate the evaluation of *CPQ*s that involve navigation patterns of given interests. iaCPQx is significantly scalable compared with CPQx because it reduces both index construction time and memory utilization, and leads to further acceleration of query processing.

(4) Extensive experimental study (Sec. VI). We demonstrate through an experimental study using 14 graphs that our indexes are maintainable and can accelerate query processing by up to *three orders of magnitude* over the state-of-the-art methods, with smaller index sizes. Our *complete C++ codebase* is provided as open source.[3]

Through our four contributions provide a positive answer to our main research question: *CPQ-aware path indexing shows clear promise for providing practical help to significantly accelerate CPQ query processing.*

II. Related Work

The study of graph querying is an active topic. Angles et al. [2], [3] and Bonifati et al. [7] give recent surveys of

[3] https://github.com/yuya-s/CPQ-aware-index

TABLE I: The comparison of language-aware path indexes

Index	Graph model	Query language
DataGuides [16]	Rooted semi-structured graph	RPQ
A[k]-index [24]	Rooted semi-structured graph	RPQ
T-index [29]	Rooted semi-structured graph	RPQ
P(k)-index [15]	Tree	XPath
Our index	Complex graph	CPQ

the current graph query language design landscape. Current practical languages such as SPARQL, Cypher, PGQL, and GSQL are based on two complementary functionalities: the ability to specify complex path patterns (e.g., find all pairs of people connected by a path using only "friendOf" edges) and the ability to specify complex graph patterns (e.g., find all pairs of people who have a friend and a relative in common). The respective underlying formal query languages for these functionalities are the Regular Path Queries (*RPQ*) and the Conjunctive Graph Queries (*CQ*, also known as "basic graph patterns" or as "subgraph patterns"), which are complementary in expressive power [7]. *CPQ* is an expressive subset of *CQ*.

RPQ and *CQ* require fundamentally different indexing and processing methods because they support fundamentally different operations: *RPQ* does not support conjunctions of paths and cyclic patterns; and, *CQ* and *CPQ* do not support disjunctive patterns and Kleene star (i.e., transitive closure). We focus on *CPQ* for index design since (1) constructing *CQ* equivalence classes for *CQ*-aware indexes has impractical exponential cost [33] (even if restricted to paths), unlike *CPQ*-aware indexes which have guaranteed polynomial cost, (2) every *CQ* can be evaluated in terms of its *CPQ* sub-queries, and (3) *CPQ* covers more than 99% of query shapes appearing in practice [8], [9], even though *CPQ* is a subset of *CQ*. An important topic beyond the scope of this paper is to study practical indexing for supporting richer languages such as the *Conjunctive Regular Path Queries* [7].

A rich literature exists on path indexing [14], [15], [17], [20], [23], [29], [35], [40]. As highlighted Section 1, there are two major approaches to path indexing: query language *unaware* and query language *aware* (see also Section III-C for formal definitions). Language-unaware path indexes are designed for supporting path patterns [14], [35]. While effective for accelerating simple graph path queries, they are not sufficient to accelerate complex path queries arising in practice, as they do not leverage the richer topological structure of graphs exposed by languages such as *CPQ*, as heavily used in real query workloads [9]. Examples of language-unaware indexes include Cooper et al. [10] who proposed a trie tree-based path index, and Baolin and Bo [5] who proposed an RDF triple-based path index. The state-of-the-art language-unaware path index is an inverted index for label sequences proposed by Fletcher et al. [14].

Language-aware path indexing, which does leverage richer graph structures, has been developed in the context of semi-structured, XML, and RDF data [15], [23], [29], [32], [40]. All prior work on language-aware path indexing, however, has focused on data models and/or query languages incomparable with *CPQ*, e.g., DataGuides [16], A[k]-index [24], and T-index [29] for *RPQ* on rooted semi-structured graphs, and the P(k)-index [15] for XPath queries on trees. We summarize the data models and query languages of existing language-aware path indexes and our proposal in Table I. These language-aware path indexes are not applicable to *CPQ* for the following three reasons. First, existing indexes are for rooted semi-structured graphs and trees, and thus their construction methods are not applicable to arbitrary graphs. Indeed, it is well known that reasoning about bisimulation structures on trees is much cheaper than on arbitrary graphs with cycles [21]. Second, *RPQ* and XPath do not support cyclic query patterns and/or conjunction of paths, so their structural characterizations are inapplicable to *CPQ*. Third, all indexes except for T-index and P(k)-index are *vertex-based* in the sense that the indexes are built over partitions of the set of vertices in the graph. Vertex-based indexes do not support general path queries because path queries require reasoning over the start and end vertices of paths. For example, the A[k]-index partitions the set of vertices based on the structural notions of vertex-bisimulation on rooted semi-structured graphs, which does not correspond to the semantics of *CPQ* on arbitrary cyclic graph structures. In summary, to the best of our knowledge no earlier indexes can be adapted for *CPQ*-aware path indexing on complex graphs, and furthermore it does not follow from prior work that path-bisimulation-based indexes can be effectively constructed and used in practice.

Methods for computing bisimulation equivalence typically focus on partitioning the *vertex set* of a graph [1], [27]. We propose here a practical method for partitioning the *set of paths* in a graph, which is novel in the literature.

Methods developed for exact subgraph pattern matching (i.e., *CQ* evaluation) can be used to process *CPQ*. Here, matching has been studied mainly under two matching semantics: isomorphic and homomorphic. Systems for isomorphic subgraph matching, e.g., [12], [18], [19], [26], are not suitable for *CPQ* which has homomorphic matching semantics (see Section 3.2). Isomorphic subgraph matching methods can return incorrect results when processing *CPQ*. Systems for homomorphic subgraph matching including RDF engines such as RDF-3X [30], Virtuoso [11], and Tentris [6], and subgraph matching algorithms such as TurboHom++ [25] are applicable to process *CPQ*. To the best of our knowledge, TurboHom++ and Tentris are the state-of-the-art algorithm and RDF engine for a homomorphic subgraph matching, respectively. We compare our methods with TurboHom++ and Tentris in our experimental study.

III. Preliminaries

We study the evaluation of conjunctive path queries on directed edge-labeled graphs using path-based index data structures. In this section we define these concepts.

A. Graphs, paths, and label sequences

A *graph* is a triple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ where $\mathcal{V}$ is a finite set of *vertices* and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V} \times \mathcal{L}$ is a set of labeled directed *edges*, i.e., $(v, u, \ell) \in \mathcal{E}$ denotes an edge from head vertex v

to tail vertex u with label $\ell \in \mathcal{L}$. $\mathcal{L}$ is a finite non-empty set of labels.[4] To support traversals in the inverse direction of edges, we extend $\mathcal{L}$ with ℓ^{-1} for $\ell \in \mathcal{L}$ and $\mathcal{E}$ with (u, v, ℓ^{-1}) for $(v, u, \ell) \in \mathcal{E}$.

We refer to pairs of vertices $(v, u) \in \mathcal{V} \times \mathcal{V}$ as *source-target vertex pairs*, where v and u are the sources and targets, respectively. We define $\mathcal{P}^{\leq k}$, for $k \geq 0$, to be the set of all those source-target pairs such that there is a path of length at most k in $\mathcal{G}$ from the source of the path to its target. We note that $\mathcal{P}^{\leq k} \subseteq \mathcal{V} \times \mathcal{V}$ for any k. In the sequel, we call source-target vertex pairs as *s-t pairs*.

For a non-negative integer k, a *label sequence of length* k is a sequence of k elements from $\mathcal{L}$ (including the inverse of labels). We denote the set of all label sequences of length at most k by $\mathcal{L}^{\leq k}$ and a label sequence in $\mathcal{L}^{\leq k}$ by $\overline{\ell} = \langle \ell_1, \ldots, \ell_j \rangle$ (where $j \leq k$). Further, we denote by $\mathcal{L}^{\leq k}(v, u)$ the set of all those elements $\overline{\ell}$ of $\mathcal{L}^{\leq k}$ such that $\overline{\ell}$ is the sequence of edge labels along a path from v to u in $\mathcal{G}$. We define γ as the average size of $\mathcal{L}^{\leq k}(v, u)$, over all s-t pairs $(v, u) \in \mathcal{P}^{\leq k}$.

Example 3.1: For $\mathcal{G}_{ex}$ of Figure 1, $\mathcal{P}^{\leq 2}$ includes, for example, (ada, ada) and (joe, sue). $\mathcal{L}^{\leq 2}(ada, ada)$ and $\mathcal{L}^{\leq 2}(joe, sue)$ include $\{\langle \mathsf{f}, \mathsf{f}^{-1} \rangle, \langle \mathsf{v}, \mathsf{v}^{-1} \rangle, \langle \mathsf{f}^{-1}, \mathsf{f} \rangle\}$ and $\{\langle \mathsf{f}^{-1} \rangle, \langle \mathsf{f}, \mathsf{f} \rangle, \langle \mathsf{v}, \mathsf{v}^{-1} \rangle\}$, respectively. □

B. Conjunctive path queries

We express conjunctive path queries algebraically. *Conjunctive path query* (*CPQ*) expressions are all and only those built recursively from the nullary operations of identity '*id*' and edge labels 'ℓ', using the binary operations of join '$\circ$' and conjunction '$\cap$'. We have the following grammar for CPQ expressions (for $\ell \in \mathcal{L}$):

$$CPQ ::= id \mid \ell \mid CPQ \circ CPQ \mid CPQ \cap CPQ \mid (CPQ).$$

Let $q \in CPQ$. Given graph $\mathcal{G}$, the semantics $[\![q]\!]_{\mathcal{G}}$ of evaluating q on $\mathcal{G}$ is defined recursively on the structure of q, as follows:

$$\begin{aligned}
[\![id]\!]_{\mathcal{G}} &= \{(v, v) \mid v \in \mathcal{V}\}, \\
[\![\ell]\!]_{\mathcal{G}} &= \{(v, u) \mid (v, u, \ell) \in \mathcal{E}\}, \\
[\![q_1 \circ q_2]\!]_{\mathcal{G}} &= \{(v, u) \mid \exists m \in \mathcal{V} : (v, m) \in [\![q_1]\!]_{\mathcal{G}} \\
&\qquad\qquad\qquad \text{and } (m, u) \in [\![q_2]\!]_{\mathcal{G}}\}, \\
[\![q_1 \cap q_2]\!]_{\mathcal{G}} &= \{(v, u) \mid (v, u) \in [\![q_1]\!]_{\mathcal{G}} \text{ and } (v, u) \in [\![q_2]\!]_{\mathcal{G}}\}, \\
[\![(q_1)]\!]_{\mathcal{G}} &= [\![q_1]\!]_{\mathcal{G}}.
\end{aligned}$$

Note that the output of a *CPQ* is always a set of s-t pairs in $\mathcal{G}$.

Figure 2 illustrates a visual representation of a *CPQ* query, where s and t denote the source and target vertices, resp., of paths in the query results (in this case, they are the same vertex, due to conjunction with identity). Essentially, evaluating a *CPQ* amounts to finding all embeddings of the pattern specified by the query into the graph. Note that *CPQ* has homomorphic pattern embedding semantics, as practical graph query languages such as SPARQL, G-CORE, and SQL/GQL use homomorphic semantics. *CPQ* does not support conjunctions of the same label sequences.

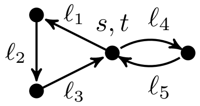


Fig. 2: Visual representation of the query $[(\ell_1 \circ \ell_2 \circ \ell_3) \cap (\ell_4 \circ \ell_5)] \cap id$, with output (s, t).

For an expression $q \in CPQ$, we define the *diameter* $\mathrm{dia}(q)$ of q. Intuitively, the diameter of an expression is the maximum number edge labels to which the join operation is applied. We compute query diameter as follow. The identity operation has diameter zero; every edge label has diameter one; $\mathrm{dia}(q_1 \cap q_2) = \max(\mathrm{dia}(q_1), \mathrm{dia}(q_2))$; and, $\mathrm{dia}(q_1 \circ q_2) = \mathrm{dia}(q_1) + \mathrm{dia}(q_2)$. For non-negative integer k, we denote by *CPQ*$_k$ the set of all expressions in *CPQ* of diameter at most k.

C. Language-aware and -unaware path index

Language-aware indexing is a general methodology which leverages language-specific structural filtering in index design [7], [15], [24]. Given a query language L, the basic idea of L-aware path indexing is to partition the set of paths in a graph $\mathcal{G}$ into L-equivalence classes and builds an index on the set of equivalence classes. Here, an L-equivalence class is a set of paths in $\mathcal{G}$ which cannot be distinguished by any query in L, i.e., for every query $q \in L$, either all paths in the class appear in the evaluation of q on $\mathcal{G}$, or none of the paths appear; for evaluating L the paths in the class can be processed together, instead of individually, leading to accelerated query evaluation.

