Compressed Graph Representations for Evaluating Regular Path Queries*

Gonzalo Navarro and Josefa Robert

Millennium Institute for Foundational Research on Data (IMFD)
Department of Computer Science, University of Chile, Chile

Abstract. Regular Path Queries (RPQs) are at the core of graph database query languages like SPARQL. They consist, essentially, of regular expressions that must match the sequence of edge labels of paths in the database graph. A way to answer them is to traverse the graph and the automaton of the RPQ in synchronization, reporting the graph nodes where the automaton reaches final states. We implement this approach on top of a compact graph representation that is particularly well suited for this task. The result is an index using considerably less space and/or query time than all existing approaches.

1 Introduction and Related Work

Graph databases offer a powerful way to model relationships within data, making them particularly useful in fields like social networking, bio-informatics, linguistics, and recommendation systems. This article focuses on directed labeled graph databases, where relationships have both direction and labels, and regular path queries (RPQs) over them, which search for paths of arbitrary length whose sequence of labels matches a given regular expression.

Consider, for example, the toy RDF [21] database from Fig. 1, which depicts a network of academics using relations of the form $x \xrightarrow{\text{expr}} y$, indicating that x is related to y by the relation expr. Then, a query like $(\text{Grace}, \text{coauthorOf}+, x?)$ asks for direct and transitive collaborators of Grace ($x = \text{Dan}$ and $x = \text{Eve}$).

RPQs are especially useful in scenarios where either the path's structure or length is not known in advance, or for studying the graph's topology, and represent a classic challenge in the field. RPQs can be extended to traverse edges in both directions, yielding what is known as two-way RPQs or 2RPQs. A central feature of SPARQL, the standard query language for RDF databases, are property path queries, which are a slight extension of 2RPQs. With the widespread adoption of SPARQL, (2)RPQs have become a popular feature [2]: out of 208 million SPARQL queries in the public logs from the Wikidata Query Service [20], about 24% use at least one RPQ feature [11]. Subsequence efforts, like like PGQL [26], Cypher [15], G-CORE [1], TigerGraph [13], and GQL [12], also support RPQ-like features.

* Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1-230755

Subject	Predicate	Object
Alice	mentored	Bob
Alice	cited	Alice
Alice	cited	Dan
Bob	refereedFor	Dan
Dan	cited	Alice
Dan	coauthorOf	Grace
Dan	coauthorOf	Eve
Dan	cited	Bob
Eve	cited	Grace
Eve	mentored	Grace
Eve	mentored	Dan
Eve	cited	Bob
Eve	coauthorOf	Dan
Grace	refereedFor	Alice
Grace	coauthorOf	Dan

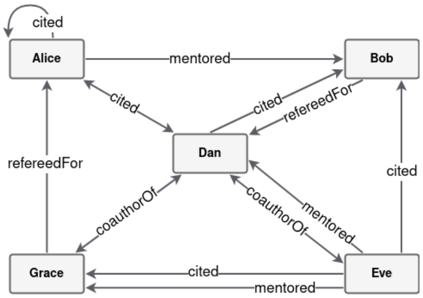


Fig. 1. The table on the left describes an RDF database of researchers and their relationships, while the graph on the right is its graphical representation.

An algorithmically sound approach to evaluate (2)RPQs [30] is to represent the regular expression of the RPQ as a finite automaton and use it to traverse the database graph in synchronization with the automaton. Formally, this is regarded as traversing a virtual “product graph” [22], whose nodes are the Cartesian product of the automaton and the graph nodes. Many systems that handle property path queries in SPARQL, instead, extend the relational algebra to support computing transitive closures, and then translate RPQs into (extended) relational algebra operations.

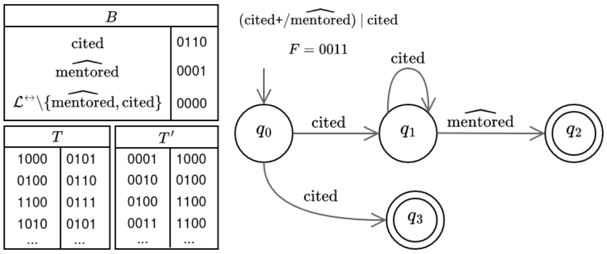
With big and growing available database graphs, indexing them within compact space is of interest in order to preserve memory space, increasing the chances of solving queries in main memory, which is much faster than the disk. While classic systems that solve graph queries, like Virtuoso [14] and Blazegraph [28], use 60 to 90 bytes per graph edge (bpe), recent research handles the most important queries within much less space. The Ring [3, 4], for example, handles the core SPARQL queries, including RPQs, within 16 bpe; a faster version called Ring$_{\text{AB}}$ uses 28 bpe. An even more recent development based on Boolean matrices [5, 6] uses as little as 4.3 bpe, though it is markedly slower than the Ring.

In this paper we build on the same product-graph approach as the Ring, but halve its space while sharply improving its speed on the most popular RPQs, where one end is fixed. Instead of using the Ring graph representation [8], which was developed with the aim of solving basic graph patterns (the other main kind of graph database queries), we resort to a compact representation of labeled graphs that had not been implemented so far [24, Sec. 9.1.4]. Ours is also, arguably, a more natural representation of the database graph, which may simplify the implementation of other graph algorithms (e.g., traversals) on it.

