Accelerating Path Querying with Structural Indexes for Complex Graphs

ABSTRACT

Conjunctive path queries (*CPQ*) are one of the most frequently used queries for complex graph analysis. However, no graph indexes fully support the power of query languages to express *CPQ*s, resulting in poor query processing performance. We propose a structural index tailored for *CPQ* to accelerate evaluation of *CPQ*. Our structural index partitions the set of paths in the graph based on the notion of *path-bisimulation* where each index partition block holds paths indistinguishable with respect to *CPQ*s. We present methods to support the full index life cycle: index construction, maintenance, and query processing with our index. We also develop *interest-aware structural* indexing to reduce index construction and memory utilization overhead while accelerating query evaluation for queries of interest. We demonstrate through extensive experiments on eleven 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.

KEYWORDS

graph databases, homomorphic subgraph matching, bisimulation

ACM Reference Format:

. 2020. Accelerating Path Querying with Structural Indexes for Complex Graphs. In *Woodstock '20: ACM Symposium on Neural Gaze Detection, June 03–05, 2020, Woodstock, NY*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Complex graphs such as hyperlinks, social networks, and knowledge graphs are ubiquitous [7, 40]. Edge labels in these graphs indicate the semantics of relationships. For example, in a social network vertices represent people, webpages, and blog posts, and edges between people denote social relationships with labels such as "friendOf" or "relativeOf", whereas non-social relationships between vertices might be labeled "likes", "visits", or "bookmarked."

Analytics on path and graph patterns is fundamental in applications of complex graphs. *Conjunctive Path Queries* (*CPQ*) is a basic graph query language supporting path navigation patterns, cyclic path patterns, and conjunctions of patterns with homomorphic embedding semantics (the typical semantics for database query languages) [7]. In a recent analysis of the Wikidata and other query logs [8, 9], *CPQ* covers more than 99% of query shapes appearing in practice.[1] In general, many graph applications require subgraph structure matching, e.g., analytics on motifs [30, 48]. *CPQ* is at the core of practical query languages such as SPARQL [36] and Cypher [33] used in graph analytics systems [7].

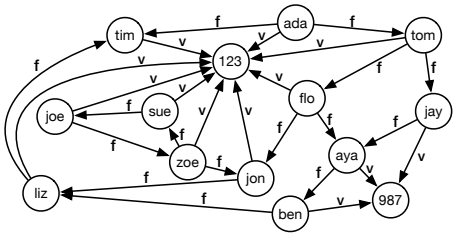

Figure 1: A graph $\mathcal{G}_{ex}$ with edge labels $\mathcal{L} = \{f, v\}$

As graphs grow in size, graph analytics systems need effective indexing methods so as to evaluate *CPQ* efficiently. State-of-the-art *path indexes* which can be used for accelerating *CPQ* evaluation essentially materialize sets of paths in the graph associated with given label sequences, e.g., [14, 41]. However, these methods are designed just for simple path navigation patterns, and therefore can result in poor *CPQ* query processing performance.

Structural indexing is a general methodology which leverages language-specific structural filtering in index design [7, 25]. Given a query language L and a database instance I, the basic idea of structural indexing is to partition the set of data objects of I under L-equivalence (i.e., into equivalence classes, where objects of the same class cannot be distinguished by any query in L), and build an index on the set of equivalence classes. Query processing with the index aggressively prunes out irrelevant data objects. Unfortunately, prior structural indexing methods do not support correct full processing of *CPQ* since these methods crucially rely on data models and/or query languages such as XPath, e.g., [5, 12, 15, 24, 31] which do not generalize to the expressive power of *CPQ* on arbitrary graphs; see detailed discussion in Table 1 and Section 2. In particular, *there are no known structural indexes for CPQ*.

Fortunately, it has been shown that *CPQ*-equivalence is equivalent to the tractable notion of *path-bisimilarity* [13]. This equivalence notion is tightly coupled to the expressive power of *CPQ* in the sense that 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. To date, however, *there has been no study of the practical usability of this formal characterization in the design of structural indexing for CPQ evaluation.*

Example. We illustrate the practical benefits of path-bisimulation-based structural indexing with an example *CPQ* query. Figure 1 shows a social media network $\mathcal{G}_{ex}$ of twelve user accounts (Ada,

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

Woodstock '20, June 03–05, 2020, Woodstock, NY

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

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. An example query is to find people and their followers who are in a triad (i.e., a cycle of length three) [23]. More precisely, this query finds the set of paths (v, u) that satisfy the conjunction of path navigation patterns ff and f^{-1} (where f^{-1} means navigating an edge labeled f from its target to source). This is a typical *CPQ*, which evaluates on our graph as $\{(sue, zoe), (joe, sue), (zoe, joe)\}$.

Consider an index on all paths of length at most 2 in $\mathcal{G}_{ex}$, with label sequences such as ff and f^{-1}. Given a label sequence, a state of the art non-structural path index retrieves a set of source-target paths associated with the label sequence. For example, given search key ff, a path index on $\mathcal{G}_{ex}$ would return 15 paths $\{(sue, zoe), \ldots, (flo, liz)\}$. Altogether there are 150 paths of length at most 2 in $\mathcal{G}_{ex}$. Not all of these 150 paths are distinguishable by path-bisimulation (and hence by *CPQ*). For example, there is no *CPQ* query which can distinguish the paths (sue, zoe), (joe, sue), and (zoe, joe) from each other, i.e., for every query q, either all three appear together in the evaluation of q or none of them do. These three paths form a single equivalence class under path-bimulation, and as far as query processing is concerned, they can be processed as a single block. For $\mathcal{G}_{ex}$ there are 30 such equivalence classes of paths of length at most 2, a five-fold reduction in the search space for *CPQ* evaluation.

This reduced search space can have tremendous impact on query evaluation costs. Consider again our example query above. With a state-of-the-art non-structural path index, this would be an intersection of lists of length 15 and 15, resp., each element of which is a pair of vertex identifiers, i.e., in total, comparisons over 60 object identifiers. However, with a structural path index on path-bisimulation equivalence classes, this would be an intersection of lists of class identifiers of length 3 and 3, resp., i.e., in total, comparisons over 6 object identifiers, followed by a retrieval of the single equivalence class forming the complete result set. This order-of-magnitude reduction of work significantly accelerates query evaluation performance.

Research challenges. Given the inapplicability of earlier structural indexing solutions, it is not immediately evident that the formal notion of path-bisimulation necessary for *CPQ* structural indexing (which has never been applied to graph indexing) can even be used in practice on real graphs. This raises the main research question we investigate in this paper: *Can structural indexing help to practically accelerate the evaluation of CPQ?*

As we illustrated above with our example, indexing methods that take advantage of the structural characterization of *CPQ* have excellent potential to help significantly accelerate query evaluation. However, to realize this potential, 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 path-bisimulation partitions can be efficiently computed, succinctly represented, maintained, and effectively indexed for query processing. Indeed, 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, 28]. Since prior studies were either theoretical or not applicable to our problem, we must build new bridges between theory and practice in order to realize practical structural indexing methods for accelerated *CPQ* evaluation.

Our contributions. In this paper, we initiate the study of structural indexing for *CPQ*, through seven contributions which together give a positive answer to our main research question. We propose (1) the first *structural index* for *CPQ*. The structural index is based on the notion *path-bisimulation* [13] which is tightly coupled to the expressive power of *CPQ*. We support the full index life cycle by our algorithms for (2) efficient index *construction* including computation of path-bisimulation, (3) efficient index *maintenance*, and for (4) accelerated *query processing* with the index. These contributions are achieved via a simple and effective index design with formal guarantees for the correctness of query results. Furthermore, as many practical application scenarios are interest-driven (i.e., users have specific navigation patterns for analysis), we also develop (5) an *interest-aware structural index* that reduces both index construction and memory utilization overhead, and leads to further acceleration of query processing. We demonstrate through (6) an extensive *experimental study* 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. Finally, (7) our *complete C++ codebase* is provided as open source.[2]

2 RELATED WORK

The study of graph querying is an active topic. Angles et al. [3, 4] and Bonifati et al. [7] give recent surveys of the current graph query language design landscape. At the heart of all current languages such as SPARQL [36], Cypher [33], PGQL [46], GSQL [18], G-CORE [2], and SQL/PGQ [42] is the language of Regular Queries (*RQ*), which support highly expressive queries combining both subgraph pattern matching and recursive path navigation functionalities [38]. However, there does not exist a practical (i.e., computable in polynomial time) structural characterization of the full *RQ* by which to partition graphs for structural indexing [39]. *CPQ* and the Regular Path Queries (*RPQ*) are important sublanguages of *RQ* [7]. *CPQ* and *RPQ* require fundamentally different indexing and processing methods because these two queries have different operations; *RPQ* does not support conjunctions of paths and cyclic patterns and *CPQ* does not support disjunctive patterns and Kleene star (i.e., transitive closure). In this paper, we study practical indexes for *CPQ*, but an important topic beyond the scope of this paper is to study practical indexing methods supporting both *CPQ* and *RPQ*.

