Noname manuscript No.
(will be inserted by the editor)

Optimizing RPQs over a Compact Graph Representation

Diego Arroyuelo · Adrián Gómez-Brandón · Aidan Hogan · Gonzalo Navarro · Javiel Rojas-Ledesma

Received: date / Accepted: date

Abstract We propose techniques to evaluate regular path queries (RPQs) over labeled graphs (e.g., RDF). We apply a bit-parallel simulation of a Glushkov automaton representing the query over a *ring*: a compact wavelet-tree-based index of the graph. To the best of our knowledge, our approach is the first to evaluate RPQs over a compact representation of such graphs, where we show the key advantages of using Glushkov automata in this setting. Our scheme obtains optimal time, in terms of alternation complexity, for traversing the product graph. We further introduce various optimizations, such as the ability to process several automaton states and graph nodes/labels simultaneously, and to estimate relevant selectivities. Experiments show that our approach uses 3–5× less space, and is over 5× faster, on average, than the next best state-of-the-art system for evaluating RPQs.

Keywords Regular path queries · Ring index · Succinct data structures · Graph databases

This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002; Fondecyt Grant 1-230755; Fondecyt Grant 1221926; Xunta de Galicia, FEDER Galicia 2014-2020 80%, SXU 20% Grant ED431G 2019/01; GAIN/Xunta de Galicia Grant ED431C 2021/53 (GRC); Xunta de Galicia/FEDER-UE Grant IN852D 2021/3; Ministerio de Ciencia e Innovación Grants PID2020-114635RB-I00 and PDC2021-120917-C21

D. Arroyuelo
IMFD & DCC, Escuela de Ingeniería, Pontificia Universidad Católica de Chile, Santiago, Chile
E-mail: diego.arroyuelo@uc.cl

✉ A. Gómez-Brandón
IMFD & CITIC, Universidade da Coruña, A Coruña, Spain
E-mail: adrian.gbrandon@udc.es

A. Hogan, G. Navarro & J. Rojas-Ledesma
IMFD & DCC, University of Chile, Santiago, Chile
E-mail: {ahogan,gnavarro,jrojas}@dcc.uchile.cl

1 Introduction

Many graph databases support *regular path queries* (*RPQs*) [22] that match arbitrary-length paths satisfying a regular expression over edge labels [4]. Fig. 1 shows a graph representing an urban transport network, indicating stations and the mode(s) of transport available between them (`l1`, `l2` and `l5` denote three metro lines). An RPQ $x \xrightarrow{(\texttt{l1}|\texttt{l2}|\texttt{l5})^+} y$ returns all 25 pairs of stations that are reachable by metro: x and y are node variables, while $(\texttt{l1}|\texttt{l2}|\texttt{l5})^+$ is a regular expression matching paths of length one-or-more with edges labeled `l1`, `l2`, or `l5`. An RPQ $x \xrightarrow{\texttt{l1}^+|\texttt{l2}^+|\texttt{l5}^+} y$ rather returns the 19 pairs of stations that are reachable via metro without changing line. We can also replace node variables with constants, where `Los Heroes` $\xrightarrow{\texttt{l2/bus}^*} y$ returns 3 stations reachable from `Los Heroes` via one leg of line 2, followed by zero-or-more bus legs.

While RPQs were proposed in the 80's [22, 49], a key milestone was the inclusion of *property paths* [43] in SPARQL 1.1 [38], that is, RPQs extended with inverse labels (known as two-way regular path queries (2RPQs)) and negated label sets. Of 208 million SPARQL queries in publicly-available logs from the Wikidata Query Service [46], 24% use at least one RPQ feature [15]. Later graph query languages followed suit, adding support for RPQ-like features [67, 4, 30, 5, 23, 24]. These developments necessitate techniques to evaluate RPQs efficiently.

A classical approach to evaluate RPQs is to represent the regular expression as an automaton and search over its product with the data graph, which is expanded lazily [49]. Recent approaches further propose using recursive queries [73, 74, 40], parallelism [50], distribution [58, 72, 21, 48, 37], specialized indexes [36, 29, 44, 45], multi-query optimization [2], approximation [70], just-in-time compilation [64], etc., to reduce runtimes for evaluating RPQs.

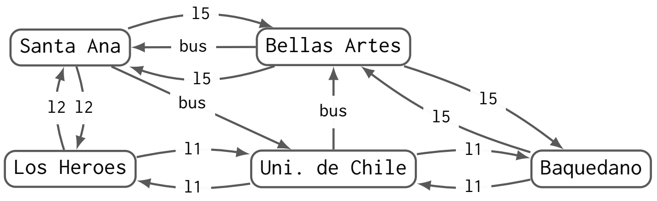


Fig. 1 Santiago metro stations with metro lines and buses

Our Contribution: We propose techniques to evaluate 2RPQs over a *ring* [7]: a compact index that represents the graph, and can support join queries (a.k.a. basic graph patterns [4]) in worst-case optimal time using space close to a plain representation of the graph. The ring uses a Burrows–Wheeler Transform (BWT) [18] to convert the graph into a sequence that is then encoded as wavelet trees [35]. The technique we propose for evaluating RPQs over a ring combines (1) BWT's backward search capabilities [27]; (2) the ability of the wavelet trees to work efficiently on ranges of nodes or edge labels; and (3) the regularity of the Glushkov automaton [34] for the regular expression and the versatility of its bit-parallel simulation [56]. The resulting algorithm traverses the product subgraph induced by the query optimally in terms of the so-called *alternation complexity* [11]. Furthermore, our algorithm can search over several paths in the product graph simultaneously. To support RPQs, our data structure uses space close to a compact representation; to support 2RPQs, the space doubles. Experiments show that for 2RPQs we use 3–5 times less space than competing databases, and are on average 2.8× faster than the nearest system: Blazegraph [65].

In this extended version of our conference paper [8], in addition to analyzing our algorithm in terms of the alternation complexity, we explore a number of optimization techniques to further improve performance. The first such technique better exploits the wavelet trees' ability to work efficiently over ranges of nodes and labels by dictionary encoding constants using a BFS order that aims to consolidate the ranges encountered while traversing the graph. With this optimization, our algorithm becomes 5.1× faster than Blazegraph. A second key optimization technique is based on the idea of splitting complex regular expressions at some selective intermediate point [42, 58, 72, 74, 57, 50] by efficiently counting the number of nodes that can be matched with intermediate automaton states. By roughly doubling the space of our basic representation (which is still 1/2–1/3 that of the other systems), this second optimization further speeds up the index on complex RPQs by 3.7× on average, and by 20× considering median times. These improvements are demonstrated by additional experiments involving a new data- and query-set based on YAGO2s [13, 74], extending previous results over Wikidata [69].

2 Related Work

We present related work on efficiently evaluating RPQs, and RPQ-like fragments such as property paths in SPARQL 1.1.

Evaluating path queries: Techniques for evaluating RPQs can be loosely divided into *navigational* (graph-based) and *relational* (table-based) approaches. The former approaches may invoke graph search algorithms (BFS, DFS, etc.), while the latter approaches may rather invoke recursive joins and other relational operators. As seen later, both approaches end up being largely analogous, and/or can be combined.

Given an RPQ $x \xrightarrow{e} y$, a natural navigational approach is to start at nodes matching x, and for each one, try to navigate paths outwards (using BFS, DFS, etc.) that match the regular expression e, reporting nodes reached at the end of matching paths as y. Conversely, we might rather opt to start at y, if, for example, y is constant and x is variable. But for some RPQs, the regular expression may give us strong hints that it would be more efficient to start in the "middle", effectively splitting the RPQ into two sub-RPQs that are later joined. Koschmieder and Leser [42] propose to split an RPQ by its *rare labels* – i.e., labels with fewer than m edges – in order to ensure more selective start/end points, where the splits are later joined. Nolé and Sartiani [58] evaluate RPQs using the concept of *Brzozowski derivatives*, whereby the regular expression is rewritten based on the symbols already read such that the rewritten expression matches suffixes that complete the path. Wang et al. [72] evaluate RPQs based on *partial answers* that can be connected, allowing for these answers to not only be prefixes, but also infixes and suffixes. Nguyen and Kim [57] split RPQs similarly to the rare labels strategy, but rather attempt to minimize the cost of the most costly sub-RPQ resulting from the split. Wadhwa et al. [70] compute approximate RPQ results using bidirectional random walks from the source and target node.

An alternative direction is to rather view the graph as a single relation of the form $G(s, p, o)$, or a set of binary relations of the form $p_i(s, o), \ldots, p_n(s, o)$, and translate an RPQ into recursive relational operators. Dey et al. [25] evaluate RPQs using either Datalog rules or recursive SQL queries. Yakovets et al. [73] translate SPARQL property paths into recursive SQL queries. Jachiet et al. [40] propose an extended relational algebra with a fixpoint operator capable of expressing RPQs. Fionda et al. [28] propose *extended property paths* – including difference and intersection of paths, custom constraints on nodes along the path, etc. – that are compiled into recursive SPARQL queries.

Seeing both navigational and relational approaches as complementary, Yakovets et al. [74] propose hybrid "waveplans" that enable novel query plans not possible in either base approach. Abul-Basher [2] propose an extension called "swarmguide" for optimizing multiple RPQs at once.

Recent approaches leverage acceleration techniques. Miura et al. [50] exploit field programmable gate arrays (FPGAs) to split and evaluate RPQs in parallel. Tetzel et al. [64] compile RPQs into C++ code that evaluates them.

Other authors propose specialized indexes for RPQs. Gubichev et al. [36] design an index called FERRARI [63] that encodes the transitive closure of edges with the same label as node intervals. Wang et al. [71] propose a specialized indexing scheme on edge labels to evaluate RPQs. Fletcher et al. [29] propose a *k-path index* that indexes all paths of length at most k in a B$^+$-tree. Kuijpers et al. [44] then use k-path indexes to optimize RPQs over Neo4j, while Liu et al. [45] populate k-path indexes with frequent paths.

Other settings: While our focus is on evaluating RPQs over a static graph on a single machine, we can briefly mention that other works have looked at evaluating RPQs on graphs partitioned over multiple machines [58, 72, 21, 48, 37], websites [10, 39], or temporal windows [59].

RPQ fragments: Some works focus on fragments of RPQs, such as *label-constrained reachability queries* (*LCRs*) [41], which match paths of arbitrary length such that each edge label on the path is in a given set $\{p_1, \ldots, p_n\}$, corresponding to RPQs of the form $x \xrightarrow{(p_1|\ldots|p_n)^*} y$ [66], which are common in practice [15, 16]. Specialized indexes for LCRs have been proposed, encoding tuples of the form (u, v, L) such that node v is reachable from node u via a path whose edges only use labels from the set L, often such that L is minimal [41, 75, 66, 60, 61]. Since such tuples can be great in number, often only a subset are indexed. Jin et al. [41] combine a spanning tree and a partial index of the transitive closure from which the full closure can be computed. Zou et al. [75] propose to decompose and build separate LCR indexes for each (strongly connected) component. Valstar et al. [66] construct a partial LCR index that is complete for k "landmark" vertices with highest degree. Peng et al. [60] propose a pruned LCR index inspired by *2-hop labeling*, which allows a tuple (u, v, L) to be pruned from the index if covered by joining two other tuples in the index. Other works have explored distance-aware variants of LCRs, including label-constrained shortest path queries (LCSPs) [14]; and label constrained k-reachability queries (LCKRs) [61].

Novelty: We introduce a novel technique to evaluate (2)RPQs that is efficient both in time and space. To the best of our knowledge, our approach is the first that evaluates RPQs on a compressed representation of the graph, and the first to show key advantages of using Glushkov automata [34] in this setting: not only does it enable a more space-efficient bit-parallel simulation of the automaton [56], its transitions exhibit a regularity that is crucial for efficiently evaluating RPQs. The combination of the backward search capabilities of the BWT [18], the ability of the wavelet trees [35] to work on ranges of nodes/labels, and the regularity of Glushvov's automaton, allow us to find the needed edges of the product graph in optimal time. The bit-parallel simulation, with access to ranges of nodes and labels, further enables processing sets of nodes of the product graph *simultaneously*, speeding up the classical strategy. The use of wavelet trees further opens up some novel optimization possibilities that we explore in this extended version. First, we can take advantage of the efficient processing of ranges by adopting a dictionary encoding of constants that consolidates ranges found during a BFS-style navigation. Second, we show how wavelet-tree-like structures can support efficient and accurate computation of the selectivities needed to decide whether or not (and where) to split an RPQ, allowing us to implement this optimization as proposed by previous works with little space cost [42, 58, 72, 74, 57, 50].

3 Key Concepts

3.1 Graphs and Regular Path Queries

Let Σ denote a set of symbols. We start by defining the directed labeled graphs that encode our data as sets of triples.

Definition 1 We define a (directed edge-labeled) *graph* $G \subseteq \Sigma \times \Sigma \times \Sigma$ to be a finite set of triples of symbols of the form (s, p, o), denoting (subject,predicate,object). Each triple of G can be viewed as a labeled edge of the form $s \xrightarrow{p} o$. Given a graph G, we define its *nodes* as $V = \{x \mid \exists y, z,\ (x, y, z) \in G \vee (z, y, x) \in G\}$.

We now define paths in such graphs.

Definition 2 A *path* ρ from x_0 to x_n in a graph G is a string of the form $x_0\, p_1\, x_1\, \ldots\, p_n\, x_n$ such that $(x_{i-1}, p_i, x_i) \in G$ for $1 \leq i \leq n$. Abusing notation, we may write that $\rho \in G$ if ρ is a path in G. Given a path ρ, we call $\mathsf{word}(\rho) = p_1 \ldots p_n \in \Sigma^*$ the *word* of ρ.

We reuse the concept of regular expressions, with a slightly adapted syntax that is more popular for RPQs [38].

Definition 3 A *regular expression* is defined inductively as follows. First, ε and any element of Σ are regular expressions. Furthermore, if E, E_1 and E_2 are regular expressions, then E^* (Kleene star), E_1/E_2 (concatenation) and $E_1|E_2$ (disjunction) are also regular expressions. We may further abbreviate E^*/E as E^+, and $\varepsilon|E$ as $E^?$.

Two-way RPQs (2RPQs) also allow for traversing edges in the inverse direction. This is formalized as follows.

Definition 4 We denote by $\hat{}\Sigma = \{\hat{}s \mid s \in \Sigma\}$ the *inverses* of symbols in Σ, and by $\Sigma^{\leftrightarrow} = \Sigma \cup \hat{}\Sigma$ the set of symbols and their inverses. We assume that $\Sigma \cap \hat{}\Sigma = \emptyset$ and that $s = \hat{}(\hat{}s)$. We call $\hat{}G = \{(y, \hat{}p, x) \mid (x,p,y) \in G\}$ the *inverse* of a graph G, and $G^{\leftrightarrow} = G \cup \hat{}G$ the *completion* of G. If E is a two-way regular expression, then so is $\hat{}E$.

For our purposes, a two-way regular expression over Σ is simply a regular expression over $\Sigma^{\leftrightarrow}$. The conversion is easily done by pushing the inversion symbol to the leaves, using the rules $\hat{}(E^*) = (\hat{}E)^*$, $\hat{}(E_1/E_2) = \hat{}E_2/\hat{}E_1$, $\hat{}(E_1|E_2) = \hat{}E_1|\hat{}E_2$, and $\hat{}\varepsilon = \varepsilon$. We now define the concept of matching a regular expression in a graph.

Definition 5 A path ρ *matches* a regular expression E iff $\mathsf{word}(\rho) \in L(E)$, where $L(E)$ denotes the language of E.