Language-*un*aware path indexes do not leverage L-equivalence classes. The major drawbacks of language-unaware path indexes, relative to our *CPQ*-aware path index presented in Section IV, are (1) the failure to capture cyclic and conjunctive path structures, so they cannot efficiently evaluate queries with cycles and conjunctions and (2) storing the same paths multiple times in the index, leading to increased index size. The state-of-the-art language-*unaware* path index [14] is an inverted index that outputs a set of paths corresponding to a given label sequence as a search key. More precisely, given $\overline{\ell} = \langle \ell_1, \ldots, \ell_j \rangle \in \mathcal{L}^{\leq k}$ for some $j \leq k$, the language-unaware path index retrieves all paths associated with the label sequence. The size of the path index [14] is $O(\gamma|\mathcal{P}^{\leq k}|)$ because each path is stored γ times on average.

IV. CPQ-Aware Path Index

In this section, we present (1) our *CPQ*-aware path index, CPQx, (2) an algorithm for constructing CPQx, (3) an algorithm for efficient query processing with the index, and (4) a method for effective maintenance of the index under graph updates.

[4] For simplicity we do not consider vertex labels. Extending our methods to accommodate labels on vertices is straightforward.

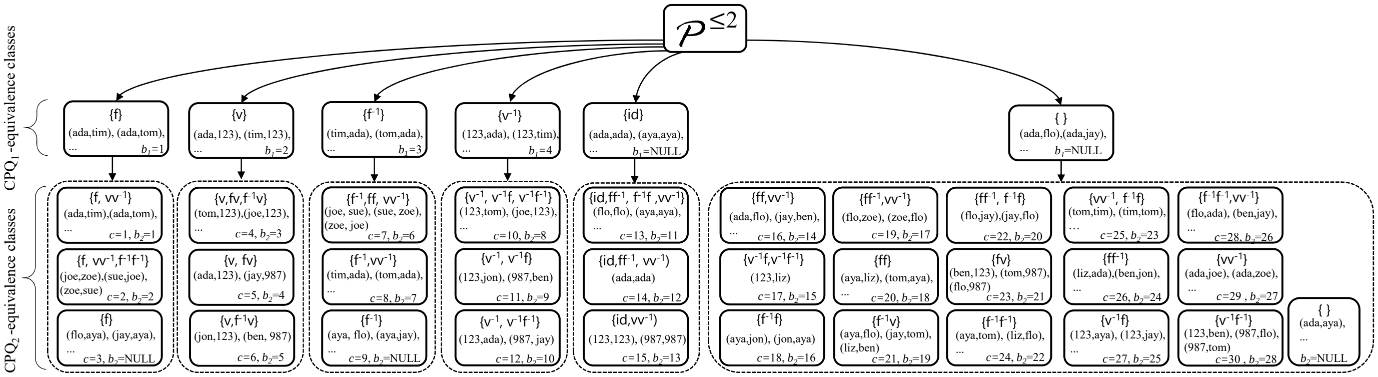

$\mathcal{P}^{\leq 2}$
CPQ$_1$ -equivalence classes
CPQ$_2$ -equivalence classes
{f} (ada,tim), (ada,tom), ... b_1=1
{v} (ada,123), (tim,123), ... b_1=2
{f^{-1}} (tim,ada), (tom,ada), ... b_1=3
{v^{-1}} (123,ada), (123,tim), ... b_1=4
{id} (ada,ada), (aya,aya), ... b_1=NULL
{ } (ada,flo),(ada,jay), ... b_1=NULL
{f, vv^{-1}} (ada,tim),(ada,tom), ... c=1, b_2=1
{f, vv^{-1},f^{-1}f^{-1}} (joe,zoe),(sue,joe), (zoe,sue) c=2, b_2=2
{f} (flo,aya), (jay,aya), ... c=3, b_2=NULL
{v,fv,f^{-1}v} (tom,123),(joe,123), ... c=4, b_2=3
{v, fv} (ada,123), (jay,987) c=5, b_2=4
{v,f^{-1}v} (jon,123), (ben, 987) c=6, b_2=5
{f^{-1},ff, vv^{-1}} (joe, sue), (sue, zoe), (zoe, joe) c=7, b_2=6
{f^{-1},vv^{-1}} (tim,ada), (tom,ada), ... c=8, b_2=7
{f^{-1}} (aya, flo), (aya,jay), ... c=9, b_2=NULL
{v^{-1}, v^{-1}f, v^{-1}f^{-1}} (123,tom), (joe,123), ... c=10, b_2=8
{v^{-1}, v^{-1}f} (123,jon), (987,ben) c=11, b_2=9
{v^{-1}, v^{-1}f^{-1}} (123,ada), (987, jay) c=12, b_2=10
{id,ff^{-1}, f^{-1}f ,vv^{-1}} (flo,flo), (aya,aya), ... c=13, b_2=11
{id,ff^{-1}, vv^{-1}) (ada,ada) c=14, b_2=12
{id,vv^{-1}) (123,123), (987,987) c=15, b_2=13
{ff,vv^{-1}} (ada,flo), (jay,ben), ... c=16, b_2=14
{v^{-1}f,v^{-1}f^{-1}} (123,liz) c=17, b_2=15
{f^{-1}f} (aya,jon), (jon,aya) c=18, b_2=16
{ff^{-1},vv^{-1}} (flo,zoe), (zoe,flo) c=19, b_2=17
{ff} (aya,liz), (tom,aya), ... c=20, b_2=18
{f^{-1}v} (aya,flo), (jay,tom), (liz,ben) c=21, b_2=19
{ff^{-1}, f^{-1}f} (flo,jay),(jay,flo) c=22, b_2=20
{fv} (ben,123), (tom,987), (flo,987) c=23, b_2=21
{f^{-1}f^{-1}} (aya,tom), (liz,flo), ... c=24, b_2=22
{vv^{-1}, f^{-1}f} (tom,tim), (tim,tom), ... c=25, b_2=23
{ff^{-1}} (liz,ada),(ben,jon), ... c=26, b_2=24
{v^{-1}f} (123,aya), (123,jay), ... c=27, b_2=25
{f^{-1}f^{-1},vv^{-1}} (flo,ada), (ben,jay), ... c=28, b_2=26
{vv^{-1}} (ada,joe), (ada,zoe), ... c=29 , b_2=27
{v^{-1}f^{-1}} (123,ben), (987,flo), (987,tom) c=30 , b_2=28
{ } (ada,aya), ... b_2=NULL

Fig. 3: The *CPQ*-aware index of $\mathcal{G}_{ex}$ with $k = 2$, where c and b indicate class and block identifiers, respectively. Each equivalence class is labeled with the set of label sequences $\mathcal{L}^{\leq k}(v, u)$, for an s-t pair (v, u) in the class (note that $\mathcal{L}^{\leq k}(v, u)$ is the same for all s-t pairs of the class, due to Definition 4.2). An arrow from block B to block B' (dashed rectangles, resp.) indicates that B' (blocks within the dashed rectangles, resp.) refines B.

A. Overall idea

The novelty of CPQx is to partition the s-t pairs $\mathcal{P}^{\leq k}$ in a given graph $\mathcal{G}$ into *CPQ*-equivalence classes. This is achieved by partition based on the notion *k-path-bisimulation*, tightly coupled to the expressive power of CPQ_k (see Theorem 4.1, below). k-path-bismulation satisfies the property: if two s-t pairs are k-path-bisimilar, the two s-t pairs are indistinguishable by any $q \in CPQ_k$. This partition brings us two benefits: efficient pruning for CPQs and compact index representations.

CPQx is built over *CPQ*-equivalence classes in two data structures, I_{l2c} and I_{c2p} (Sec. IV-B), which are practically constructed (Sec. IV-C) and maintainable while guaranteeing the correctness of query results (Sec. IV-E). Intuitively, I_{l2c} is a map from label sequences to *CPQ*-equivalence classes and I_{c2p} is a map from *CPQ*-equivalence classes to s-t pairs. The index is used to evaluate queries in two stages (Sec. IV-D). In the first stage, the query is processed over the set of *CPQ*-equivalence classes. This stage allows us to filter out paths which will not contribute to the query result. In the second stage, standard query processing proceeds on the s-t pairs contained in the *CPQ*-equivalence classes. The main advantage of our new query processing algorithm is that we compare *CPQ*-equivalence classes while computing conjunctions of path queries instead of comparing paths themselves, resulting in significantly accelerated *CPQ* evaluation.

B. Index definition

CPQx is based on *CPQ*-equivalence class under the notion of *k-path-bisimulation*. We choose this notion as it captures precisely the expressive power of CPQ_k. Intuitively, s-t pairs (v, u) and (x, y) are k-path-bisimilar when all steps along any paths in the graph of length at most k from v to u and from x to y can be performed in unison, every move along the way in one of the paths being mimicable in the other.

Definition 4.1 (k-path-bisimulation): Let $\mathcal{G}$ be a graph, k be a non-negative integer, and $v, u, x, y \in \mathcal{V}$. The s-t pairs (v, u) and (x, y) are *k-path-bisimilar*, denoted $(v, u) \approx_k (x, y)$, if and only if

1) $v = u$ if and only if $x = y$; (i.e., cycle or not)
2) if $k > 0$, then for each $\ell \in \mathcal{L}$,
   a) if $(v, u, \ell) \in \mathcal{E}$, then $(x, y, \ell) \in \mathcal{E}$; and, if $(u, v, \ell) \in \mathcal{E}$, then $(y, x, \ell) \in \mathcal{E}$;
   b) if $(x, y, \ell) \in \mathcal{E}$, then $(v, u, \ell) \in \mathcal{E}$; and, if $(y, x, \ell) \in \mathcal{E}$, then $(u, v, \ell) \in \mathcal{E}$; and,
3) if $k > 1$, then
   a) for each $m \in \mathcal{V}$, if (v, m) and (m, u) are in $\mathcal{P}^{\leq k-1}$, then there exists $m' \in \mathcal{V}$ such that (x, m') and (m', y) are in $\mathcal{P}^{\leq k-1}$, and, furthermore, $(v, m) \approx_{k-1} (x, m')$ and $(m, u) \approx_{k-1} (m', y)$;
   b) for each $m \in \mathcal{V}$, if (x, m) and (m, y) are in $\mathcal{P}^{\leq k-1}$, then there exists $m' \in \mathcal{V}$ such that (v, m') and (m', u) are in $\mathcal{P}^{\leq k-1}$, and, furthermore, $(x, m) \approx_{k-1} (v, m')$ and $(m, y) \approx_{k-1} (m', u)$.

k-path-bisimulation is a structural characterization of the expressive power of CPQ_k, in the following sense [13].

Theorem 4.1: Let $\mathcal{G}$ be a graph, k be a non-negative integer, and $v, u, x, y \in \mathcal{V}$. If $(v, u) \approx_k (x, y)$, then for every $q \in CPQ_k$ it holds that $(v, u) \in [\![q]\!]_{\mathcal{G}}$ if and only if $(x, y) \in [\![q]\!]_{\mathcal{G}}$.

Towards leveraging Theorem 4.1 for CPQx design, we define the notion of a CPQ_k-equivalence class based on the k-path-bisimulation. Partitioning $\mathcal{P}^{\leq k}$ into CPQ_i-equivalence classes provides a basic building block of our index.

Definition 4.2: Let $\mathcal{G}$ be a graph, $v, u \in \mathcal{V}$, i be a non-negative integer, and $(v, u) \in \mathcal{P}^{\leq i}$. The CPQ_i-equivalence class of (v, u) is the set

$$[(v, u)]_i(\mathcal{G}) = \{(x, y) \mid x, y \in \mathcal{V} \text{ and } (v, u) \approx_i (x, y)\}.$$

We denote equivalence class blocks by B and a set of blocks by $\mathbb{B}$. We define $\mathbb{B}_i(\mathcal{G}) = \{[(v, u)]_i(\mathcal{G}) \mid v, u \in \mathcal{V}\}$.

As a corollary of Theorem 4.1, we have that query processing is tightly coupled to $\mathbb{B}_k(\mathcal{G})$.