2 Basic Concepts

2.1 Labeled Graphs and Regular Path Queries

Let V and $\mathcal{L}$ be finite sets of nodes and labels, respectively. We assume these sets have been already integer-encoded as $V = \{1, \ldots, n\}$ and $\mathcal{L} = \{1, \ldots, \lambda\}$. A



B	
cited	0110
$\widehat{\text{mentored}}$	0001
$\mathcal{L}^{\leftrightarrow}\backslash\{\widehat{\text{mentored}}, \text{cited}\}$	0000

T	
1000	0101
0100	0110
1100	0111
1010	0101
...	...

T'	
0001	1000
0010	0100
0100	1100
0011	1100
...	...

(cited+/$\widehat{\text{mentored}}$) | cited
$F = 0011$
q_0
cited
q_1
cited
$\widehat{\text{mentored}}$
q_2
cited
q_3

Fig. 2. Glushkov automaton for the regular expression $($`cited+/.`$\widehat{\texttt{mentored}})|$`cited`.

directed edge-labeled graph G is a finite set containing e triples $(s, p, o) \in V \times \mathcal{L} \times V$ representing the graph edges $s \xrightarrow{p} o$ from vertex s to vertex o with label p. In the RDF model [21], s is called a *subject*, p a *predicate*, and o an *object*.

A *path* ρ from a node x_0 to node x_k in G is a string $x_0 p_1 x_1 \cdots x_{k-1} p_k x_k$ such that $(x_{i-1}, p_i, x_i) \in G$ for $1 \leq i \leq k$. Given a path ρ, we denote $\mathsf{word}(\rho) = p_1 \cdots p_k$ the string labeling path ρ. Two-way RPQs (2RPQs) also allow traversing reversed edges. Hence, we define the set of inverse labels as $\hat{\mathcal{L}} = \{\hat{p} : p \in \mathcal{L}\}$, and let $\mathcal{L}^{\leftrightarrow} = \mathcal{L} \cup \hat{\mathcal{L}}$ be the set of predicates and their inverses. We define the *inverse graph* as $\hat{G} = \{(y, \hat{p}, x) : (x, p, y) \in G\}$, and its *completion* as $G^{\leftrightarrow} = G \cup \hat{G}$. A *two-way regular expression* (2RE) is then formed from the following rules: ε is a 2RE; if $c \in \mathcal{L}^{\leftrightarrow}$, then c is a 2RE; if E, E_1 and E_2 are 2REs, then so are E^* (Kleene star), E_1/E_2 (concatenation), and $E_1 \mid E_2$ (disjunction). If E is a 2RE, we abbreviate E^*/E as E^+ and $\varepsilon|E$ as $E^?$.

The *language* $L(E)$ of a 2RE E over the alphabet of terminals $\mathcal{L}$ is the language of the regular expression E over $\mathcal{L}^{\leftrightarrow}$. We say that a path ρ *matches* a 2RE E if $\mathsf{word}(\rho) \in L(E)$. A *two-way regular path query*, or 2RPQ for short, is a query of the form (x, E, y), which has as solution all the pairs of nodes (s, o) such that there is a path $\rho = s p_1 \ldots p_k o$ in $G^{\leftrightarrow}$ where $\mathsf{word}(\rho) \in L(E)$; x and/or y can be constants (thus fixing the value of s and/or o, respectively), or variables.

2.2 Glushkov Automata

Let R be a regular expression on m symbols. The *Glushkov automaton* [16, 10] of R is a particular non-deterministic finite automaton (NFA) $A_R = (Q, \mathcal{L}_R, \Delta, q_0, F)$ recognizing the language of R. It satisfies, in particular, that (i) it has no ε-transitions; (ii) it has exactly $m + 1$ states, that is, $|Q| = m + 1$; (iii) all the transitions arriving at a state have the same label. The automaton can be constructed in $O(m^2)$ time and uses $\Theta(m^2)$ space. As an example, consider the set of terminals formed by the predicates in Fig. 1. The regular expression $($`cited+/`$\widehat{\texttt{mentored}})|$`cited` is represented by the Glushkov automaton of Fig. 2.

The properties of the Glushkov automaton allow for an efficient, bit-parallel processing [25, 4], which uses the following data structures:

- D, a word containing $m+1$ bits, which represents a non-deterministic state of A_R, *i.e.*, a subset of Q,
- $B[1..\sigma]$, a table containing bitvectors of length $m+1$, in which $B[c]$ indicates with 1s the states targeted by transitions labeled c,
- $T[0..2^{m+1}-1]$, a table storing in $T[X]$, where X is a $(m+1)$-bit argument representing a subset of Q, the set of states reachable from X in one step using any symbol.

We use the symbol & to denote the bit-wise and operator. Then, it can be shown [25] that $T[D]$ & $B[c]$ is the bitvector of length $m+1$ indicating the states of Q that are reachable from state D using transitions labeled c. The *forward* traversal of A_R, which recognizes strings in $L(R)$ prefixing a sequence, starts with $D \leftarrow 2^{m+1}$ (which activates the initial state q_0), and then repeatedly (i) recognizes the word seen so far if D & $F \neq 0$ (*i.e.*, a final state was reached), (ii) finishes if $D = 0$ (*i.e.*, there are no active states in the NFA), and (iii) updates $D \leftarrow T[D]$ & $B[c]$ with the next symbol c.

It is also possible to recognize R while reading the input in reverse. For this, we first build a table $T'[0..2^{m+1}-1]$ such that $T'[X]$ marks with 1s the states that reach some state in X in one step. Then, we activate the final states with $D \leftarrow F$ and, repeatedly, (i) recognize the word seen so far if D & $2^{m+1} \neq 0$, (ii) finish if $D = 0$, and (iii) update $D \leftarrow T'[D$ & $B[c]]$ with the next symbol c.