A rich literature exists on path indexing [6, 14, 17, 21, 34, 47]. State-of-the-art non-structural path indexing methods are effective for path navigation patterns (e.g., [14, 43]) but they do not leverage the richer topological structure of graphs exposed by *CPQ* such as cyclic patterns and conjunctions of paths. Structural indexing for path queries, which does leverage richer graph structures, has been successfully studied for XML and semi-structured data, e.g., [5, 12, 15, 24, 31]. All prior work on graph structural indexing, however, has focused on data models and/or query languages incomparable with *CPQ*, e.g., DataGuides [16], A[k]-index [25], and T-index [31] for *RPQ* on rooted semi-structured graphs, and the

[2] *anonymized repository.* https://github.com/temporarygithubcode/www2021

Table 1: The comparison of properties of structural indexes

Index	Graph model	Query language
DataGuides [16]	Rooted semi-structured graph	RPQ
A[k]-index [25]	Rooted semi-structured graph	RPQ
T-index [31]	Rooted semi-structured graph	RPQ
P(k)-index [15]	Tree	XPath
Our index	Complex graph	CPQ

P(k)-index [15] for XPath queries on trees. We summarize the data models and query languages of existing structural indexes and our proposal in Table 1. These structural indexes are not applicable to *CPQ* on complex graphs by the following two reasons. First, existing indexes are for rooted semi-structured graphs and trees, and thus their construction methods are not applicable to complex graphs. Indeed, it is well known that reasoning about bisimulation structures on trees is much cheaper than on arbitrary graphs with cycles [22]. Second, *RPQ* and XPath do not support cyclic query patterns and/or conjunction of paths, so their structural characterizations are inapplicable to *CPQ*. These methods crucially rely on query semantics and/or graph structures which do not generalize to conjunctive cyclic path patterns on complex graphs (i.e., arbitrary graphs structures with cycles). In addition, 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. The vertex-based index does 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.

Furthermore, structural indexes on RDF graphs have been also proposed [29, 37, 44]. However, this work builds on structural characterizations which correspond only to acyclic query patterns or have no formal relationships to a given query language (and hence, there are no guarantees of correct query evaluation results for *CPQ*). In summary, to the best of our knowledge no earlier indexes can be adapted for structural indexing for *CPQ*, 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, 28]. 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 matching on edge-labeled graphs 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., [11, 19, 20, 27], are not suitable for *CPQ* which has homomorphic matching semantics. Isomorphic subgraph matching methods can return incorrect results when processing *CPQ*. On the other hand, systems for homomorphic subgraph matching such as RDF-3X [32], Virtuoso [10], and TurboHom++ [26] are applicable to process *CPQ*. To the best of our knowledge, TurboHom++ is the state-of-the-art system for homomorphic subgraph matching. We compare our methods with TurboHom++ in our experimental study.

3 PRELIMINARIES

In this paper 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 and explain the state-of-the-art *path index* [14].

3.1 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.[3] To support traversals in the inverse direction of edges, we always extend $\mathcal{L}$ and $\mathcal{E}$ with ℓ^{-1} for each $\ell \in \mathcal{L}$ and (u, v, ℓ^{-1}) for each $(v, u, \ell) \in \mathcal{E}$, respectively.

We will refer to pairs of vertices $(v, u) \in \mathcal{V} \times \mathcal{V}$ as *source-target paths*, where v is the source of the path and u is its target. We define $\mathcal{P}^{\leq k}$, for $k \geq 0$, to be the set of all those source-target paths 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 will refer to source-target paths.

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 paths $(v, u) \in \mathcal{P}^{\leq k}$.

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 $\{\langle \mathrm{f}, \mathrm{f}^{-1}\rangle, \langle \mathrm{v}, \mathrm{v}^{-1}\rangle, \langle \mathrm{f}^{-1}, \mathrm{f}\rangle\}$ and $\{\langle \mathrm{f}^{-1}\rangle, \langle \mathrm{f}\rangle, \langle \mathrm{v}, \mathrm{v}^{-1}\rangle\}$, respectively.*

3.2 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:

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

Note that the output of a *CPQ* is always a set of paths in $\mathcal{G}$.

Example 3.2. *Let us consider queries on $\mathcal{G}_{ex}$ of Figure 1.*

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

- *Users u and v such that u follows v and both users visit the same blog:* $[\![\mathsf{f} \cap (\mathsf{v} \circ \mathsf{v}^{-1})]\!]_{\mathcal{G}_{ex}} = \{(ada, tim), (ada, tom), \ldots, (zoe, sue)\}$.
- *Users that follow a user visiting the same blog:* $[\![(\mathsf{f} \circ \mathsf{v} \circ \mathsf{v}^{-1}) \cap id]\!]_{\mathcal{G}_{ex}} = \{(ada, ada), (flo, flo), \ldots, (zoe, zoe)\}$.

For an expression $q \in CPQ$, we define the *diameter* $\mathrm{dia}(q)$ of q as follows. 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, $\mathrm{dia}(q_1 \circ q_2) = \mathrm{dia}(q_1) + \mathrm{dia}(q_2)$. Intuitively, the diameter of an expression is the maximum number edge labels to which the join operation is applied. In Example 3.2, the queries are of diameter 2 and 3, respectively. For non-negative integer k, we denote by CPQ_k the set of all expressions in CPQ of diameter at most k.

3.3 Path indexing

The state-of-the-art path index [14] is essentially inverted indexes that output the set of paths corresponding to a given label sequence as search key. More precisely, given $\bar{\ell} = \langle \ell_1, \ldots, \ell_j \rangle \in \mathcal{L}^{\leq k}$, for some $j \leq k$, conceptually a k-path index $I^k_{\mathcal{G}}$ retrieves all paths associated with this label sequence , i.e., $I^k_{\mathcal{G}}(\bar{\ell}) = [\![\ell_1 \circ \cdots \circ \ell_j]\!]_{\mathcal{G}}$.

Some major drawbacks of non-structural k-path indexes, relative to our structural index presented in the next section, are (1) repetitive storage, as paths can be associated with multiple label sequences (e.g., (ada, ada) is associated with both $\langle \mathsf{f}, \mathsf{f}^{-1} \rangle$ and $\langle \mathsf{v}, \mathsf{v}^{-1} \rangle$ in Figure 1) and (2) failure to capture cyclic and conjunctive path structures (e.g., there are two distinct label sequences satisfied by (ada, tom), namely, $\langle \mathsf{f} \rangle$ and $\langle \mathsf{v}, \mathsf{v}^{-1} \rangle$). These drawbacks lead to an increase of index size and inefficient CPQ evaluation, as illustrated by our example in Section 1. The size of a non-structural k-path index is $O(\gamma |\mathcal{P}^{\leq k}|)$ because each path is stored γ times on average.

4 STRUCTURAL INDEX

In this section, we present (1) our structural index, (2) an algorithm for efficient query processing with the index, and (3) a method for effective maintenance of the index under graph updates.

4.1 Overall idea

The basic idea of our novel structural indexing is to partition the paths $\mathcal{P}^{\leq k}$ in a given graph $\mathcal{G}$ into disjoint blocks such that the paths within each block are indistinguishable with respect to CPQ_k queries (i.e., for each block, for every query $q \in CPQ_k$, either all paths or no paths of the block appear in $[\![q]\!]_{\mathcal{G}}$). In this way, the number of partition blocks can in practice be orders of magnitude smaller than the number of paths, as illustrated in our example in Sec. 1. We partition the paths 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 paths are k-path-bisimilar, the two paths are indistinguishable by any $q \in CPQ_k$. We design the first algorithm to practically compute equivalence partitioning based on k-path-bisimulation (Sec. 4.3).

The structural index is built over these blocks in two data structures, I_{l2h} and I_{h2p} (Sec. 4.2), which are maintainable while guaranteeing the correctness of query results (Sec. 4.5). Intuitively, I_{l2h} is a map from label sequences to block identifiers and I_{h2p} is a map from block identifiers to blocks (i.e., k-path-bisimulation equivalence classes). The index is used to evaluate queries in two stages (Sec. 4.4). In the first stage, the query is processed over the set of blocks. This stage allows us to filter out paths which will not contribute to the query result. In the second stage, the blocks identified in the first stage are retrieved, and then standard query processing proceeds on the paths contained in these blocks. The main advantage of our new query processing algorithm is that we compare the block identifiers for conjunctions of path queries instead of on paths themselves, resulting in significantly accelerated CPQ evaluation.