We now define regular path queries, which specify a regular expression and conditions or variables to bind in the extremes of the matching paths.

Definition 6 Let Φ denote a set of variables. Let $\mu : \Phi \to \Sigma$ denote a partial mapping from variables to symbols. We denote the domain of μ as $\mathsf{dom}(\mu)$, i.e., the set of variables for which μ is defined. If E is a regular expression, $s \in \Phi \cup \Sigma$ and $o \in \Phi \cup \Sigma$, then we call (s,E,o) a *regular path query* (*RPQ*). Let x_μ be defined as $\mu(x)$ if $x \in \mathsf{dom}(\mu)$, or x otherwise. We define the *evaluation* of (s,E,o) on G as:

$$(s,E,o)(G) = \{\mu \mid \mathsf{dom}(\mu) = \{s,o\} \cap \Phi \text{ and there exists a path } \rho \text{ from } s_\mu \text{ to } o_\mu \text{ in } G \text{ matching } E\}.$$

Example 1 Take the graph G of Fig. 1 and the RPQ $(x, (\mathtt{l1}|\mathtt{l2}|\mathtt{l5})^+, y)$, where $x, y \in \Phi$ are variables. Infinitely many paths in G match $(\mathtt{l1}|\mathtt{l2}|\mathtt{l5})^+$, including:

$$\mathtt{UCh} \xrightarrow{\mathtt{l1}} \mathtt{LH} \xrightarrow{\mathtt{l1}} \mathtt{UCh}$$
$$\mathtt{UCh} \xrightarrow{\mathtt{l1}} \mathtt{LH} \xrightarrow{\mathtt{l1}} \mathtt{UCh} \xrightarrow{\mathtt{l1}} \mathtt{LH}$$
$$\mathtt{Baq} \xrightarrow{\mathtt{l1}} \mathtt{UCh} \xrightarrow{\mathtt{l1}} \mathtt{LH} \xrightarrow{\mathtt{l2}} \mathtt{SA}$$

and so forth (abbreviating node labels). The evaluation of the RPQ on G will return all mappings such that x maps to the start node of some such path, and y maps to the end node of the same path. For example, from the first path, we will return a solution μ such that $\mu(x) = \mathtt{UCh}$, $\mu(y) = \mathtt{UCh}$. □

When E is a regular expression over $\Sigma^{\leftrightarrow}$ (i.e., a two-way regular expression over Σ), we call (s,E,o) a *two-way regular path query* (2RPQ), and its evaluation is defined on $G^{\leftrightarrow}$ instead of G. We will speak of RPQs and 2RPQs indistinctly, assuming the corresponding alphabet and graph.

3.2 Product Graph

A key approach for evaluating an RPQ (s,E,o) on G uses the *product graph* $G_{\mathcal{M}}(V_{\mathcal{M}}, A_{\mathcal{M}})$ of G and a nondeterministic finite automaton (NFA) $\mathcal{M}$ of E [49]. Let $\mathcal{M} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA, built for example with Thompson's construction, where Q denotes the set of states, $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ the transitions, $q_0 \in Q$ the initial state, and $F \subseteq Q$ the set of accepting states. Then we build $G_{\mathcal{M}}$ from G and $\mathcal{M}$ as follows.

Definition 7 Let $\mathcal{M} = (Q, \Sigma, \Delta, q_0, F)$ be an NFA and G be a labeled graph with labels in Σ and nodes in V, then the *product graph* of G and $\mathcal{M}$ is a directed graph $G_{\mathcal{M}}(V_{\mathcal{M}}, A_{\mathcal{M}})$ with nodes $V_{\mathcal{M}} = V \times Q$ and edges

$$\begin{aligned} A_{\mathcal{M}} = \{((x,q),(y,p)) \mid \exists\, l : (x,l,y) \in G \wedge (q,l,p) \in \Delta\} \\ \cup \{((x,q),(x,p)) \mid x \in V \wedge (q,\varepsilon,p) \in \Delta\}. \qquad (1) \end{aligned}$$

The RPQ can then be evaluated using graph search (e.g., BFS or DFS) to find paths in the product graph $G_{\mathcal{M}}$ that start from some node $(x,q_0) \in V_{\mathcal{M}}$ and end in some node $(y,q_f) \in V \times F$ (where $x = s$ if $s \in \Sigma$, and $y = o$ if $o \in \Sigma$). Each traversal strategy we use on $G_{\mathcal{M}}$ for a given query induces a subgraph $G'_{\mathcal{M}}$ of $G_{\mathcal{M}}$, as follows.

Definition 8 Given a traversal strategy of the product graph to solve RPQs, the *induced subgraph* of an automaton $\mathcal{M}$ on a graph G is the subgraph $G'_{\mathcal{M}}(V'_{\mathcal{M}}, A'_{\mathcal{M}})$ of $G_{\mathcal{M}}$ induced by the nodes $V'_{\mathcal{M}}$ visited by the traversal and all the edges $A'_{\mathcal{M}}$ that connect nodes of $V'_{\mathcal{M}}$.

3.3 Cost Models

Per Def. 8, we say that a *traversal strategy* determines the induced subgraph $G'_{\mathcal{M}}$ – that is, the set of nodes to traverse in order to solve the query – but not *how* to compute $G'_{\mathcal{M}}$. A corresponding *traversal algorithm* implements the traversal strategy, computing $G'_{\mathcal{M}}$ from $G_{\mathcal{M}}$. The number $|A'_{\mathcal{M}}|$ of edges in the induced subgraph is arguably a *lower bound* on the time complexity of the traversal algorithm. This lower bound would be reachable only if the algorithm could guess, in optimal time, which are the edges in $A_{\mathcal{M}}$, leaving every node in $V'_{\mathcal{M}}$, that belong to $A'_{\mathcal{M}}$.

Actual traversal algorithms require more time than this lower bound. For example, an algorithm that at every node $(x,q) \in V'_{\mathcal{M}}$ verifies each possible label l of an edge leaving from $x \in V$, to see if it leads from state $q \in Q$ to some state $p \in Q$ (cf. Eq. (1)), incurs a cost of at least $|lab_G(x)|$, where

$$lab_G(x) \;=\; \{l \mid \exists\, y : (x,l,y) \in G\}.$$

The total cost of such a traversal algorithm is then $Cost_G = \sum_{(x,q)\in V'_{\mathcal{M}}} |lab_G(x)|$, multiplied by the time it takes to determine where a label leads in $\mathcal{M}$. Instead, an algorithm that verifies, from $(x,q) \in V'_{\mathcal{M}}$, every possible symbol l leading somewhere in $\mathcal{M}$ from q, to see if there is an edge $(x,l,y) \in G$, incurs a cost of at least $|lab_{\mathcal{M}}(q)|$, where

$$lab_{\mathcal{M}}(q) = \{l \mid \exists\, p : \mathcal{M} \text{ has a path from } q \text{ to } p \text{ consuming } l\}.$$

The total cost of such a traversal algorithm is then $Cost_{\mathcal{M}} = \sum_{(x,q)\in V'_{\mathcal{M}}} |lab_{\mathcal{M}}(q)|$, multiplied by the time it takes to determine where a label leads in G. A clever algorithm choosing the smaller of both at each node of $V'_{\mathcal{M}}$ could obtain at best $Cost_{GM} = \sum_{(x,q)\in V'_{\mathcal{M}}} \min(|lab_G(x)|, |lab_{\mathcal{M}}(q)|)$.

Per Eq. (1), what we need to compute in every node $(x,q) \in V'_{\mathcal{M}}$ is the *intersection* $lab_G(x) \cap lab_{\mathcal{M}}(q)$. The costs just derived correspond to particular ways to compute that intersection. A lower bound to the number of comparisons needed to compute an intersection is given in terms of the so-called *alternation complexity*, defined next.

Definition 9 [11] The *alternation complexity* $\alpha(X,Y)$ between sets X and Y is the minimum number of times we must switch from one to the other in an ordered traversal, so as to enumerate $X \cup Y$.

For example, consider the sets $X = \{1,2,5,6,7,9,10,13,14\}$ and $Y = \{3,4,6,7,8,9,10,11,12\}$. Their union is $X \cup Y = \{1,\ldots,14\}$, their intersection is $X \cap Y = \{9,10\}$, and $\alpha(X,Y) = 4$, because we can traverse $X \cup Y$ with 4 switches between X and Y, for instance as follows:

```
  1   2           5   6   7       9   10              13   14
__________     ___________                         ____________
          3   4       6   7   8   9   10   11   12
          _____        _____________________________
```

The alternation complexity yields the following lower bound on the cost of intersecting two sets:

Lemma 1 *[11] If the elements of X and Y belong to a universe of size U, then $\alpha(X,Y)\log\frac{U}{\alpha(X,Y)}$ is a lower bound to the cost of intersecting X and Y.*

It is easy to see that $\alpha(lab_G(x), lab_{\mathcal{M}}(q))$ lower bounds $2\min(|lab_G(x)|, |lab_{\mathcal{M}}(q)|)$, and it could be much lower. For example, if $lab_G(x)$ has k elements, all smaller than the k elements of $lab_{\mathcal{M}}(q)$, then $\min(|lab_G(x)|, |lab_{\mathcal{M}}(q)|) = k$, but $\alpha(lab_G(x), lab_{\mathcal{M}}(q)) = 1$. A better lower bound for the cost of a traversal algorithm is then:
$Cost_A = \sum_{(x,q)\in V'_{\mathcal{M}}} \alpha(lab_G(x), lab_{\mathcal{M}}(q)) \le Cost_{GM}$.

Per Lemma 1, a lower bound for any traversal algorithm that builds $G'_{\mathcal{M}}$ is then $alt(G'_{\mathcal{M}}, U)$, a shorthand for

$$\sum_{(x,q)\in V'_{\mathcal{M}}} \alpha(lab_G(x), lab_{\mathcal{M}}(q)) \log \frac{U}{\alpha(lab_G(x), lab_{\mathcal{M}}(q))}. \quad (2)$$

The cost of our algorithm, interestingly, will be *upper bounded* by Eq. (2). The reason is that we will manage to process several nodes of $V_{\mathcal{M}}$ simultaneously. This is obtained by combining Glushkov automata (which yields automata $\mathcal{M}$ with small sets $lab_{\mathcal{M}}(q)$), bit-parallelism (which processes together several states of $\mathcal{M}$), and the ring data structure (which processes together several nodes of G); all those are described next.

3.4 Glushkov Automaton

Glushkov [34, 12] proposed an alternative to the more popular Thompson's construction for building an NFA

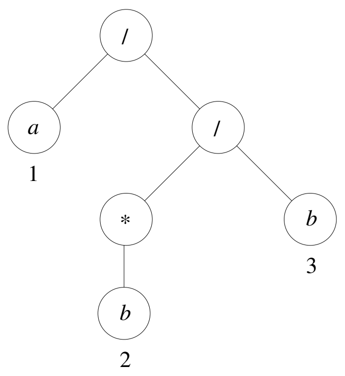
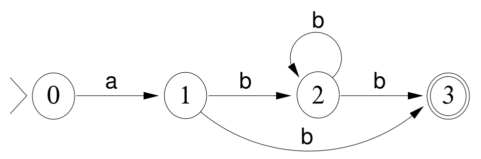


Fig. 2 The syntax tree of the regular expression $E = a/b^*/b$, indicating the positions of leaves.



T[0000] = 0000
T[0001] = 0000
T[0010] = 0011
T[0011] = 0011
T[0100] = 0011
. . .
T[1111] = 0111

F = 0001
B[a] = 0100
B[b] = 0011

Fig. 3 The Gluskov automaton of the regular expression $a/b^*/b$, and its bit-parallel representation on the bottom.

from a regular expression E. The bit-parallel simulation of Glushkov's construction has been shown to be more efficient than Thompson's, and also than classical DFA constructions when the regular expressions are small [56]. Since this is the case for most RPQs in practice (in particular, of all those we found in real query logs) we have chosen the bit-parallel simulation of Glushkov's NFA for our index.

Let E have m occurrences of symbols in Σ. Its syntax tree then has m leaves that represent symbols of Σ. We say that the kth left-to-right leaf corresponding to some $c \in \Sigma$ represents the *position* k of E, and we say $p_k = c$. The algorithm creates an NFA of $m+1$ states, one per position plus the initial state 0.

Example 2 Fig. 2 shows the syntax tree of the regular expression $E = a/b^*/b$. It has three positions, with $p_1 = a$, $p_2 = b$, and $p_3 = b$. The corresponding NFA is shown on the top of Fig. 3. □

Compared to Thompson's construction of an NFA from E, Glushkov's has the disadvantage of generating $\Theta(m^2)$ transitions in the worst case, and needing $O(m^2)$ construction time [17]. In exchange, it has various useful properties:

1. The NFA has no ε-transitions.
2. The NFA has exactly $m+1$ states (numbered 0 to m), which is worst-case optimal.

3. All the transitions arriving at state $k > 0$ have the label p_k, while no transition leads to state 0.

These properties imply the following fact [56], which leads to yet another advantage of using Glushkov's construction, as we see in the following.

Fact 1 *In a Glushkov NFA, the states reached in one step from a set X of states by symbol c are the intersection of those reached from X in one step and those reached by c from any state.*

Example 3 Consider again the top of Fig. 3 and take the states $X = \{0, 2\}$. The states reachable from X in one step via b are $\{2, 3\}$, that is, the intersection of $\{1, 2, 3\}$ reachable in one step from X via any symbol and $\{2, 3\}$ reachable in one step via b from any state. $\square$

3.5 Bit-parallel Glushkov Automata

Fact 1 enables a particularly efficient *bit-parallel* simulation of the NFA [56]. This simulation represents the NFA states as a sequence of bits, so each configuration of active and inactive states (bits set to 1 and 0, respectively), correspond to a state in the DFA according to the classic powerset construction. The simulation operates on the states in parallel by using the classic arithmetical and logical operations on computer words. Assume that the alphabet is an integer range $\Sigma = [1\,..\,\sigma]$. The simulation uses the following variables:

– A bit sequence D of length $m+1$ tells, at every step, the active NFA states, as discussed. Assume the initial state corresponds to the leftmost bit.
– A table $B[1\,..\,\sigma]$ of bit sequences indicates with 1s, at each $B[c]$, the states targeted by transitions labeled c.
– A table $T[0\,..\,2^{m+1}-1]$ stores in $T[X]$, for each possible $(m+1)$-bit argument X representing a set of states, the states reachable from X in one step by any symbol.
– A bit sequence F marks with 1s the final NFA states.

The automaton can be used, in particular, to traverse a sequence and report all (the ending positions of) its prefixes matching a word in the language accepted by the NFA. The bit-parallel simulation is as follows:

1. We set $D \leftarrow 10^m$ (a 1 followed by m 0s) to activate the initial state, and start the scan from position zero.
2. If $D \,\&\, F \neq 0^{m+1}$, we have reached a final state and report the current position (where '&' is the bitwise-and).
3. If $D = 0^{m+1}$, we have run out of active states and finish.
4. We read a new input symbol c and use Fact 1 to update D as follows, so the new active states are those reachable from the current ones and by symbol c:

$$D \;\leftarrow\; T[D] \,\&\, B[c]. \tag{3}$$

5. Return to point 2.

Example 4 The Glushkov automaton of $a/b^*/b$ and its bit-parallel representation are shown in Fig. 3. Given a string $S = abba$, we initialize $D \leftarrow 1000$ with the initial state 0 activated. We read $S[1] = a$ and update $D \leftarrow T[1000] \,\&\, B[a] = 0100 \,\&\, 0100 = 0100$, activating state 1. We read $S[2] = b$ and update $D \leftarrow T[0100] \,\&\, B[b] = 0011 \,\&\, 0011 = 0011$, activating states 2 and 3. We report here the endpoint of a match since $D \,\&\, F = 0011 \,\&\, 0001 \neq 0000$. To find other endpoints, we next read $S[3] = b$ and update $D \leftarrow T[0011] \,\&\, B[b] = 0011 \,\&\, 0011 = 0011$, reporting this position as well. Finally, we read $S[4] = a$ and update $D \leftarrow T[0011] \,\&\, B[a] = 0011 \,\&\, 0100 = 0000$. At this point we run out of active states and finish. $\square$

A similar simulation can be used to read the text in reverse [56] by building a table $T'[0\,..\,2^m-1]$ where $T'[X]$ marks with 1s the states that can reach a state in X in one step, initializing $D \leftarrow F$ and, for each symbol c, updating

$$D \;\leftarrow\; T'[D \,\&\, B[c]], \tag{4}$$

and accepting when $D \,\&\, 2^m \neq 0$.