The data structures use $O(2^m + |\mathcal{L}_R|)$ bits. Precomputing B and T (or T') takes $O(2^m)$ time. Fig 2 shows the tables for our example.

2.3 The Product-Graph Approach and its Ring Implementation

Data structures. The Ring [4, 7] represents the completion $G^{\leftrightarrow}$ (not G) using the sequences L_p, C_o, L_s, and C_p defined next. Let us write $G^{\leftrightarrow} = \{(s_i, p_i, o_i) : 1 \le i \le 2e\}$ in such a way that the edges are lexicographically sorted, that is, first by o, in case of ties by s, and still in case of ties by p (we call this order OSP), and define the array $L_\mathrm{p} = p_1, \ldots, p_{2e}$. Observe that L_p can be written as a concatenation of segments $L_\mathrm{p}^1 \cdot L_\mathrm{p}^2 \cdots L_\mathrm{p}^n$, where L_p^o contains those p_i such that $o_i = o$. We define the bit-vector $C_\mathrm{o} = 10^{|L_\mathrm{p}^1|}10^{|L_\mathrm{p}^2|} \cdots 10^{|L_\mathrm{p}^n|}$. Suppose now that $G^{\leftrightarrow}$ is sorted in order POS, and let $L_\mathrm{s} = s_1, \ldots, s_{2e}$. We can similarly write $L_\mathrm{s} = L_\mathrm{s}^1 \cdots L_\mathrm{s}^\lambda$ and define $C_\mathrm{p} = 10^{|L_\mathrm{s}^1|} \cdots 10^{|L_\mathrm{s}^\lambda|}$.

Solving RPQs. The Ring reduces double-variable queries (x, R, y) to solving the single-variable queries (x, R, o) for every node o. It also reduces queries of the form (s, R, x) to the case $(x, \hat{R}, s)$, where $\hat{R}$ is the reversed regular expression.

To solve an RPQ (y, R, o) with variable y, they first build the bit-parallel Glushkov automaton A_R for the regular expression R, as shown in Section 2.2. They then conceptually traverse the *product graph* $A_R \times G^{\leftrightarrow}$, starting from (F, o). This is equivalent to simultaneously traversing A_R and $G^{\leftrightarrow}$, starting from the final states of A_R and from the node $o \in G^{\leftrightarrow}$. Each step consists of three parts:

1. Identify the predicates associated with the current object o (*i.e.*, labeling arrows that arrive at or leave from o), which are the elements of L_p^o.

2. For each identified predicate p, feed A_R with p in reverse traversal mode, and abandon this branch if A_R becomes out of active states.
3. Determine the subjects s of edges of the form $(s, p, o) \in G^{\leftrightarrow}$, which are the elements of L_{s}^{p}. Report s if the initial state of A_R is active.
4. Interpret s as an object o, and create a new branch that starts from step 1.

In addition, graph nodes are marked with the active states of A_R they have been already visited with, to avoid loops in the traversal of the product graph.

The Ring represents L_{p} and L_{s} using wavelet trees [19]. This allows obtaining the predicates and subjects in steps 1 and 3 in batch using its so-called backward search. Further, it can efficiently process ranges of symbols in steps 1 and 3. Finally, they can integrate steps 1 and 2 so as to directly produce the predicates p that are associated with o *and* lead to active states in A_R, in a time they prove to be optimal in terms of computing the intersection of both sets.

A variant called $\mathrm{Ring}_{\mathrm{AB}}$ [4] uses more space but improves the time, by starting the traversal from a "split" point of R, not necessarily from its end, and traversing $G^{\leftrightarrow}$ in both direction from the split point. If the split point is an uncommon predicate, much fewer nodes need be visited.

3 Our Graph Representation

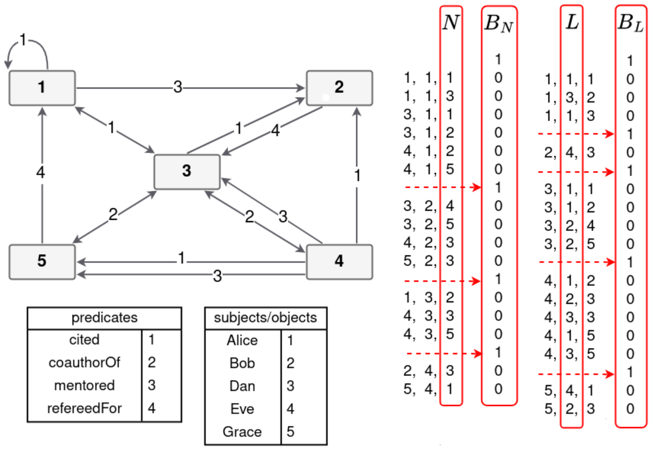
Data structures. Our approach represents G (not $G^{\leftrightarrow}$) using sequences L, B_L, N, and B_N, which we define as follows. We sort $G = \{(s_i, p_i, o_i) : 1 \le i \le e\}$ with order SOP, and define the array $L = p_1, \ldots, p_e$. Array L is the concatenation of segments $L_1 \cdot L_2 \cdots L_n$, where L_s contains those p_i such that $s_i = s$. We encode the lengths of the subarrays L_s in the bitvector $B_L = 10^{|L_1|} \ldots 10^{|L_n|}$. The sequences N and B_N are defined similarly: we sort G with order PSO and define the array $N = o_1, \ldots, o_e$, which can also be decomposed as $N = N_1 \cdots N_\lambda$, with N_p containing those o_i such that $p_i = p$. Finally, we set $B_N = 10^{|N_1|} \ldots 10^{|N_\lambda|}$.

