Investigations on path indexing for graph databases

Jonathan M. Sumrall[1], George H. L. Fletcher[2], Alexandra Poulovassilis[3], Johan Svensson[1], Magnus Vejlstrup[1], Chris Vest[1], and Jim Webber[1]

[1] Neo Technology, {max.sumrall,johan, magnus.vejlstrup, chris.vest, jim.webber}@neotechnology.com
[2] Eindhoven University of Technology, g.h.l.fletcher@tue.nl
[3] Birkbeck, University of London, ap@dcs.bbk.ac.uk

Abstract. Graph databases have become an increasingly popular choice for the management of the massive network data sets arising in many contemporary applications. We investigate the effectiveness of path indexing for accelerating query processing in graph database systems, using as an exemplar the widely used open-source Neo4j graph database. We present a novel path index design which supports efficient ordered access to paths in a graph dataset. Our index is fully persistent and designed for external memory storage and retrieval. We also describe a compression scheme that exploits the limited differences between consecutive keys in the index, as well as a workload-driven approach to indexing. We demonstrate empirically the speed-ups achieved by our implementation, showing that the path index yields query run-times from 2x up to 8000x faster than Neo4j. Empirical evaluation also shows that our scheme leads to smaller indexes than using general-purpose LZ4 compression. The complete stand-alone implementation of our index, as well as supporting tooling such as a bulk-loader, are provided as open source for further research and development.

1 Introduction

Massive graph-structured data collections are increasingly common in modern application scenarios such as social networks and linked open data. Consequently, there has been a flurry of development of graph database systems to support scalable analytics on massive graphs. The selection and manipulation of *paths* forms the core of querying graph datasets. However, the feasibility of a path-centric approach to indexing massive graphs is an open problem and, to date, no study has been performed on the benefits of path indexing for processing graph queries in industry-strength graph databases. To our knowledge, this work is the first to provide a design and implementation of a path index specifically for graph databases, as well as an empirical study of the performance of such indexes.

Related Work. The study of path indexing has a long history, with a rich variety of strategies developed in the context of object-oriented [3] and XML

[16] databases, and more recently in the indexing of graph data [17]. Related work includes approaches to creating structural summaries of semi-structured data, such as DataGuides [8], T-index [12], AK-index [11] and DK-index [4]. IndexFabric [6] indexes paths in tree-structured data by representing every path in the tree as a string and storing it in a Patricia tree. GraphGrep [14] uses a hash-based method to find occurrences of paths within subgraphs of a graph. For a more detailed review of previous approaches to indexing graph-structured data, we refer the reader to [15]. To our knowledge, the novel approach to path indexing that we present in this paper has not been studied or applied before in the context of any actively supported graph database system.

Contributions. We introduce a path-oriented index for graph-structured data and highlight its benefits for accelerating graph query processing, focusing on the processing of *path queries*. Our index implementation, which is based on the venerable B^+tree data structure, has been custom-built from scratch specifically to be based in external memory and to support and leverage the path structures found in graph datasets. The complete index implementation, as well as supporting tooling such as a bulk-loader, are available open source for further research and development.[4]

We show that use of our index yields, on average, orders of magnitude faster query processing times compared with Neo4j[5], a popular open-source native graph database which offers features such as being fully transactional and supporting a declarative graph query language, Cypher. We stress that our performance studies here compare our standalone index with a fully-fledged graph DBMS. Hence, the performance figures must be interpreted in this light. Nonetheless, the significantly faster query processing times achieved by our index is a clear indication that our solution warrants further investigation towards practical deployment in graph DBMSs. We also highlight the design and benefits of a simple yet highly effective path-centric compression scheme used in our index. We note that, to our knowledge, the proposed approach to path indexing is not found in any current graph database system, and thus the contributions of this paper and their potential for practical impact extend beyond our specific demonstration here by comparison to Neo4j.

Organization. In the next section we define our graph data model and path queries. In Section 3 we describe our path index implementation, including index design, initialization and compression. In Section 4 we present an empirical evaluation of our implementation. We conclude in Section 5 with a summary of our contributions and directions of further work.