Bit-parallelism uses the RAM model of computation, where all the arithmetical and logical operations over a w-bit word take constant time; it is usual to assume $w = \Theta(\log n)$, where n is the data size. In our case, if $m+1 \leq w$, we can use a single computer word to hold each bit sequence. In this case, the space of the simulation is $O(2^m+\sigma)$ words, instead of the worst-case $O(2^m\sigma)$ of a classic DFA implementation. The tables are built in time $O(2^m)$ [56] if we use lazy initialization for B. Current computer words with $w = 64$ make this to be the case with most RPQs found in practice.

If $m+1 > w$, then we need to use $\lceil (m+1)/w \rceil$ computer words to hold D, F, and every entry of B and T. In this case, all the time and space complexities get multiplied by $O(m/w)$. If we want to avoid the exponential space and time $O(2^m)$, we can split table T vertically into $\lceil (m+1)/d \rceil$-bit subtables $T_1, \ldots, T_d$, so that if we partition $X = X_1 \cdots X_d$, then $T[X] = T_1[X_1] \mid \cdots \mid T_d[X_d]$, where "|" denotes the bitwise-or. This reduces the space to $O(d\,2^{m/d}+\sigma)$ and further multiplies time by $O(d)$, for any desired $1 \leq d \leq m+1$ [56]. We assume for simplicity that $m = O(w)$ and use $O(2^m)$ space in the paper, but in Theorem 1 we recall that we can curb the exponential space.

Rather than using bit-parallelism to simulate the DFA, we use it to traverse the product graph while acting on several NFA states simultaneously (e.g., we may have a set of active NFA states that do not correspond to any DFA state in the classic powerset construction). This corresponds to acting on several nodes of the product graph simultaneously.

(SA,l2,LH)	(l1,UCh,LH)	(SA,UCh,^bus)
(SA,l5,BA)	(l1,UCh,Baq)	(SA,LH,l2)
(SA,bus,UCh)	(l1,LH,UCh)	(SA,BA,l5)
(SA,^bus,BA)	(l1,Baq,UCh)	(SA,BA,bus)
(UCh,l1,LH)	(l2,SA,LH)	(UCh,SA,bus)
(UCh,l1,Baq)	(l2,LH,SA)	(UCh,LH,l1)
(UCh,bus,BA)	(l5,SA,BA)	(UCh,BA,^bus)
(UCh,^bus,SA)	(l5,BA,SA)	(UCh,Baq,l1)
(LH,l1,UCh)	(l5,BA,Baq)	(LH,SA,l2)
(LH,l2,SA)	(l5,Baq,BA)	(LH,UCh,l1)
(BA,l5,SA)	(bus,SA,BA)	(BA,SA,l5)
(BA,l5,Baq)	(bus,UCh,SA)	(BA,SA,^bus)
(BA,bus,SA)	(bus,BA,UCh)	(BA,UCh,bus)
(BA,^bus,UCh)	(^bus,SA,UCh)	(BA,Baq,l5)
(Baq,l1,UCh)	(^bus,UCh,BA)	(Baq,UCh,l1)
(Baq,l5,BA)	(^bus,BA,SA)	(Baq,BA,l5)
L_o	L_s	L_p

Nodes

1	SA
2	UCh
3	LH
4	BA
5	Baq

Edges

1	l1
2	l2
3	l5
4	bus
5	^bus

Fig. 4 Triples representing the completion of the graph of Fig. 1, adding a reverse edge labeled ˆbus for each edge labeled bus (l1, l2 and l5 are considered bidirectional). Three rotations of triples are shown; the last column in each rotation forms a component of the ring.

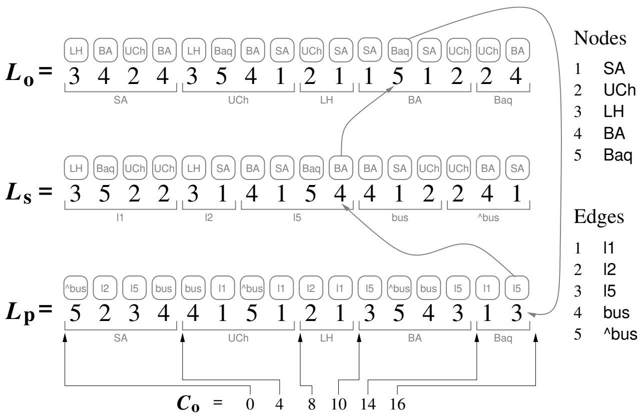


Fig. 5 The ring for the completion of the graph of Fig. 1, adding a reverse edge labeled ˆbus for each edge labeled bus (l1, l2 and l5 are bidirectional). We also show how the last triple in L_p is tracked.

3.6 The Ring

The *ring* [7] is a recently proposed representation for a set of triples (s,p,o), supporting worst-case optimal joins [68]. It regards triples with different rotations, (s,p,o), (p,o,s), or (o,s,p). What the ring actually stores are the numeric identifiers of objects, subjects, and predicates of the triples separated into three sequences, L_o, L_s, and L_p, respectively, as follows; the sortings are done on the numeric identifers.

Definition 10 The *ring* of a graph G formed by n triples of the form (s,p,o) consists of three sequences:

– $L_\mathrm{o}[1\,..\,n]$ enumerates the objects o from the list of the lexicographically sorted triples (s,p,o).
– $L_\mathrm{s}[1\,..\,n]$ enumerates the subjects s from the list of the lexicographically sorted triples (p,o,s).
– $L_\mathrm{p}[1\,..\,n]$ enumerates the predicates p from the list of the lexicographically sorted triples (o,s,p).

The concatenation $L_\mathrm{o} \cdot L_\mathrm{s} \cdot L_\mathrm{p}$ is analogous to the Burrows–Wheeler Transform (BWT) [18] of the concatenation of all triples in the graph [7].

Example 5 Fig. 4 shows the triples for the completion of the graph of Fig. 1, using abbreviated node labels. On the right we map node and edge labels to integer numbers, and sort according to those numbers. The leftmost rotation lists triples sorted by (s,p,o); the last column (o) of this rotation forms the sequence L_o. The middle rotation sorts triples by (p,o,s), and its last column (s) forms the sequence L_s. Finally, the rightmost rotation sorts the triples by (o,s,p) and its last column (p) forms L_p. The ring is then the tuple of the three sequences, L_o, L_s and L_p.

With this arrangement, a range in L_o corresponds to a lexicographic interval of triples (s,p,o) wherein a range may represent all the triples with a specific subject s (i.e., starting with s), and a smaller range may represent all the triples with subject s and predicate p (i.e., starting with (s,p)). A range in L_o can also represent a range of subjects $s_b\,..\,s_e$, and even a subject s followed by a range of predicates $p_s\,..\,p_e$. Analogously, ranges in L_s correspond to lexicographic intervals of triples (p,o,s) and ranges in L_p correspond to lexicographic intervals of triples (o,s,p). In the three strings, the range $[1\,..\,n]$ represents all the triples and a range of size 1 represents an individual triple.

Example 6 In Fig. 4, the range of positions $[5\,..\,8]$ of L_o corresponds to the triples (s,p,o) where $s = \mathsf{UCh}$, and the smaller range $[5,6]$ to the triples (s,p,o) where $s = \mathsf{UCh}$ and $p = \mathsf{l1}$. If we need a range for the triples where $p = \mathsf{l5}$ and $o = \mathsf{BA}$, we instead use the range $[8\,..\,9]$ of L_s, which is sorted in order (p,o,s). □

The ring retrieves triples using *LF-steps* [27], which we now define on array L_p (and analogously on L_s and L_o). Given a position i and the symbol $c = L_\mathrm{p}[i]$, let $C_\mathrm{p}[c]$ count occurrences of symbols smaller than c in L_p, and $\mathsf{rank}_c(L_\mathrm{p}, i)$ count occurrences of the symbol c in $L_\mathrm{p}[1\,..\,i]$. The corresponding LF-step returns the position in L_s of the subject associated with the predicate c, as follows:

$$\mathsf{LF}_\mathrm{p}(i) \;=\; C_\mathrm{p}[c] + \mathsf{rank}_c(L_\mathrm{p}, i). \tag{5}$$

The subject of the triple corresponding to the predicate at $L_\mathrm{p}[i]$ is $L_\mathrm{s}[i']$ for $i' = \mathsf{LF}_\mathrm{p}(i)$, and the object is $L_\mathrm{o}[i'']$ for $i'' = \mathsf{LF}_\mathrm{s}(i')$. We close the cycle with $i = \mathsf{LF}_\mathrm{o}(i'')$, where the predicate is (again) at $L_\mathrm{p}[i]$.

Example 7 Fig. 5 shows the ring of Fig. 4, now directly as sequences L_o, L_s, and L_p of integers. We write the abbreviated names over the numbers for readability. Note that, for example, L_p can be partitioned into the triples (o,s,p) starting with objects 1 (SA), 2 (UCh), 3 (LH), 4 (BA), and 5 (Baq), which we indicate below the sequence, and whose endpoints are marked in the array C_o, shown on the bottom.

Consider $L_\mathrm{p}[16] = 3$ (15), which gives the predicate of a triple whose object is 5 (Baq) because it belongs to the range $L_\mathrm{p}[15\,..\,16] = L_\mathrm{p}[C_\mathrm{o}[5]{+}1\,..\,C_\mathrm{o}[5{+}1]]$. To find this triple's subject, we note that this is the fourth 3 (15) in L_p. If we go to the fourth position in the area of 15 in L_s, $L_\mathrm{s}[7\,..\,10]$, which is $L_\mathrm{s}[10]$, we learn that the subject is $L_\mathrm{s}[10] = 4$ (BA). Indeed, $\mathsf{LF}_\mathrm{p}(16) = 10$. Thus, the triple is $\mathsf{BA} \xrightarrow{\mathsf{15}} \mathsf{Baq}$. Furthermore, $L_\mathrm{s}[10]$ is the second 4 in L_s, so if we go to the corresponding position $L_\mathrm{o}[12]$ (note $\mathsf{LF}_\mathrm{s}(10) = 12$) we cyclically find $L_\mathrm{o}[12] = 5$ (Baq), the object of the triple. We indeed return to position $L_\mathrm{p}[16]$ if we map $L_\mathrm{o}[12]$, the second 5 in L_o, to L_p. Again, $\mathsf{LF}_\mathrm{o}(12) = 16$. □

Multijoins can then be solved using *backward search* [27] over the ring, which computes in batch all of the LF-steps in a range. Consider a range $L_\mathrm{p}[b_o\,..\,e_o]$ listing all triples with a specific object o. The backward search by some specific predicate p gives the range $L_\mathrm{s}[b_p\,..\,e_p]$ corresponding to all triples with object o and predicate p. This is computed with the following formulas, which extend the LF-steps (Eq. (5)) to ranges [27, 7]:

$$b_p = C_\mathrm{p}[p] + \mathsf{rank}_p(L_\mathrm{p}, b_o - 1) + 1, \qquad (6)$$

$$e_p = C_\mathrm{p}[p] + \mathsf{rank}_p(L_\mathrm{p}, e_o). \qquad (7)$$

Listing the subjects s in $L_\mathrm{s}[b_p\,..\,e_p]$ then yields all the triples with that specific predicate p and object o, for example.

Example 8 Continuing our example, assume we wish to find all subjects for triples with predicate 3 (15) and object 4 (BA). Let us start from $L_\mathrm{p}[11\,..\,14]$, corresponding to object BA. If we apply a backward search step from $b_o = 11$ and $e_o = 14$, on the label 3 (15) using Eqs. (6) and (7), we obtain $L_\mathrm{s}[b_s\,..\,e_s] = L_\mathrm{s}[8..9] = \langle 1, 5\rangle$, meaning we arrive at BA by 15 from sources $L_\mathrm{s}[8] = 1$ (SA) and $L_\mathrm{s}[9] = 5$ (Baq). □

The ring uses *wavelet trees* [35], described next, to index each sequence L_o, L_s, and L_p. This representation enables backward searches in $O(\log|\Sigma|)$ time, and worst-case optimal joins with m triple patterns in time $O(Q^* m \log|\Sigma|)$, where Q^* is the AGM bound of the query [9, 7].

3.7 Wavelet Trees

The wavelet tree is a versatile data structure that, as we discuss now, represents a string or sequence within essentially the same space of its plain representation, while also being able to efficiently perform a number of useful queries on it.

Definition 11 The *wavelet tree* represents a string $L[1\,..\,n]$ with alphabet $[1\,..\,\sigma]$ as a perfect binary tree with σ leaves such that the c^{th} left-to-right leaf represents symbol c. Each internal wavelet tree node v that is the ancestor of leaves $c_s\,..\,c_e$ represents the subsequence $L_{\langle c_s, c_e\rangle}$ of L formed by the symbols in $[c_s\,..\,c_e]$. Instead of storing $L_{\langle c_s, c_e\rangle}$, node v stores a bitvector $W_{\langle c_s, c_e\rangle}$, so that $W_{\langle c_s, c_e\rangle}[i] = 0$ iff the leaf representing symbol $L_{\langle c_s, c_e\rangle}[i]$ descends by the left child of v. The leaves are not actually materialized.

The wavelet tree replaces L in the sense that it can obtain any $L[i]$ in $O(\log\sigma)$ time, as follows. Let v be the wavelet tree root, which stores bitvector $W = W_{\langle 1,\sigma\rangle}$ where $W[i] = 0$ indicates that $L[i] \in [1\,..\,\sigma/2]$; otherwise $L[i] \in [\sigma/2 + 1\,..\,\sigma]$ (we assume σ to be a power of 2 for simplicity). In the first case, $L[i] = L_{\langle 1,\sigma\rangle}[i]$ corresponds to $L_{\langle 1,\sigma/2\rangle}[i']$, where $i' = \mathsf{rank}_0(W, i)$ and we continue recursively by the left child of v with position i'. In the second case, $L[i]$ corresponds to $L_{\langle \sigma/2+1,\sigma\rangle}[i'']$, where $i'' = \mathsf{rank}_1(W, i)$ and we continue recursively by the right child of v with position i''.

Operation rank on bitvectors can be done in $O(1)$ time adding only sublinear space on top of the bitvector [19, 51]. Thus, in time $O(\log\sigma)$ we find a leaf and determine $L[i]$.