Fig. 3 depicts the sequences N, L, B_L, and B_N for the database obtained by integer-encoding the example from Fig. 1. While our structures already exist [24, Sec. 9.1.4], our presentation here has the added value of exposing their relation with the Ring representation [4]: we store predicates with order SOP in L and objects in order PSO in N; the Ring stores predicates in order OSP in L_{p} and subjects in order POS in L_{s}. The ring doubles our space because it stores the reverse edges (with reverse labels) explicitly.

We utilize plain bitvectors [23] to store B_L and B_N, and GMR-arrays [18] to store L and N. These choices allow storing bitvectors B using $(1 + o(1))|B|$ bits, and general arrays A with values in the range $[1\,..\,k]$ using $(1 + o(1))|A| \log |k|$ bits. Therefore, our representation of G uses $(1 + o(1))(e \log(\lambda n) + n + \lambda)$ bits, which, unlike the space of the Ring, is asymptotically optimal.

Note that we do not explicitly store the triples of G. To recover the original triples and other useful information about the graph, we can use RSA operations on the sequences L, B_L, N, and B_N. These are defined for any array A as follows:

- $\mathsf{access}(A, i)$ returns the letter $A[i]$, for any $1 \le i \le n$.



predicates	
cited	1
coauthorOf	2
mentored	3
refereedFor	4

subjects/objects	
Alice	1
Bob	2
Dan	3
Eve	4
Grace	5

Fig. 3. On the left, the integer-encoded version of the graph from Fig. 1, along with its translation to the original terms. On the right we show how N is derived from the triples in order PSO. The arrows highlight the limits of the segments N_p and point to the 1s in B_N. The equivalent diagram for L and B_L uses the triples in order SOP.

- $\mathsf{rank}_a(A, i)$ returns the number of occurrences of a in $A[1..i]$, for any $1 \leq i \leq n$. We assume $\mathsf{rank}_a(A, 0) = 0$.
- $\mathsf{select}_a(A, j)$ returns the position of the j-th occurrence of a in A, for any $j \geq 0$. We assume $\mathsf{select}_a(A, 0) = 0$ and $\mathsf{select}_c(A, j) = n + 1$ if $j > \mathsf{rank}_a(A, n)$.

Our data structure choices yield constant time RSA operations for B_L and B_N [23], constant time select for L and N, $O(\log \log n)$ time rank and access for N, and $O(\log \log \lambda)$ time rank and access for L [18].

Retrieving graph edges. We now detail how to recover the triple associated to a position in L or in N. If $(s, o, p) \in G$ is the triple corresponding to position i of L, then $p = \mathsf{access}(L, i)$, whereas s and o can be recovered as follows:

$$s = \mathsf{select}_0(B_L, i) - i;\;\; q = \mathsf{select}_1(B_N, p) - p;\;\; j = q + \mathsf{rank}_p(L, i);\;\; o = N[j]. \quad (1)$$

Similarly, if $(s, o, p) \in G$ is the triple corresponding to position j of N, then $o = N[j]$, whereas p and s can be recovered as follows:

$$\begin{aligned} &p = \mathsf{select}_0(B_N, j) - j;\;\; r = \mathsf{select}_1(B_N, p) - p;\;\; i = \mathsf{select}_p(L, j - r); \quad (2)\\ &s = \mathsf{select}_0(B_L, i) - i. \end{aligned}$$

We note that formulas (1) and (2) resemble the way the Ring recovers its triples using forward and backward direction, respectively [3].

Operations on the structure. The representation supports several useful queries on the graph. We describe next those we use to solve 2RPQs. We call $G_p = \{(s, p, o) \in G\}$ the triples of G labeled p.

- $\mathsf{neigh}_p(G, s)$ returns all neighbors of s in G_p, *i.e.*, $\{o : (s, p, o) \in G\}$.
- $\mathsf{rneigh}_p(G, o)$ returns all reverse neighbors of o in G_p, *i.e.*, $\{s : (s, p, o) \in G\}$.
- $\mathsf{sources}_p(G)$ returns the nodes s that are the origin of an edge with label p, *i.e.*, $\{s : \exists o, (s, p, o) \in G\}$.
- $\mathsf{targets}_p(G)$ returns the nodes o that are the target of an edge with label p, *i.e.*, $\{o : \exists s, (s, p, o) \in G\}$.

We now describe the algorithms for neigh_p, rneigh_p, $\mathsf{sources}_p$, and $\mathsf{targets}_p$.

The neighbors of s in G_p are exactly the elements o in the segment $N_{p,s}$ of N_p that corresponds to the edges $(s, p, o) \in G$. To identify $N_{p,s}$, we first note that the beginning of the segment N_p in N is $r = \mathsf{select}_1(B_N, p) - p$. Then, we get the offset at which the objects associated with s begin by counting the number of edges whose predicate is p and whose subject is some $s' < s$. This is done with

$$q_1 = \mathsf{rank}_p(L, l_1), \text{ where } l_1 = \mathsf{select}_1(B_L, s) - s.$$

Similarly, the start of $N_{p,s+1}$ in N_p is at

$$q_2 = \mathsf{rank}_p(L, l_2), \text{ where } l_2 = \mathsf{select}_1(B_L, s + 1) - (s + 1).$$

Since, in N_p, the subsegment $N_{p,s+1}$ immediately follows $N_{p,s}$, it follows that

$$\mathsf{neigh}_p(G, s) = N_p[q_1 + 1 \mathinner{.\,.} q_2] = N[r + q_1 + 1 \mathinner{.\,.} r + q_2].$$

This also yields that $\mathsf{outdegree}_p(s)$, the outdegree of s in G_p, is equal to $q_2 - q_1$. Overall, we compute any element of $\mathsf{neigh}_p(G, s)$ in time $O(\log \log n)$.