2 Graphs and path queries

Data Model. Although modern graph DBMSs such as Neo4j support a richer property graph data model, we restrict our attention to just the path structure of graphs. In particular, we adopt a basic model of finite, edge-labeled, directed

[4] `https://github.com/jsumrall/Path-Index`
[5] `http://neo4j.com/docs/stable/`

graphs $G = \langle N, E, \mathcal{L} \rangle$ where: N is a finite set of nodes; $\mathcal{L}$ is a finite set of edge labels; and $E \subseteq N \times \mathcal{L} \times N$ is a set of labeled directed edges.

Given a graph G, our interest is in indexing *paths* in G. The simplest paths are edges between adjacent nodes. In particular, for each edge $(s, \ell, t) \in E$, we say there is a path of length one from s to t (resp., from t to s) having label ℓ (resp., ℓ^{-1}).[6]

In general, for $k > 0$, let $paths_k(G)$ denote the set of all vectors of nodes $(n_1, \ldots, n_{j+1}) \in \underbrace{N \times \cdots \times N}_{j+1 \text{ times}}$, for $1 \leqslant j \leqslant k$, such that there is a path of length one from n_i to $n_i + 1$ in G, for each $1 \leqslant i \leqslant j$. The *label-path* of a given path $(n_1, \ell_1, n_2), (n_2, \ell_2, n_3), \ldots, (n_j, \ell_j, n_{j+1})$ is the sequence of edge labels $\ell_1 \ell_2 \cdots \ell_j$ along the path.

As an example, consider a graph G_{ex} with node set $\{sue, tom, zoe, chem101\}$ and edge set

$$\{(sue, takesCourse, chem101), (zoe, teacherOf, chem101),$$
$$(tom, takesCourse, chem101), (sue, knows, tom), (tom, knows, zoe)\}.$$

Then there are two distinct paths in G_{ex} of length two from sue to zoe, with respective label-paths $knows \cdot knows$ and $takesCourse \cdot teacherOf^{-1}$.

Queries. We focus on the evaluation of *path queries*, which are specified by projections on label-paths over $\mathcal{L}$. Given a label-path $\bar{\ell} = \ell_1 \ell_2 \cdots \ell_k$ and, for some $r \geq 0$, a list of indices $i_1, \ldots, i_r$ each in the range $[1, k+1]$, the semantics of evaluating $\pi_{i_1,\ldots,i_r}(\bar{\ell})$ on G is the set of all vectors of nodes $(m_1, \ldots, m_r) \in \underbrace{N \times \cdots \times N}_{r \text{ times}}$ such that there is a path $(n_1, \ell_1, n_2), (n_2, \ell_2, n_3), \ldots, (n_k, \ell_k, n_{k+1})$ in G with $m_j = n_{i_j}$ for each $1 \leq j \leq r$.

As an example, the following query selects all node pairs (x, z) such that x takes a course taught by z:

$$\pi_{1,3}(takesCourse \cdot teacherOf^{-1}).$$

It evaluates to the result set $\{(sue, zoe), (tom, zoe)\}$ on the graph G_{ex} above.

Here, we consider the execution of path queries of length at most k, for some fixed k. Compilation strategies for arbitrary graph queries targeting our path indexes is outside the scope of this paper and is a topic of ongoing study. In particular, preliminary work along these lines is reported by Fletcher *et al.* [7] which studies the use of path indexing for accelerating *regular path queries* on graphs.

3 Path Indexing

In this section we describe our path indexing approach, focusing on the requirements, design, initialization and compression of our path indexes. The main

[6] ℓ^{-1} denotes the inverse of edge label ℓ, which we just treat as normal edge label.

objective is to maintain an index on the set $paths_k(G)$ of a graph G, for some fixed k, so as to accelerate the execution of path queries. A secondary goal is to design methods for optimizing the index structure, so as to reduce the overall size of the index and the cost of building, using and updating it. The size of $paths_k(G)$ may exceed the amount of internal memory available and hence the index design must target external memory. For detailed discussion of the design space considered and the design choices made, we refer the reader to [15].

3.1 Index design

Path keys. Given a graph G, our path index maintains an index on the set $paths_k(G)$, for some fixed k. Members of this set need to be represented in a standard fashion, using a scheme such that specific elements of a path can be identified, different paths can be compared to each other, and paths can be serialized. This indexible form of a path is called a *key*.