4.2 Index definition

The structural index is based on path equivalence under the notion of *k-path-bisimulation*. We choose this notion as it captures precisely the expressive power of CPQ_k. Intuitively, source-target paths (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 paths (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 $x = y$;
(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 $(m, y) \approx_{k-1} (m', u)$.

Note that this is a weaker structural notion than graph isomorphism. Furthermore, algorithmically bisimulation can be computed in polynomial time (see Section 4.3), whereas computing graph isomorphism is intractable.

Example 4.1. *Figure 2 shows k-path bisimulation of paths in $\mathcal{G}_{ex}$ of Figure 1 for $k = 1, 2$. For example, the $k = 2$ equivalence class labeled $\{\mathsf{f}^{-1}, \mathsf{ff}\}$ refines the $k = 1$ equivalence class labeled $\{\mathsf{f}^{-1}\}$, and all elements of this class are in the result set of the queries $[\![\mathsf{f}^{-1}]\!]_{\mathcal{G}_{ex}}$, $[\![\mathsf{f} \circ \mathsf{f}]\!]_{\mathcal{G}_{ex}}$, and $[\![\mathsf{f}^{-1} \cap (\mathsf{f} \circ \mathsf{f})]\!]_{\mathcal{G}_{ex}}$.*

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 structural index design, we define the notion of a k-path-bisimulation equivalence class of

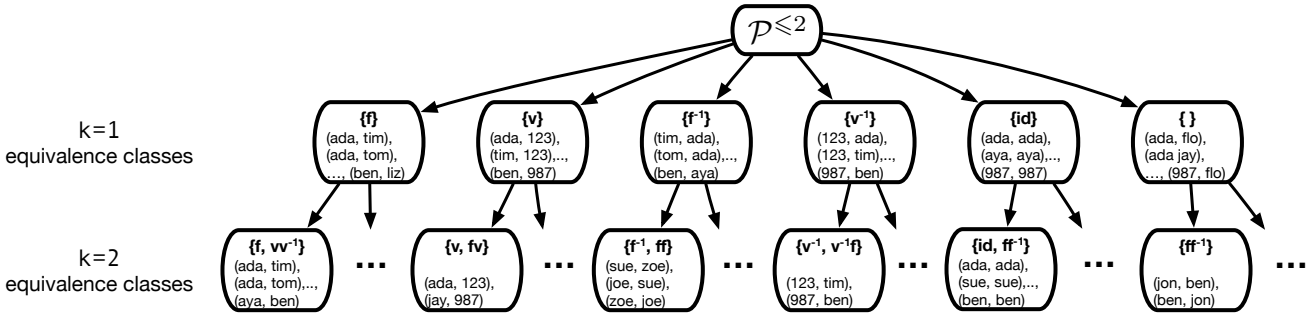

Figure 2: The 1-path and 2-path bisimulation partitions of $\mathcal{G}_{ex}$. Each equivalence class is labeled with the set of all CPQ_k expressions which select all members of the class. An arrow from block A to block B indicates that B refines A, i.e., $B \subseteq A$.

paths. The partition of $\mathcal{P}^{\leq k}$ into equivalence classes provides the basic building blocks 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 i-path-bisimulation 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 call an equivalence class a block *and we define the set of blocks as $B_i(\mathcal{G}) = \{[(v,u)]_i(\mathcal{G}) \mid v,u \in \mathcal{V}\}$.*

As an example, the second row ($k = 2$) of Figure 2 is $B_2(\mathcal{G}_{ex})$. Here, for example, we have $\{(sue, zoe), (joe, sue), (zoe, joe)\}$ as a block of $\approx_2$-equivalent paths.

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

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

Towards leveraging Corollary 4.1 for query processing, we assign an identifier $b_i(v,u)$ to each block $[(v,u)]_i$ of B_i. If two paths are in the same block, they have the same block identifier. Here, if two paths belong to the same block in B_i, they also belong to the same block in B_{i-1}, as i-path-bisimilar paths are $(i-1)$-path-bisimilar, i.e., $\approx_i$ refines $\approx_{i-1}$ for all $i > 0$. We can now define the k-path-bisimulation partition of a graph.

Definition 4.3. *For non-negative integer k, the k-path bisimulation partition of a graph $\mathcal{G}$ is the set*

$$[\mathcal{G}]_k = \{B_i(\mathcal{G}) \mid 0 \leq i \leq k\}.$$

Each path $(v,u) \in \mathcal{P}^{\leq k}$ has an associated sequence of k block identifiers $\langle b_0(v,u), b_1(v,u), \ldots, b_k(v,u)\rangle$. We call the sequence of k block identifiers of a path its *history*. It is easy to establish that k-path-bisimilar paths are uniquely identified by their common history. Each history consists of a distinct sequence of block identifiers, so we can assign a *history identifier* h to each history, and we define $\mathcal{H}$ as the set of all history identifiers. We also define $\mathbb{P}(h) \subseteq \mathcal{P}^{\leq k}$ and $\mathcal{H}(\bar{\ell}) \subseteq \mathcal{H}$ as the set of paths that belong to h, and the set of history identifiers that belong to $\bar{\ell}$, respectively.

We can now define our structural index based on the k-path-bisimulation partition of $\mathcal{G}$.

Definition 4.4 (Structural Index). *Given $[\mathcal{G}]_k$, a structural index $I_{[\mathcal{G}]_k}$ is a pair of data structures I_{l2h} and I_{h2p} such that I_{l2h} maps label sequences in $\mathcal{L}^{\leq k}$ to sets of history identifiers and I_{h2p} maps history identifiers to sets of paths in $\mathcal{P}^{\leq k}$, as follows:*

$$I_{l2h}(\bar{\ell}) = \{h \mid h \in \mathcal{H}(\bar{\ell})\},$$
$$I_{h2p}(h) = \{(v,u) \mid (v,u) \in \mathbb{P}(h)\}.$$

Choosing the appropriate value for k depends on the diameters of queries to be processed.

Our structural index achieves smaller size than the-state-of-the-art path indexes because each path in our index is associated with a single history, while each path in state-of-the-art path indexes can be associated with multiple label sequences, as we observed in Section 3.3. Therefore, our structural index enables us to efficiently find the set of paths that satisfy queries with a smaller footprint than the state-of-the-art path indexes.

Theorem 4.2. *The size of structural index is $O(\gamma|\mathcal{H}| + |\mathcal{P}^{\leq k}|)$.*

Proof: I_{l2h} stores the set of history identifiers associated with each label sequence. Each history identifier appears on average γ times in I_{l2h}. Thus, the size of I_{l2h} is $O(\gamma|\mathcal{H}|)$. 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 the structural index is $O(\gamma|\mathcal{H}| + |\mathcal{P}^{\leq k}|)$. □

The size of structural index $O(\gamma|\mathcal{H}| + |\mathcal{P}^{\leq k}|)$ is not larger than that of path index $O(\gamma|\mathcal{P}^{\leq k}|)$ because $|\mathcal{H}|$ is at most $|\mathcal{P}^{\leq k}|$.

4.3 Index construction

In this section, we describe how to construct the structural index efficiently. The main contribution here is that we develop an efficient algorithm to compute k-path-bisimulation partitions whose time and space complexity are polynomial. After computing the k-path bisimulation partition, we then construct $I_{[\mathcal{G}]_k} = (I_{l2h}, I_{h2p})$.

The algorithm for computing k-path-bisimulation partitions uses a bottom-up approach because i-path-bisimulation is derived from $(i-1)$-path-bisimulation in Definition 4.1. The efficient computation is coming from our idea that two paths are k-path-bisimilar iff (1) they have the same set of pairs of block identifiers in B_{i-1} and B_1 and (2) both paths are cycle or both are not. Furthermore, to efficiently compute the i-path-bisimulation for $1 \leq i \leq k$, we skip computing equivalence classes of paths that are not connected by label sequences of length i (e.g., when $i = 1$, we skip (ada, flo)

Algorithm 1: Computing k-path-bisimulation

input : Graph $\mathcal{G}$, natural number k
output: $[\mathcal{G}]_k$

procedure kPathBisimulation($\mathcal{G}$, k)
  $\mathbb{S}^i_{(v,u)} = \emptyset$ for $i = 1, \ldots, k$ and $\forall(v,u) \in \mathcal{P}^{\leq i} - \mathcal{P}^{\leq i-1} + (\mathcal{P}^{\leq i} \cap \mathcal{P}^{\leq i-1})$;
  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 B_1;
  for $i = 2, \ldots, k$ **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 $\forall(v, u) \in \mathcal{P}^{\leq i} - \mathcal{P}^{\leq i-1} + (\mathcal{P}^{\leq i} \cap \mathcal{P}^{\leq i-1})$ as B_i;
    if $i \neq 2$ **then** Clear $\mathbb{S}^{i-1}$:
  return $[\mathcal{G}]_k = \{B_1, \ldots, B_k\}$;
end procedure