Note that all bitvectors stored at an internal wavelet tree level amount to n bits, and thus the wavelet tree requires $n\log_2\sigma$ bits, that is, the same as a plain representation of L. The total space of the wavelet tree is $n\log_2\sigma + o(n\log\sigma) + O(\sigma\log n)$ bits. The term $o(n\log\sigma)$ covers the extra space required by the sublinear-space rank data structures, whereas the $O(\sigma\log n)$ space is used to store the $O(\sigma)$ wavelet tree pointers (a pointer requires $w = \Theta(\log n)$ bits in the RAM model of computation).

A similar algorithm can be used to compute $\mathsf{rank}_c(L, i)$. We start at the wavelet tree root v and, if c descends by the left child, we recursively go left with $i \leftarrow \mathsf{rank}_0(W, i)$; otherwise we recursively go right with $i \leftarrow \mathsf{rank}_1(W, i)$. When we arrive at the leaf c, the current value of i is the answer.

We can then compute, in time $O(\log\sigma)$, the values needed in the LF and the backward search formulas (Eqs. (5) to (7)). Note that the arrays C_x needed in those formulas require $O(\sigma\log n)$ bits to store $O(\sigma)$ $\log(n)$-bit numbers. This is already accounted for in the space we gave for wavelet trees.

Example 9 Fig. 6 shows the wavelet tree of sequence L_p for our running example (ignore the slanted bitvectors for now). To compute $\mathsf{rank}_4(L_\mathrm{p}, 5)$, we start at position $i \leftarrow 4$ of the root (the short diagonal arrows track our position). Since leaf 4 is to the right, we go right and set $i \leftarrow \mathsf{rank}_1(W_{\langle 1,5\rangle}, 5) = 3$. On the right child of the root, we see that leaf 4 descends to the left, so we go left with $i \leftarrow \mathsf{rank}_0(W_{\langle 4,5\rangle}, 3) = 2$, arriving at the leaf of 4. Thus $\mathsf{rank}_4(L_\mathrm{p}, 5) = i = 2$. Added to $C_\mathrm{p}[4] = 10$, we obtain position $12 = \mathsf{LF}_\mathrm{p}(5)$. □

3.8 Solving other problems with Wavelet Trees

Wavelet trees can be used for many other purposes [32, 55]. We will indeed make use of their extended capabilities for our algorithm. A good warmup is the following problem.

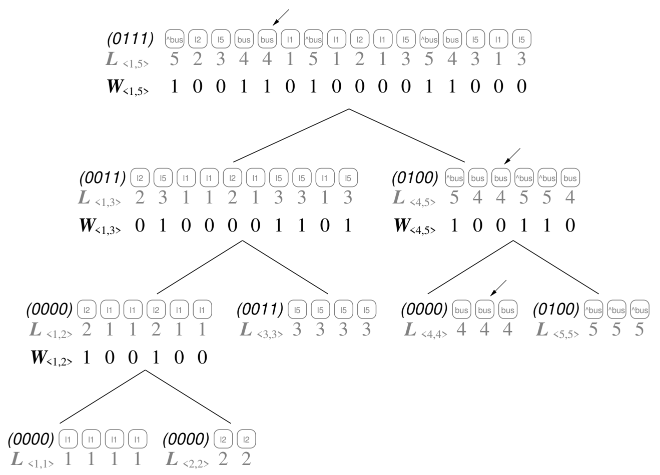


Fig. 6 The wavelet tree of the sequence L_{p} of Fig. 5. The short diagonal arrows track $L[5]$. The slanted bitvectors on the nodes refer to the B entries of the automaton of Fig. 8, and will be discussed later.

Definition 12 The *range listing problem* is that of listing all the distinct values in a range $L[b\,..\,e]$ of a string L.

This problem can be solved on the wavelet tree of L as follows. We start at the root's bitvector W and descend left with the interval $L_{\langle 1,\sigma/2\rangle}[b'\,..\,e']$, where $b' = \mathsf{rank}_0(W, b-1)+1$ and $e' = \mathsf{rank}_0(W, e)$. We also descend right with the interval $L_{\langle \sigma/2+1,\sigma\rangle}[b''\,..\,e'']$, where $b'' = \mathsf{rank}_1(W, b-1)+1$ and $e'' = \mathsf{rank}_1(W, e)$. We abandon every empty interval and instead report every leaf we arrive at.

Since every node with a nonempty interval has at least one child with a nonempty interval, we can charge to the reported leaves the $O(\log\sigma)$ cost of traversing the path up to them. The total time is then $O(\log\sigma)$ per distinct symbol reported, irrespective of the total number of symbols. Note that this algorithm first visits the leftmost leaves, and thus it reports the symbols in increasing order.

A related problem is that of intersecting the symbols in ranges of L.

Definition 13 The *range intersection problem* is that of listing all the distinct common values in two ranges $L[b_1\,..\,e_1]$ and $L[b_2\,..\,e_2]$ of a string L.

To solve this problem on the wavelet tree of L, we traverse it keeping track of both ranges, as for range listing, but stop descending when any of the two ranges becomes empty. Only the paths that reach the leaves with both nonempty intervals report an element of the intersection.

This time our search can finish at internal wavelet tree nodes, so we do not obtain $O(\log\sigma)$ time per reported element. Still, the described algorithm matches the lower bound for computing intersections given in Lemma 1 [32].

Our third problem is to count, rather than list, the distinct values in a range.

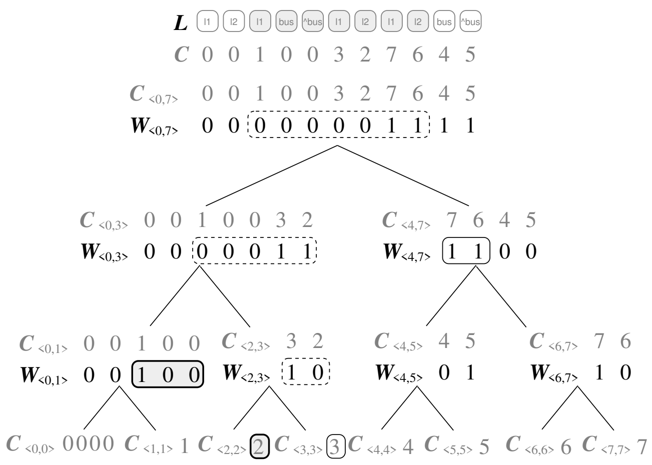


Fig. 7 A sequence L and its array C. We show how to solve the colored range query on $L[3\,..\,9]$.

Definition 14 The *colored range counting problem* is that of counting the number of distinct values in a range $L[b\,..\,e]$ of a string L.

We can solve this problem in $O(\log n)$ time (where n is the length of L) by adding $n\log_2 n + o(n\log n)$ extra bits of space on top of L [53, 33]. This technique defines an array $C[1\,..\,n]$ from $L[1\,..\,n]$. We set $C[i] = 0$ if i is the first occurrence of symbol $L[i]$. Otherwise, we set $C[i] = \max\{j \mid j < i,\ L[i] = L[j]\}$, that is, the previous position of L where the symbol $L[i]$ appears. It is not hard to see that the number of different values in $L[b\,..\,e]$ is exactly the number of symbols strictly smaller than b in $C[b\,..\,e]$.

The number of symbols that belong to $[0\,..\,b-1]$ within the range $C[b\,..\,e]$ can be counted in $O(\log n)$ time provided we represent C as a wavelet tree. This is a specific case of the *range counting query* [55]. In order to solve it, the algorithm descends in the wavelet as for range reporting $C[b\,..\,e]$. Let the $[b\,..\,e]$ become $[b'\,..\,e']$ when mapped to a wavelet tree node of C corresponding to the alphabet range $[l\,..\,r]$ (a subrange of the alphabet $[0\,..\,n-1]$ of C). The traversal stops in the node if $b' > e'$, $l \geq b$, or $r < b$. The first two cases do not count any element because there are no useful values below those nodes: the range $[b\,..\,e]$ does not map to the node or no symbol smaller than b appears below the node, respectively. In the third case, instead, all the symbols below the node belong to $[0\,..\,b-1]$, so we add the $e' - b' + 1$ elements to the count.

The total time is $O(\log n)$ since the range $[b\,..\,e]$ can be covered with $O(\log n)$ maximal nodes of the wavelet tree of C, which have $O(\log n)$ ancestors in total. The cost of the algorithm can be charged to that of visiting those nodes [55].

Example 10 Fig. 7 shows an example of a colored range query over a sequence L. Below the sequence L we show the array C and its wavelet tree. For example, $C[6] = 3$ because the previous occurrence of symbol $L[6] = \mathsf{l1}$ is at

$L[3]$. To query the range $L[3\,..\,9]$, we count all the elements in $C[3\,..\,9]$ that are < 3. A top-down traversal of the wavelet tree reports 4 different elements. The rounded rectangles track the range in each node. Dashed rectangles mean the traversal continues and solid ones represent nodes where the algorithm stops. Those that count some solutions have a gray background. For example, the algorithm stops in $W_{\langle 4,7\rangle}$ because all the elements below that node are in $[6\,..\,7]$, thus its subtree does not contain any solution. In $W_{\langle 0,1\rangle}$, instead, the alphabet is smaller than 3. Therefore, the 3 elements within the range of that node are counted. The leaf $C_{\langle 2,2\rangle}$ adds the last element, for a total of 4 different elements. □

4 Our Approach

We consider evaluation of (2)RPQs under set semantics, where, as discussed by Arenas et al. [6], evaluating recursive RPQs under bag semantics is costly as it involves counting a potentially exponential number of (simple) paths. To evaluate RPQs, we navigate backwards all the paths that match them, conceptually performing a BFS traversal of the product graph. Our data structure will include part of the ring structure, more precisely, the wavelet trees representing sequences L_{p} and L_{s}, as well as all the arrays C_*.

The sets of subjects and objects are equal and correspond to the nodes V in the graph; each node may act as a subject (i.e., edge source) or as an object (i.e., edge target). The set of predicates $P \subseteq \Sigma^{\leftrightarrow}$ corresponds to the edge labels of $G^{\leftrightarrow}$. In order to represent nodes and predicates in wavelet trees, we map the identifiers to integer intervals, identifying $V = [1\,..\,|V|]$ and $P = [1\,..\,|P|]$.

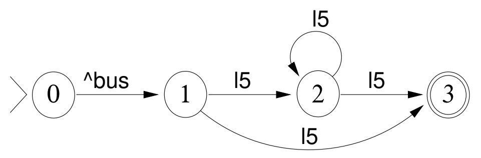
We will first focus on RPQs of the form (x, E, o), where $x \in \Phi$ and $o \in V$. We will build the Glushkov automaton for E and use it to navigate backwards, from objects to subjects. Since we use the NFA backwards, we will start from its final states, $D = F$, use the reverse Glushkov simulation of Eq. (4), and report a valid binding $x = s$ at every node $s \in V$ where the initial NFA state is activated. The navigation starts from the range of o in L_{p}.

This technique also supports RPQs of the form (s, E, y), where $s \in V$ and $y \in \Phi$, by reversing E and searching instead for $(y, \hat{}E, s)$. We will later consider the other kinds of RPQs.

We note that since the alphabet of E is P, our vector $B[1\,..\,|P|]$ for the bit-parallel NFA simulation is of size $O(|P|)$, but still preprocessing the RPQ takes time $O(2^m)$ with lazy initialization. This adds a working space usage of $O(2^m + |P|)$ on top of the ring.

Example 11 The RPQ (`Baq`, `l5`$^+$/`bus`, y) tells us where we can reach from `Baq` via line 5 then taking one bus. The reversed regular expression is $\hat{}E$ = `^bus/l5`*`/l5`, equivalent to the example $a/b^*/b$ of Fig. 3. We have converted `bus` to `^bus` to reverse the edge direction (we do not do this for `l5`, which is bidirectional). Fig. 8 shows the Glushkov automaton for this regular expression; note that B[`^bus`] corresponds to $B[a]$ and B[`l5`] to $B[b]$ in Fig. 3, and that the alphabet of the regular expression is the set of predicates.

We first start from node 5 (`Baq`) and work backwards. We thus start from $L_{\mathrm{p}}[C_{\mathrm{o}}[5] + 1\,..\,C_{\mathrm{o}}[6]] = L_{\mathrm{p}}[15\,..\,16]$, and report all the nodes that we can reach in reverse from there that activate the initial state of our automaton, 0. □



T'[0000] = 0000	F = 0001
T'[0001] = 0110	B[l1] = 0000
T'[0010] = 0110	B[l2] = 0000
T'[0011] = 0110	B[l5] = 0011
T'[0100] = 1000	B[bus] = 0000
...	B[^bus] = 0100
T'[1111] = 1110	

Fig. 8 The Glushkov automaton for the regular expression `^bus/l5`*`/l5`, its bitvector F and table B, and the transition table T' of its reversed automaton.

As said, we will virtually traverse our induced subgraph $G'_{\mathcal{M}}$ of the product graph $G_{\mathcal{M}}$ backwards. To simulate this process, we perform a sequence of (backward) NFA steps, traversing in reverse the possible paths ρ that match $\hat{}E$. The traversal abandons every branch where the NFA runs out of active states. Every time it reaches the initial state we report the current node. Each NFA step starts and ends at a range of L_{p} corresponding to the current object (initially, o), and is simulated in the following three parts:

1. We find all the predicates labeling edges that point to the current object. This leads us from the interval in L_{p} (corresponding to the object) to several intervals in L_{s} (corresponding to distinct predicates for that object).
2. We find the subjects of edges labeled with each such predicate. This leads us from each interval in L_{s} (corresponding to a predicate leading to our object) to several intervals in L_{o} (corresponding to distinct subjects).
3. We regard each of those subjects as an object again, by mapping each resulting range in L_{o} to the corresponding range in L_{p}. We only need C_{o} to do this, not L_{o}.

After steps 1 and 2, we abandon the branch if the resulting range is empty. After step 2, we perform the NFA transition and abandon the branch if we run out of active states ($D = 0$). We report the subject if the initial state is active in D.

Note that, in step 1, we are only interested in predicates that lead to some node in $G_{\mathcal{M}}$. That is, we want predicates that lead not only to the current object, but also to active NFA states. In step 2, we are only interested in subjects that have not been visited before with the same NFA states, so as to avoid falling into cycles of $G_{\mathcal{M}}$.

In terms of the product graph, visiting a node v of G with a set D of active NFA states corresponds to traversing *simultaneously* all nodes of $G'_{\mathcal{M}}$ that combine v with an active state in D. Thus, bit-parallelism enables us to perform less work than classical techniques that visit $G'_{\mathcal{M}}$ node by node. Similarly, the ability of wavelet trees to work on ranges of symbols will let us work simultaneously on various nodes of G. Furthermore, we will combine this ability with Fact 1 to carry out steps 1 and 2 in such a way that the time spent is upper bounded by Eq. (2). We now detail each part.

4.1 Part one: Finding predicates from objects

The first part finds the distinct predicates p that lead to (i.e., cyclically precede in the (o, s, p) triples) the current range of objects. We can use the wavelet tree of L_p to discover all the distinct predicates p in $L_\mathrm{p}[b_o\,..\,e_o]$, using the range listing algorithm described in Section 3.8. Next, we can identify which of those predicates p lead to a currently active NFA state, that is, such that $D \,\&\, B[p] \neq 0$ per Eq. (4).

Listing and checking Eq. (4) on every distinct predicate p, however, may involve checking many predicates that lead to no node in $G_{\mathcal{M}}$, which we aim to avoid as it leads to a suboptimal cost in terms of $Cost_G$ (Section 3.2).

Instead, we will find the useful predicates p efficiently thanks to Fact 1 and to an enhancement of the wavelet tree of L_p, where we will have $B[\cdot]$ entries not only for the predicates p, but also for all the other $|P|-1$ nodes in the wavelet tree of L_p: Let v be a wavelet tree node; then $B[v]$ will be the bitwise-or of the $B[p]$ entries of all the symbols p descending from v. This allows us to quickly determine that whole ranges of values of p will activate no NFA state.