To make a transformation from label-paths to keys, we first assign an ordering to the elements of $\mathcal{L}$. Under this ordering, we convert each label to an integer value in the range $1, \ldots, |\mathcal{L}|$. As noted above, we also consider the inverse of labels: for a label identified by integer i, the inverse of the label is assigned the value $|\mathcal{L}| + i$. A k-label-path can now be uniquely identified by a vector $(v_1, \ldots, v_k)$ where each v_i is in the range $1, \ldots, 2|\mathcal{L}|$. Based on this vector representation of label-paths, a unique integer is assigned to each label-path: the *label-path's identifier*. These identifiers are stored in a mapping dictionary, implemented using a hash map. During query evaluation, the mapping dictionary is consulted to identify the corresponding identifier for that particular label-path.

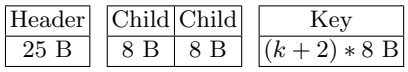
To represent specific paths of G, the sequence of *nodes* along a path must also be considered. Each node is differentiated from all other nodes in the graph by a unique integer identifier (e.g. as generated by the graph DBMS; in the case of Neo4j, this corresponds to the physical address of the node). Concatenating the identifier of a path's label-path with the identifiers of the nodes along the path, a path can be represented as a vector consisting of first its label-path identifier followed by its node identifiers. Therefore our data representation of a key is as a series of integer values, and for a path of length k, the size of the key is $k+1$ integer values (of 8 bytes each).

Storage and search. We use a B$^+$tree [5] as the underlying storage mechanism for keys. This allows keys to be stored and retrieved in sorted order efficiently for large sets of keys which may exceed the amount of internal memory in the system. It also allows for searching using any prefix of the keys stored in the index, e.g. a label-path identifier. Moreover, our path index implementation can also support alternative sort orderings of the paths, which may be desirable for join processing as part of a fully-fledged query processor; further discussion of this is can be found in [15].

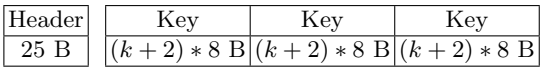
Page design. Our index is designed to be disk-based, and therefore careful attention has been paid to how the bytes of the internal and leaf pages of the index are arranged. All pages contain a header with essential information including sub-tree references and the number of elements in the page. Individual

elements are assumed to be of equal size, and therefore delimiter values between elements are not needed.

Figure 1 details the structure of internal pages and leaf pages. The internal pages contain a 25 byte header, followed by references to children pages, followed by the keys which sort the children pages. Leaf pages contain the 25 byte header, followed by the keys. Since the header contains information about the number of keys in the page, it is possible to directly navigate to specific keys by calculating an offset value based on the size of the keys and the ordered position of the desired key.



(a) Internal Page



(b) Leaf Page

Fig. 1: Layout of the internal pages and leaf pages of the index.

3.2 Index compression

We recall that the first value of a key is a label-path id and the subsequent values are node ids, i.e. a key is of the form $pathID$, $nodeID_1$, $nodeID_2$, $nodeID_3$, ... Within the index, keys are sorted lexicographically, first by $pathID$, then by $nodeID_1$, then by $nodeID_2$, and so on. This ordering causes neighbouring keys to be similar. Indeed, many keys will often have the same values of $pathID$ and $nodeID_1$ in particular, since many neighboring keys have the same label-path ids and the same starting node ids along the path. This is similar to the observation of Neumann and Weikum [13] on efficiently storing RDF triples, and allows for a similar compression scheme. The compression method we use involves not storing the full key, but only storing the difference between successive keys. This results in a high compression, as the change between keys is very often quite small.

For each value in a key, the delta (i.e., integer distance) to obtain this value from the value in the same position in the previous key is calculated. Once each delta is obtained, the minimum number of bytes necessary to store the *largest* delta for this key is found. Each delta is then standardized in length to only that minimum number of bytes. A header byte contains a value representing the size of all these deltas. The largest possible delta would require 8 bytes and the minimum delta we consider is 1 byte.