Algorithm 2: Construction of the structural index

input : Graph $\mathcal{G}$, natural number k
output: Structural index $I_{[\mathcal{G}]_k} = \{I_{h2p}, I_{l2h}\}$

procedure StructuralIndex($\mathcal{G}$, k)
  $[\mathcal{G}]_k \leftarrow$ kPathBisimulation($\mathcal{G}$, k);
  for $(v, u) \in \mathcal{P}^{\leq k}$ **do**
    $h \leftarrow hash(\langle b^1_{v,u}, \ldots, b^k_{v,u}\rangle)$;
    if h *is NULL* **then**
      $h \leftarrow h_{new}$;
      $hash(\langle b^1_{v,u}, \ldots, b^k_{v,u}\rangle) \leftarrow h$;
      $\mathcal{H} \leftarrow \mathcal{H} \cup \{h\}$;
      Update h_{new};
    $I_{h2p}.append(h, (v, u))$;
  for $h \in \mathcal{H}$ **do**
    for $(v, u) \in I_{h2p}(h)$ **do**
      for $\overline{\ell} \in \mathcal{L}^{\leq k}(v, u)$ **do**
        $I_{l2h}.append(\overline{\ell}, h)$;
  sort (v, u) in I_{h2p} and h in I_{l2h};
  return $I_{[\mathcal{G}]_k}$;
end procedure

in Figure 2) because the number of such paths can be very large. The history $\langle b_0(v,u),\ b_1(v,u), \ldots, b_k(v,u)\rangle$ of any path (v,u) is uniquely identified even if partial (or whole) block identifiers are not assigned to (v,u) (i.e., $b_i(v,u) = Null$). The bottom-up approach enables to control the complexity by changing k, so we can specify k depending on the computational resources. After computing the k-path-bisimulation partition, the structural index is constructed in a simple way. It generates history identifiers from histories, and then insert a pair of $\overline{\ell}$ and $h \in \mathcal{H}(\overline{\ell})$ into I_{l2h} and a pair of history identifier h and $(v,u) \in \mathbb{P}(h)$ into I_{h2p}. Note that the set of paths with the same history identifier has the same label sequence due to the definition of k-path-bisimulation.

Algorithms 1 and 2 show pseudo-code for computing the k-path-bisimulation partition and constructing the structural index, respectively. In Algorithm 1, we sort elements of $\mathbb{S}^i$ so that i-path-bisimilar paths are sequentially listed for efficiently assigning the block identifiers. In Algorithm 2, we generate history identifier h of (v,u) by using a hash function for each history $\langle b_0(v,u), b_1(v,u), \ldots, b_k(v,u)\rangle$. If two paths have the same history, it assigns the same history identifiers to the two paths.

We here describe the space and time complexity for constructing the structural index.

Theorem 4.3 (Space complexity). *Given a graph $\mathcal{G}$ and positive number k, the space complexity of index construction is $O((k+d)|\mathcal{P}^{\leq k}| + \gamma|\mathcal{H}|)$, where d is the maximum vertex degree.*

Proof: The algorithm stores block identifiers (or label sequences) for paths in $\mathcal{P}^{\leq i} - \mathcal{P}^{\leq i-1} + (\mathcal{P}^{\leq i} \cap \mathcal{P}^{\leq i-1})$ 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 set of block identifiers (i.e., $B_1, \ldots, B_k$). 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 $O(\gamma|\mathcal{H}| + |\mathcal{P}^{\leq k}|)$. Therefore, its space complexity is $O((k+d)|\mathcal{P}^{\leq k}| + \gamma|\mathcal{H}|)$. □

Theorem 4.4 (Time complexity). *Given a graph G and positive number k, the time complexity of index construction is $O(k(d|\mathcal{P}^{\leq k}| + |\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|) + \gamma|\mathcal{H}|\log\gamma|\mathcal{H}|)$, where d is the max vertex degree.*

Proof: The algorithm for constructing the structural index has two steps (1) computing $[\mathcal{G}]_k$ and (2) constructing $I_{[\mathcal{G}]_k} = (I_{l2h}, I_{h2p})$. 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 $I_{[\mathcal{G}]_k} = (I_{l2h}, I_{h2p})$, it sorts the set of pairs of label sequences and history identifiers for I_{l2h} and the set of pairs of history identifiers and paths for I_{h2p}. Since these sizes are $O(|\mathcal{P}^{\leq k}|)$ and $O(\gamma|\mathcal{H}|)$, resp., it takes $O(|\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|)$ and $O(\gamma|\mathcal{H}|\log\gamma|\mathcal{H}|)$, resp. Thus, the total time complexity is $O(k(d|\mathcal{P}^{\leq k}| + |\mathcal{P}^{\leq k}|\log|\mathcal{P}^{\leq k}|) + \gamma|\mathcal{H}|\log\gamma|\mathcal{H}|)$. □

4.4 Query processing with structural index

We accelerate query processing by using the structural index, instead of the original graph, through the effective use of histories, which mitigates the cost of unnecessarily comparing paths which do not participate in the query result.

Our query processing method evaluates a given query $q \in CPQ$ following its parse tree, where label sequences are processed from left to right in k-sized prefixes (see Figure 3 for an example). Each node on the parse tree represents logical operations of CPQ: LookUp, Conjunction, Join, and Identity. We process starting from the root node of q, recurring on the left and right, as necessary. This method heuristically derives an execution plan. Further query optimization and planning with our index is an interesting rich topic for future research.

Our query processing method, in particular, accelerates Conjunction and Identity thanks to the path-bisimulation-based

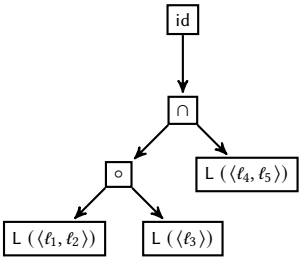

Figure 3: Parse tree of query $[(\ell_1 \circ \ell_2 \circ \ell_3) \cap (\ell_4 \circ \ell_5)] \cap id$, when $k = 2$. Here, $\ell_1 \circ \ell_2 \circ \ell_3$ is processed left to right, with an index look up $\langle \ell_1, \ell_2 \rangle$ of joined with a look up of $\langle \ell_3 \rangle$.

partitioning. Recall that paths with the same history identifiers represent k-path-bisimilar paths. If a history identifier is included in both sets of history identifiers regarding to two label sequences, the set of paths regarding to the history identifier is through both label sequences. That is, we can find conjunction of the two paths without comparing the set of paths. In the case of Identity, since k-path-bisimilar paths are partitioned into cycle or not, we can evaluate Identity by just checking the first path in the set of paths of history identifiers. Since the number of history identifiers $|\mathcal{H}|$ is much smaller than that of paths $|\mathcal{P}^{\leq k}|$, the computation cost decreases significantly (see Table 3 in experimental study).

Proposition 4.5 (Conjunction correctness). *Given two sets of history identifiers $\mathbb{H}$ and $\mathbb{H}'$, the set of paths $\mathbb{P}(h)$ for all $h \in \mathbb{H} \cap \mathbb{H}'$ is same as $\mathbb{P}(h) \cap \mathbb{P}(h')$ for all $h \in \mathbb{H}$ and $h' \in \mathbb{H}'$.*

Algorithm 3 shows pseudocode for our query processing method. $\mathbb{P}$ and $\mathbb{H}$ denote the sets of paths and history identifiers that are found during query processing, respectively. Our query processing algorithm repeatedly traverses the nodes on a given query tree. We use some optimization techniques to reduce computation cost with guaranteeing the correctness. First, in Conjunction we compute $\mathbb{H}_1 \cap \mathbb{H}_2$ where $\mathbb{H}_1, \mathbb{H}_2 \subseteq \mathcal{H}$ to obtain conjunctive paths instead of comparing paths. Second, we use sort merge join as a physical operator for Conjunction and Join. Third, in Identity since we can optimize $q \circ id = q$, we handle only $q \cap id$ as Identity. Additionally, Identity is executed with the other three operators to avoid inserting the paths that are deleted by Identity.

4.5 Index maintenance

Our structural index is easily updated when the graph is updated. Our update method lazily updates the structural index, while maintaining correctness 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, some paths change their label sequences (also may disappear/appear) and k-path-bisimilar paths may become non-bisimilar. If two non-bisimilar paths are assigned the same history identifier, then query results would be incorrect. Thus, our lazy update method divides the set of paths if they become not k-path-bisimilar due to edge deletion. On the other hand, it does not merge two sets of paths even if they become k-path-bisimilar. This is because even if k-path-bisimilar paths belong to the different history identifiers, query processing still ensures correct results.