Example 12 The $B[\cdot]$ entries for all the nodes of the wavelet tree of L_p are written as slanted bitvectors on the nodes in Fig. 6. Those on the leaves correspond to the entries in Fig. 8, and those on internal nodes to the bitwise-or of their children. Thus, for example, the bitvector *0011* on the left node of level 2 indicates that via its descendents 11, 12 and 15, one reaches only states 2 and 3 in the NFA of Fig. 8. □

This enhancement can be built with lazy initialization from the $B[p]$s in $O(m \log |P|)$ time, by starting with all $B[v] = 0$ and working upwards only from the nonzero entries $B[p]$, doing $B[v] \leftarrow B[v] \mid B[p]$ for every ancestor v of p. The extra space is still $O(|P|)$, and we can store the entries $B[v]$ in heap order, following the (perfectly balanced) wavelet tree of L_p.

With this extension of B, we proceed as follows. We start from the root v of the wavelet tree of L_p, with the range $[b\,..\,e] = [b_o\,..\,e_o]$ and bitvector D. If $D \,\&\, B[v] = 0$, we stop. Otherwise, if v is a leaf p, then we report the interval $L_\mathrm{s}[b\,..\,e]$. Otherwise, we recursively continue with the left and right children v_l and v_r of v, with the intervals $[b\,..\,e] = [\mathsf{rank}_0(W, b-1)+1\,..\,\mathsf{rank}_0(W, e)]$ for v_l and $[b\,..\,e] = [\mathsf{rank}_1(W, b-1)+1\,..\,\mathsf{rank}_1(W, e)]$ for v_r.

Example 13 To start the search from $L_\mathrm{p}[15\,..\,16]$ and $D = 0001$, we must first find all distinct values in the range that label transitions leading to an state active in D. We start from the wavelet tree root $v_{\langle 1,5\rangle}$ of Fig. 6, with the range $L_{\langle 1,5\rangle}[15\,..\,16]$. We descend to the left child, $v_{\langle 1,3\rangle}$ since $B[v_{\langle 1,3\rangle}] \,\&\, D = 0011 \,\&\, 0001 \neq 0000$ and thus there are relevant transition labels below it. When descending, we map the range to $L_{\langle 1,3\rangle}[9\,..\,10]$ (because $\mathsf{rank}_0(W_{\langle 1,5\rangle}, 15-1)+1 = 9$ and $\mathsf{rank}_0(W_{\langle 1,5\rangle}, 16) = 10$). From $v_{\langle 1,3\rangle}$, we do not descend to $v_{\langle 1,2\rangle}$ since $B[v_{\langle 1,2\rangle}] \,\&\, D = 0000 \,\&\, 0001 = 0000$ and thus no relevant transition labels descend from it (though there is a 1 in our range $L_{\langle 1,3\rangle}[9\,..\,10]$ indicating an l1 reaching Baq, it does not lead to active NFA states). Instead, we descend to $v_{\langle 3,3\rangle}$ because $B[v_{\langle 3,3\rangle}] \,\&\, D = 0011 \,\&\, 0001 \neq 0000$. Since it is a leaf, we have found a relevant label (3, i.e., l5) reaching our range (i.e., Baq). Its range is $L_{\langle 3,3\rangle}[4\,..\,4]$, which added to the number of leaves in l1 and l2 (equivalent to $C_\mathrm{p}[3] = 6$) yields the range $L_\mathrm{s}[10\,..\,10]$, completing the backward search step for symbol l5 (recall Eqs. (6) and (7)).

On the other hand, we do not descend from $v_{\langle 1,5\rangle}$ to its right child, $v_{\langle 4,5\rangle}$, because $B[v_{\langle 4,5\rangle}] \,\&\, D = 0100 \,\&\, 0001 = 0000$. If we did, we would obtain an empty interval in $L_{\langle 4,5\rangle}$ because there are no 4s or 5s in $L_{\langle 1,5\rangle}[15\,..\,16]$. □

In order to analyze this process, let us adapt the definition of $lab_{\mathcal{M}}$ to sets of states D and to the reverse traversal done with our Glushkov automaton:

$$lab(D) \;=\; \{p \mid D \,\&\, B[p] \neq 0\}.$$

Let us also adapt lab_G to a set of predicates in L_p:

$$lab(b_o, e_o) \;=\; \{L_\mathrm{p}[i] \mid b_o \leq i \leq e_o\}.$$

The process we described corresponds to intersecting, using the wavelet tree of L_p enriched with the array $B[\cdot]$, $lab(D)$ with $lab(b_o, e_o)$. The cost of such an intersection algorithm is analyzed by Gagie et al. [31, Thm. 8], in terms of the number of wavelet tree nodes that cover both sets. Their result is that, if the sets of wavelet tree leaves corresponding to both sets are X and Y, then the cost of the wavelet tree intersection, or equivalently, the number of nodes that are ancestors of leaves in both X and Y, is upper-bounded by $\alpha(X, Y) \log \frac{|P|}{\alpha(X,Y)}$. Our algorithm proceeds in exactly the same way: the wavelet tree nodes that are traversed correspond to the ancestors of leaves that are both in $lab(D)$ and in $lab(b_o, e_o)$. As a result, the cost of our intersection is

$$O\left(\alpha(lab(D), lab(b_o, e_o)) \log \frac{|P|}{\alpha(lab(D), lab(b_o, e_o))}\right). \quad (8)$$

Note that this scheme works thanks to Fact 1, since we must intersect every $B[p]$ with *the same* set D of active states. Furthermore, in terms of the product graph traversal, we are simultaneously processing all nodes that combine o with the active states in D, thus allowing us to obtain all the distinct edges of $G'_{\mathcal{M}}$ that we can traverse from the current nodes of $G'_{\mathcal{M}}$. This leads to a cost that can be even lower than the formula of Eq. (2), because $\alpha(X_1 \cup X_2, Y) \leq \alpha(X_1, Y) + \alpha(X_2, Y)$ and $\alpha(X, Y_1 \cup Y_2) \leq \alpha(X, Y_1) + \alpha(X, Y_2)$. Therefore, $\alpha(lab(D), lab(b_o, e_o)) \leq \sum_{q \in D,\, x \in L_{\mathrm{p}}[b_o..e_o]} \alpha(lab_{\mathcal{M}}(q), lab_G(x))$ and thus our cost in Eq. (8) is a lower bound to $alt(G'_{\mathcal{M}}, |P|)$ (Eq. (2)).

4.2 Part two: Finding subjects from predicates

The second part of the process starts at each of the ranges $L_{\mathrm{s}}[b_p..e_p]$ reported by the first part, and traverses the wavelet tree of L_{s} using the range reporting algorithm of Section 3.8 to find all the distinct subjects s in that range, mapping each to an interval $L_{\mathrm{o}}[b_s..e_s]$. By Fact 1, the set of active NFA states will be the same, $D \leftarrow T'[D \,\&\, B[p]]$ (Eq. (4)), for all those subjects. If D contains the initial state, we report that subject s starts a path of the 2RPQ (i.e., we report (s, o) as an answer to the query).

Example 14 Once we obtain the range $L_{\mathrm{s}}[10..10] = 4$ (BA) from edge label 3 (l5), giving the edge BA $\xrightarrow{\text{l5}}$ Baq, we update $D \leftarrow T'[D \,\&\, B[3]] = T'[0001 \,\&\, 0011] = T'[0001] = 0110$, activating states 1 and 2 in our NFA (see Fig. 8). This new state D is independent of the subject we arrived at. □

We need to prevent falling into cycles, however: If we arrive at a subject s with a subset of the NFA states we have already visited s with, we must stop because we are repeating nodes in the product graph.

To avoid cycles, we store an array $S[1..|V|]$ that, for each subject s, stores a bitvector $S[s]$ with all the active NFA states we already reached s with. This adds $O(|V|)$ working space, but S can be zeroed in constant time with lazy initialization. If we arrive at s and $D \mid S[s] = S[s]$, then s can be skipped because we are only repeating NFA states on it (and cycling in $G_{\mathcal{M}}$). Otherwise, we set $D \leftarrow D \,\&\, \sim S[s]$ and then $S[s] \leftarrow D \mid S[s]$, where "$\sim$" is bitwise-not. This leaves only the NFA states that are new to s in D, and adds to $S[s]$ the new active states we have arrived at s with. At the beginning of our traversal, after zeroing D we mark the states F on the node o where we start the search.

The problem with this simple solution is that, again, we are generating each subject s and then verifying if it has already been visited. Once again, we can do better.

To implement this filter more efficiently, we will again exploit Fact 1 and enhance the wavelet tree of L_{s}. We use the same technique of storing $S[\cdot]$ entries at wavelet tree nodes

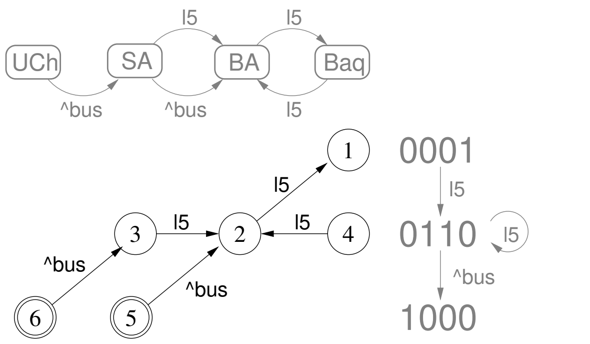


Fig. 9 The visited part of the product graph to match the NFA of Fig. 8 in our graph of Figs. 1 and 5. We show in the rows the D values (i.e., sets of NFA states) we visit, and label the graph edges for readability.

v of L_{s}, so as to avoid descending by a branch if all the subjects below it have already been visited with all the active states in D. For this, $S[v]$ must be the intersection of those $S[s]$ cells below v. When reaching v along the range listing algorithm on $L_{\mathrm{s}}[b_p..e_p]$, we prune the traversal at v if $D \mid S[v] = S[v]$; otherwise we continue via both left and right children. If we arrive at a useful wavelet tree leaf s, we set $S[s] \leftarrow D \mid S[s]$ as explained, and then climb up the wavelet tree setting $S[v] \leftarrow S[v_l] \,\&\, S[v_r]$ for every ancestor v of s with children v_l and v_r (we can stop this upward update as soon as v does not change).

Just as in Section 4.1, this process can be analyzed in terms of the alternation complexity between the set of subjects that lead to the given object by the given label, and the set of subjects not yet reached with all the states active in D. For simplicity, we omit this analysis and simply charge the algorithm $O(\log |V|)$ time for the wavelet tree traversal of every edge in $A'_{\mathcal{M}}$, that is, $O(|A'_{\mathcal{M}}| \log |V|)$ overall.

Not visiting s with a subset of the NFA states of previous visits ensures that we never work more than classical product graph traversals: every time we reprocess a node s of G, we must be including a new NFA state in D (instead, we can be faster because we handle several NFA states together, as explained). Again, we can efficiently filter the subjects with the wavelet tree thanks to Fact 1, because all the subjects s are visited with the same set of states D.

4.3 Part three: Mapping subjects back to objects

In the third part, we report each useful subject s we must consider, with its corresponding state D. In order to proceed with the next step of the simulation, we must map this range of subjects to the same range of nodes seen as objects. This is done with the array C_{o}, where $C_{\mathrm{o}}[s]$ is the number of symbols smaller than s in L_{o}. Thus, $L_{\mathrm{p}}[C_{\mathrm{o}}[s]+1..C_{\mathrm{o}}[s+1]]$ corresponds to the interval of L_{p} that is aligned to object s.

Then, as explained, we restart part one with each s for which $C_{\mathrm{o}}[s+1] > C_{\mathrm{o}}[s]$, with state D.

Example 15 Fig. 9 (bottom left) shows the traversed part of the product graph for our running example. Numbers represent the order in which nodes are traversed. Edges are labeled for readability only. The nodes of the product graph are formed from pairs of graph nodes (Fig. 1) and automaton states (Fig. 8). Columns indicate graph nodes; e.g., nodes 1 and 4 correspond to Baq, being in the same column. Rows indicate states, where on the right we show bitvectors denoting *sets* of NFA states active on that row; for example, for nodes 2, 3 and 4, the bitvector 0110 indicates that we are processing states 1 and 2 simultaneously at these steps. Thus, e.g., node 4 actually corresponds to two nodes in the product graph that we are processing simultaneously: $(\mathsf{Baq}, 1)$ and $(\mathsf{Baq}, 2)$. The examples illustrated in parts one, two and three of this section solve the (reverse) traversal $2 \xrightarrow{\mathsf{l5}} 1$. The same process can be applied successively in order to evaluate all six steps, as illustrated in the Appendix. □

When some of the reported subjects s are successive and share the same state D after removing $S[s]$ from it, we can speed up the algorithm. Since k successive subjects $[s \mathinner{..} s+k-1]$ have consecutive intervals in L_{p}, we can map all of them together to the same interval $L_{\mathrm{p}}[C_{\mathrm{o}}[s]+1 \mathinner{..} C_{\mathrm{o}}[s+k]]$. Note that parts one and two do not require that the intervals in L_{p} and L_{s} correspond to a single target node. Therefore, we can efficiently restart part one with only one range instead of k different intervals, further exploiting the ability of our algorithm to simultaneously process several nodes of G.

Checking whether the subjects are consecutive is easy because, when we enumerate the distinct subjects using our modified range listing on $L_{\mathrm{s}}[b_p \mathinner{..} e_p]$, the subjects are output in increasing order regardless of their position in L_{s}.

4.4 Time complexity

The following theorem summarizes the cost of our data structure and traversal algorithm.

Theorem 1 *Let G be a labeled graph with nodes in V and labels in P. Consider an RPQ (x, E, y) where x is constant and y is variable or vice versa, E has m literals, and the computer word holds $w = \Omega(m)$ bits. The ring representation of G can return all the matching pairs (s, o) for the RPQ in time $O(2^m + m \log|P| + |A'_{\mathcal{M}}| \log|V| + alt(G'_{\mathcal{M}}, |P|))$, where $G'_{\mathcal{M}}(V'_{\mathcal{M}}, A'_{\mathcal{M}})$ is the subgraph, of the product graph of G and Glushkov's NFA $\mathcal{M}$ of E, induced by a BFS traversal from the constant node, and $alt(G'_{\mathcal{M}}, |P|)$ is defined in Eq. (2). The working space of the query is $O(2^m + |P| + |V| + |V'_{\mathcal{M}}|)$ words. For any desired parameter $1 \le d \le m$, we can reduce the space to $O(d\,2^{m/d} + |P| + |V| + |V'_{\mathcal{M}}|)$ words, and the time becomes $O(d\,2^{m/d} + m^2 + m \log|P| + d|V'_{\mathcal{M}}| + |A'_{\mathcal{M}}| \log|V| + alt(G'_{\mathcal{M}}, |P|))$. If $w = o(m)$, all the space and time complexities are multiplied by $O(m/w)$.*

Proof First consider the query (x, E, o) for $s \in \Phi$ and $o \in V$. The algorithm virtually visits the nodes of $G'_{\mathcal{M}}$ in reverse order. It starts simultaneously from all the nodes $(o, f) \in V'_{\mathcal{M}}$, for all the final NFA states $f \in F$ (we regard F and D as sets of NFA states). The algorithm preserves the invariant that, if it is at node $v \in V$ with the active NFA states D, then it is the first time it simulates the visit of the node $(v, d) \in V'_{\mathcal{M}}$ for any NFA state $d \in D$. Edges from new nodes $(v', d') \in V'_{\mathcal{M}}$ to (v, d) are found in three parts. In the first part, it finds every distinct label $p \in P$ of edges in $A'_{\mathcal{M}}$ that lead to (v, d) for some $d \in D$. As seen in Section 4.1, the total cost for this part can be bounded by $O(alt(G'_{\mathcal{M}}, |P|))$. In the second part, for each label p found, it finds every distinct node $v' \in V$ such that we reach (v, d) (for some $d \in D$) from some unvisited node (v', d') of $V'_{\mathcal{M}}$ via label p (it obtains simultaneously the set D' of all those states d'). We can then charge the cost to the edges $((v', d'), (v, d)) \in A'_{\mathcal{M}}$, yielding a total cost of $O(|A'_{\mathcal{M}}| \log|V|)$ as shown in Section 4.2. The third part takes $O(1)$ time per node arrived at, which becomes the current node in the next iteration.