To obtain the reverse neighbors of o in G_p, we recall that $N_p = N[r_1 + 1 \mathinner{.\,.} r_2]$, where $r_1 = \mathsf{select}_1(B_N, p) - p$ and $r_2 = \mathsf{select}_1(B_N, p+1) - (p+1)$. The number of reverse neighbors of o in G_p is then $\mathsf{indegree}_p(G, o) = \mathsf{rank}_o(N, r_2) - \mathsf{rank}_o(N, r_1)$. Moreover, the position j in N of the k-th reverse neighbor of o in G_p is

$$j = \mathsf{select}_o(N, l), \text{ where } l = \mathsf{rank}_o(N, r_1) + k.$$

Formula (2) then yields the corresponding subject s, which is then the kth element of $\mathsf{rneigh}_p(G, v)$, computed in total time $O(\log \log n)$.

To implement $\mathsf{sources}_p(G)$, we start by computing the position i of the first occurrence of p in L, with $i = \mathsf{select}_p(L, 1)$. We then report the subject s associated to position i of L using formula (1). To avoid adding duplicates (which arise if the same subject is connected to two different objects by the same label), we move forward until the end of L_s, which is at position $q = \mathsf{select}_1(B_L, s + 1) - (s + 1)$. Note that the last occurrence of p in L_s is the $\mathsf{rank}_p(L, q)$-th one in L. We iterate by looking for the next occurrence of p (at position $l = \mathsf{select}_p(L, \mathsf{rank}_p(L, q)+1)$) until we reach the end of L. The cost is $O(\log \log \lambda)$ per element reported.

It rests to implement $\mathsf{targets}_p(G)$. Note that the objects of the edges with label p are the elements of N_p. So, it is enough to collect the elements in $N_p = N[r_1 + 1 \mathinner{.\,.} r_2]$ and then remove the duplicates with, for example, integer sorting. The total time is $O(|G_p| \log \log n)$.

4 Solving RPQs on our Representation

4.1 Single-variable 2RPQs

We start by discussing a symmetry in the queries. A pair (s, o) is a solution for the 2RPQ (s, R, x) iff (o, s) is a solution for $(y, \hat{R}, s)$, where $\hat{R}$ is the reverse of R. Hence, when solving single-variable queries, we can choose the position of the variable. Queries of the form $(s, R, x)/(x, R, o)$ involve traversing A_R forwards/backwards. This gives us an important degree of freedom: we can choose the direction to traverse A_R, with yields slightly different algorithms.

We first present the algorithm for a query (s, R, x) and then show how it is modified to solve the other class of queries. The basic strategy of the algorithm is analogous to that of the Ring [4], but the implementation of the ideas differs. The solutions to (s, R, x) are the ends of paths ρ in G starting at s such that $\mathsf{word}(\rho)$ is accepted by A_R. Thus, the problem boils down to traversing the product graph $A_R \times G$. Along our traversal we maintain a pair (D, v), meaning that we are visiting, at the same time, all the nodes $(q, v) \in A_R \times G$ such that q is represented in D. Our initial pair is $(2^{m+1}, s)$. From the current pair (D, v), we find all the labels p that leave some state in D. For each such label, we compute the new set of states D' using the formula seen in Section 2.2. In G, we jump from v to every node $v' \in \mathsf{neigh}_p(G, v)$ or $v' \in \mathsf{rneigh}_p(G, v)$, depending on whether p is an inverse label or not. For each resulting node v', we recursively continue our traversal by pair (D', v').

There are two critical checks that we make when considering a new pair (D, v). First, we have to determine whether v belongs to the solution. This is done by verifying that D contains an accepting state. Second, we need to avoid loops in the traversal of the product graph. To that end, we maintain a structure that keeps track of the states of A_R with which we have already visited every node of G. We now enter into details.

Avoiding redundancies and loops. If we are processing (D, v) and have previously visited another pair (D', v), we should remove from D the states that are also in D', as those have already been processed and may lead to loops. We maintain a table $\mathsf{seen}[1\,..\,n]$ containing in $\mathsf{seen}[v]$ the bitvector of length $m + 1$ that represents all the states that have been active in previously seen pairs of the form (D', v). This array is initialized as $\mathsf{seen}[i] = 0$ before the query starts. When arriving at (D, v), we remove from D the states in $\mathsf{seen}[v]$, and add those to $\mathsf{seen}[v]$, as follows (where $\sim$ and $|$ are the bitwise logical not and or, respectively):

$$D \leftarrow D \mathbin{\&} {\sim}\mathsf{seen}[v] \text{ and } \mathsf{seen}[v] \leftarrow \mathsf{seen}[v] \mid D.$$

Reporting solutions. If $D \mathbin{\&} F \neq 0$, the automaton A_R accepts the path towards node v, so v is a solution to the query. Note, however, that v may be reachable from multiple accepting paths. To avoid reporting duplicated answers, we must verify that there is no previously seen pair (D', v), where D' contains a final state. This is done by verifying that $\mathsf{seen}[v] \mathbin{\&} F = 0$ before updating seen.