Algorithm 3: Query processing

```
input : Node on query tree q, structural index I_[G]_k
output: set of paths P, set of histories H
procedure Evaluation(q, I_[G]_k)
if operation of q is LookUp then
    return LookUp (q.ℓ, I_[G]_k);
else if operation of q is Identity then
    return Identity(P, H, I_[G]_k);
else
    P_l, H_l ← Evaluation(q_l, I_[G]_k);
    P_r, H_r ← Evaluation(q_r, I_[G]_k);
    if operation of q is Join then
        return Join (P_l, P_r, H_l, H_r, I_[G]_k);
    else if operation of q is Conjunction then
        return Conjunction (P_l, P_r, H_l, H_r, I_[G]_k);
if q is the root of query tree then
    P ← P ∪ I_h2p(h) for all h ∈ H;
return P, H;
end procedure
```

Proposition 4.6 (Update correctness). *After edge deletion or edge insertion, query processing with Algorithm 3 ensures correct query results.*

We next 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 paths involved in the deleted edge by bread-first search. The label sequences of these paths 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_{h2p} if the label sequences of the paths change. Here, for efficiently finding history identifier h' according to the deleted paths, we use inverted index whose keys are paths. We then add new $\mathbb{P}(h')$ that includes only the path into I_{h2p} unless their label sequences are empty (i.e., paths disappear). This update does not check whether or not the affected path is k-path-bisimilar to other paths.

Edge insertion. The procedure is similar to that for edge deletion. The difference is enumerating paths involving the new edge.

Other graph updates. We can handle the following additional updates by combinations of edge deletion and insertion.

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

Theorem 4.7. *The time complexity for edge deletion or insertion is $O(d|\mathcal{P}_u| + |\mathcal{P}_u| \log |\mathcal{P}^{\leq k}| + |\mathcal{H}| \log |\mathcal{H}|)$, where $\mathcal{P}_u$ and d are the set of paths that are involved updates and the maximum degrees among vertices in $\mathcal{P}_u$, resp.*

5 INTEREST-AWARE INDEX

In many application scenarios, users are often interested in only a specific set of queries. The interest set of queries can be decomposed into the specific set of label sequences appearing in the queries. The structural index, however, stores all label sequences, including inverse of labels, up to length k. Motivated by this, we develop an interest-aware structural index based on a given set of queries. The index supports processing of any *CPQ*, yet is tailored to especially accelerate processing of all *CPQ* queries (i.e., not just those in the set of interest) which use any of the label sequences of interest.

Interest-aware structural indexing. Towards an interest-aware structural index, 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 paths (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) $\mathcal{L}^{\leq k}(v, u) \cap \mathcal{L}_q = \mathcal{L}^{\leq k}(x, y) \cap \mathcal{L}_q$.

Intuitively, $\mathcal{L}_q$ are the label sequences of interest in a given set of queries. When we construct the interest-aware structural index, 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 the basic and the interest-aware structural indexes is that the former and the latter assign same history identifiers to the set of k-path bisimilar paths and the set of interest-aware path-equivalent paths, 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 paths have the same history identifiers (i.e., partition blocks are bigger). Therefore, the size of interest-aware structural index is much smaller (and hence faster to use) than that of the basic structural index.

Index construction and query processing. The index construction and query processing methods are almost the same as those for the structural index. The difference for the construction algorithm is that it enumerates paths only with given label sequences and two paths have same history identifiers if they are interest-aware path-equivalent. Since the construction of the interest-aware structural index decreases the number of paths, it becomes more efficient than that of the structural index. 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.

The space and time complexity of constructing the index are similar to Theorems 4.3 and 4.4, resp. In interest-aware structural indexes, costs decrease as $\mathcal{L}_q \subseteq \mathcal{L}^k$ decreases in size since the number of indexed paths in the graph decreases.

Interest-aware index maintenance. The interest-aware structural index can be easily updated in a similar lazy fashion as we did for the structural index.

Graph update: The graph update procedures are almost the same as those for the basic structural index given in Section 4.5. The difference is that we do not process the set of paths whose label sequences are not included in the given set of label sequences.

Table 2: 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	88,234	16	
Epnions	131,828	840,799	16	
Advogato	5,417	1,724,554	8	✓
BioGrid	64,332	862,277	14	✓
StringHS	16,956	2,483,530	14	✓
StringFC	15,515	4,089,600	14	✓
WikiTalk	2,394,385	10,042,820	16	
WebGoogle	875,713	10,210,074	16	
Youtube	15,088	21,452,214	10	✓
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	

Label sequence deletion: When we delete label sequences from the given set of label sequences, we can just delete history identifiers from I_{l2h} of the deleted label sequence. After deleting the label sequences, 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 4.6.

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

6 EXPERIMENTAL STUDY

We next present the results of an experimental evaluation of our methods. We designed the experiments to clarify the questions: (1) *Does structural indexing accelerate query processing?* (Section 6.1); (2) *How compact are structural indexes?* (Section 6.2); (3) *Can structural indexes be effectively updated?* (Section 6.3); and, (4) *Are structural indexes well-behaved as k grow?* (Section 6.4).

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 2 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 hyperlinks, social networks, knowledge graphs, citation networks, and biological networks. These datasets except for ego-Facebook, webGoogle, WebTalk, CitPatents, and Wikidata are provided by the authors [35, 45]. In Wikidata, we extract vertices that represent URI from original Wikidata[4]. WebGoole, WebTalk, and CitPatents are available at SNAP[5]. Since these three graphs have no edge labels, we assign edge labels that are exponentially

[4]https://www.wikidata.org/
[5]http://snap.stanford.edu/

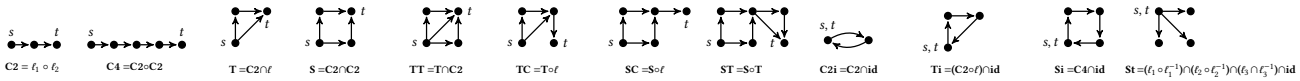

Figure 4: Query templates, where s and t denote the source and target of paths, respectively.

distributed with $\lambda = 0.5$ which follows the distribution of edge labels on YAGO. As StringHS and StringFC often have similar result trends, we omit the results of StringHS at some points due to space constraints.

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 4. These query structures correspond to practical structures appearing, e.g., in the Wikidata query logs [8, 9]. 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). We use as abbreviations C, T, S, and St for Chain, Triangle, Square, and Star, respectively.

For each template and data set, we generate ten queries with random labels, with a mix of queries having empty and non-empty result sets. 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, Yago and Freebase have half non-empty and half empty queries. 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.

Methods. We compare the following methods: **Structural**, our structural index of Section 4; **IA-Structural**, our interest-aIAre structural index of Section 5; **Path**, the state-of-the-art path index proposed in [14]; **IA-Path**, **Path** where only label sequences included in the given interest are indexed; **TurboHom++**, the state-of-the-art algorithm for homomorphic subgraph matching [26]; and, **BFS**, index-free breadth-first-search query evaluation [7]. We implemented all methods (available in our open source codebase) except for TurboHom++ for which we used the authors' binary code [26]. To be fair, we used the same query plans for all methods, except for TurboHom++ which performs its own planning. We also note that the implementation of TurboHom++ just counts the number of answers, while our query processing outputs all answers, so our method is more expensive in this sense.

We also tested an RDF engine **Virtuoso**, but its performance is significantly lower than TurboHom++. **RDF3X** was shown to be outperformed by TurboHom++ [26]. A simple **relational database approach** is essentially the same as Path with $k = 1$, which is lower performance that with $k = 2$. 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.

6.1 Does structural indexing accelerate query processing?

In summary, yes. Figure 5 shows the average query time of each method for each of the twelve query templates on the real datasets. The structural index accelerates conjunctions as mentioned at Section 4.4, so query times of T, S, TT, and St with Structural and IA-Structural are significantly lower than those with all methods, up to thousands times faster on queries with Conjunction and without Join. For TC, SC, and ST, the fastest method depends on datasets, Structural or Path. When Conjunction is heavy, Structural is advantageous. For queries with Join and without Conjunction such as C2, C4, Ti, and Si, since Structural takes two accesses to both I_{l2h} and I_{h2p}, it has higher costs than Path, but the difference between them is small. Query time of C2i is smaller than that of C2 in both Structural and Path. This is because the size of answers decreases, and thus a cost for inserting paths to the answer sets reduces. Efficient Identity operation works well on some datasets such as Robots, StringFC, and YAGO, while the efficiency highly depends on specified labels. For Ti and Si, TurboHom++ works well on some datasets because it joins only paths that satisfy cycle but other methods check whether paths are cycle or not after join. Compared with TurboHom++, our methods have significant improvement for many query templates such as T, S, TT, St, C2, and C2i.