As for the initialization costs, Glushkov's construction takes $O(m^2)$ time [17] to mark all bits in $B[1 \mathinner{..} |P|]$ after the constant-time lazy initialization of B. The construction of the bit-parallel tables takes time $O(2^m)$ [56], dominated by table T. Computing the cells B for the internal wavelet tree nodes of L_{p} using lazy initialization adds $O(m \log|P|)$ time since only $O(m)$ wavelet tree leaves have nonzero cells in B. The lazy initialization of S for the wavelet tree nodes of L_{s} adds $O(1)$ time.

The working space is $O(m\,2^m)$ bits (i.e., $O(2^m)$ words when $m = O(w)$) for the bit-parallel simulation of the NFA, $O(m(|P| + |V|))$ bits for the tables B/S on the wavelet tree nodes of $L_{\mathrm{p}}/L_{\mathrm{s}}$, $O(|P| + |V|)$ bits for the compact structures for the lazy initialization of tables B/D (see App. C, [54]), and $O(|V'_{\mathcal{M}}|)$ words for the BFS traversals (to store the state D and the range in L_{p} of each node of $G'_{\mathcal{M}}$ in the queue).

Partitioning the table T into $d > 1$ subtables to reduce the exponential term 2^m can be done whenever it is convenient. As mentioned in Section 3.5, the construction time of T and its space become $O(d\,2^{m/d})$ words of m bits. This also impacts on the traversal time: since we recompute D using T for each label we process along the traversal of $G'_{\mathcal{M}}$, splitting T adds $O(d\,|V'_{\mathcal{M}}|)$ to the cost.

Finally, if the computer word cannot hold $m + 1$ bits, we store $\lceil (m+1)/w \rceil$ words for D, F, and the cells of table T and arrays B and S. This impacts every term of our space complexity, which must be then multiplied by $O(m/w)$. Similarly, all the terms of our time complexity involve processing some of those cells, and thus our time complexity is multiplied by $O(m/w)$, too.

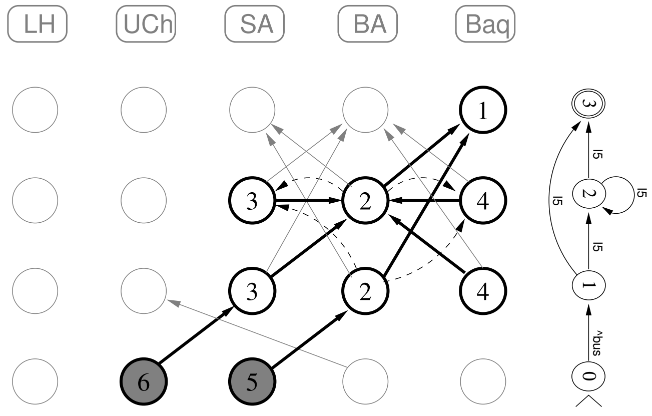


Fig. 10 The product graph $G_{\mathcal{M}}$ of the graph of Fig. 1 and the NFA of Fig. 8, highlighting in black the subgraph $G'_{\mathcal{M}}$ that is traversed (backwards) in Fig. 9. Dashed arrows lead to the loops we avoid.

To solve queries (s, E, y), where $s \in V$ and $y \in \Phi$, we just reverse the query and proceed as before on $(y, \hat{} E, s)$ (as we did in our running example). □

The theorem does not reflect that we can process several NFA states d simultaneously when traversing the nodes (v, d) of $G'_{\mathcal{M}}$, thanks to the bit-parallel simulation of the NFA. On the other extreme, we can choose $d = m$ (i.e., simulate the automaton state-by-state) so as to completely remove the exponential dependence on m; we then obtain a time complexity of $O(m(m+\log|P|+|V'_{\mathcal{M}}|)+|A'_{\mathcal{M}}|\log|V|+alt(G'_{\mathcal{M}},|P|))$. In practice, if we have long RPQs, we can increase d, which will decrease the exponential term in m from both time and space, but will multiply (data-dependent) terms in the time by d. If we have short RPQ expressions (and lots of data), we can decrease d, which will have the inverse effect.

Example 16 Fig. 10 shows the product graph $G_{\mathcal{M}}$ of our running example, with the nodes and edges of $G'_{\mathcal{M}}$ in bold. The rows now correspond to the NFA states. The dashed edges form cycles in $G'_{\mathcal{M}}$ and we avoid them. The shaded nodes correspond to reported results. Comparing the figure with Fig. 9, however, one can see that our simulation processes the nodes of the second and third rows of $G_{\mathcal{M}}$ simultaneously (in row 0110 of Fig. 9). We maintain in Fig. 10 the numbering of the nodes of Fig. 9, which helps see the nodes we visit simultaneously. □

4.5 Other kinds of RPQs

We have so far considered RPQs where one extreme is fixed and the other is variable. For RPQs where both $s, o \in V$ are fixed, we can start from o and process $\hat{} E$, stopping when we find s or run out of active states (or vice versa with E).

The most complex case, (x, E, y) with $x, y \in \Phi$, has variables for both subject and object, where we must find all the pairs (s, o) connected by a path matching E. We could handle this query by launching $|V|$ queries (s, E, y), one per possible subject s, but this could be very inefficient if many subjects s do not lead to answers (s, o).

A more efficient solution uses the ability of the ring and of wavelet trees to work on ranges of symbols, in order to spot the useful sources s. We determine which RPQs (s, E, y) produce some output by working backwards from the objects o using queries (x, E, o) and collecting the useful values s for x. Instead of starting with each specific object o, however, we will start with the full range $L_{\mathrm{p}}[1..n]$. Exactly the same algorithm we have described for queries (x, E, o), now started with the full range, obtains all the subjects s leading to *some* object by E. Then, for every subject s we arrived at, we run the RPQ (s, E, y), and report (s, o) for each object o found in this search.

We can, symmetrically, first find the objects o reachable via E from some subject, and then run only the useful queries (x, E, o). Which is better depends on the data. In the next section we discuss more sophisticated ways to handle RPQs with two variables (as are often the most expensive).

5 Optimizations

We present two novel optimizations for RPQs, based on node ordering and splitting RPQs.

5.1 Node ordering

As described in Section 4, the ring maps nodes and predicates to ranges of integer identifiers. Any mapping is equivalent in terms of correctness, though some can be preferred for other reasons; e.g., using the lexicographic rank of the original string identifiers enables a more compact storage of the dictionary that maps strings to identifiers and back [47].

Other identifier orderings can be beneficial for the time performance of our index, however. In part three of our algorithm (Section 4.3), we can reduce the number of intervals that restart our search when some of the reported subjects from $L_{\mathrm{s}}[b_{\mathrm{p}}..e_{\mathrm{p}}]$ are consecutive and share the same NFA state. We can then speed up our performance by ordering the nodes in a way that maximizes the number of consecutive subjects reported by L_{s}. Since our RPQ algorithm traverses the graph in BFS order, encoding the nodes in that order is a promising strategy. While this does improve times in practice, we obtained better results by exploiting the fact that our algorithm always departs from a range of edges with the same predicate and then enumerates the corresponding source nodes (part two, Section 4.2).

Our actual strategy uses a node ordering that locally preserves the BFS order for the nodes that are sources of edges with the same labels. We navigate the whole graph with a

BFS that visits backwards every node, starting a new BFS from an arbitrary remaining unvisited node. Each node visited for the first time through an edge labeled p is appended to a list $\mathcal{L}_p$. Once the BFS traversals are complete, the nodes in each list $\mathcal{L}_p$ are given consecutive identifiers.

While this does not guarantee that all the nodes in $L_s[b_p \,..\, e_p]$ will have consecutive identifiers, longer successions of identifiers will tend to appear in those intervals, as we will show in the experimental results.

5.2 Splitting RPQs

Our key contribution is a data representation that enables efficient traversal of the induced subgraph $G'_{\mathcal{M}}$, processing several nodes and states together, Still, other than using Glushkov's instead of Thompson's NFA to build $G_{\mathcal{M}}$, the subgraph $G'_{\mathcal{M}}$ itself is relatively standard, resulting from navigating in BFS order from the fixed node. When both extremes of the RPQ are variable, this requires trying out a potentially large number of starting points for the BFS traversal. A good part of the related work reviewed in Section 2 focuses instead on generating smart query plans via defining a much smaller subgraph $G'_{\mathcal{M}}$ to navigate. Their strategy is to split the RPQ into sub-RPQs at infrequent labels that must occur in all matching paths, so they are searched from a few starting nodes only. Their results are later combined to obtain the final answer. In this section we show how some extensions to our data representation enable the generation of highly competitive query plans of this kind, which can then be executed using our data structures.

Example 17 Assume we have a graph with information about museums visited by people (visits), bands where some violinists play (violinist_in), and events where different bands play (plays_in). Fig. 11 (top) shows the number of different source/target nodes of each predicate for the RPQ $E = \hat{}\text{visits}/\text{violinist\_in}/\text{plays\_in}$. For example, 10,000 different museums were visited by 100,000 different people. By starting the query with ^visits, from the 10,000 museum nodes, the algorithm advances to 100,000 nodes of people that have visited a museum. In the next step, the algorithm has to check if each of those persons plays the violin. Instead, by splitting the query at the violinists (source of violinist_in) we obtain the 300 nodes that are violinists, and move from them to the museums, without checking jobs of 100,000 people. Evaluating the query from the target of violinist_in to the end works analogously. Solving both sub-expressions (^visits and violinist_in/plays_in), and combining their results is likely much faster than computing the query from either end. □

Let us first focus on queries of the form (x, E, y) where $x, y \in \Phi$, that is, when both source and target are variables.

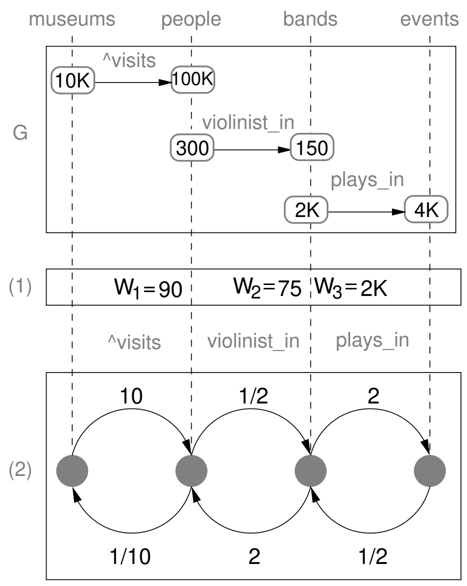


Fig. 11 The top rectangle exemplifies the number of graph nodes participating in the edges of an RPQ where $E = \hat{}\text{visits}/\text{violinist\_in}/\text{plays\_in}$. Each value within the bounded rectangles indicates the number of different nodes that are either sources or targets of a predicate. The rectangles in the middle and in the bottom illustrate the first and second steps of our query plan, respectively.

Those are likely to be the most expensive queries. Say we choose to split $E = E_1/E_2$ and then execute the subqueries $rpq_1 = (x, E_1, z)$ and $rpq_2 = (z, E_2, y)$ for each possible constant $z \in Z$, for some $Z \subseteq V$. For each solution (a, z) of rpq_1 and (z, b) of rpq_2, we add (a, b) to a set S_z. The final result is the union of all the sets S_z for every $z \in Z$.

To ensure correctness, we can choose $Z = V$. However, we can optimize by choosing a smaller set Z that still obtains all the results for (x, E, y). In particular, Z can be set to the intersection of the nodes E_1 can arrive at, and the nodes E_2 can start from. For example, if E_1 ends with a single predicate p_1, then it suffices that Z contain the targets of edges labeled p_1, where $\hat{}p_1$ forms a range in L_s. If E_2 starts with a single predicate p_2, then it suffices that Z contain the sources of edges labeled p_2, which form a range in L_s. If both conditions hold, Z can be the intersection of both ranges.

We will use the syntax tree of the regular expression (Section 3.4) to define positions where it can be safely split.

Definition 15 A position of E is *mandatory* iff all the ancestors of its corresponding leaf in the syntax tree are concatenation nodes.

Example 18 In the syntax tree of Fig. 2, positions 1 and 3 are mandatory, whereas 2 is not. □

Recall that, by property (3) of Glushkov's automaton (Section 3.4), the state k of Glushkov's NFA receives only transitions with label p_k. The following fact establishes the importance of mandatory positions: every path matching E must include an edge with their label.

Fact 2 *If k is a mandatory position of E, then every path from s to o that matches the RPQ (s, E, o) includes an edge labeled p_k.*

Let us call $\mathsf{source}(p)$ and $\mathsf{target}(p)$ the set of source and target nodes of edges labeled p, respectively. Fact 2 implies that, if k is a mandatory position, then we can set $Z = \mathsf{source}(p_k)$ or $Z = \mathsf{target}(p_k)$ and still get all the results. Our wavelet tree-based representation enables a more sophisticated way to define Z based on the following fact.

Fact 3 *If both k and $k+1$ are mandatory positions of an expression E, then every path from s to o that matches the RPQ (s, E, o) includes two consecutive edges labeled p_k and p_{k+1}.*

This fact implies that another valid set Z in such a case is $Z = \mathsf{target}(p_k) \cap \mathsf{source}(p_{k+1})$. We then define Z as the set of smallest cardinality among those induced by mandatory positions (Fact 2) and pairs of consecutive mandatory positions (Fact 3). Note that, since Fact 3 is more restrictive than Fact 2, we will always choose the former when we have consecutive mandatory positions.

Formally, let k be a mandatory position, then W_k is the minimum-cardinality set between $\mathsf{source}(p_k)$, $\mathsf{target}(p_k)$ and, if $k+1$ is also mandatory, $\mathsf{target}(p_k) \cap \mathsf{source}(p_{k+1})$. Ties can be broken arbitrarily. A rough plan then consists in setting Z to a set W_k of minimum cardinality.

Example 19 Fig. 11 (middle) defines the size of each possible W_k. For example, since the first mandatory position, with predicate $p_1 = \hat{}\mathsf{visits}$, is followed by another mandatory position, with predicate $p_2 = \mathsf{violinist\_in}$, W_1 is the intersection between $\mathsf{target}(\hat{}\mathsf{visits})$ and $\mathsf{source}(\mathsf{violinist\_in})$. For the example, we assume that $|W_1| = 90$, i.e., that 90 violinists that have visited a museum. On the other hand, since the last position, with predicate $p_3 = \mathsf{plays\_in}$, cannot apply Fact 3 and has more distinct targets than sources, it has $W_3 = \mathsf{source}(\mathsf{plays\_in})$. Once all the W_ks are computed, the minimum size of Z is 75, which is obtained from the intersection between $\mathsf{target}(\mathsf{violinist\_in})$ and $\mathsf{source}(\mathsf{plays\_in})$. Therefore, Z contains 75 nodes of bands and the two subexpressions are $E_1 = \hat{}\mathsf{visits}/\mathsf{violinist\_in}$ and $E_2 = \mathsf{plays\_in}$. This plan avoids checking if each possible band (2,000) plays in an event or if each museum (10,000) has been visited, which would be necessary if we started from the last or the first extreme of E, respectively. □

The rough approach of minimizing $|Z|$ does not always lead to the minimum number of steps to solve E_1 and E_2, however. Since we have to navigate the product graphs of both subexpressions, the cost of solving an RPQ depends, more precisely, on the number of traversed edges.