Computing the new pairs. We now explain in detail how we produce the new pairs (D', v') from the current one, (D, v).

First, we find the set P_D of all the elements $p \in \mathcal{L}_R$ that label an edge in A_R leaving D. As seen in Section 2.2, a label p is in P_D iff $T[D] \;\&\; B[p] \neq 0$. We can then obtain P_D by iterating through all the labels $p \in \mathcal{L}_R$ and keeping those for which the formula holds, which takes only $O(m)$ time. Since we may encounter D multiple times throughout the process, however, we define a lazy-initialized table $P[0\,..\,2^{m+1}-1]$ and store P_D in $P[D]$ as a bitvector of length $m+1$.

For each $p \in P_D$, we generate the new set of active states $D' = T[D] \;\&\; B[p]$. The set of nodes v' we reach by following p is, as explained, $\mathsf{neigh}_p(v)$ if $p \in \mathcal{L}$, and $\mathsf{rneigh}_{\hat{p}}(v)$ if $p \in \hat{\mathcal{L}}$. Those sets are computed as described in Section 3.

Queries of the form (y, R, o). The solution to these queries is analogous, but we traverse the edges of A_R and of G backwards. Concretely, the initial NFA state is F and the final NFA state is 2^{m+1}, and their roles are exchanged everywhere in the description above. We start from pair (F, o) and report v when D contains the initial state. We compute the NFA edges backwards with the formula $D' = T'[D \;\&\; B[p]]$, recall Section 2.2.

Finally, $P[D]$ now corresponds to the *ingoing* predicates, which are determined as those $p \in \mathcal{L}_R$ such that $D \;\&\; B[p] \neq 0$. In G we move to the reverse neighbors of v with label p if $p \in \mathcal{L}$ and to its neighbors with label $\hat{p}$ if $p \in \hat{\mathcal{L}}$.

4.2 Double-variable 2RPQs

It remains to treat the case of double-variable queries (x, R, y). The naive approach is to solve the single-variable queries (v, R, y) for all nodes $v \in V$ (or, symmetrically, the queries (x, R, v)). This is highly inefficient as many objects v may not lead to any solution. We improve this basic method by first obtaining a *feasible set* V', which is a (typically small) subset of V that contains all the nodes v such that (v, R, y) leads to at least one solution to (x, R, y). Let $\mathcal{L}'$ be the set of labels of edges leaving the initial state in A_R. We then define

$$V' \;=\; \bigcup_{p \in \mathcal{L}' \cap \mathcal{L}} \mathsf{sources}_p(G) \;\cup\; \bigcup_{p \in \mathcal{L}' \cap \hat{\mathcal{L}}} \mathsf{targets}_{\hat{p}}(G).$$

It is easy to see that every subject that is a solution to the query (x, R, o), $o \in V$, is in V'. Therefore, the solution set of (x, R, y) is equal to the collection of all solutions of the queries (v, R, y), $v \in V'$.

Let us now describe in detail how to get the elements of V'. First, the labels p connected to q_0 are those that lead to at least one active state when jumping from q_0, that is, those such that $T[2^{m+1}] \;\&\; B[p] \neq 0$. Then, for each such p, we need all subjects of edges in $G^{\leftrightarrow}$ with label p. These are given by $\mathsf{sources}_p(G)$ if p is not inverted and by $\mathsf{targets}_{\hat{p}}(G)$ otherwise.

5 Experimental Results

5.1 Benchmark

Benchmark database. We use the Wikidata Graph Pattern Benchmark (WGPB) as our benchmark[29]. This graph has $e = 958,844,164$ edges, $n = 348,945,080$ nodes, and $\lambda = 5,419$ labels (*i.e.*, predicates). From the nodes, $|S| = 106,736,662$ act as subjects and $|O| = 295,611,216$ as objects. This amounts to a total of 10.7 GB in plain form (with 32-bit integers for each triple component, and thus 12 bytes per edge, or bpe) and 7.9 GB in packed form (*i.e.*, using $\lceil \log |S| \rceil + \lceil \log |P| \rceil + \lceil \log |O| \rceil$ bits, or 8.63 bpe).

Queries. In order to get challenging, real-world RPQs, the authors of the Ring [4] extracted all the RPQs posed to the Wikidata Query Service that threw timeout error, that is, that needed more than 60 seconds to complete, from the Wikidata Query Logs [20]. After filtering RPQs using Wikidata-specific features, mentioning constants not used in the dataset, having one label, normalizing variable names, and removing duplicates, this process yielded 1,952 unique queries. Furthermore, they only keep the 1,567 queries with less than 1 million unique results for comparability reasons (as Virtuoso has a hard-coded limit of $2^{20} \approx 1$ million results). All queries are run with a timeout of 60 seconds under set semantics (using DISTINCT in the case of SPARQL).

Systems compared. We compare with the Ring and with its larger and faster version, Ring$_{\text{AB}}$. A completely different approach, which translates RPQs to operations on sparse Boolean matrices, has recently appeared [5, 6]. We compare with both their uncompressed baseline (BM) and their compressed version that uses k^2-trees (k^2-BM). Additionally, we compare our algorithm to the following well-known platforms for managing and querying RDF databases.

1. *Blazegraph* is the official SPARQL endpoint used by Wikidata and by other large customers.
2. *Apache Jena* is a widely used graph database and the reference implementation of the SPARQL standard.
3. *Virtuoso* is a multi-model database that accommodates RDF data, which hosts the public DBpedia endpoint, among others [14].