Corollary 4.1: Let $\mathcal{G}$ be a graph, k be a non-negative integer, and $q \in CPQ_k$. There exists $\mathbb{B} \subseteq \mathbb{B}_k(\mathcal{G})$ such that $[\![q]\!]_{\mathcal{G}} = \bigcup_{B \in \mathbb{B}} B$.

s-t pairs are disjointly partitioned into blocks in $\mathbb{B}_k$ based on the notion of k-path-bisimulation. Towards leveraging Corollary 4.1 for query processing, we assign a *class identifier*

c to each block in $\mathbb{B}_k(\mathcal{G})$, and we define $\mathcal{C}$ as the set of class identifiers. We also define $\mathcal{P}(c) \subseteq \mathcal{P}^{\leq k}$ and $\mathcal{C}(\bar{\ell}) \subseteq \mathcal{C}$ as the set of s-t pairs that belong to c, and the set of class identifiers that belong to $\bar{\ell}$ (i.e., those equivalence classes whose elements are in the evaluation result of $\bar{\ell}$ interpreted as a query), respectively. We can now define CPQx based on the *CPQ*$_k$-equivalence class of $\mathcal{G}$.

Definition 4.3 (CPQ-aware Index CPQx*):* Given $\mathbb{B}_k(\mathcal{G})$, CPQx I_k is a pair of data structures I_{l2c} and I_{c2p} such that I_{l2c} maps label sequences in $\mathcal{L}^{\leq k}$ to sets of class identifiers and I_{c2p} maps class identifiers to sets of s-t pairs in $\mathcal{P}^{\leq k}$, as follows:

$$\begin{aligned} I_{l2c}(\bar{\ell}) &= \{c \mid c \in \mathcal{C}(\bar{\ell})\}, \\ I_{c2p}(c) &= \{(v,u) \mid (v,u) \in \mathcal{P}(c)\}. \end{aligned}$$

CPQx is essentially an inverted index to find the set of s-t pairs associated with given label sequences through class identifiers. We note that s-t pairs are not stored in CPQx if they are not connected by path with at most k. It enables us to efficiently find the set of s-t pairs that satisfy *CPQ*s thanks to the *CPQ*$_k$-equivalence classes.

Theorem 4.2: The size of CPQx is $O(\gamma|\mathcal{C}| + |\mathcal{P}^{\leq k}|)$.

Proof: I_{l2h} stores the set of class identifiers associated with each label sequence. Each class identifier appears on average γ times in I_{l2h}. Thus, the size of I_{l2h} is $O(\gamma|\mathcal{C}|)$. In I_{h2p}, since each path is stored as single entry, the size of I_{h2p} is $O(|\mathcal{P}^{\leq k}|)$. Therefore, the size of CPQx is $O(\gamma|\mathcal{C}| + |\mathcal{P}^{\leq k}|)$. □

The size of CPQx $O(\gamma|\mathcal{C}|+|\mathcal{P}^{\leq k}|)$ is not larger than that of language-unaware path index [14] $O(\gamma|\mathcal{P}^{\leq k}|)$ because $|\mathcal{C}|$ is at most $|\mathcal{P}^{\leq k}|$. Each s-t pair in CPQx is associated with a single class, while each s-t pair in state-of-the-art path indexes can be associated with multiple label sequences, as we observed in Section III-C. CPQx achieves smaller size than the state-of-the-art language-unaware path index.

Example 4.1: Figure 3 shows *CPQ*$_k$-equivalent classes of $\mathcal{G}_{ex}$ of Figure 1 for $k = 2$. The first and second rows of Figure 3 depict $\mathbb{B}_1(\mathcal{G}_{ex})$ and $\mathbb{B}_2(\mathcal{G}_{ex})$, respectively.

Suppose that we find s-t pairs of paths labeled with both f^{-1} and ff. In CPQx, $I_{l2c}(\mathsf{f}^{-1})$ and $I_{l2c}(\mathsf{ff})$ return $\{7,8,9\}$ and $\{7,16,20\}$, respectively. From these results, we can see that the s-t pairs of paths labeled with $\{\mathsf{f}^{-1},\mathsf{ff}\}$ belongs only to 7. We do not need to check s-t pairs in other blocks. □

C. Index construction

We describe an efficient construction method for CPQx defined in Section IV-B. The main contribution here is that we develop an efficient algorithm to compute *CPQ*$_k$-equivalence class whose time and space complexity are polynomial for our index design. After computing the *CPQ*$_k$-equivalence class, we construct $I_k = (I_{l2c}, I_{c2p})$.

We use a bottom-up approach for computing *CPQ*$_k$-equivalence class. We here note that *CPQ*$_k$-equivalence class in our index construction is not exactly same as the original definition. Our index does not need to distinguish paths with conjunctions divided at different locations if two paths are merged at targets. The bottom-up approach is suitable for computing *CPQ*$_k$-equivalence class in our index design.

A straightforward algorithm is that (1) for computing 1-path-bisimulation, it checks all s-t pairs if they are connected edges with the same labels and cyclic patterns (i.e., self-loop), (2) it obtains pairs of paths that are connected 2 length path and computes 2-path-bisimulation, and (3) it repeats similar process of the second one until obtaining k-path-bisimulation. However, this is inefficient because we need to compute i-path bisimulation for all s-t pairs.

The idea of our algorithm leverages the characteristics of bisimilar paths, which (1) paths uniquely belong to blocks in $\mathbb{B}_i$ and (2) i-path-bisimilation refines $(i-1)$-path-bisimilation. We assign a *block identifier* $b_i(v,u)$ to each block $[(v,u)]_i$ and (v,u) of $\mathbb{B}_i$. Each s-t pair $(v,u) \in \mathcal{P}^{\leq k}$ has an associated sequence of k block identifiers $\langle b_1(v,u),\ldots,b_k(v,u)\rangle$. It is easy to establish that k-path-bisimilar s-t pairs are uniquely identified by their common sequences.

Based on the above idea, our algorithm effectively skips both computing i-path-bisimulation and assigning block identifiers to the s-t pairs that have no paths. For obtaining block identifiers of $\mathbb{B}_i$, we join two s-t pairs in $\mathbb{B}_{i-1}$ and $\mathbb{B}_1$. We can assign the same block identifiers to s-t pairs in $\mathbb{B}_i$ if (1) joined s-t pairs have the same block identifiers in $\mathbb{B}_{i-1}$ and $\mathbb{B}_1$ and (2) both s-t pairs are cycle or not. Furthermore, we can skip assigning block identifiers to blocks in $\mathbb{B}_i$ if the s-t pairs in $\mathbb{B}_i$ are not connected at i length paths because k-path-bisimilar s-t pairs are uniquely identified even if $b_i(v,u) = \mathit{Null}$.

After computing the *CPQ*$_k$-equivalence class, CPQx is constructed in a simple way. It generates class identifiers from sequences of block identifiers, and then insert a pair of $\bar{\ell}$ and $c \in \mathcal{C}(\bar{\ell})$ into I_{l2c} and a pair of class identifier c and $(v,u) \in \mathcal{P}(c)$ into I_{c2p}. Note that the set of source-target paths with the same class identifier has the same label sequence due to the definition of k-path-bisimulation.

Algorithms 1 and 2 show pseudo-code for computing the *CPQ*$_k$-equivalence class and constructing CPQx, respectively. In Algorithm 1, we sort elements of $\mathbb{S}^i$ so that i-path-bisimilar s-t pairs are sequentially listed for efficiently assigning the block identifiers. In Algorithm 2, we generate class identifier c of (v,u) by using a hash function for each class $\langle b_1(v,u),\ldots,b_k(v,u)\rangle$. If two s-t pairs have the same class, it assigns the same class identifiers to the two s-t pairs.

We here describe the time and space complexity for constructing CPQx. d indicates the maximum vertex degree.

Theorem 4.3 (Time complexity): Given a graph G and positive number k, the time complexity of index construction is $O(k(d|\mathcal{P}^{\leq k}| + |\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|) + \gamma|\mathcal{C}|\log\gamma|\mathcal{C}|)$.

Proof: The algorithm for constructing CPQx has two steps (1) computing $[\mathcal{G}]_k$ and (2) constructing $I_k = (I_{l2c}, I_{c2p})$. For computing $[\mathcal{G}]_k$, the algorithm enumerates the set of block identifiers for each path, which takes $O(d|\mathcal{P}^{\leq k}|)$. Then, it compares the block identifiers by a sorting algorithm, which takes $O(|\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|)$. Since it repeats k times, it takes $O(k(d|\mathcal{P}^{\leq k}| + |\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|))$. For constructing

Algorithm 1: Computing CPQ_k-equivalence class

```
input : Graph $\mathcal{G}$, natural number $k$
output: Set of blocks $[\mathcal{G}]_k$
procedure cpqPathPartition($\mathcal{G}$, $k$)
$\mathbb{S}^i_{(v,u)} = \emptyset$ for $i = 1, \dots, k$ and
    $\forall(v,u) \in (\mathcal{P}^{\leq i} - \mathcal{P}^{\leq i-1}) \cup (\mathcal{P}^{\leq i} \cap \mathcal{P}^{\leq i-1})$;
for $e = (v, u, \ell) \in \mathcal{E}$ do
    $\mathbb{S}^1_{(v,u)} \leftarrow \mathbb{S}^1_{(v,u)} \cup \{\ell\}$;
Sort $\mathbb{S}^1$ according to edge labels and $(v, u)$;
Set $b_1(v, u)$ for $\forall(v, u) \in \mathcal{P}^{\leq 1}$ as $\mathbb{B}_1$;
for $i = 2, \dots, k$ do
    for $\forall\mathbb{S}^{i-1}_{(v,m)}$ do
        for $\forall\mathbb{S}^1_{(m,u)}$ do
            $\mathbb{S}^i_{(v,u)} \leftarrow \mathbb{S}^i_{(v,u)} \cup \{b_{i-1}(v, m), b_1(m, u)\}$;
    Sort $\mathbb{S}^i$ according to block identifiers and $(v, u)$;
    Set $b_i(v, u)$ for
        $\forall(v, u) \in (\mathcal{P}^{\leq i} - \mathcal{P}^{\leq i-1}) \cup (\mathcal{P}^{\leq i} \cap \mathcal{P}^{\leq i-1})$ as $\mathbb{B}_i$;
    if $i \neq 2$ then Clear $\mathbb{S}^{i-1}$:
return $[\mathcal{G}]_k = \{\mathbb{B}_1, \dots, \mathbb{B}_k\}$;
end procedure
```

Algorithm 2: Construction of CPQx

```
input : Graph $\mathcal{G}$, natural number $k$
output: CPQx $I_k = \{I_{c2p}, I_{l2c}\}$
procedure Construction($\mathcal{G}$, $k$)
$[\mathcal{G}]_k \leftarrow$ cpqPathPartition($\mathcal{G}$, $k$);
for $(v, u) \in \mathcal{P}^{\leq k}$ do
    $c \leftarrow hash(\langle b^1_{v,u}, \dots, b^k_{v,u}\rangle)$;
    if $c$ is Null then
        $c \leftarrow c_{new}$;
        $hash(\langle b^1_{v,u}, \dots, b^k_{v,u}\rangle) \leftarrow c$ ;
        $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$;
        Update $c_{new}$;
    $I_{h2p}.append(c, (v, u))$;
for $c \in \mathcal{C}$ do
    for $(v, u) \in I_{c2p}(c)$ do
        for $\overline{\ell} \in \mathcal{L}^{\leq k}(v, u)$ do
            $I_{l2c}.append(\overline{\ell}, c)$;
sort $(v, u)$ in $I_{c2p}$ and $c$ in $I_{l2c}$;
return $I_k$;
end procedure
```

$I_k = (I_{l2c}, I_{c2p})$, it sorts the set of pairs of label sequences and class identifiers for I_{l2c} and the set of pairs of class identifiers and paths for I_{c2p}. Since these sizes are $O(|\mathcal{P}^{\leq k}|)$ and $O(\gamma|\mathcal{C}|)$, resp., it takes $O(|\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|)$ and $O(\gamma|\mathcal{C}|\log\gamma|\mathcal{C}|)$, resp. Thus, the total time complexity is $O(k(d|\mathcal{P}^{\leq k}| + |\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|) + \gamma|\mathcal{C}|\log\gamma|\mathcal{C}|)$. □

Theorem 4.4 (Space complexity): Given a graph $\mathcal{G}$ and positive number k, the space complexity of index construction is $O((k + d)|\mathcal{P}^{\leq k}| + \gamma|\mathcal{C}|)$.