Often, the prefix of a key can be identical to that of the previous key in the page, while the final value in the key can require a large delta. In the compression scheme above, we allocate a number of bytes to store the large delta, but the

delta for the first few values would be zero. To compress even further, the first 5 bits in the header can be used to signal when the corresponding value has a delta of zero, essentially forming a gap in the series of deltas stored for this key. We call these "gap bits". By enabling a gap bit, we can avoid writing the delta for that value altogether, and only write the values which have a non-zero delta. An illustration of our compressed key structure can be found in Figure 2.

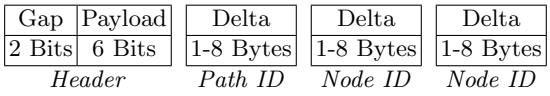


Fig. 2: Structure of a compressed key with gap bits for a path with $k = 1$.

Compression is applied to individual leaf pages, not across pages. Compressing larger portions of the index would produce a smaller index, but at a cost of greater complexity in maintaining the index under updates. By only compressing individual pages, we can still traverse to any leaf page and immediately begin reading keys. If larger portions of the index were compressed together, then those additional portions would need to be fetched and decompressed before beginning to read keys.

Compression is also not applied to pages representing internal pages in the index. Internal pages account for a much smaller share of the total number of pages in the index, as most pages are leaves. Further, we assume that internal pages will be accessed often during traversals, and the additional decompression time on these pages may not justify the possible space savings.

3.3 Index initialization: full vs. workload-based indexing

We have explored two approaches to populating the index. The first is to generate and store all possible paths up to length k. We first perform an external merge sort on the length-1 paths (i.e. the graph's edge set E), and their inverses, and bulk load them into our path index. With the $k = 1$ index constructed, the $k = 2$ index is constructed by performing a merge join on the opposing end nodes of the length-1 paths. In general, the $k = n + 1$ index is constructed by performing a join on two full length-k_1 and length-k_2 indexes, where $k_1 + k_2 = n + 1$ holds true. For large values of k, this requires an extensive time and space commitment, as we see below. The payoff is that the expected query execution time on any arbitrary k-path query will be very low.

As an alternative to this off-line construction of all paths up to length k, it is possible to index on demand (i.e. as a background process during query execution time) only those paths needed to fulfill a given query workload, i.e. to index a finite set of path queries of arbitrary length. Such an index is first initialized with the length-1 paths, i.e. with the graph's edge set E. Then, as encountered in the query workload, longer paths (of arbitrary length) are dynamically built

and added to the index by performing joins on the initial 1-paths and subsequent longer paths which have already been indexed. We refer to the indexes for the first method as *full k indexes* and the latter as *workload-based indexes*.

4 Evaluation

We now describe a set of experiments that investigate our index compression scheme, index sizes, and query execution times using path indexing. All experiments were performed on a 2.0GHz i7 processor with 8 GB of main memory and a solid state drive, running OSX 10.10. Experiments were run on three different datasets, drawn from different sources and of different sizes. Two datasets, the Lehigh University Benchmark (LUBM) dataset [10] and the Linked Data Benchmark Council (LDBC) dataset [2] are synthetic datasets, while the Advogato dataset [1] is a real-world dataset. All experiments were conducted using the latest version of Neo4j available at the time, Neo4j 2.3.0-M01. We focus here on our experiments with the LUBM and refer the reader to [15] for details of the experiments with the other two datasets.

LUBM graphs model a university scenario (e.g., nodes represent universities, departments, students, teachers, ...). We generated a graph with 50 universities, containing approximately 6.8 million unique edges. We followed the same data preparation steps as taken by Gubichev and Then [9], except our dataset was not enriched with inferred facts derived from ontology rules. For example, nodes of type *Associate Professor* do not also get the more general label *Professor*. LUBM is provided with 14 different queries. Here, we use roughly the same queries as used by Gubichev and Then [9], with substitutions for the length-0 queries. Our queries are listed in the Appendix.