Machine and implementation. Our experiments were conducted on an isolated Intel(R) Xeon(R) CPU E5-2630 running at 2.30GHz, with 15 MB of cache and 384 GB of RAM. The operating system is GNU/Linux Devuan 2.1, with kernel `4.9.0-18-amd64`. We used the SDSL library [17], which implements our bitvectors and GMR arrays, and the Glushkov automata implementation of the Ring authors [9]. Our implementation is written in `C++11`, using the compiler `g++` version 6.3.0 and the flags `-std=c++11`, `-O3` and `-msse4.2`. All experiments are single-threaded. Recall that our algorithm allows both traversal directions of the NFA. Given that we observed slightly better performance with forward traversal, we have used this direction in our experiments. The complete source code and the instructions for compiling it can be found in the repository [27].

Table 1. Index space (in bpe) and construction time (in hours) for the different systems. The integer-encoding times are included in the construction.

	Ours	Ring	Ring_{AB}	BM	k^2-BM	Jena	Virtuoso	Blazegraph
Index space	7.17	16.41	27.93	16.45	4.33	95.83	60.07	90.79
Indexing time	5.5	7.5	8.3	10.7	5.5	37.4	3.0	39.4

5.2 Results

Index construction. Constructing the integer-encoded database takes 5.2 hours using the code by Arroyuelo *et al.* [4]. Constructing our index then takes 0.3 additional hours. The resulting index uses 6.87 GB, or 7.17 bpe. The total space used at query time is higher, 10.28 bpe. The excess is dominated by the seen table (3.11 bpe); the $O(2^m)$ space related to NFA preprocessing is negligible.

Table 1 shows the construction time and resulting space usage of our index compared to the other systems. Our index is the second most space-efficient, only surpassed by the k^2-BM. Note that the Ring uses slightly more than twice the space of our index, which is consistent with the fact that we do not duplicate the edges for dealing with the inverted predicates. BM has a similar space requirement as the Ring, while the rest of the systems use much more.

Our index exhibits a competitive construction time of 5.5 hours, matching k^2-BM and ranking second only to Virtuoso's leading time of 3 hours.

Querying. Table 2 details the average, median, and the number of timeouts for the different types of queries across the competing algorithms. Fig. 4 shows the performance distribution across all queries and specifically for double-variable queries, with the algorithms ordered on the x-axis according to their space usage.

Our index shows average performance, 0.6 seconds per query, is outperformed only by the 0.4 seconds of Ring_{AB}; we remind that this index uses 4 times more space than ours. The only index using less space than ours (60% of our space), k^2-BM, is 5 times slower on average. The other systems use more space and average time than ours. The median times of our index are also far better than the others, except for BM, which in exchange uses twice as much space.

The advantage of our index is most evident in queries with a single variable, where it achieves nearly the best performance in both median and average times, while using less space. For double-variable queries, our index remains competitive but does not stand among the fastest, and its particular its median time is close to the highest. Despite its high median times, our index produces fewer timeouts, which demonstrates higher reliability for complex queries.

Fig. 4 compares space and time distributions, showing on the left that our index is a key point in the Pareto-optimal curve, formed also by the k^2-BM (which uses 60% of the space but an order of magnitude more time) and the BM (which uses half the time but twice the space); Ring_{AB} has a good average but not such a good distribution. On the right, for 2 variables, the Pareto-optimal curve is formed by the Boolean-Matrix-based indices. Our index does outperform k^2-BM on the average, but the latter distributes better.

Table 2. Performance comparison of the different algorithms, detailing the number of timeouts (execution time over 60 seconds) and the average and median times, not considering timeouts. The notation $1v/2v$ denotes single-/double-variable queries.

	Ours	Ring	Ring$_{\text{AB}}$	BM	k^2-BM	Jena	Virtuoso	Blazegraph
Average	0.61	0.95	0.41	1.39	3.25	4.51	2.08	3.23
Median	0.01	0.07	0.03	0.005	0.35	0.21	0.13	0.13
Timeout	2	4	1	14	39	84	1	41
Average $1v$	0.28	0.48	0.25	1.19	2.84	3.62	1.79	3.24
Median $1v$	0.009	0.06	0.03	0.005	0.35	0.19	0.11	0.13
Timeout $1v$	0	0	0	12	30	58	1	39
Average $2v$	7 .41	10.91	3.66	5.45	11.92	22.83	8.17	2.98
Median $2v$	3.70	1.45	0.93	0.01	0.87	1.57	3.89	0.14
Timeout $2v$	2	4	1	2	9	26	0	6

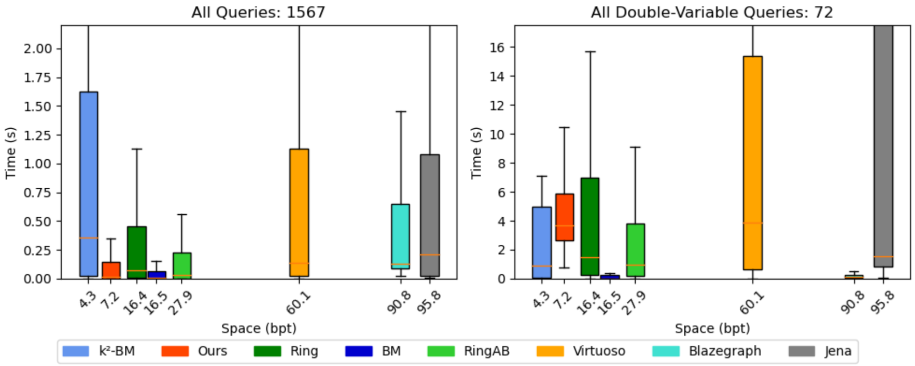


Fig. 4. Time taken by the different algorithms across all queries and for those with two variables (2v). The figure does not include the timeouts.