Proof: The algorithm stores block identifiers (or label sequences) for paths in $(\mathcal{P}^{\leq i} - \mathcal{P}^{\leq i-1}) \cup (\mathcal{P}^{\leq i} \cap \mathcal{P}^{\leq i-1})$ and the number of block identifiers for each path is at most d. Thus, the size of $\mathbb{S}^i$ is $O(d|\mathcal{P}^{\leq k}|)$. Additionally, it stores k sets of block identifiers (i.e., $\mathbb{B}_1, \dots, \mathbb{B}_k$). Since the size of each set is $O(|\mathcal{P}^{\leq k}|)$, the size is totally $O(k|\mathcal{P}^{\leq k}|)$. To store the index, it takes $O(\gamma|\mathcal{C}| + |\mathcal{P}^{\leq k}|)$. Therefore, its space complexity is $O((k + d)|\mathcal{P}^{\leq k}| + \gamma|\mathcal{C}|)$. □

Example 4.2: We explain how to construct CPQx by using Figure 3. Our algorithm first computes the 1-equivalence classes according to edge labels. It assigns block identifiers to blocks with {f}, {v}, $\{\mathsf{f}^{-1}\}$, and $\{\mathsf{v}^{-1}\}$, while it does not assign block identifiers to blocks with {id} and {} because the s-t pairs are not connected one length paths (i.e., edges).

Then, it joins two edges to obtain paths with two length, for instance, it joins $(ada, 123)$ and $(123, tim)$, and then obtains (ada, tim) with vv^{-1}. Similarly, it joins $(ada, 123)$ and $(123, tom)$, and then obtains (ada, tom). Since joined paths of (ada, tim) and (ada, tom) are from the same s-t pairs of blocks (i.e., $(123, tim)$ and $(123, tom)$ belong to the same block), (ada, tim) and (ada, tom) belong to the same block. We do not assign block identifiers to blocks with {f}, $\{\mathsf{f}^{-1}\}$, and {} because the s-t pairs in the block are not connected two length path.

s-t pairs in the same equivalent classes have the same sequences of block identifiers. Even if we do not assign block identifiers to some blocks, we can identify CPQ_2-equivalence classes. For instance, {f} in CPQ$_2$-equivalence class has sequence $\langle 1, Null\rangle$, and other blocks do not have the same sequence.

Our construction algorithm effectively reduces the index construction costs because most s-t pairs have no paths. In such small example of Figure 3, we can skip computing many bisimilar s-t pairs. In this example, the possible number of s-t pairs is 196 and the number of s-t pairs that are connected paths at most two length is 150. As graph sizes increase, the effectiveness of our index construction algorithm increases. □

D. Query processing with CPQx

We accelerate query processing by using CPQx, instead of the original graph. The effective use of classes mitigates the cost of unnecessarily comparing paths which do not participate in the query result.

Our query processing method builds a parse tree according to a given query $q \in CPQ$ and CPQx (see Figure 4 for an example) and evaluates the query following the parse tree. Each node of the parse tree represents a logical operation of *CPQ*: LookUp (i.e., given label sequence $\overline{\ell} \in \mathcal{L}^{\leq k}$, find the corresponding set of class identifiers by I_{l2c}), Conjunction, Join, and Identity. Since LookUp nodes depend on the length k of CPQx, we split label sequences longer than k into sub-label sequences whose sizes are at most k. This method derives an execution plan, with index LookUp as leaf nodes. We process starting from the root node of q, recurring on the left and right, as necessary. Further query optimization is an interesting rich topic for future research.

Our query processing method, in particular, accelerates Conjunction and Identity. In the case of Conjunction, we can efficiently compute the conjunction of the two sub queries without directly comparing the corresponding set of

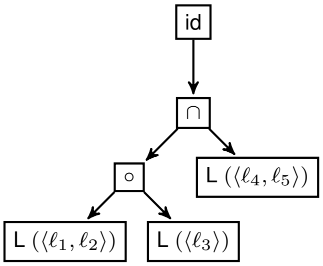


Fig. 4: Parse tree of query $[(\ell_1 \circ \ell_2 \circ \ell_3) \cap (\ell_4 \circ \ell_5)] \cap id$, when $k = 2$. Here, $\ell_1 \circ \ell_2 \circ \ell_3$ is processed left to right, with an index look up $\langle \ell_1, \ell_2 \rangle$ joined with a look up of $\langle \ell_3 \rangle$.

s-t pairs in the graph. We compare class identifiers obtained by two sub queries to obtain a set of s-t pairs that satisfy both of the sub queries. The correctness of conjunction evaluation is based on the following proposition:

*Proposition 4.1 (*Conjunction correctness): Given two sets of class identifiers $\mathbb{C}$ and $\mathbb{C}'$, the set of s-t pairs $\{(v, u) \mid \forall c \in \mathbb{C}, c' \in \mathbb{C}' : (v, u) \in \mathcal{P}(c) \text{ and } (v, u) \in \mathcal{P}(c')\}$ is the same as $\{(v, u) \mid \forall c \in \mathbb{C} \cap \mathbb{C}' : (v, u) \in \mathcal{P}(c)\}$.

In the case of Identity, since k-path-bisimilar s-t pairs are partitioned according to their cyclic patterns, we can evaluate Identity by just checking the first s-t pairs in the set of s-t pairs of class identifiers. These processes decrease computation cost significantly because the number of class identifiers $|\mathcal{C}|$ is much smaller than that of paths $|\mathcal{P}^{\leq k}|$ (see Table III in experimental study).

Additionally, we use optimization techniques to reduce computation cost while guaranteeing correctness. First, we use a sorted merge join as a physical operator for Conjunction and Join, as CPQx stores sorted class identifiers and s-t pairs. Second, in Identity since we can optimize $q \circ id = q$, we handle only $q \cap id$ as Identity. Third, Identity is executed with the other three operators to avoid inserting the s-t pairs that are deleted by Identity.

We describe the time complexity of query processing by using CPQx.

Theorem 4.5 (Time complexity): Let assume a given query q consisting of α_1 times of Join and α_2 times of Conjunction. Given graph $\mathcal{G}$ and index I_k, the time complexity of processing q is $O((\alpha_1 + \alpha_2)(\max((d^k)^{\alpha_1}|\mathbb{P}_q|, |V|^2) \log(\max((d^k)^{\alpha_1}|\mathbb{P}_q|, |V|^2)))$ if $\alpha_1 > 0$ or $O(\alpha_2|\mathbb{C}_q|)$ if $\alpha_1 = 0$, where the maximum numbers of paths $|\mathbb{P}_q|$ and class identifiers $|\mathbb{C}_q|$ returned by a single LookUp, and the maximum degree d.

Proof: The query processing consists of two main process; Join and Conjunction. Other operations are ignorable due to small costs. First, in Join, it takes $O(|\mathcal{P}_q| \log |\mathcal{P}_q|)$ for the first Join. The number of paths could increase at most $d^k|\mathcal{P}_q|$ after the first Join. When q consists of α_1 times of Join, the cost of Join is $O(\alpha_1(\max((d^k)^{\alpha_1}|\mathcal{P}_q|, |V|^2) \log \max((d^k)^{\alpha_1}|\mathcal{P}_q|, |V|^2))$. Second, in Conjunction before Join, it takes $O(|\mathbb{C}_q|)$ because it compares the number overlapped class identifiers. The number of class identifiers after Conjunction does not increase, so the costs of Conjunction $O(\alpha_2|\mathbb{C}_q|)$. If Conjunction follows Join, our algorithm sorts paths and check overlapped paths. It takes $O(\alpha_2(\max(d^k)^{\alpha_1}|\mathcal{P}_q|, |V|^2) \log(\max(d^k)^{\alpha_1}|\mathcal{P}_q|, |V|^2))$.

Therefore, the time complexity of processing q is $O((\alpha_1 + \alpha_2)(\max((d^k)^{\alpha_1}|\mathbb{P}_q|, |V|^2) \log(\max((d^k)^{\alpha_1}|\mathbb{P}_q|, |V|^2)))$ if $\alpha_1 > 0$ or $O(\alpha_2|\mathbb{C}_q|)$ if $\alpha_1 = 0$. □

Algorithm 3: Query processing

```
input  : Node on query tree $q$, structural index $I_k$
output: set of paths $\mathbb{P}$, set of classes $\mathbb{C}$
procedure Evaluation($q$, $I_k$)
if operation of $q$ is LookUp then
    return LookUp ($q.\bar{\ell}$, $I_k$);
else if operation of $q$ is LookUP with Identity then
    return LookUpId ($q.\bar{\ell}$, $I_k$);
else
    $\mathbb{P}_l, \mathbb{C}_l \leftarrow$ Evaluation($q_l$, $I_k$);
    $\mathbb{P}_r, \mathbb{C}_r \leftarrow$ Evaluation($q_r$, $I_k$);
    if operation of $q$ is Join then
        return Join ($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$);
    else if operation of $q$ is Conjunction then
        return Conjunction ($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$);
    else if operation of $q$ is Join with Identity then
        return JoinId ($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$);
    else if operation of $q$ is Conjunction with Identity
      then
        return ConjunctionId ($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$);
if $q$ is the root of query tree then
    $\mathbb{P} \leftarrow \mathbb{P} \cup I_{c2p}(c)$ for all $c \in \mathbb{C}$;
return $\mathbb{P}, \mathbb{C}$;
end procedure
```

Algorithms 3 and 4 show pseudocode for our query processing method and operations, respectively. $\mathbb{P}$ and $\mathbb{C}$ denote the sets of s-t pairs and class identifiers that are found during query processing, respectively. Our query processing algorithm repeatedly traverses the nodes on a given query tree.

Example 4.3: Let us consider evaluating $\mathsf{ff} \cap \mathsf{f}^{-1}$. Our algorithm first LookUP for ff and f^{-1}, and finds the sets of class identifiers $\{7, 16, 20\}$ and $\{7, 8, 9\}$, respectively. Then, it takes Conjunction for the two sets, and then obtains $\{7\}$. Finally, we obtain three paths $\{(sue, zoe), (joe, sue), (zoe, joe)\}$ by $I_{c2p}(7)$. Since we do not need to compare any paths for the conjunction operation, the query processing is very fast.

If we use the language-unaware path index, we first find 15 and 15 s-t pairs for ff and f^{-1}, resp., each element of which is a pair of vertex identifiers, i.e., in total, comparisons over 60 vertex identifiers. With CPQx, this would be an intersection of lists of class identifiers of length 3 and 3, resp., i.e., in total, comparisons over 6 class identifiers This 10x reduction demonstrates the significant acceleration of CPQx. □

E. Index maintenance

CPQx is easily updated when the graph is updated. Our update method lazily updates CPQx, while maintaining cor-

Algorithm 4: Operations

```
procedure LookUp($\bar{\ell}$, $I_k$)
return $\emptyset, I_{c2p}(\bar{\ell})$;

procedure Join($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$)
$\mathbb{P}_l \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_l\}$;
$\mathbb{P}_r \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_r\}$;
$\mathbb{P} \leftarrow \{(v,y) \mid (v,u) \in \mathbb{P}_l \land (x,y) \in \mathbb{P}_r \land u = x\}$
return $\mathbb{P}, \emptyset$;

procedure Conjunction($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$)
if $\mathbb{C}_l \neq \emptyset$ and $\mathbb{C}_r \neq \emptyset$ then
    $\mathbb{C} \leftarrow \mathbb{C}_l \cap \mathbb{C}_r$;
    return $\emptyset, \mathbb{C}$;
else
    if $\mathbb{P}_l \neq \emptyset$ then
        $\mathbb{P}_l \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_l\}$;
    if $\mathbb{P}_r \neq \emptyset$ then
        $\mathbb{P}_r \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_r\}$;
    $\mathbb{P} \leftarrow \mathbb{P}_l \cap \mathbb{P}_r$;
    return $\mathbb{P}, \emptyset$;

procedure JoinId($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$)
$\mathbb{P}_l \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_l\}$;
$\mathbb{P}_r \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_r\}$;
$\mathbb{P} \leftarrow \{(v,y) \mid (v,u) \in \mathbb{P}_l \land (x,y) \in \mathbb{P}_r \land u = x \land v = y\}$;
return $\mathbb{P}, \emptyset$;

procedure ConjunctionId($\mathbb{P}_l, \mathbb{P}_r, \mathbb{C}_l, \mathbb{C}_r$, $I_k$)
if $\mathbb{C}_l \neq \emptyset$ and $\mathbb{C}_r \neq \emptyset$ then
    $\mathbb{C} \leftarrow \{c \mid c \in \mathbb{C}_l \land c \in \mathbb{C}_r \land (v,u) \in I_{c2p}(c) \land v = u\}$;
    return $\emptyset, \mathbb{C}$;
else
    if $\mathbb{P}_l \neq \emptyset$ then
        $\mathbb{P}_l \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_l\}$;
    if $\mathbb{P}_r \neq \emptyset$ then
        $\mathbb{P}_r \leftarrow \{(v,u) \mid (v,u) \in I_{c2p}(c) \land c \in \mathbb{C}_r\}$;
    $\mathbb{P} \leftarrow \{(v,u) \mid (v,u) \in \mathbb{P}_l \land (v,u) \in \mathbb{P}_r \land v = u\}$;
    return $\mathbb{P}, \emptyset$;

procedure Identity($\mathbb{P}, \mathbb{C}$, $I_k$)
if $\mathbb{C} \neq \emptyset$ then
    $\mathbb{C}' \leftarrow \{c \mid c \in \mathbb{C} \land (v,u) \in I_{c2p}(c) \land v = u\}$;
    return $\emptyset, \mathbb{C}'$;
else
    $\mathbb{P}' \leftarrow \{(v,u) \mid (v,u) \in \mathbb{P} \land v = u\}$;
    return $\mathbb{P}', \emptyset$;
```

rectness of query evaluation. That is, to reduce update cost, it does not maintain the same index entries with the index that is constructed from scratch. This approach enables efficient index updates with a small deterioration of the index performance.