Comparing Structural with IA-Structural, IA-Structural achieves smaller query time because the numbers of paths and history identifiers are smaller. In particular, for C2i, IA-Structural is much faster than the Structural because it reduces the number of LookUp operations. Here, we note that IA-Path does not become faster than Path because both of the indexes have the same number of paths regarding to label sequences.

Pruning power. We show the reason why our structural indexes accelerate query processing more than the-state-of-the-art path indexes. Recall that our query processing algorithm accelerates queries with Conjunction by comparing history identifiers instead of paths. Table 3 shows the average numbers of history identifiers in Structural and IA-Structural and the number of paths in Path, which

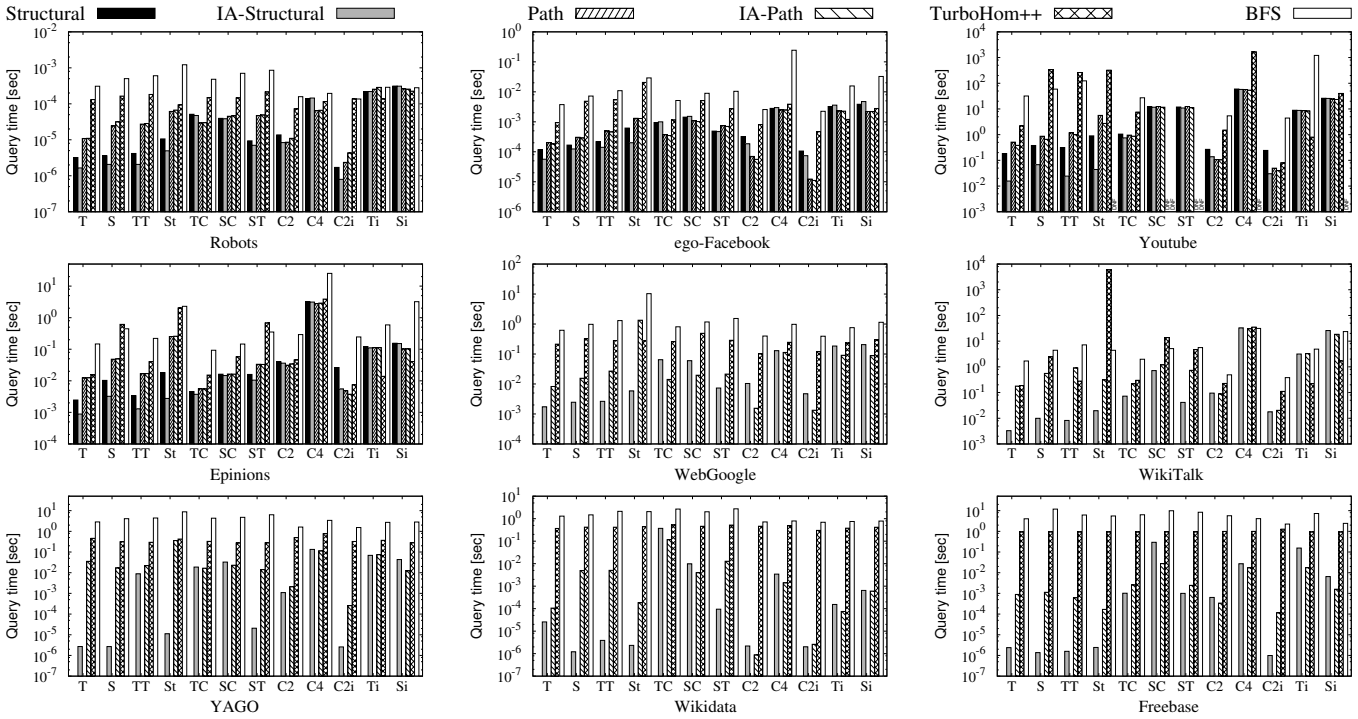

Figure 5: Average query time for 12 query templates on real datasets. DNF denotes did not finish within 24 hours. Note that Structural and Path are not reported for YAGO, WikiTalk, WebGoogle, CitPatents, and Freebase due to out of memory.

are involved on evaluating S queries which include Conjunction. The smaller numbers indicate higher pruning power. The numbers of history identifiers that are involved during the query evaluation in Structural and IA-Structural are much smaller than the number of paths in Path. This result shows that partitioning paths 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++. TurboHom++ searches for answers by a depth-first manner which is fast to find the first result, while our query algorithm only reports answers after the last logical operator.

Figure 6 shows the query time on empty and non-empty queries on Yago and Freebase of IA-Structural and TurboHom++. In TurboHom++ all query nodes are given as output, whereas *CPQ*'s have binary output. 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 6. From these result, we can see that IA-Structural is much faster than TurboHom++ for both empty and non-empty queries. For finding the first answer in queries with Conjunction (e.g., T and S), IA-Structural is faster than TurboHom++, so we can confirm that our methods are faster than TurboHom++. In IA-Structural, the query time on empty queries is generally smaller that that on non-empty queries because of two reasons; (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. While, some non-empty queries are faster than empty queries because the intermediate results on empty queries are possibly very large even when the answers are empty.

Scalability: Figure 7 shows the **IA-Structural** average query time for varying graph size of synthetic datasets. Our method scalably evaluates *CPQ*s as graphs grow larger.

6.2 Are structural indexes compact?

We can also give a positive answer to this question. Table 4 shows the index sizes and times. Structural achieves smaller size than Path, because it stores a single path regarding to a history while Path stores multiple paths regarding to label sequences. The IA-Structural is much smaller than Structural because it stores paths in the given interest. In Yago, WikiTalk, WebGoogle, CitPatents, 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. IA-Structural can reduce its size well when the graph structures and labels have large skews.

Indexing time in Structural is larger than that in Path because constructing Structural requires computing k-path-bisimulation, while Path just enumerates paths with label sequences. However, since the index time increases linearly as the size of graph increases, constructing Structural is practical. The interest-aware indexes clearly take less time for construction.

Table 3: The numbers of histories on Structural and IA-Structural and the number of paths on Path, for evaluating S queries.

Dataset	Structural	IA-Structural	Path
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

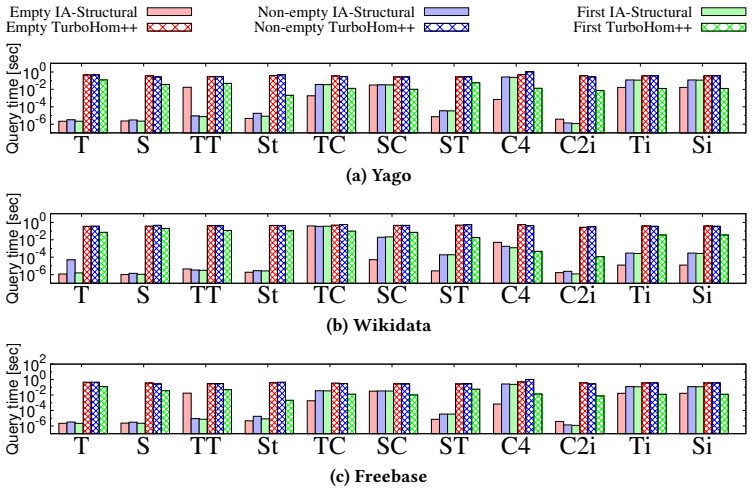

Figure 6: Average query time of empty and non-empty queries, and for obtaining the first result of non-empty queries

Table 4: Index size (IS), index time (IT), and memory consumption for constructing indexes (IM), where "-" indicates out of memory.