In order to approximate the number of traversed edges, we assume that the edges of each predicate distribute uniformly between its linked nodes. That is, in the subgraph with just the edges labeled p, all the nodes in $\mathsf{source}(p)$ have the same out-degree, and all the nodes in $\mathsf{target}(p)$ have the same in-degree. Those average degrees can then be computed, respectively, as

$$\mathsf{deg}^+(p) = \frac{|\mathsf{target}(p)|}{|\mathsf{source}(p)|}, \quad \mathsf{deg}^-(p) = \frac{|\mathsf{source}(p)|}{|\mathsf{target}(p)|}.$$

For simplicity, let us rename $p_1, \ldots, p_k$ to be the labels of the mandatory positions in E_1, and $p_{k+1}, \ldots, p_m$ those of E_2. The algorithm then traverses approximately $\mathsf{deg}^-(p_k)$ edges in E_1 and $\mathsf{deg}^+(p_k)$ in E_2 for each node in W_k. Focusing on E_1, the approximate number of traversed edges is $|W_k|\mathsf{deg}^-(p_k)$. This is also an approximation of the number of active nodes in the second step. The number of traversed edges in the second step is then approximated as $|W_k|\mathsf{deg}^-(p_k)\mathsf{deg}^-(p_{k-1})$, and so on. The computation is analogous for E_2, using $\mathsf{deg}^+(p)$ instead of $\mathsf{deg}^-(p)$. The number of traversed edges is then approximated as

$$|W_k| \left(\sum_{i=1}^{k} \prod_{j=i}^{k} \mathsf{deg}^-(p_j) + \sum_{i=k+1}^{m} \prod_{j=k+1}^{i} \mathsf{deg}^+(p_j) \right). \quad (9)$$

Our execution plan then splits E by computing each possible W_k according to Facts 2 and 3, setting Z to the W_k that minimizes Eq. (9). Note that this might still imply solving E left to right as a whole (i.e., not splitting), if its first position is mandatory and $W_1 = \mathsf{source}(p_1)$ turns out to be the best option, or right to left if its last position is mandatory and $W_m = \mathsf{target}(p_m)$ is the best option. We also solve E as a whole if it has no mandatory positions.

Example 20 In Fig. 11 (bottom), the rightward arrows are labeled with the out-degree of their predicate and the leftward ones with its in-degree (e.g., $\mathsf{deg}^+(\hat{}\mathsf{visits}) = 10$ and $\mathsf{deg}^-(\hat{}\mathsf{visits}) = 0.1$). With our rough strategy, we would set Z to the 75 nodes of bands with a violinist. By applying Eq. (9), that option yields an estimation of 225 traversed edges. Instead, with $E_1 = \hat{}\mathsf{visits}$ and $E_2 = \mathsf{violinist\_in}/\mathsf{plays\_in}$, the set Z consists of the 90 violinists that visit museums, and its estimated number of traversed edges decreases to 144. We then deem this partition to be more convenient. □

Since each term of Eq. (9) indicates the cost of solving a sub-RPQ, we can prioritize the subqueries depending on those costs. That is, we start solving rpq_1 and then rpq_2 when the first term of Eq. (9) is smaller than the second; otherwise we evaluate rpq_2 and then rpq_1. This quickly detects elements of Z that have no solution for one of the subqueries, and avoids evaluating the other subquery for them.

In order to use this evaluation plan, for Facts 2 and 3, we need to compute all the sizes $|\mathsf{source}(p_k)|$, $|\mathsf{target}(p_k)|$, and the intersection sizes $|\mathsf{target}(p_k) \cap \mathsf{source}(p_{k+1})|$. Computing $|\mathsf{source}(p_k)|$ is equivalent to obtaining the number of distinct elements in $L_s[C_p[p_k]+1\,..\,C_p[p_k+1]]$, with the $O(\log|V|)$-time algorithm described in Section 3.8 for colored range counting. This algorithm poses an extra space of $O(n \log n)$ bits, however; a low-space alternative is to actually produce the set $\mathsf{source}(p_k)$ with the algorithm for range listing described in the same section. Since we store $G^{\leftrightarrow}$, we can also compute $|\mathsf{target}(p_k)|$ by counting the number of distinct values in $L_s[C_p[\hat{~}p_k]+1\,..\,C_p[\hat{~}p_k+1]]$. The size of the intersections $\mathsf{target}(p_k) \cap \mathsf{source}(p_{k+1})$, instead, can only be computed by actually intersecting the intervals $[C_p[\hat{~}p_k]+1\,..\,C_p[\hat{~}p_k+1]]$ and $[C_p[p_{k+1}]+1\,..\,C_p[p_{k+1}+1]]$ in L_s and producing the output, with the range intersection algorithm also described in Section 3.8.

Our technique can be generalized to RPQs of the form (x, E, y) where x and/or y are constants. Note that at the extremes we can only apply Fact 2. We now must compute each W_k considering that x and/or y is an individual node of G. If $x = s \in V$, then we obtain $W_1 = \{s\}$. Analogously, when $y = o \in V$ we have $W_m = \{o\}$. These restrictions do not affect the average degrees of the nodes, so the computation of Eq. (9) stays the same. It is much more likely, however, that the equation recommends not to split E, but to process it as a whole starting from a constant extreme. We can also extend our technique to find multiple splitting points; we do not further follow this path because most RPQs are not large enough to make it profitable.

6 Implementation and Experiments

We implemented our scheme in C++11 using the succinct data structures library (SDSL, `https://github.com/simongog/sdsl-lite`). Our code is single-threaded, and does not use special CPU instructions. We ran our experiments on an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores, 15 MB of cache, and 384 GB of RAM. Our code was compiled using g++ with flags `-std=c++11`, `-O3`, and `-msse4.2`. The source code and data are available at `https://github.com/adriangbrandon/Ring-RPQ`.

6.1 Wikidata Benchmark

We first evaluate our approach on a Wikidata graph [69] of $n = 958{,}844{,}164$ edges, $|V| = 348{,}945{,}080$ nodes, $|S| = 106{,}736{,}662$ subjects, $|P| = 5{,}419$ predicates, and $|O| = 295{,}611{,}216$ objects. This graph occupies 7.9 GB (or 8.63 bytes per triple, bpt) in packed form (i.e., using $\lceil\log|S|\rceil + \lceil\log|P|\rceil + \lceil\log|O|\rceil$ bits per tuple) or 2.8 GB (3.1 bpt) by using Vertical Partitioning [1] (i.e., using $|P| \cdot \lceil\log n\rceil + \sum_{p=1}^{|P|} SO_p \cdot \lceil\log SO_p\rceil$, where SO_p is the number of edges of property p). Our benchmark considers three different variants of the Ring. We denote as Ring our approach of Section 4, without any optimization. The variant that only considers the optimization of Section 5.1 is denoted Ring_A. Finally, Ring_AB uses both optimizations (Sections 5.1 and 5.2).

We compare them with the following graph database systems, in terms of both space and time:[1]

Jena: A reference implementation of the SPARQL standard.

Virtuoso: A widely used graph database hosting the public DBpedia endpoint, among others [26].

Blazegraph: The graph database system [65] hosting the official Wikidata Query Service [46].

Datalog: An in-memory Datalog-based engine that we cannot identify due to licensing terms.

Systems are configured per vendor recommendations, per previous work [7]. Jena, Virtuoso and Blazegraph implement RPQs per the semantics of property paths in SPARQL 1.1 whereby paths without `*` or `+` are translated into SPARQL graph patterns without RPQs and evaluated under bag semantics. All systems apply set semantics for arbitrary-length paths, per the SPARQL standard. In order to make the results comparable, we add the `DISTINCT` keyword to apply set semantics for all queries.[2] Jena and Blazegraph implement a navigational BFS-style function called ALP (Arbitrary Length Paths) defined by the SPARQL 1.1 standard [38]. Virtuoso uses a transitive closure operator implemented over its relational database engine. The three SPARQL engines were loaded with RDF graphs using their bulk-loaders to minimize transactional overhead; these engines dictionary encode the graph internally, where later we will discuss the overheads associated with this dictionary.

For the Datalog system, we store the graph using the extensional predicate $E(s,p,o)$ for edges, and an intensional (materialized) predicate $V(n)$ to capture nodes through the two rules $V(n) \leftarrow E(n,p,o)$, $V(n) \leftarrow E(s,p,n)$. We use the dictionary-encoded graph (terms are integers). We then translate the RPQs to (positive) Datalog queries. We first applied a base translation of the syntax tree of RPQs into Datalog. Thereafter we tested a number of transformations and optimizations over this base translation, as follows (the transformations are chained, starting with the base translation): (1) *inlining* of non-recursive intermediate predicates, such that rules with that predicate in the head are removed from the query, and rules with that predicate in the body have the corresponding atoms replaced by the bodies

[1] To the best of our knowledge, ArangoDB, Neo4j and OrientDB do not support RPQs declaratively with the standard semantics as we define, though Neo4j does provide some RPQ-like features. While our queries could be run in TigerGraph, its licenses forbid benchmarking.

[2] We could support property paths under SPARQL semantics by evaluating recursive operators under set semantics using the techniques described here and rewriting non-recursive parts to (unions of) basic graph patterns evaluated on the Ring [7].

of rules with the predicate in the head; (2) *linearizing* the query by inlining all but one of the recursive body predicates (where possible), reducing the arity of intermediate predicates; (3) *pushing constants* in the query (resulting from constant node(s) in the RPQ) away from the goal predicate of the query towards the base graph predicate(s); (4) *pruning* to remove duplicate rules, duplicate atoms, and trivially satisfied atoms such as $V(x)$ in a body $V(x), E(x, p, y)$. The direction of the linear recursion in (2) was decided based on what would allow optimization (3) to push the constants through to the base predicate, depending on which node was constant. We performed experiments with and without each optimization, where optimization (3) yielded major performance gains, avoiding the computation of the complete transitive closure, rather computing it from a specific constant. Before querying, we built all four index permutations for constant predicate – namely POS, PSO, SPO and OPS – as well as an index on nodes. Since we evaluate RPQs, where predicates are constant, we omit the two permutations – OSP and SOP – where the predicate is in the last position as these can only be useful when the predicate is unknown.

In order to test on real-world queries, we extract RPQs from the Wikidata Query Logs [46]. However, these logs contain in the order of 579 million queries, of which 50 million use non-trivial RPQs (i.e., not a simple label) [15]. Based on the average runtimes observed later, sequentially evaluating all such RPQs for the engines we compare would take multiple decades on one machine. Thus we rather extracted all non-trivial RPQs from the smaller set of code-`500` (timeout) sections of all seven intervals of the Wikidata Query Logs [46]; these queries reached a one-minute timeout on the public query service, and thus should tend to offer more challenging instances for our experiments. After filtering RPQs mentioning constants not used in the dataset, normalizing variable names, and removing duplicates, this process yielded 1,952 unique queries. From those, we selected the 1,589 that we could confirm produced less than one million results (in some system), for compatibility with Virtuoso, which has a hard-coded limit of 2^{20} results. Queries are run with a 60 second timeout. We classify the RPQs of our log into patterns by mapping nodes to constant/variable types and erasing their predicates; for example, (x, p_1/p_2^*, y) has the pattern `c / * c`, `c / * v`, `v / * c`, or `v / * v`, depending on whether x and y are constant (`c`) or variable (`v`). Table 1 shows the most frequent patterns in our log.

Index construction: The Ring works with a dictionary-encoded version of the graph as described in Section 4, where we complete the graph by adding the reversed edges with inverse labels: If an edge is labeled with predicate p, its reverse edge has predicate $\hat{}p = p + |P|$. This doubles the number of edges and predicates. To construct our index, we build arrays L_s and L_p (and the corresponding C_p and C_o) using a suffix array [7]. We represent L_s and L_p using wavelet matrices [20], a particular implementation of wavelet trees to handle big alphabets efficiently. We use plain bitvectors to implement the wavelet-matrix nodes. Array C_o is represented using a plain bitvector, whereas C_p is represented as a simple array. Our index is constructed in 1.1 hours, using 73.37 GB of RAM. Prior dictionary encoding takes 5.2 additional hours, for a total of 6.3 hours.

Table 1 The 18 most frequent RPQ patterns in our log.

1st–6th	#	7th– 12th	#	13th–18th	#
v/*c	450	v*/*c	30	v|v	11
v*c	421	v|*c	30	v|c	9
v+c	107	v*/*/*/*/*c	28	v*v	8
c*v	98	v/v	20	v/+c	7
c/*v	95	v/?c	20	v/*v	5
v/c	48	v^v	14	v+v	2

Implementing queries: To evaluate queries in the Ring without any optimization, we use our generic query algorithm of Section 4, but handle the query patterns `v^v`, `v/^v`, `v|v`, `v||v`, and `v/v` more efficiently using just backward search and the extended functionality of wavelet trees: For a variable-to-variable query (x, p, y) (analogously, $(x, \hat{}p, y)$), we start by extracting all subjects s from $L_\mathrm{s}[C_\mathrm{p}[p]..C_\mathrm{p}[p+1]-1]$, using the wavelet tree. Then, for each s in that range, we start at range $[C_\mathrm{o}[s]..C_\mathrm{o}[s]-1]$ in L_p and carry out a backward search step using $\hat{}p$. This yields the range of L_s containing all values o such that (s, p, o) is a graph edge, so we report (s, o). Query $(x, p_1|p_2, y)$ (similarly, $(x, p_2|p_3|p_4, y)$) is decomposed into queries (x, p_1, y) and (x, p_2, y), which are computed as explained before. To detect duplicate pairs (s, o), we use a hash table (`std::unordered_set` in C++). For query $(x, p_1/p_2, y)$ (similarly, $(x, p_1/\hat{}p_2, y)$) we first find all nodes z that are the target of an edge labeled p_1, and the source of an edge labeled p_2. This is done by intersecting the ranges $L_\mathrm{s}[C_\mathrm{p}[\hat{}p_1]..C_\mathrm{p}[\hat{}p_1+1]-1]$ and $L_\mathrm{s}[C_\mathrm{p}[p_2]..C_\mathrm{p}[p_2+1]-1]$, using the wavelet tree capabilities [32]. Then, for every such z in the intersection, we carry out a backward search for $p_1 z$, to find all nodes s such that (s, p_1, z) is a graph edge. Similarly, we do a backward search for $\hat{}p_2 z$, to find all nodes o such that (z, p_2, o) is a graph edge. Then, for every such s and o we report (s, o), again avoiding duplicates. Finally, for queries $(x, p_1/(p_2)^*, y)$ we start the search always from the sources of p_1. In general, this filters candidates more efficiently. For all the remaining queries (x, E, y), we choose to start from the end whose predicate has the smallest cardinality.