We proceed as follows. When edges are deleted/inserted, label sequences of paths between s-t pairs change (also paths may disappear/appear) and k-path-bisimilar s-t pairs may become non-bisimilar. If two non-bisimilar s-t pairs are assigned the same class identifier, then query results would be incorrect. Thus, our lazy update method divides the set of paths for all k length paths related to graph update and does not merge two sets of s-t pairs even if they become k-path-bisimilar. Query processing still ensures correct results even if k-path-bisimilar s-t pairs belong to the different class identifiers.

Proposition 4.2 (Update correctness): After edge deletion or edge insertion, query processing in Sec. IV-D ensures correct query results.

We explain how we handle five cases: edge deletion, edge insertion, label change, vertex deletion, and vertex insertion.

Edge deletion. We explain a procedure for edge deletion. We first enumerate all s-t pairs involved in the deleted edge by bread-first search. The label sequences of these s-t pairs may change unless there are alternative paths through same label sequences, so we check whether there are alternative paths. Next, we delete paths from I_{c2p} if the label sequences of the s-t pairs change. Here, for efficiently finding class identifier c' according to the deleted s-t pairs, we use inverted index whose keys are s-t pairs. We then add new $\mathbb{P}(c')$ that includes only the path into I_{c2p} unless their label sequences are empty (i.e., paths disappear). This update does not check whether or not the affected s-t pairs is k-path-bisimilar to other s-t pairs.

Example 4.4: Suppose that we delete $\{(ada, tim)\}$ with f from $\mathcal{G}_{ex}$ in Figure 3. We first list all related s-t pairs with at most 2 length such as $\{(ada, \mathit{123})\}$ and $\{(tom, tim)\}$. Among them, $\{(ada, \mathit{123})\}$ has alternative paths through fv. The other s-t pairs are deleted from corresponding blocks, and then new blocks with new class identifiers are created that include a single s-t pair for all deleted pairs. □

Edge insertion. The procedure is similar to that for edge deletion. The difference is enumerating s-t pairs involving the new inserted edge.

Other graph updates. We can handle the following additional updates by combinations of edge deletion and insertion. For example, in the vertex deletion, we delete all edges that connect to the deleted vertex, and then delete the vertex.

The update cost is much smaller than reconstructing the index from scratch. After update, the set of k-path bisimilar s-t pairs may belong to different class identifiers. We guarantee the correctness of query results even if the set of k-path bisimilar s-t pairs belong to different class identifiers.

Theorem 4.6: The time complexity for edge deletion or insertion is $O(d|\mathcal{P}_u| + |\mathcal{P}_u| \log |\mathcal{P}^{\leq k}| + |\mathcal{C}| \log |\mathcal{C}|)$, where $\mathcal{P}_u$ and d are the set of s-t pairs that are involved updates and the maximum degrees among vertices in $\mathcal{P}_u$, respectively.

Proof: The update method first finds $\mathcal{P}_u$ by bread-first search starting from end vertices of an inserted/deleted edge. This takes $O(d|\mathcal{P}_u|)$. Then, it searches for s-t pairs in $\mathcal{P}_u$ from I_{c2p}, and then it searches for the class identifiers associated with s-t pairs in $\mathcal{P}_u$ from I_{l2c}. Since these data structures are sorted list, paths and class identifiers are found by two binary search, that is $O(|\mathcal{P}_u| \log |\mathcal{P}^{\leq k}| + |\mathcal{C}| \log |\mathcal{C}|)$. Therefore, the time complexity is $O(d|\mathcal{P}_u| + |\mathcal{P}_u| \log |\mathcal{P}^{\leq k}| + |\mathcal{C}| \log |\mathcal{C}|)$. □

V. Interest-Aware Index

In many application scenarios, users are often interested in only a specific set of label sequences, i.e., navigation patterns.

We call the given label sequences *interests*. Users are clearly interested in not all label sequences. CPQx, however, stores all label sequences, including inverse of labels, up to length k, which leads to the scalability problem (i.e., large index construction costs and index size).

Motivated by this, we develop an interest-aware CPQx, namely iaCPQx based on a given set of label sequences. The iaCPQx solves the scalability problem of CPQx. The index supports processing of any CPQ, yet is tailored to especially accelerate processing of all CPQ queries which use any of the label sequences of interest.

A. Index design

Towards an interest-aware CPQx, we propose the notion of *interest-aware path-equivalence* as follows.

Definition 5.1 (Interest-aware Path-Equivalence): Let $\mathcal{G}$ be a graph, $v, u, x, y \in \mathcal{V}$ and $\mathcal{L}_q \subseteq \mathcal{L}^{\leq k}$ be a set of label sequences. The s-t pairs (v, u) and (x, y) are *interest-aware path-equivalent*, denoted $(v, u) \approx_i (x, y)$, if and only if the followings hold:

1) $v = u$ if and only if $x = y$;
2) $\mathcal{L}^{\leq k}(v, u) \cap \mathcal{L}_q = \mathcal{L}^{\leq k}(x, y) \cap \mathcal{L}_q$.

$\mathcal{L}_q$ is the set of interests. When we construct iaCPQx, we always include all sequences of length one (i.e., all edge labels) in $\mathcal{L}_q$. Thus, even queries containing label sequences without users' interests can be still evaluated.

The difference between CPQx and iaCPQx is that the former and the latter assign same class identifiers to the set of k-path bisimilar s-t pairs and the set of interest-aware path-equivalent s-t pairs, respectively. Since interest-aware path-equivalence is weaker than k-path bisimulation (i.e., it is easy to show that $\approx_k$ refines $\approx_i$, when k is at least as large as the length of the longest sequence in $\mathcal{L}_q$), more s-t pairs have the same class identifiers (i.e., partition blocks are bigger). Therefore, the size of iaCPQx is much smaller (and hence faster to use) than that of the basic CPQx, and we can control the size of index by adjusting the size of interests.

Theorem 5.1: The size of iaCPQx is $O\left(\frac{|\mathcal{L}_q|}{|\mathcal{L}^{\leq k}|}\left(\gamma|\mathcal{C}| + |\mathcal{P}^{\leq k}|\right)\right)$.

Proof: The number of paths that are stored in iaCPQx linearly decreases by the size of $|\mathcal{L}_q|$, compared with the size of CPQx. Therefore, the size of iaCPQx is $O\left(\frac{|\mathcal{L}_q|}{|\mathcal{L}^{\leq k}|}\left(\gamma|\mathcal{C}| + |\mathcal{P}^{\leq k}|\right)\right)$. □

B. Index construction and query processing

The index construction and query processing methods are almost the same as those for CPQx. The difference for the construction algorithm is that it enumerates s-t pairs only with given label sequences and two paths have same class identifiers if they are interest-aware path-equivalent. Since the construction of iaCPQx decreases the number of paths, it becomes more efficient than that of CPQx. The difference for query processing is that we divide label sequences into sub-label sequences if the label sequences are not included in the given label sequences.

TABLE II: Dataset overview: $|\mathcal{E}|$ and $|\mathcal{L}|$ include inverse edges and labels, respectively.

Dataset	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\mathcal{L}\rvert$	Real label?
Robots	1,484	5,920	8	✓
ego-Facebook	4,039	176,468	16	
Advogato	5,417	102,654	8	✓
Youtube	15,088	21,452,214	10	✓
StringHS	16,956	2,483,530	14	✓
StringFC	15,515	4,089,600	14	✓
BioGrid	64,332	1,724,554	14	✓
Epnions	131,828	1,681,598	16	
WebGoogle	875,713	10,210,074	16	
WikiTalk	2,394,385	10,042,820	16	
YAGO	4,295,825	24,861,400	74	✓
CitPatents	3,774,768	33,037,896	16	
Wikidata	9,292,714	110,851,582	1054	✓
Freebase	14,420,276	213,225,620	1556	✓
g-Mark-1m	1,006,802	15,925,506	12	
g-Mark-5m	5,005,992	84,994,500	12	
g-Mark-10m	10,005,721	183,748,319	12	
g-Mark-15m	15,003,647	255,538,724	12	
g-Mark-20m	20,004,856	393,797,046	12	

The space and time complexity of constructing iaCPQx are similar to Theorems 4.3 and 4.4, resp. The construction cost of iaCPQx decrease as $|\mathcal{L}_q|$ decreases since the number of paths related to its index construction decreases.

The time complexity of query processing on iaCPQx is the same as that on CPQx. The difference is the numbers of paths and class identifiers by LOOKUP. iaCPQx reduces them, so the query processing becomes fast.

C. Maintenance

iaCPQx is easily updated in a similar fashion for CPQx.

Graph update: The graph update procedures are almost the same as those for CPQx given in Section IV-E. The difference is that we do not process the set of s-t pairs whose label sequences are not included in the given set of label sequences.

Label sequence deletion: When we delete a label sequence from the given set of label sequences, we can just delete the deleted label sequence from I_{l2c}. After deleting the label sequence, two paths may become interest-aware path-equivalent. While we do not merge two sets of paths, we can still guarantee correct query answers in a fashion analogous with Proposition 4.2.

Label sequence insertion: For inserting new label sequences, we insert new s-t pairs to the index. Thus, we first enumerate the set of s-t pairs that have the inserted label sequences, and then take the same procedure as for inserting new edges.

VI. EXPERIMENTAL STUDY

We next present the results of an experimental evaluation of our methods. We designed the experiments to clarify the questions: (1) Does *CPQ*-aware indexing accelerate query processing? (Section VI-A); (2) Are *CPQ*-aware indexes compact? (Section VI-B), (3) Are *CPQ*-aware indexes maintainable? (Section VI-C); and, (4) *Are CPQ-aware indexes well-behaved as k grows?* (Section VI-D).

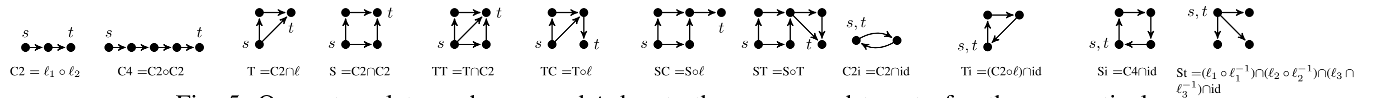


Fig. 5: Query templates, where s and t denote the source and target of paths, respectively.

Experiments were performed on a Linux server with 512GB of memory and an Intel(R) Xeon(R) CPU E5-2699v3 @ 2.30GHz processor. All algorithms are single-threaded.

Datasets. Table II provides an overview of the datasets used in our study consisting of nine datasets with real labels and five datasets without edge labels. The graphs range over several different scenarios, such as social networks, biological networks, and knowledge graphs. These datasets are provided by the authors [31], [38] except for ego-Facebook, webGoogle, WebTalk, CitPatents, and Wikidata. In Wikidata, we extract vertices that represent URI from original Wikidata [39]. ego-Facebook, WebGoogle, WebTalk, and CitPatents are available at SNAP [36]. Since these four graphs have no edge labels, we assign edge labels that are exponentially distributed with $\lambda = 0.5$ which follows the distribution of edge labels on YAGO. Note that the graphs are of the same size and complexity as those used in recent studies [18], [31], [37], [38].

The synthetic datasets model citation networks with three types of vertices, researcher, venue, and city, and six edge labels, cites from/to researchers, supervises from/to researchers, livesIn from researcher to city, workisIn from researcher to city, publishesIn from researcher to venue, and heldIn from venue to city. We use the synthetic datasets for evaluating scalability, varying the number of vertices and edges from roughly 1 and 8 million (**g-Mark-1m**) to 20 and 200 million (**g-Mark-20m**), resp.

Queries. We used twelve *CPQ* templates as described in Figure 5. These query structures correspond to practical structures appearing, e.g., in the Wikidata query logs [8], [9]. We use as abbreviations C, T, S, and St for Chain, Triangle, Square, and Star, respectively. We especially chose these templates to (1) better understand the interaction of all basic constructs of the language and (2) exemplify query structures occurring frequently in practice such as chains (e.g., C4), stars (St), cycles (e.g., Ti), and flowers (e.g., ST).