Dataset	Structural			IA-Structural			Path			IA-Path		
	IS [B]	IT [s]	IM [B]	IS [B]	IT [s]	IM [B]	IS [B]	IT [s]	IM [B]	IS [B]	IT [s]	IM [B]
Robots	1.78M	0.26	84.0M	0.46M	0.057	19.4M	2.0M	0.085	35.0M	0.52M	0.046	19.9M
ego-Facebook	97.4M	18.3	2.1G	15.4M	2.1	322.2M	116.9M	6.0	974.3M	20.9M	131.5	324.1M
Youtube	27.6G	57,653	389.6G	2.3G	8,705	103.3G	34.0G	28,870	195.5G	8.7G	8,284	107.2G
Epinions	3.5G	849.6	84.7G	1.0G	229.6	25.3G	4.5G	264.1	36.6G	1.3G	1.63	25.3G
Advogato	56.7M	11.9	1.8G	20.5M	3.84	752M	74.4M	3.5	808M	29.0M	2.4	754M
BioGrid	1.7G	495.1	44.9G	0.40G	89.2	10.1G	2.48G	137.0	19.6G	0.63G	50.8	10.1G
StringHS	1.4G	521.4	71.0G	1.3G	374.8	46.0G	6.0G	260.4	42.3G	2.8G	221.3	45.3G
StringFC	1.0G	852.7	60.4G	0.97G	761.7	37.7G	5.1G	596.0	36.6G	2.3G	606.3	37.5G
WebGoogle	-	-	-	4.7G	782.3	98.0G	-	-	-	5.2G	479.1	98.3G
WikiTalk	-	-	-	14.4G	3,226	286.2G	-	-	-	16.0G	1,060	278.6G
YAGO	-	-	-	3.8G	1,001	152.8G	-	-	-	3.9G	776.9	152.2G
Wikidata	-	-	-	1.44G	461.6	29.4G	-	-	-	1.47G	77.7	33.0G
CitPatents	-	-	-	2.2G	535.1	45.5G	-	-	-	2.5G	257.4	46.5G
Freebase	-	-	-	1.0G	6,884	314.9G	-	-	-	3.7G	6,426	319.1G
g-Mark-1m	-	-	-	0.63G	104.9	13.6G	-	-	-	0.64M	55.7	13.9G
g-Mark-5m	-	-	-	4.1G	728.0	64.2G	-	-	-	4.1G	257.4	65.4G
g-Mark-10m	-	-	-	9.3G	1,715	167.3G	-	-	-	9.4G	574.2	171.3G
g-Mark-15m	-	-	-	13.8G	2,741	201.9G	-	-	-	13.9G	862.5	204.1G
g-Mark-20m	-	-	-	20.3G	4,251	337.9G	-	-	-	20.6G	1,590	346.0G

Table 5: Update time on structural index

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 6: Update time on IA-structural index

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]

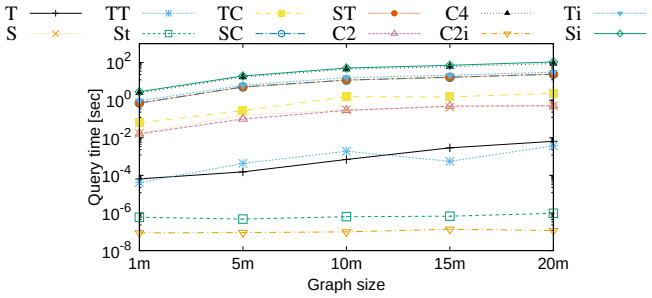

Figure 7: IA-structural query performance as graph size grows

6.3 Are structural indexes maintainable?

In short, we can also answer this question affirmatively. Our structural 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 updates and interest updates, we delete and insert ten edges and ten label sequences, respectively, and report the average response time of each operation. Tables 5 and 6 show the update time on Structural and IA-Structural, respectively. Our indexes can be quickly updated compared to the initial index construction time. The IA-Structural can be updated for graph updates more efficiently than Structural because the number of edges that are involved with graph update is smaller. The interest changes also can be handled with low cost.

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 8 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 7 shows the increasing ratio of index size after updates. Our update method does not merge two history identifiers even if the set of paths regarding to the history identifiers are k-path-bisimilar, and thus the size of index increases. This result shows that the increasing ratio is small even if the number of updates is large. The increase ratio of index sizes increases as the number of updates increases.

6.4 Are structural indexes well-behaved as k grow?

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

Figure 9 shows the query time for IA-Structural varying with k. We can see that the query time decreases from $k = 1$ to $k = 2$. While some query times increase when k increases from two. This is because structural index divides paths into too fine granularity for some query templates, and then it takes additional lookup costs. This result implies that a smaller k is better for evaluating *CPQ* whose label sequences is not larger than k.

Figure 10 shows the index size, index time, and memory usage for index construction for IA-structural varying with k, respectively. The index size exponentially increases with increasing k generally. The size of IA-structural does not increase from $k = 3$ to $k = 4$ much. This is because the numbers of length 3 paths and length 4 paths do not increase. The index time and memory usage increase as increasing k index size.

6.5 Discussion

We have given positive answers to all four questions posed at the beginning of this section. Our structural index accelerates query processing by up to three orders of magnitude while being maintainable without increasing index size over the state-of-the-art methods, and is well-behaved as k grows. In query processing, each method has its own advantages for specific query templates and datasets such as T and S for Structural, C2 and C4 for Path, and Ti and Si for TurboHom++. We can select methods depending on which query templates are often posed. Our structural indexes provide the best performance among all the methods for the largest variety of query templates and datasets.

7 CONCLUDING REMARKS

We initiated the study of structural 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. These results provide a positive answer to our main research question: *structural indexing shows clear promise for providing practical help to significantly accelerate CPQ query processing.*

We conclude by highlighting three directions for further study. (1) In practice, edges and vertices can also carry local data (e.g., user vertices might have their names and dates of birth) [7]. We study extensions to our methods for supporting selection and vertices joins. (2) We study methods to control the construction costs adaptively based on the density of graphs for improving scalability. (3) We study query compilation and optimization strategies for *CPQ* and more expressive languages such as *RQ* in the presence of structural indexes, e.g., indexing and optimization for *CPQ* extended with unbounded path navigation via the Kleene star.

REFERENCES

[1] Luca Aceto, Anna Ingolfsdottir, and Jiri Srba. 2011. The algorithmics of bisimilarity. In *Advanced Topics in Bisimulation and Coinduction*. CUP, 100–172.
[2] Renzo Angles et al. 2018. G-CORE: A core for future graph query languages. In *SIGMOD*. 1421–1432.
[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40.
[4] Renzo Angles, Juan L. Reutter, and Hannes Voigt. 2019. Graph query languages. In *Encyclopedia of Big Data Technologies*. Springer.
[5] Sofía Brenes Barahona. 2011. *Structural summaries for efficient XML query processing*. Ph.D. Dissertation. Indiana University, Bloomington.

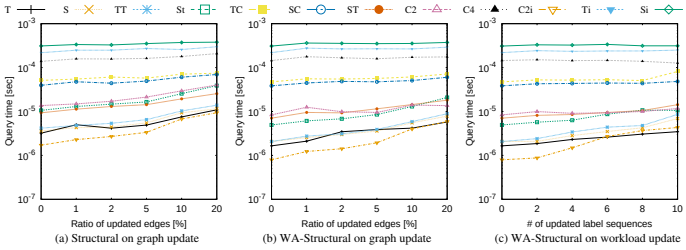

Figure 8: Impact of maintenance to query time on Robots

Table 7: The increasing ratio of index size on update on Robots

Index	Ratio of updated edges				
	1	2	5	10	20
Structural	1.02	1.04	1.11	1.35	1.63
IA-Structural	1.03	1.06	1.13	1.31	1.53
Index	**# of updated label sequences**				
	2	4	6	8	10
IA-Structural	1.002	1.06	1.15	1.24	1.48