4.1 Index compression evaluation

Evaluation of our compression scheme shows that it results in significantly reduced index sizes compared to the uncompressed index size. Further, our compression method outperforms general-purpose LZ4 compression[7] in terms of both speed and scale of compression. A comparison of the size of indexes resulting from each compression technique is shown in Table 1, while a comparison of the speed of the compression techniques is shown in Table 2. The comparison was undertaken by inserting sequentially increasing keys into the index and measuring throughput time and final index size. The evaluation was undertaken for indexes with key sizes of $k = 1, 2, 3$. However, the size of the $k = 3$ index without compression and with the LZ4 algorithm was either too large for our test system or took a significant amount of time. We also include here results for our workload-based index, built using the query workload of the LUBM benchmark, which significantly lowers storage overhead and compression time. Overall, the comparison shows that our scheme outperforms the LZ4 algorithm in terms of both speed and scale of compression.

[7] `https://github.com/jpountz/lz4-java`

Table 1: Index size.

Index	Uncompressed	LZ4	Path Index
$k = 1$	0.16 GB	0.053 GB	0.02 GB
$k = 2$	15.99 GB	3.67 GB	1.69 GB
$k = 3$	-	-	41.58 GB
workload-based	-	-	0.1 GB

Table 2: Indexing time, rounded to the nearest minute.

Index	Uncompressed	LZ4	Path Index
$k = 1$	< 1 Minute	4 Minutes	< 1 Minute
$k = 2$	28 Minutes	266 Minutes	27 Minutes
$k = 3$	-	-	178 Minutes
workload-based	-	-	4 Minutes

4.2 Index size evaluation

The right-most column of Table 1 shows the index sizes. These results show that the size of the index as k increases becomes a limiting factor to the usability of the index. However, while the index sizes may be large, the evaluation time for path queries using the index remains very low (see below). Moreover, although the full indexes can grow to be quite large, the workload-based index has very low overhead while still supporting efficient query processing, as we see next.

4.3 Query execution evaluation

We compare query execution time using our path index with that using Neo4j, subject to the provisos discussed in the Introduction. Only the time needed to retrieve the results is compared for each query: the time needed to open and close the database or index, and to open and close a transaction event is ignored. Six runs were conducted for each query, with each run consisting of 5 executions of the query. Between each run, the system's caches were flushed. The first execution after a cache flush was considered a "cold" run, with empty caches, and the subsequent runs were considered "warm" runs, where caching is likely to result in lower evaluation times. For each query therefore, we obtained 6 "cold run" timings and 24 "warm run" timings. For each set of cold-run and warm-run timings of each query, we excluded 10% of the data from each end of the range of recorded results, eliminating outliers due to nondeterminism in the runtime environment. We report here the mean of the remaining values, focusing on the warm run experiments. A full analysis, including both the warm and the cold runs, can be found in [15].

Full indexes. We first consider path query performance on a full $k = 3$ index. Results are reported in Table 3, where we give the time to the first result and the time to the last result. For both Neo4j and our path index, the time to the first result is measured as the time from immediately before Neo4j's or the

Table 3: Average times (ms) to retrieve the first result and the last result in Neo4j and in the Path Index.

		Neo4j	**Index $k=3$**	**Index $k=2$**	**Index $k=1$**	**Speedup**
Q1	First Result	480	-	-	0.19	2526x
	Last Result	2080	-	-	37	56x
Q2	First Result	2014	-	-	1	2014x
	Last Result	2014	-	-	1	2014x
Q3	First Result	413	-	-	0.05	8260x
	Last Result	1352	-	-	4	338x
Q4	First Result	774	-	0.8	173	967x
	Last Result	3741	-	112	10932	33x
Q5	First Result	457	-	2	45	228x
	Last Result	13303	-	1439	4645	9x
Q6	First Result	437	-	2	47	218x
	Last Result	2225	-	107	2831	20x
Q7	First Result	8	2	2.4	-	4x
	Last Result	2221	32	179	-	69x
Q8	First Result	1	1	2	-	1x
	Last Result	5319	1992	493	-	2x
Q9	First Result	1	2	2	-	0.5x
	Last Result	1378	8	179	-	172x
Q10	First Result	1	3	2	-	0.3x
	Last Result	1392	4	16	-	348x
Avg	First Result	458	2	1	< 1	1444x
	Last Result	3502	509	552	14	306x

path index's *find* operation is executed, and the time immediately after the first result is found. The time to the last result is measured as the time immediately before Neo4j's or the path index's *find* operation is executed, until the time immediately after the last result is found.