For each template and dataset, we generate ten queries with random labels. We only use queries in which all (sub-)paths of length two are non-empty, but the answers of some queries may be empty. To evaluate the difference between query times of non-empty and empty queries, queries on Yago, Wikidata, and Freebase have 50% non-empty and 50% empty queries except for C2. Queries for other datasets consist of mostly non-empty queries though we randomly set labels. We report for each query template the average response time over all ten queries. Note that the answers of some queries may be empty, but intermediate results are non-empty.

Methods. We compare the following methods: CPQx, our *CPQ*-aware index of Section IV; iaCPQx, our interest-aware CPQx of Section V; **Path**, the state-of-the-art lungauge-unaware path index proposed in [14]; **iaPath**, **Path** where only label sequences included in the given interest are indexed; **TurboHom++**, the state-of-the-art algorithm for homomorphic subgraph matching [25][5]; **Tentris**, the state-of-the-art RDF engine [6][6]; and, **BFS**, index-free breadth-first-search query evaluation [7]. We implemented all methods (see our open source codebase) except for TurboHom++ and Tentris. To be fair, we used the same query plans for all methods, except for TurboHom++ and Tentris which perform their own planning.

A **relational database approach** is essentially the same as Path with $k = 1$, which has lower performance than with $k = 2$. **RDF3X** and **Virtuoso** were shown to be outperformed by TurboHom++ [25] and Tentris [6]. Thus, we exclude Virtuoso, RDF3X, and the relational graph approach in our experiments.

We varied path length k from one to four, with a default value of two. For the interest-aware indexes on the datasets, we specify all label sequences in the set of queries as the interests. We divide label sequences larger than k length into prefix label sequences of length k and the rest. On synthetic datasets, we specify five label sequences as interests; cites-cites, cites-supervises, publishesIn-heldIn, worksIn-heldIn^{-1}, and livesIn-worksIn^{-1}.

Index implementation. In this study we use simple in-memory data structures; the study of alternative physical index representations is an interesting topic beyond the scope of this paper. Identifiers of vertices and labels are 32-bit integers, following TurboHom++. Indexes are implemented as standard C++ vectors. For further details, please see our open-source codebase.

A. Does CPQ-aware indexing accelerate query processing?

In summary, we can answer "yes" for this question. Figure 6 shows the average query time of each method for each of the twelve query templates on nine datasets. We do not show results on other five datasets due to page limitation.

CPQ-aware indexes accelerate CONJUNCTION as mentioned at Section IV-D, so query times of T, S, TT, and St with CPQx and iaCPQx are significantly smaller than those with all methods. For TC, SC, and ST, either iaCPQx or Path are the fastest on many datasets. When CONJUNCTION is heavy, iaCPQx is advantageous. For queries with JOIN and without CONJUNCTION such as C2, C4, Ti, and Si, since *CPQ*-aware indexes take two accesses to both I_{l2c} and I_{c2p}, they have higher costs than Path, but the difference between them is small. Query time of C2i is smaller than that of C2 in both iaCPQx and Path. This is because the size of answers decreases, and thus a cost for inserting s-t pairs to the answer sets reduces. Efficient IDENTITY works well on some datasets such as Robots, YAGO, and Freebase while

[5] The binary code of TurboHom++ was provided by the authors [25].

[6] https://github.com/dice-group/tentris

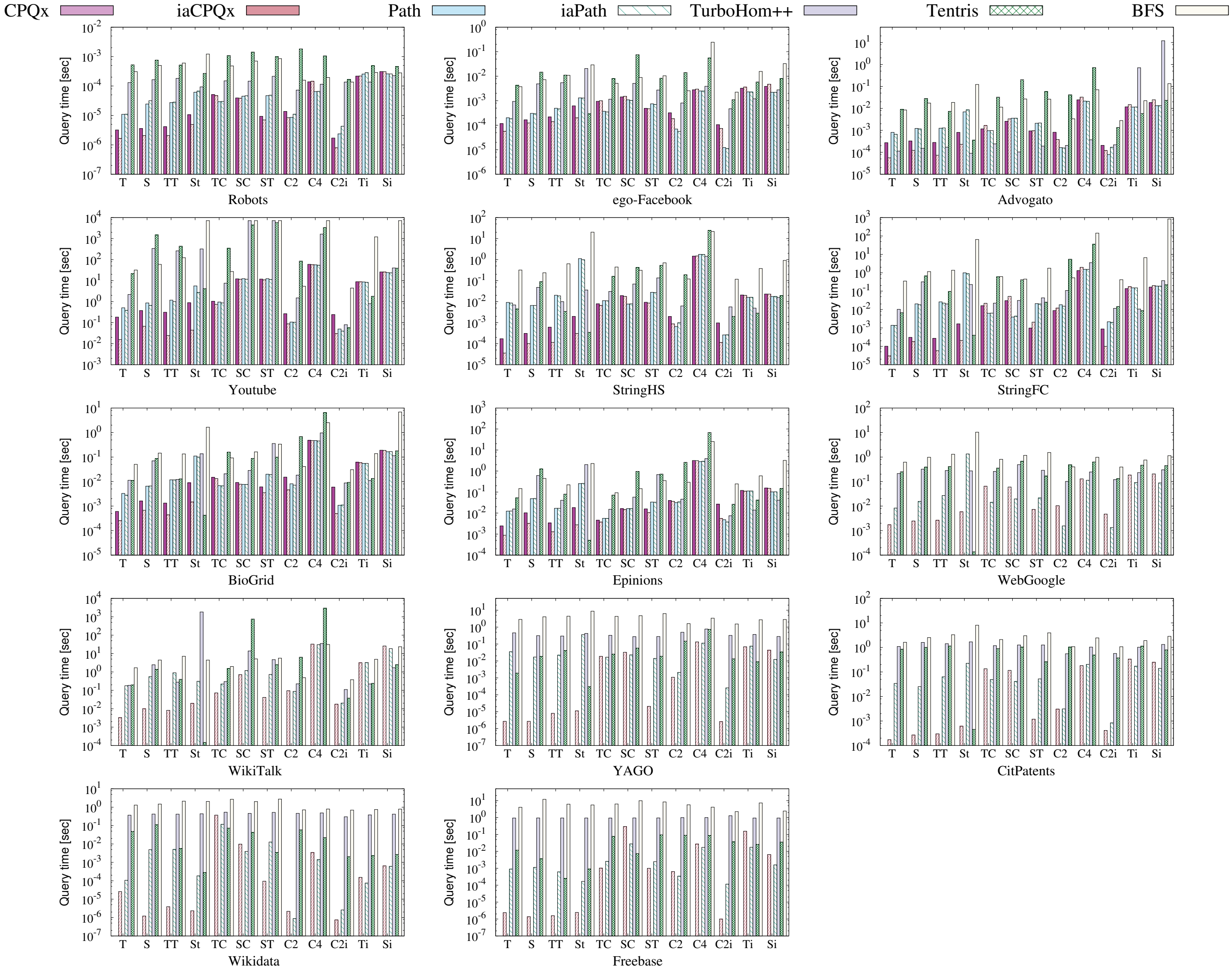


Fig. 6: Average query time for 12 query templates on real datasets. We terminated queries if the queries did not finish within two hours. Note that CPQx and Path are not reported for WebGoogle, WikiTalk, YAGO, CitPatents, Wikidata, and Freebase due to out of memory.

the efficiency highly depends on specified labels. For Ti and Si, TurboHom++ and Tentris perform well on some datasets because it joins only s-t pairs that satisfy cycle but other methods check whether s-t pairs are cycle or not after join. Tentris additionally performs well for St query on some dataset due to its query optimization. Compared with TurboHom++ and Tentris, our methods have significant improvement for many query templates such as T, S, TT, St, C2, and C2i. In particular, TurboHom++ and Tentris do no finish some queries within two hours in experiments; whereas iaCPQx and Path finish all queries within two hours.

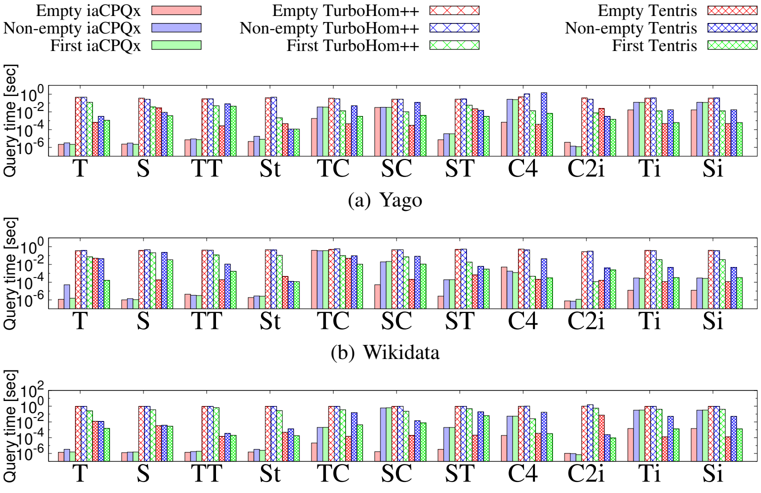
Comparing CPQx with iaCPQx, iaCPQx is more efficient because the numbers of class identifiers is smaller. In particular, for C2i, iaCPQx is much faster than CPQx as it reduces the number of LookUp. Here, we note that iaPath does not become faster than Path because both of the indexes have the same number of s-t pairs regarding to label sequences.

We evaluate synthetic benchmarking queries for three datasets, YAGO2 [**?**], LUBM [**?**], and WatDiv [**?**]. We use Y1–Y4 queries generated in [**?**] for YAGO2, L1–L7 queries for LUBM and L1–L5 and S1–S7 queries for WatDiv. For these benchmarking queries, we transform them into CPQs with keeping query shapes and their edge labels. We assign source and targets to the queries by ourselves.

Pruning power. We show the reason why our *CPQ*-aware indexes accelerate query processing more than the-state-of-the-art language-unaware path indexes. Recall that our query processing algorithm accelerates queries with Conjunction by comparing class identifiers instead of s-t pairs. Table III shows the average numbers of class identifiers in CPQx and iaCPQx and the average number of s-t pairs in iaPath, which are involved on evaluating S queries. The smaller numbers

TABLE III: The numbers of class identifiers on CPQx and iaCPQx and the number of s-t pairs on iaPath, for evaluating S queries.

Dataset	CPQx	iaCPQx	iaPath
Robots	0.4K	0.13K	2.4K
ego-Facebook	22K	19K	23K
Youtube	18M	2.0M	21M
Epinions	715K	222K	1.8M
Advogato	38.0K	6.4K	93.6K
BioGrid	275K	71.2K	499K
StringHS	49.7K	11.2K	750K
StringFC	33.3K	3.7K	388K
Yago	-	75	967K
WikiTalk	-	915K	19M
WebGoogle	-	287K	760K
CitPatents	-	36K	1.1M
Wikidata	-	3	286M
Freebase	-	8.6	79K



(a) Yago

(b) Wikidata

(c) Freebase

Fig. 7: Average query time of empty and non-empty queries, and for obtaining the first result of non-empty queries

indicate higher pruning power.

The numbers of class identifiers that are involved during the query evaluation in CPQx and iaCPQx are much smaller than the number of s-t pairs in iaPath. This result shows that partitioning s-t pairs based on k-path-bisimulation is effective for evaluating *CPQ*.

Impact of empty and non-empty queries. We evaluate the impact of empty and non-empty results on query time. The purposes here are (1) to gain further insight into the performance of our methods and (2) to compare the search strategy of our algorithm with that of TurboHom++ and Tentris. Figure 7 shows the query time on empty and non-empty queries on Yago, Wikidata, and Freebase of iaCPQx, TurboHom++, and Tentris. TurboHom++ outputs subgraphs, whereas *CPQ*s have binary output (i.e., s-t pairs). For a closer comparison, we also evaluate the query time for finding the first answer (thereby offsetting the cost of enumerating all non-binary answers with TurboHom++), also shown in Figure 7.

From these results, we can see that iaCPQx is much faster than TurboHom++ and Tentris for both empty and non-empty queries on most datasets and query templates. The query time on empty queries is generally smaller than that on non-empty queries because (1) empty queries do not have insert cost to the answers and (2) empty queries might terminate on the way of query evaluation due to empty intermediate results. Some non-empty queries are faster than empty queries when the intermediate results on empty queries are large.

Scalability: Figure 11 shows the average query time of iaCPQx for varying graph size on synthetic datasets. Our method scalably evaluates *CPQ*s as graphs grow larger.