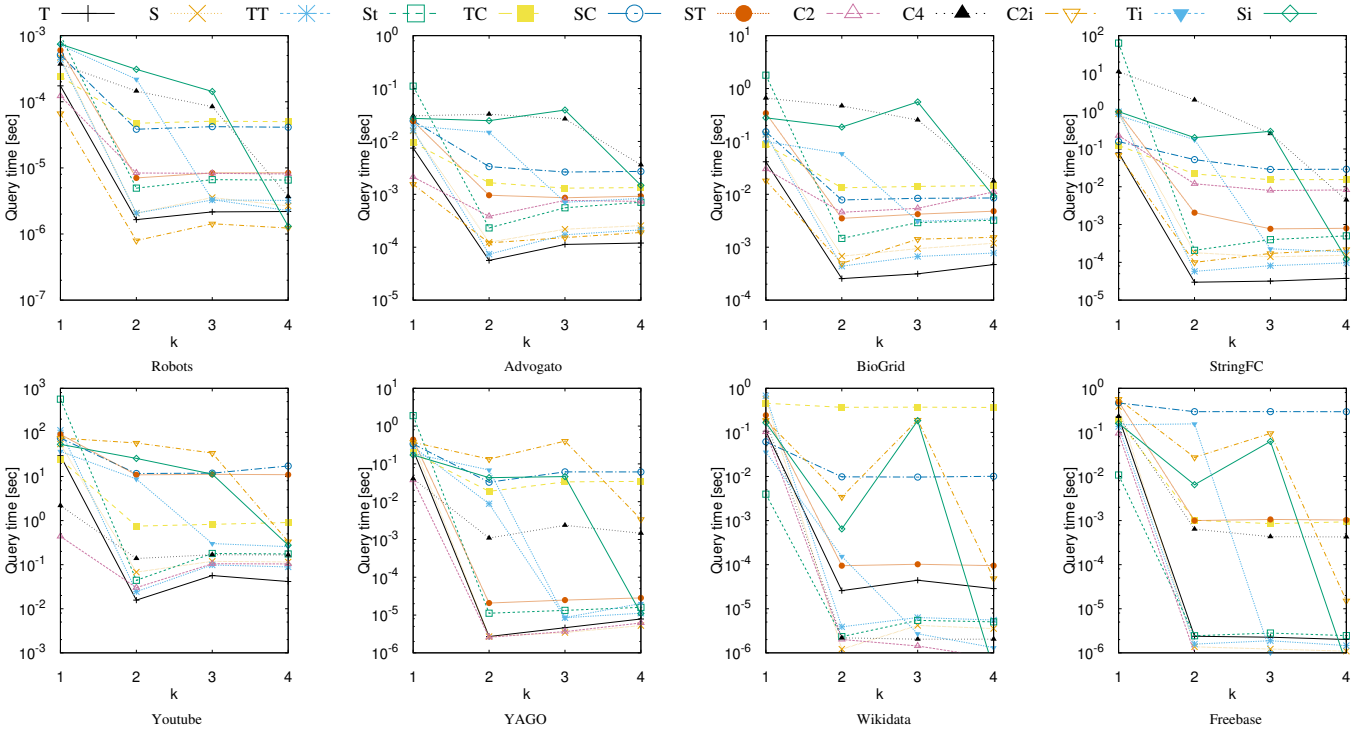

Figure 9: Impact of k on query time with IA-Structural index

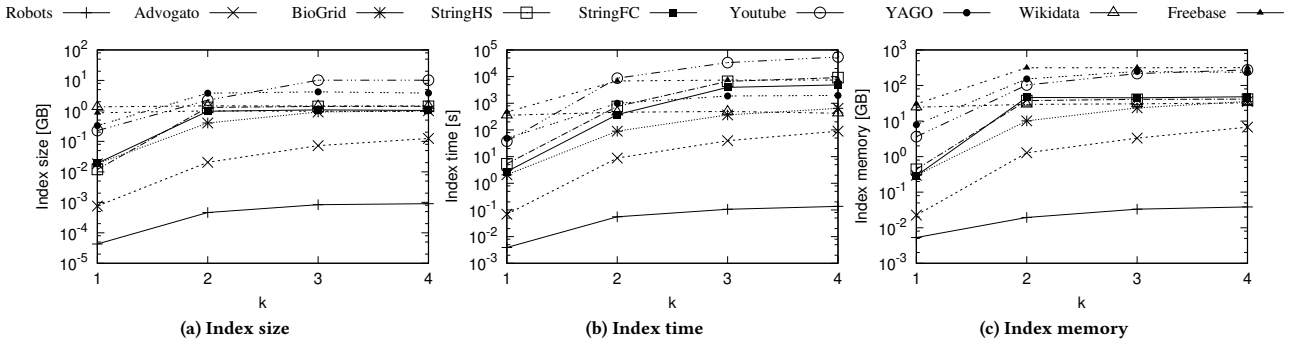

Figure 10: Impact of k on index construction of IA-structural index

[6] Maciej Besta and Torsten Hoefler. 2018. Survey and taxonomy of lossless graph compression and space-efficient graph representations. In *CoRR abs/1806.01799.*
[7] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool.
[8] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *WWW.* 127–138.

[9] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* (2020), 655–679.

[10] Orri Erling and Ivan Mikhailov. 2009. RDF Support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*. 7–24.

[11] Grace Fan, Wenfei Fan, Yuanhao Li, Ping Lu, Chao Tian, and Jingren Zhou. 2020. Extending Graph Patterns with Conditions. In *SIGMOD*. 715–729.

[12] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *SIGMOD*. 157–168.

[13] George Fletcher, Marc Gyssens, Dirk Leinders, Jan Van den Bussche, Dirk Van Gucht, and Stijn Vansummeren. 2015. Similarity and bisimilarity notions appropriate for characterizing indistinguishability in fragments of the calculus of relations. *Journal of Logic and Computation* 25, 3 (2015), 549–580.

[14] George Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *EDBT*. 636–639.

[15] George Fletcher, Dirk Van Gucht, Yuqing Wu, Marc Gyssens, Sofia Brenes, and Jan Paredaens. 2009. A methodology for coupling fragments of XPath with structural indexes for XML documents. *Information Systems* 34, 7 (2009), 657–670.

[16] Roy Goldman and Jennifer Widom. 1997. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*. 436–445.

[17] Gang Gou and Rada Chirkova. 2007. Efficiently querying large XML data repositories: a survey. *IEEE Trans. Knowl. Data Eng.* 19, 10 (2007), 1381–1403.

[18] GSQL. [n. d.]. GSQL query language. https://www.tigergraph.com/gsql/.

[19] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*. 1429–1446.

[20] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. $\text{Turbo}_{\text{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*. 337–348.

[21] Su-Cheng Haw and Chien-Sing Lee. 2011. Data storage practices and query processing in XML databases: A survey. *Knowledge-Based Systems* 24, 8 (2011), 1317 – 1340.

[22] Jelle Hellings, George HL Fletcher, and Herman Haverkort. 2012. Efficient external-memory bisimulation on DAGs. In *SIGMOD*. 553–564.

[23] Paul W Holland and Samuel Leinhardt. 1976. Local structure in social networks. *Sociological methodology* 7 (1976), 1–45.

[24] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. 2002. Covering indexes for branching path queries. In *SIGMOD*. 133–144.

[25] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. 2002. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*. 129–140.

[26] Jinha Kim, Hyungyu Shin, Wook-Shin Han, Sungpack Hong, and Hassan Chafi. 2015. Taming subgraph isomorphism for RDF query processing. *PVLDB* 8, 11 (2015).

[27] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB* 6, 2 (2012), 133–144.

[28] Yongming Luo, George Fletcher, Jan Hidders, Yuqing Wu, and Paul De Bra. 2013. External memory k-bisimulation reduction of big graphs. In *CIKM*. 919–928.

[29] Yongming Luo, François Picalausa, George HL Fletcher, Jan Hidders, and Stijn Vansummeren. 2012. Storing and indexing massive RDF datasets. In *Semantic search over the web*. 31–60.

[30] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network Motifs: Simple Building Blocks of Complex Networks. *Science* 298, 5594 (2002), 824–827.

[31] Tova Milo and Dan Suciu. 1999. Index structures for path expressions. In *ICDT*. 277–295.

[32] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *VLDB J.* 19, 1 (2010), 91–113.

[33] openCypher. [n. d.]. openCypher project. http://www.opencypher.org.

[34] Marcus Paradies and Hannes Voigt. 2019. Graph representations and storage. In *Encyclopedia of Big Data Technologies*. Springer.

[35] You Peng, Ying Zhang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2020. Answering billion-scale label-constrained reachability queries within microsecond. *PVLDB* 13, 6 (2020), 812–825.

[36] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. In *International semantic web conference*. 30–43.

[37] François Picalausa, Yongming Luo, George Fletcher, Jan Hidders, and Stijn Vansummeren. 2012. A structural approach to indexing triples. In *ESWC*. 406–421.

[38] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular queries on graph databases. *Theory Comput. Syst.* 61, 1 (2017), 31–83.

[39] Benjamin Rossman. 2008. Homomorphism preservation theorems. *J. ACM* 55, 3 (2008), 15:1–15:53.

[40] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2019. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal* (Jun 2019).

[41] Dennis E. Shasha, Jason Tsong-Li Wang, and Rosalba Giugno. 2002. Algorithmics and Applications of Tree and Graph Searching. In *PODS*. 39–52.

[42] SQL/PGQ. [n. d.]. Ad Hoc on SQL Extensions Property Graphs. http://www.incits.org/committees/dm32.2-sql-property-graphs.

[43] Jonathan Sumrall, George Fletcher, Alexandra Poulovassilis, Johan Svensson, Magnus Vejlstrup, Chris Vest, and Jim Webber. 2016. Investigations on path indexing for graph databases. In *PELGA*. 532–544.

[44] Thanh Tran and Guenter Ladwig. 2010. Structure index for RDF data. In *Workshop on Semantic Data Management*, Vol. 2.

[45] Lucien D J Valstar, George Fletcher, and Yuichi Yoshida. 2017. Landmark indexing for evaluation of label-constrained reachability queries. In *SIGMOD*. 345–358.

[46] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES*.

[47] Kam-Fai Wong, Jeffrey Xu Yu, and Nan Tang. 2006. Answering XML queries using path-based indexes: a survey. *World Wide Web* 9, 3 (2006), 277–299.

[48] Ömer Nebil Yaveroğlu, Noël Malod-Dognin, Darren Davis, Zoran Levnajic, Vuk Janjic, Rasa Karapandza, Aleksandar Stojmirovic, and Nataša Pržulj. 2014. Revealing the Hidden Language of Complex Networks. *Scientific Reports* 4, 1 (2014), 4547.