In addition to using the full-length k-paths in the index, queries are also evaluated using the $(k-1)$-paths in the index for Queries 4-10, for comparison purposes. For example, looking at Query 7 in Table 3, we see under the column labeled "Index $k=2$" the time needed to evaluate Query 7 using the $k=2$ and $k=1$ subpaths of the query and joining the results (using a merge join). This gives us an indication of query evaluation times if the index only contained the smaller subpaths and not the full $k=3$ path. The column "Index $k=1$" for Query 7 is blank, as these experiments only show the times needed to perform a single (merge) join to evaluate a given query. Evaluating Query 7 using only the $k=1$ paths is possible, but would require joining two subpaths first, and then undertaking a sort merge join with the third subpath or performing a hash join with the third subpath.

Table 4: Workload experiment with paths constructed from the $k = 1$ index with joined results inserted into the index (average time to last result, in ms).

	Query Plan	**Index Construction**	**Index Query**	**Neo4j**	**Speed up**
Q4	takesCourse $\bowtie$ teacherOf^{-1}	30289	119	3741	31x
Q5	memberOf $\bowtie$ subOrganizationOf^{-1}	129499	775	13303	17x
Q6	memberOf $\bowtie$ subOrganizationOf	11113	39	2225	57x
Q7A	undergraduateDegreeFrom $\bowtie$ Query 6^{-1}	769	< 1	2221	2221x
Q7B	P_{7B} = subOrganizationOf^{-1} $\bowtie$ memberOf^{-1}				
	undergraduateDegreeFrom $\bowtie$ P_{7B}	15832	< 1	2221	2221x
Q8A	hasAdvisor $\bowtie$ Query4^{-1}	836	2	5319	2659x
Q8B	P_{8B} = teacherOf $\bowtie$ takesCourse^{-1}				
	hasAdvisor $\bowtie$ P_{8B}	2703	2	5319	2659x
Q9	P_9 = worksFor $\bowtie$ subOrganizationOf^{-1}				
	headOf^{-1} $\bowtie$ P_9	8807	2	1378	689x
Q10	P_{10} = worksFor $\bowtie$ subOrganizationOf				
	headOf^{-1} $\bowtie$ P_{10}	822	< 1	1392	1392x
Avg		22296	104	4124	1327x

Workload-based indexes. Experiments were also conducted on workload-based indexes built at runtime, where the necessary $k = 1$ paths are joined to form the paths of length 2 in the queries, or joined a third time to form the paths of length 3. Table 4 shows the cost of building and using the workload-based indexes. The alternatives A/B for Queries 7 and 8 arise from whether or not to reuse paths already constructed in the index from previous query evaluations.

Summary. The above results demonstrate that both full and workload-based path indexes have much lower evaluation times for all path queries compared to Neo4j. Our experiments on the LDBC and Advogato datasets confirm and further strengthen the results reported here, for both warm and cold runs (see [15] for details).

5 Concluding remarks

This paper has presented a new and simple path indexing approach to improve path query performance for graph database systems. Our empirical study has demonstrated the significant potential of path indexes for graph databases. Keeping in mind that Neo4j is a fully-fledged graph DBMS, our experiments show that, for every query trialled, path indexing provides a non-trivial, often multiple orders of magnitude, improvement in query evaluation time. We have demonstrated the practicality of workload-driven path indexes, where the additional time to first evaluate and store the results of a path query is relatively large, but subsequent query times using the index provide significant speedups, amortizing the index build cost over the lifetime of the query workload. Furthermore, our

workload-based indexes are an order of magnitude smaller than the full index. Our implementation includes supporting tools, e.g. for bulk loading the index with paths from the graph in an efficient way. As indicated in the Introduction, the complete codebase is available open-source for further study.

Our empirical results show that workload-based indexing offers the most promise in terms of index size, index construction time, and query performance. Further study of the design, engineering, and deployment in practical graph database systems of these types of indexes is the natural progression of this work. Additional experiments need to be conducted to identify how to best build the index based on encountered queries. Possibilities include examining query logs and building indexes based on frequent queries. Study of index maintenance under mixed transactional workloads is another interesting direction of future study, i.e. policies for updating the path indexes in the face of insertions and deletions of edges in the data graph. Efficient index updates may be achieved by supporting multiple indexes, supporting fast retrieval for multiple dimensions of label-paths. Finally, another important direction for future research is compilation strategies for richer query languages such as Cypher targeting our path indexes as one of the alternative access paths available in the DBMS.