Figure 9 shows the query time on YAGO benchmarking queries. YAGO2 includes 80 M vertices, 164 M edges, and 38 edge labels (including inverse edge labels). iaCPQx averagely achieves the smallest query time among them. Figure 10 shows the average query time of given queries varying with graph sizes of LUBM and WatDiv. iaCPQx can be built on graphs with about 280 and 220 M edges for LUBM and WatDiv, respectively. The increasing ratio of query time depends on query shapes. WatDiv benchmarking query needs more joins than ones of LUBM, so its query time largely increases as graph sizes increase.

B. Are CPQ-aware indexes compact?

We can also give a positive answer to this question. Table IV shows the index sizes and index times. CPQx achieves smaller index size than Path, because it stores a single s-t pair regarding to a class while Path stores multiple s-t pairs regarding to label sequences. *CPQ*-aware indexes can reduce its size well when the graph structures and labels have large skews. iaCPQx is much smaller than CPQx because it stores s-t pairs in the given interest. In WikiTalk, WebGoogle, Yago, Wikidata, and Freebase, the interest-*un*aware indexes cannot be constructed due to their size. The interest-aware indexes work well for large graphs, where index size is controllable by specifying the appropriate interests.

Indexing time in both CPQx and Path are generally large. Comparing CPQx with Path, CPQx is less efficient than Path because CPQx requires computing k-path-bisimulation, while Path just enumerates s-t pairs with label sequences. These difference is not very large in most datasets. Comparing the interest-aware indexes with the interest-unaware indexes, the interest-aware indexes are more practical, which clearly takes less time for construction. iaCPQx contributes to the scalable and efficient index construction with query acceleration. In this evaluation, we set k as two. We show that iaCPQx can be constructed in larger k in Sec. VI-D.

Figure 12 shows the index size on ego-Facebook varying the size of labels from 16 to 1024. This results show the index sizes of Path and CPQx gradually increase because the cardinally of label sequences and the number of class identifier increase, repsectively. While the index sizes of iaPath and iaCPQx decrease as the size of labels increases. This

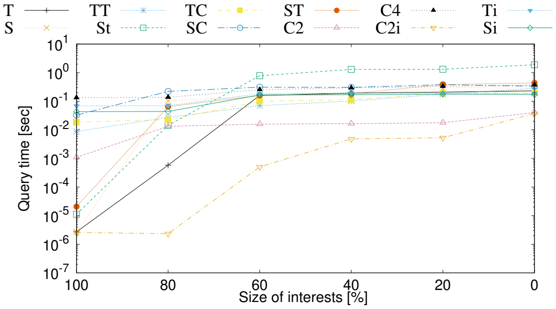


Fig. 8: Impact of interest size to query time iaCPQx on YAGO. Values on X-axis indicate the percentage of label sequences in the set of queries that we use as interests.

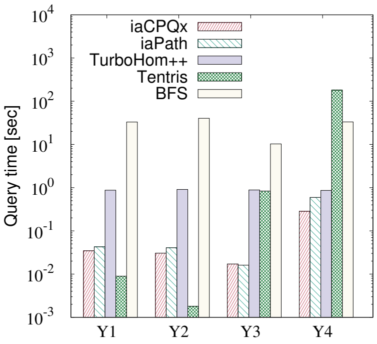


Fig. 9: Query time on YAGO2 benchmarking queries

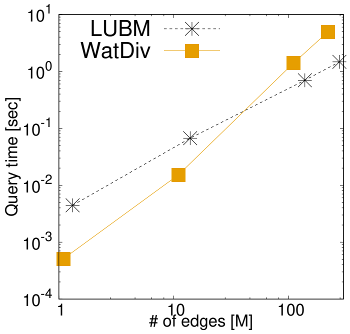


Fig. 10: Query time on LUBM and WatDiv

TABLE IV: Index size (IS) and index construction time (IT), where "-" indicates out of memory.

Dataset	CPQx		iaCPQx		Path		iaPath	
	IS [B]	IT [s]	IS [B]	IT [s]	IS [B]	IT [s]	IS [B]	IT [s]
Robots	1.78M	0.26	0.46M	0.056	2.0M	0.085	0.52M	0.043
ego-Facebook	97.4M	18.3	15.4M	2.0	116.9M	6.0	20.9M	131.5
Youtube	27.6G	57,653	2.3G	8,116	34.0G	28,870	8.7G	7,283
Advogato	56.7M	11.9	20.5M	8.53	74.4M	3.5	29.0M	2.3
StringHS	1.4G	521.4	1.3G	335.9	6.0G	260.4	2.8G	237.3
StringFC	1.0G	852.7	0.97G	664.7	5.1G	596.0	2.3G	579.8
BioGrid	1.7G	495.1	0.40G	86.6	2.5G	137.0	0.63G	47.9
Epinions	3.5G	849.6	1.0G	229.6	4.5G	264.1	1.3G	122.8
WebGoogle	-	-	4.7G	724.5	-	-	5.2G	444.7
WikiTalk	-	-	14.4G	3,064	-	-	16.0G	1,060
YAGO	-	-	3.8G	809.4	-	-	3.9G	589.7
CitPatents	-	-	2.2G	469.5	-	-	2.5G	226.4
Wikidata	-	-	1.44G	267.5	-	-	1.47G	70.4
Freebase	-	-	1.0G	5,696	-	-	3.7G	5,176
g-Mark-1m	-	-	0.63G	104.9	-	-	0.64M	55.7
g-Mark-5m	-	-	4.1G	728.0	-	-	4.1G	257.4
g-Mark-10m	-	-	9.3G	1,715	-	-	9.4G	574.2
g-Mark-15m	-	-	13.8G	2,741	-	-	13.9G	862.5
g-Mark-20m	-	-	20.3G	4,251	-	-	20.6G	1,590

TABLE V: Update time on CPQx

Dataset	Edge deletion	Edge insertion
Robots	0.0008 [s]	0.0005 [s]
Advogato	0.005 [s]	0.001 [s]
BioGrid	0.6 [s]	0.2 [s]
StringHS	0.3 [s]	0.1 [s]
StringFC	0.2 [s]	0.06 [s]
Youtube	0.9 [s]	0.3 [s]

TABLE VI: Update time on iaCPQx

Dataset	Edge deletion	Edge insertion	Label sequence deletion	Label sequence insertion
Robots	0.0002 [s]	0.0001 [s]	0.2 [μs]	0.01 [s]
Advogato	0.004 [s]	0.0004 [s]	0.3 [μs]	0.7 [s]
BioGrid	1.0 [s]	0.005 [s]	0.5 [μs]	14.2 [s]
StringHS	0.2 [s]	0.03 [s]	0.5 [μs]	15.4 [s]
StringFC	0.2 [s]	0.04 [s]	0.5 [μs]	9.8 [s]
Youtube	1.2 [s]	1.1 [s]	0.5 [μs]	255.5 [s]
YAGO	0.7 [s]	0.04 [s]	0.8 [μs]	30.3 [s]
Wikidata	0.7 [s]	0.4 [s]	1.0 [μs]	24.2 [s]
Freebase	1.7 [s]	0.2 [s]	1.1 [μs]	21.8 [s]

is because these interest-aware indexes store only paths that are matched with the given interests. So, if the number of labels is large, the number of paths that match the interests is small. Comparing CPQ-aware indexes and language-unaware path indexes, the size of indexes of CPQ-aware indexes are always smaller than those of language-unaware path indexes. This indicates that our indexes have robustness to the number of labels, in particular, iaCPQx.

C. Are CPQ-aware indexes maintainable?

In short, we can answer this question affirmatively. *CPQ-aware indexes can be efficiently updated with small deterioration of query processing and a small increase of its sizes.*

Update time. To study the impact of graph and interest updates, we delete and insert a hundred edges and label sequences of C2 queries, respectively, and report the average response time of each operation. Table VI shows the update time on iaCPQx. Our indexes can be quickly updated compared to the initial index construction time. Thus, our indexes

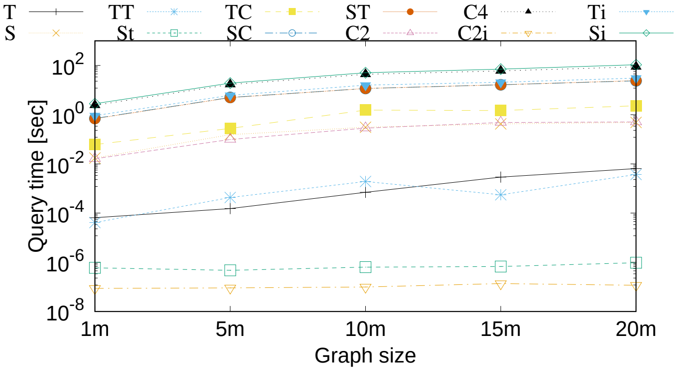


Fig. 11: Query performance of iaCPQx as graph size grows

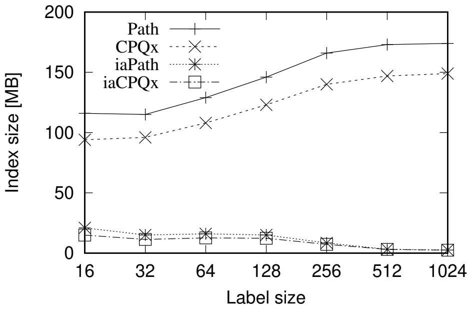


Fig. 12: Impact of the size of labels on index size on ego-Facebook

can handle graph and interest updates on large graphs.

Impact of updates on query time and index size. Our update method lazily updates our index, and thus it deteriorates performance of query time and increases the index size. We here evaluate the query time and index size after deleting $x\%$ edges (resp. x label sequences) and inserting the deleted edges (resp. label sequences).

Figure 13 shows the query time after updates. The query templates whose query times are small (e.g., C2i and T) increase their query time after updates because of increasing LOOKUP costs. On the other hand, the query templates with large JOIN costs (e.g., C4 and Si) do not increase their query time much because lookup costs are relatively small compared with JOIN costs. Note that we confirmed that the query results are the same before and after updates.

Table VII shows the increasing ratio of index size after updates. Our update method does not merge two class identifiers even if the s-t pairs in blocks with the class identifiers are k-path-bisimilar, and thus the size of index increases. We can see that our update method is practical because the increasing ratio is small even if the number of updates is large.

D. Are CPQ-aware indexes well-behaved as k grows?

We can also give a positive answer. As k increases, query processing time accelerates substantially.

Figure 14 shows the query time for iaCPQx varying with k. We can see that the query time decreases from $k = 1$ to $k = 2$. Some query times increase when $k > 2$. There are two reasons for this. First, CPQ-aware indexes divide paths into too fine granularity for some query templates, and which increases LOOKUP and CONJUNCTION costs. Second, costs of JOIN possibly increase, in particular, query time of C4 and Si queries increases when $k = 3$. As we described in Section IV-B, queries with diameters i become the fastest when $k = i$.

Figure 15 shows the index size of iaCPQx and index time for iaCPQx construction varying with k, respectively. The index size increases with increasing k generally. The increase ratio depends on the number of s-t pairs stored in iaCPQx. Therefore, the size of iaCPQx often slightly increases from $k = 3$ to $k = 4$ because the numbers of length 3 paths and length 4 paths are similar. In particular, the index size in Freebase does not increase much even when k increases. The index time increases as increasing k index size.

For deciding appropriate k, we can generally select the maximum length of interests when building iaCPQx. Otherwise, we select k and the interests to control index construction costs.

VII. CONCLUDING REMARKS

We studied language-aware path indexing for evaluation of *CPQ*, a fundamental language at the core of contemporary graph query languages. We proposed new practical indexes and developed algorithms for index construction, maintenance, and query processing to support the full index life cycle. We experimentally verified that our methods achieved up to three orders of magnitude acceleration of query processing over the state-of-the-art, while being maintainable and without increasing index size.

We highlight three research directions. (1) In practice, edges and vertices can also carry local data (e.g., user vertices might have their names and dates of birth) [7]. Study practical extensions to our methods for supporting CPQ combined with querying local data. (2) Investigate practical methods for scalable index construction that adaptively controls interests and k. (3) Now that we have practical CPQ-aware indexes, they can be used in a standard query processing pipeline, i.e., queries expressed in practical languages such as SPARQL and Cypher can use our indexes as part of a physical execution plan. Study query compilation and optimization strategies for *CPQ* combined with other languages such as *RPQ* and *CQ*.

REFERENCES

[1] L. Aceto, A. Ingolfsdottir, and J. Srba, “The algorithmics of bisimilarity,” in *Advanced Topics in Bisimulation and Coinduction*, 2011, pp. 100–172.