6 Conclusions

We have demonstrated that a compact representation designed for labeled graphs outperforms in space, and competes in time, with the Ring index [4] at solving Regular Path Queries. The main advantages of our representation are (1) it helps navigating edges bidirectionally without duplicating data, whereas the Ring navigates only backwards; (2) it builds on sequence representations [18] that are faster but with less functionality than the wavelet trees [19] used by the Ring. Both decisions in the Ring, unidirectionality and use of wavelet trees, aim at matching intersection-time lower bounds, while our index only ensures $O(m)$ time (which in practice is low anyway). On one-variable queries (which are most popular) our index provides better time distribution, losing only on the average to a Ring variant that uses 4 times more space (Ring$_{\text{AB}}$). Our representation fills an important place in the Pareto-optimal curve for one-variable queries.

Our work also uncovers subtle relations of independent interest between the Ring structure and the labeled graph representation we build on [24, Sec. 9.1.4].

References

1. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutiérrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: Proc. 44th ACM International Conference on Management of Data (SIGMOD). pp. 1421–1432 (2018)
2. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of Modern Query Languages for Graph Databases. ACM Computing Surveys **50**(5), 68:1–68:40 (2017)
3. Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Reutter, J.L., Rojas-Ledesma, J., Soto, A.: The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. ACM Transactions on Database Systems **29**(2), article 5 (2024)
4. Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Optimizing RPQs over a compact graph representation. The Very Large Databases Journal **33**, 349–374 (2024)
5. Arroyuelo, D., Gómez-Brandón, A., Navarro, G.: Evaluating regular path queries on compressed adjacency matrices. In: Proc. 30th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 35–48 (2023)
6. Arroyuelo, D., Gómez-Brandón, A., Navarro, G.: Evaluating regular path queries on compressed adjacency matrices. CoRR **2307.14930** (2024)
7. Arroyuelo, D., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Time- and space-efficient regular path queries. In: Proc. 38th IEEE International Conference on Data Engineering (ICDE). pp. 3091–3105 (2022)
8. Arroyuelo, D., Navarro, G., Reutter, J.L., Rojas-Ledesma, J.: Optimal joins using compressed quadtrees. ACM Transactions on Database Systems **47**(2), article 8 (2022)
9. Arroyuelo, D., Gómez-Brandón, A., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Ring-RPQ. `https://github.com/darroyue/Ring-RPQ` (2022)
10. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theoretical Computer Science **48**(C), 117–126 (1986)
11. Bonifati, A., Martens, W., Timm, T.: Navigating the Maze of Wikidata Query Logs. In: Proc. The World Wide Web Conference (WWW). pp. 127–138 (2019)
12. Deutsch, A., Francis, N., Green, A., Hare, K., Li, B., Libkin, L., Lindaaker, T., Marsault, V., Martens, W., Michels, J., Murlak, F., Plantikow, S., Selmer, P., van Rest, O., Voigt, H., Vrgoč, D., Wu, M., Zemke, F.: Graph pattern matching in GQL and SQL/PGQ. In: Proc. 48th ACM International Conference on Management of Data (SIGMOD). pp. 2246–2258 (2022)
13. Deutsch, A., Xu, Y., Wu, M., Lee, V.E.: Aggregation Support for Modern Graph Analytics in TigerGraph. In: Proc. 46th ACM International Conference on Management of Data (SIGMOD). pp. 377–392 (2020)
14. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Networked Knowledge – Networked Media, pp. 7–24. Springer (2009)
15. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An Evolving Query Language for Property Graphs. In: Proc. 44th ACM International Conference on Management of Data (SIGMOD). pp. 1433–1445 (2018)
16. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys **16**(5), 1–53 (1961)

17. Gog, S., Beller, T., Moffat, A., Petri, M.: Sdsl - succinct data structure library. `https://github.com/simongog/sdsl-lite` (2016)
18. Golynski, A., Munro, J., Rao Satti, S.: Rank/select operations on large alphabets: A tool for text indexing. In: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 368–373 (2006)
19. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 841–850 (2003)
20. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In: Proc. International Semantic Web Conference (ISWC). pp. 376–394 (2018)
21. Manola, F., Miller, E.: RDF Primer. W3C Recommendation (2004), `http://www.w3.org/TR/rdf-primer/`
22. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM Journal on Computing **24**(6), 1235–1258 (1995)
23. Munro, J.I.: Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). pp. 37–42. LNCS 1180 (1996)
24. Navarro, G.: Compact Data Structures — A practical approach. Cambridge University Press (2016)
25. Navarro, G., Raffinot, M.: New techniques for regular expression searching. Algorithmica **41**(2), 89–116 (2005)
26. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: Proc. 4th International Workshop on Graph Data Management: Experiences and Systems (GRADES). p. 7 (2016)
27. Robert, J.: A compact graph structure for efficiently solving RPQs. `https://github.com/j-rparra/compactGraph` (2023)
28. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata®RDF Graph Database. In: Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014)
29. Vrandecic, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Communications of the ACM **57**(10), 78–85 (2014)
30. Yakovets, N., Godfrey, P., Gryz, J.: Query Planning for Evaluating SPARQL Property Paths. In: Proc. 42th ACM International Conference on Management of Data (SIGMOD). pp. 1875–1889 (2016)