References

1. Advogato network dataset – KONECT, October 2014.
2. Renzo Angles et al. The linked data benchmark council. *ACM SIGMOD Record*, 43(1):27–31, May 2014.
3. Elisa Bertino et al. Object-oriented databases. In Elisa Bertino et al, editor, *Indexing Techniques for Advanced Database Systems*, pages 1–38. Kluwer, 1997.
4. Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index. In *Proc. SIGMOD '03*, page 134, San Diego, California, USA, June 2003. ACM Press.
5. Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.
6. Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A Fast Index for Semistructured Data. In *Proc. VLDB '01*, pages 341–350, Roma, Italy, September 2001. Morgan Kaufmann Publishers Inc.
7. George H. L. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. Efficient regular path query evaluation using path indexes. In *Proc. EDBT'16*, pages 636–639, 2016.
8. Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB '97*, pages 436–445, Athens, Greece, August 1997. Morgan Kaufmann Publishers Inc.
9. Andrey Gubichev and Manuel Then. Graph pattern matching – do we have to reinvent the wheel? In *Proc. GRADES'14*, pages 1–7, 2014.
10. Yuanbo Guo et al. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semantics*, 3(2-3):158–182, 2005.
11. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. ICDE'02*, pages 129–140, San Jose, California, USA, 2002. IEEE Comput. Soc.
12. Tova Milo and Dan Suciu. Index Structures for Path Expressions. In *Proc. ICDT'99*, pages 277–295, Jerusalem, Israel, January 1999. Springer-Verlag.

13. Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, September 2009.
14. Dennis Shasha, Jason T. L. Wang, and Rosalba Giugno. Algorithmics and applications of tree and graph searching. In *Proc. PODS '02*, page 39, Madison, Wisconsin, June 2002. ACM Press.
15. Jonathan Sumrall. Path indexing for efficient path query processing in graph databases. Master's thesis, Eindhoven University of Technology, 2015.
16. Kam-Fai Wong, Jeffrey Xu Yu, and Nan Tang. Answering XML queries using path-based indexes: A survey. *WWW J*, 9(3):277–299, 2006.
17. Xifeng Yan and Jiawei Han. Graph indexing. In Charu C. Aggarwal and Haixun Wang, editors, *Managing and Mining Graph Data*, pages 161–180. Springer, 2010.

Appendix: LUBM Cypher Queries

```
Q1: MATCH (x)-[:memberOf]->(y) RETURN ID(x),ID(y)

Q2: MATCH (x)-[:memberOf]->(y)
    WHERE x.URI = "http://www.Department0...Student207"
    RETURN ID(x), ID(y)

Q3: MATCH (x)-[:worksFor]->(y) RETURN ID(x),ID(y)

Q4: MATCH (x)-[:takesCourse]->(y)<-[:teacherOf]-(z)
    RETURN ID(x),ID(y),ID(z)

Q5: MATCH (x)-[:memberOf]->(y)<-[:subOrganizationOf]-(z)
    RETURN ID(x),ID(y),ID(z)

Q6: MATCH (x)-[:memberOf]->(y)-[:subOgranizationOf]->(z)
    RETURN ID(x),ID(y),ID(z)

Q7: MATCH (x)-[:undergraduateDegreeFrom]->(y)
           <-[:subOrganizationOf]-(z)<-[:memberOf]-(x)
    RETURN ID(x),ID(y),ID(z)

Q8: MATCH (x)-[:hasAdvisor]->(y)-[:teacherOf]->(z)<-[:takesCourse]-(x)
    RETURN ID(x),ID(y),ID(z)

Q9: MATCH (x)<-[:headOf]-(y)-[:worksFor]->(z)<-[:subOrganisationOf]-(w)
    RETURN ID(x),ID(y),ID(z),ID(w)

Q10:MATCH (x)<-[:headOf]-(y)-[:worksFor]->(z)-[:subOrganisationOf]->(w)
    RETURN ID(x),ID(y),ID(z),ID(w)
```