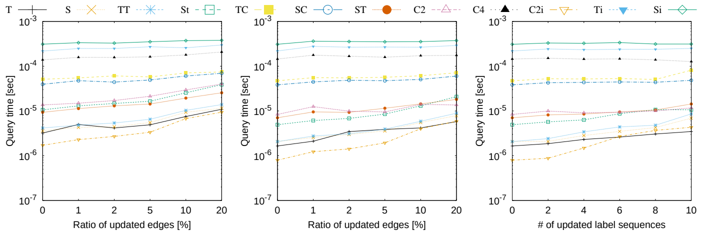
[2] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc, “Foundations of modern query languages for graph databases,” *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, 2017.

[3] R. Angles, J. L. Reutter, and H. Voigt, “Graph query languages,” in *Encyclopedia of Big Data Technologies*. Springer, 2019.

[4] G. Bagan, A. Bonifati, R. Ciucanu, G. Fletcher, A. Lemay, and N. Advokaat, “gmark: Schema-driven generation of graphs and queries,” *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 4, pp. 856–869, 2016.

[5] L. Baolin and H. Bo, “Hprd: a high performance rdf database,” in *NPC*, 2007, pp. 364–374.

[6] A. Bigerl, F. Conrads, C. Behning, M. A. Sherif, M. Saleem, and A.-C. N. Ngomo, “Tentris–a tensor-based triple store,” in *ISWC*, 2020, pp. 56–73.



(a) CPQx on graph update (b) iaCPQx on graph update (c) iaCPQx on workload update

Fig. 13: Impact of maintenance to query time on Robots

TABLE VII: The increasing ratio of index size on update on Robots

Index	Ratio of updated edges				
	1	2	5	10	20
CPQx	1.02	1.04	1.11	1.35	1.63
iaCPQx	1.03	1.06	1.13	1.31	1.53

Index	# of updated label sequences				
	2	4	6	8	10
iaCPQx	1.002	1.06	1.15	1.24	1.48

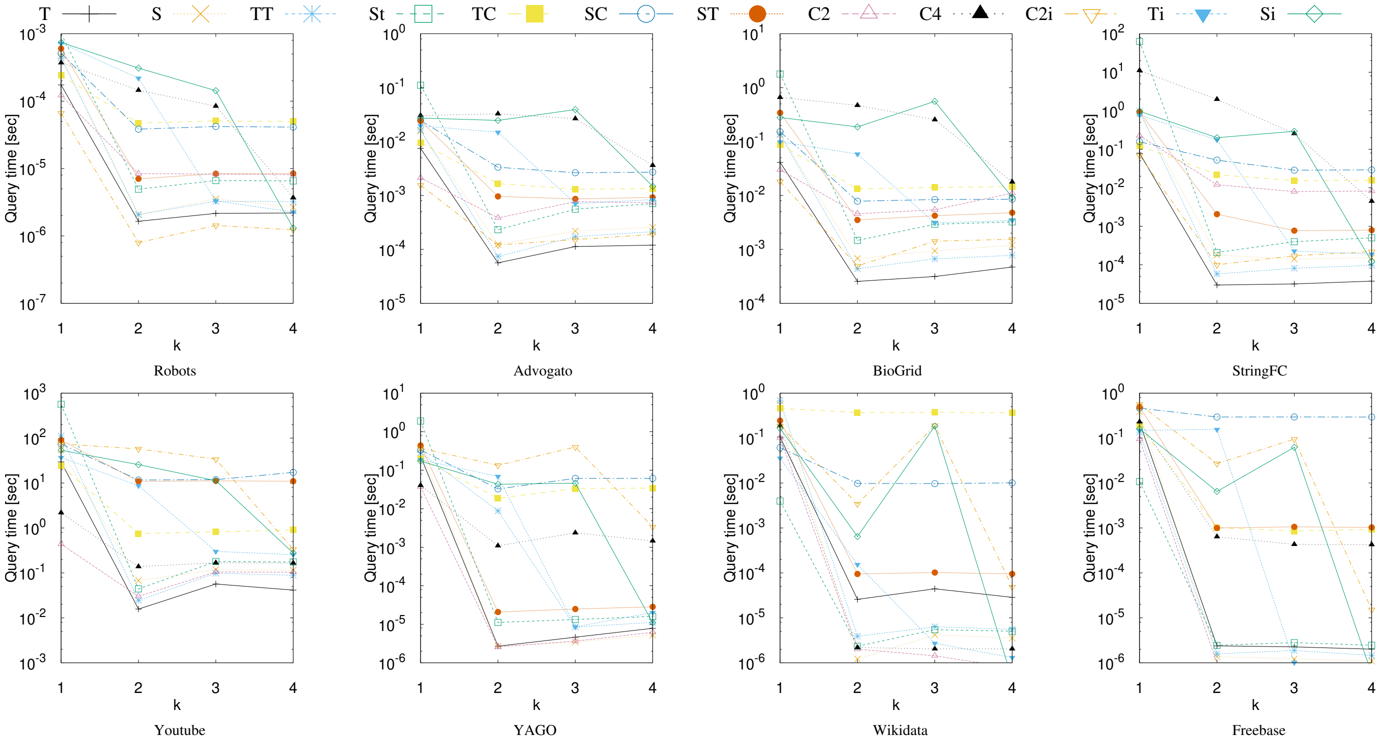
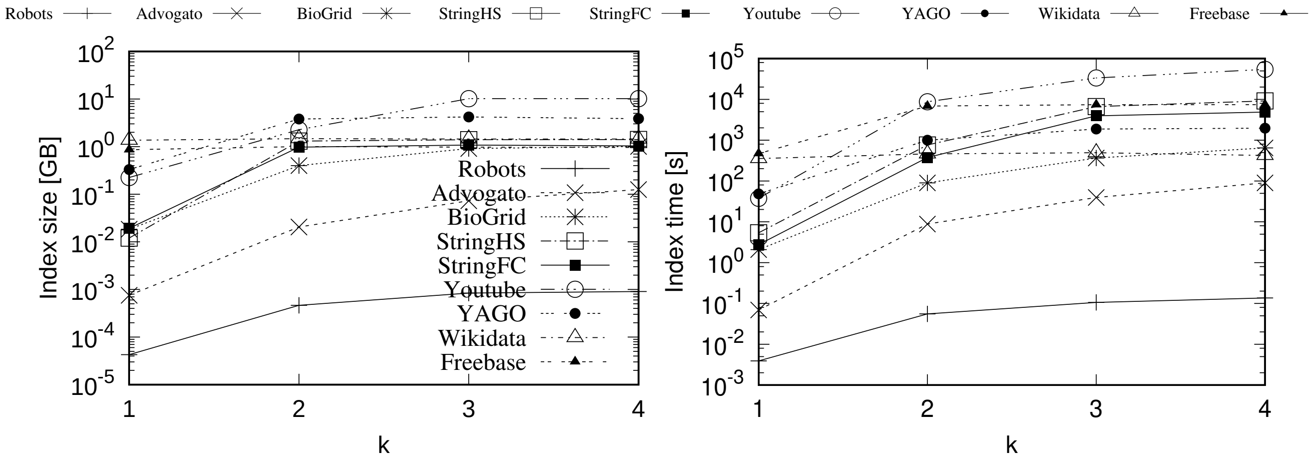


Fig. 14: Impact of k on query time with iaCPQx



(a) Index size (b) Index time

Fig. 15: Impact of k on index construction of iaCPQx

[7] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets, *Querying Graphs*. Morgan & Claypool, 2018.

[8] A. Bonifati, W. Martens, and T. Timm, “Navigating the maze of wikidata query logs,” in *WWW*, 2019, pp. 127–138.

[9] ——, “An analytical study of large SPARQL query logs,” *The VLDB Journal*, pp. 655–679, 2020.

[10] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon, “A fast index for semistructured data,” in *VLDB*, vol. 1, 2001, pp. 341–350.

[11] O. Erling and I. Mikhailov, “Rdf support in the virtuoso dbms,” in *Networked Knowledge-Networked Media*, 2009, pp. 7–24.

[12] G. Fan, W. Fan, Y. Li, P. Lu, C. Tian, and J. Zhou, “Extending graph patterns with conditions,” in *SIGMOD*, 2020, pp. 715–729.

[13] G. Fletcher, M. Gyssens, D. Leinders, J. Van den Bussche, D. Van Gucht, and S. Vansummeren, “Similarity and bisimilarity notions appropriate for characterizing indistinguishability in fragments of the calculus of relations,” *Journal of Logic and Computation*, vol. 25, no. 3, pp. 549–580, 2015.

[14] G. Fletcher, J. Peters, and A. Poulovassilis, “Efficient regular path query evaluation using path indexes,” in *EDBT*, 2016, pp. 636–639.

[15] G. Fletcher, D. Van Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens, “A methodology for coupling fragments of XPath with structural indexes for XML documents,” *Information Systems*, vol. 34, no. 7, pp. 657–670, 2009.

[16] R. Goldman and J. Widom, “Dataguides: Enabling query formulation and optimization in semistructured databases,” in *VLDB*, 1997, pp. 436–445.

[17] G. Gou and R. Chirkova, “Efficiently querying large XML data repositories: a survey,” *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 10, pp. 1381–1403, 2007.

[18] M. Han, H. Kim, G. Gu, K. Park, and W.-S. Han, “Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together,” in *SIGMOD*, 2019, pp. 1429–1446.

[19] W. Han, J. Lee, and J. Lee, “Turbo$_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases,” in *SIGMOD*, 2013, pp. 337–348.

[20] S.-C. Haw and C.-S. Lee, “Data storage practices and query processing in XML databases: A survey,” *Knowledge-Based Systems*, vol. 24, no. 8, pp. 1317 – 1340, 2011.

[21] J. Hellings, G. H. Fletcher, and H. Haverkort, “Efficient external-memory bisimulation on dags,” in *SIGMOD*, 2012, pp. 553–564.

[22] P. W. Holland and S. Leinhardt, “Local structure in social networks,” *Sociological methodology*, vol. 7, pp. 1–45, 1976.

[23] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth, “Covering indexes for branching path queries,” in *SIGMOD*, 2002, pp. 133–144.

[24] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, “Exploiting local similarity for indexing paths in graph-structured data,” in *ICDE*, 2002, pp. 129–140.

[25] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, “Taming subgraph isomorphism for RDF query processing,” *PVLDB*, vol. 8, no. 11, 2015.

[26] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, “An in-depth comparison of subgraph isomorphism algorithms in graph databases,” *PVLDB*, vol. 6, no. 2, pp. 133–144, 2012.

[27] Y. Luo, G. Fletcher, J. Hidders, Y. Wu, and P. De Bra, “External memory k-bisimulation reduction of big graphs,” in *CIKM*, 2013, pp. 919–928.

[28] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, “Network motifs: Simple building blocks of complex networks,” *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[29] T. Milo and D. Suciu, “Index structures for path expressions,” in *ICDT*, 1999, pp. 277–295.

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

[31] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang, “Answering billion-scale label-constrained reachability queries within microsecond,” *PVLDB*, vol. 13, no. 6, pp. 812–825, 2020.

[32] F. Picalausa, Y. Luo, G. Fletcher, J. Hidders, and S. Vansummeren, “A structural approach to indexing triples,” in *ESWC*, 2012, pp. 406–421.

[33] B. Rossman, “Homomorphism preservation theorems,” *J. ACM*, vol. 55, no. 3, pp. 15:1–15:53, 2008.

[34] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, “The ubiquity of large graphs and surprising challenges of graph processing: extended survey,” *VLDB J.*, vol. 29, no. 2-3, pp. 595–618, 2020.

[35] D. E. Shasha, J. T. Wang, and R. Giugno, “Algorithmics and applications of tree and graph searching,” in *PODS*, 2002, pp. 39–52.

[36] SNAP, “http://snap.stanford.edu/.”

[37] S. Sun, X. Sun, Y. Che, Q. Luo, and B. He, “Rapidmatch: a holistic approach to subgraph query processing,” *PVLDB*, vol. 14, no. 2, pp. 176–188, 2020.

[38] L. D. J. Valstar, G. Fletcher, and Y. Yoshida, “Landmark indexing for evaluation of label-constrained reachability queries,” in *SIGMOD*, 2017, pp. 345–358.

[39] Wikidata, “https://www.wikidata.org/.”

[40] K.-F. Wong, J. X. Yu, and N. Tang, “Answering XML queries using path-based indexes: a survey,” *World Wide Web*, vol. 9, no. 3, pp. 277–299, 2006.

[41] Ö. N. Yaveroğlu, N. Malod-Dognin, D. Davis, Z. Levnajic, V. Janjic, R. Karapandza, A. Stojmirovic, and N. Pržulj, “Revealing the hidden language of complex networks,” *Scientific Reports*, vol. 4, no. 1, p. 4547, 2014.