We implement array B (used to filter on L_p in Section 4.1) with an array of integers, initially zeroed. We do lazy initialization by setting the values of the different predicates of the query and their wavelet matrix ancestors, and zero-

Table 2 Index space (in bpt), indexing time (in hours), and some statistics on the query times (in seconds). Row "Timeouts" counts queries that take over 60 seconds or are rejected by the planner as too costly. RPQs with some constant node are indicated by c, and without by ¬c.

	Ring	Ring_A	Ring_AB	Jena	Virtuoso	Blazegraph	Datalog
Index space	16.41	16.41	27.93	95.83	60.07	90.79	78.32
Indexing time	6.3	7.3	8.3	37.4	3.0	39.4	6.0
Average	1.38	0.70	0.59	5.26	3.87	3.58	11.05
Median	0.09	0.03	0.03	0.20	0.14	0.13	2.71
Timeout	14	8	5	105	55	46	198
Average c	0.62	0.25	0.25	3.83	2.98	3.30	10.60
Median c	0.07	0.03	0.03	0.17	0.11	0.13	2.68
Timeout c	1	0	0	63	37	39	178
Average ¬c	14.39	8.55	6.48	29.59	18.95	8.35	18.84
Median ¬c	4.13	2.10	1.57	4.50	7.98	0.19	6.65
Timeout ¬c	13	8	5	42	18	7	20

ing them again after running the query. Array S, on the other hand, is implemented using a compact lazy initialization structure (App. C, [54]), which uses $O(|V|)$ extra bits on top of S. We use 16-bit cells for D, as queries in our log have fewer than 16 predicates (with a few exceptions that use operator |, which are handled differently as explained).

Optimizations: The index construction and the mechanism to support queries underwent some modifications to consider our optimizations. The optimization of Section 5.1 affects the index construction time as it needs to sort the nodes by navigating the graph. Computing the new order takes an extra hour. In addition, the optimization that splits a given query into two subqueries (Section 5.2) needs to build another wavelet tree to support the colored range counting problem. That construction requires an extra hour and increases the memory usage peak to 104.2 GB RAM.

The optimization that splits the RPQ into two subqueries is only applied when both extremes are variables, because our tests show that in the other queries the method chooses to run from the constant extreme, without splitting. Note that each subquery uses its own array B, which doubles the space usage of B at query time.

Regarding the shortcuts for the simplest query patterns presented before (e.g., v/v), they still remain in all the variants. However, queries of the form $(x, p_1/(p_2)^*, y)$ are not a special case when we use the optimization of Section 5.2. That is, the starting point of the query is not restricted to the source of p_1. The evaluation plan will decide whether partitioning the query by the target of p_1 is more convenient.

Space and query time: Table 2 compares the space usage, the time taken for indexing, and the query times of the systems tested. For query times, we measure the time elapsed between the query request and enumerating the last result.

Ring and Ring_A are the smallest indexes, using 16.4 bpt. This is about twice the space of the packed representation of the data, consistent with the fact that we duplicate all the edges. Array S, needed at query time, uses 3.1 additional bpt, whereas B uses 9×10^{-5} bpt. The total working space usage at query time is 19.5 bpt, $1/3$–$1/5$ of the space used by the other indexes (not considering their extra working space). Instead, the structure of Ring_AB requires 27.9 bpt and duplicates B. Its total working space usage grows to 31 bpt, which is still significantly smaller than the other indexes ($1/2$–$1/3$ of their space). It is worth noting, however, that the indexes of SPARQL engines support join queries (with variable predicates), not just RPQs. Indeed, the Ring also supports such queries when L_o is additionally provided, as discussed in previous work [7].

Virtuoso has the fastest indexing time of around 3 hours. For Datalog and Ring, which take 6.0 and 6.3 hours respectively, we include dictionary encoding, which takes the bulk of time (5.2 hours). The other variants of our approach pay an extra hour during the dictionary encoding because of ordering the nodes. In addition, Ring_AB builds another wavelet tree, thus construction takes 8.3 hours. Jena (37.4 hours) and Blazegraph (39.4 hours) took much longer to index.

Our approach offers the fastest query times on average. Even the non-optimized Ring is 2.6 times faster than Blazegraph, the next best performer. It is also the system with the fewest timeouts: 14. The best performance is obtained with Ring_AB, which is 6.1 times faster than Blazegraph and produces only 5 timeouts. Ring_A uses 60% of the space of Ring_AB and is only 20% slower on average, being twice as fast as the basic Ring and 5.1 times faster than Blazegraph.

On the queries where some node is a constant ("c" in the table, 94.5% of the log), the Ring is on average 4.8 times faster than Virtuoso, the next best competitor for this query type. The optimization of Ring_A speeds up those queries considerably, being 11.9 times faster than Virtuoso. Note that there is no difference between Ring_A and Ring_AB because splitting RPQs does not apply to queries with constants. For the queries where both nodes are variables ("¬c", 5.5% of the log), Blazegraph is 1.7 times faster, on average, than the Ring without any optimization, and it is still slightly faster than Ring_A. By splitting the RPQs, Ring_AB speeds up the queries by a factor of 1.3 in this dataset, thereby outperforming Blazegraph by 30%.

Regarding medians, the Ring is about twice as fast as the next best performer overall. However, both variants with optimizations are three times faster. Blazegraph greatly outperforms other systems in the median for RPQs with two variables. In that case, our best variant is Ring_AB (second best overall), which is 8 times slower.

Datalog is outperformed by all the other query engines, which offer native support and planning for RPQs.

Fig. 12 shows the distribution of query times for the different patterns. Note that Datalog returns few queries (specifically 7) in under a second, and thus does not appear for the scale used in the third row of plots. Also, in the case

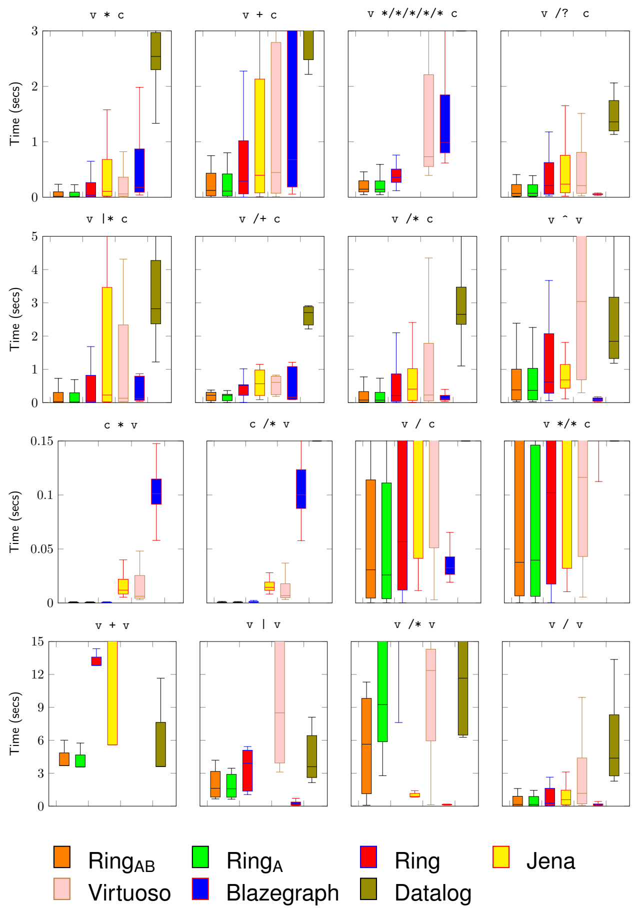


Fig. 12 Boxplots for the distribution of Wikidata query times.

of c * v and c /* v, the results for the Ring variants are so close to the x-axis that they may be difficult to see. The same occurs with Blazegraph in c | v and c /* v.

Our optimized variants, Ring_A and Ring_AB, are consistently faster than the original Ring. The difference between Ring_A and Ring_AB is noticeable only in patterns where both nodes are variables and the RPQs match paths longer than two. For instance, Ring_AB is 1.5 times faster than Ring_A on the pattern v /* v. It follows that Ring_A is always preferable over the basic Ring, as it uses the same space and is around twice as fast. Ring_AB, instead, uses more space, but can be preferred when matching complex RPQs.

We also observe that the Ring variants tend to outperform the other systems in most patterns involving Kleene star ($*$) or Kleene plus ($+$). Indeed, Ring_AB has the best performance in 9 out of 16 patterns. However, other systems sometimes outperform the Ring for RPQs matching paths of fixed length 1 or 2, because those RPQs can be converted into unions of basic graph patterns, that is, joins and unions, which can be handled more efficiently. This strategy can be applied for RPQs consisting of concatenations and disjunctions, but not for those involving Kleene star and plus.

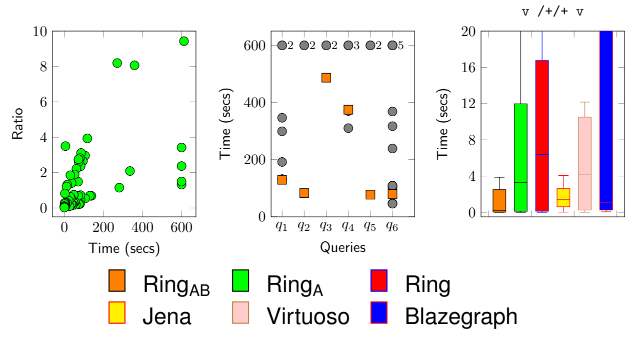


Fig. 13 Experiments on complex Wikidata RPQs, showing the ratio between Ring_A and Ring_AB compared with the time to solve each query with Ring_A (left), query times for solving 6 queries with every possible partition (middle), and boxplots for the distribution of query times in the YAGO2s benchmark for patterns of the form v /+/+ v (right).

Splitting the RPQs. Regarding the splitting of RPQs (Section 5.2), we only see a significant improvement on pattern v /* v, which is the only pattern in Fig. 12 with two variables and a concatenation (where splitting is most valuable) aside from v / v, whose instances are relatively trivial. To better understand the effect of splitting RPQs for more complex queries, we select the unique RPQs from the Wikidata logs whose nodes are variables and whose regular expression E has at least one mandatory position. We discard those of the form v / v since they can be solved as a join without navigating the product graph. We obtain 73 complex RPQs.

We run the 73 complex RPQs with a timeout of 600 seconds and no limit of results on Ring_A and Ring_AB, taking a total of 1.9 and 1.7 hours, respectively. For each query, we compute the ratio of improvement caused by splitting the RPQ as the time of Ring_A divided by the time of Ring_AB. Those results are shown on Figure 13 (left), where every query is represented as a point (x, y), where x is the time required to solve that query in Ring_A and y its ratio of improvement by splitting RPQs. We observe that the ratio tends to increase when the query takes more time. The gains of splitting RPQs are then more prominent on costly queries.

To evaluate how well our estimation of the number of traversed edges works for splitting RPQs, we choose six random queries with no constant and some mandatory positions, and solve them with all possible splits. Figure 13 (middle) shows the times of each query for each partition. The orange square represents the partition selected by Ring_AB and the numbers above show the number of timeouts. Ring_AB selects the optimal split in four of the six queries. In the other two, the splits chosen are close to the optimal ones, avoiding timeouts in every case. Although the assumption that degrees distribute uniformly for a given predicate does not hold in practice, the heuristic works quite well since it is not difficult to identify (and reject) very bad splits that imply vast amounts more work; furthermore, it does not matter if the absolute estimated values are accurate or not, but rather it only matters that the order induced by these estimates is accurate for choosing the best partition.

String dictionaries. The fact that we work directly on data and queries that are mapped from strings to integers can be unfair with the database systems managing strings, as storing and reporting strings may induce additional space and time overhead. Indeed, the size of the Wikidata dictionary is considerable: 13 GB when represented in plain form as a concatenation of strings. Jena, Virtuoso and Blazegraph include this overhead, but the Ring variants and Datalog do not. We can largely reduce the space taken by those strings by storing them using succinct dictionaries [47]. For example, the variant HTFC-rp with sampling step 64 [47] compresses the dictionary to 833 MB, thereby increasing the space of our structure by just 0.9 bpt. Within this space, the structure translates an identifier to its string or vice versa in only 3 microseconds. The impact of this translation in our query times is minimal, adding just 0.006 seconds to the times we report in Table 2 and Figures 12 and 13.

On the other hand, these compact dictionaries assume that the string code is its lexicographical position. In order to use BFS ordering in Ring_A and Ring_AB, we must add a permutation [52] to convert between lexicographical and BFS order. This permutation introduces an extra space of 1.2 GB (1.3 bpt). The impact in the query time is again negligible, under 50 microseconds per query. The construction of the dictionary demands 35 minutes and 68 GB of memory.

With dictionaries, then, the working space of Ring rises to 20.4 bpt, Ring_A to 21.7 bpt, and Ring_AB to 33.2 bpt. Those are still 2–3 times smaller than the next smallest index.

6.2 YAGO2s Benchmark

Yakovets et al. [74] proposed using a real-world dataset, YAGO2s [13], to test optimizations on more complex RPQs. This dataset contains n = 171,684,850 edges, $|V|$ = 42,599,955 nodes, $|S|$ = 7,709,355 subjects, $|P| = 99$ predicates, and $|O|$ = 37,618,098 objects. Its total space is 1.2 GB (7.5 bpt) in packed form, or 505 MB using Vertical Partitioning (3.1 bpt). The benchmark contains 55 realistic RPQs that concatenate two transitive closures of the form `v /+/+ v`, which makes it harder to predict the length of matching paths and to choose the right split.

We index YAGO2s with all variants of our system and the most competitive baselines of Section 6.1. The space of each index is similar to the one obtained with Wikidata. Ring and Ring_A need 13.1 bpt and the second index with least space usage is Ring_AB (23.8 bpt). The other systems use 2–4 times the space of Ring_AB: Virtuoso uses 50.8, Blazegraph 64.6, and Jena 81.3 bpt.

On each system, we run the 55 queries [74, App. B.1] and measure their response time. We set a timeout of 600 seconds and no limit on results, except for Virtuoso, which has a hard-coded limit of 2^{20} results. The right part of Figure 13 shows the query times. The best performance is achieved with Ring_AB, which is around 3.7 times faster than our second best configuration (Ring_A), on average. In terms of medians, Ring_AB is 20.7 times faster than Ring_A. The closest baseline to Ring_AB is Jena, where each query takes, on average, 14 seconds, while Ring_AB needs only 3 seconds. Considering medians, the closest system is instead Blazegraph, with 1.4 seconds, but Ring_AB is 7 times faster: 0.2 seconds. Although Virtuoso limits result sizes, its median is the worst of the baselines. Even without limit, Ring_AB sharply dominates the time performance and uses only 29%–37% of the space of its competitors. Its optimization based on splitting RPQs yields a large speedup compared to Ring_A on complex queries.

String dictionaries. The strings in this dataset take 1.2 GB in plain form, but again they can be compressed to 210 MB, increasing the size of our data structure by only 1.3 bpt. The time performance is the same, and its impact on the query times is again negligible. The permutation to support BFS ordering adds other 172 MB (1.1 bpt). The construction of the dictionary takes 44 minutes and uses 6.6 GB.

7 Extensions

In this section we explore some less standard extensions to RPQs that our data structures could efficiently support.

7.1 Restrictions on nodes

The language of RPQs restricts only the labels of the edges traversed by the paths, but sets no conditions on the nodes. However, some proposed extensions of RPQs (see, e.g., [62, 3, 28]) do allow for restrictions, called “node tests”, to be expressed over intermediate nodes that participate in a path. A feature our representation can efficiently support is to set arbitrary conditions on such nodes. Our data structures can efficiently enforce such restrictions, and even discard whole ranges of undesired nodes.

More concretely, we can set a condition on the source node of any predicate in an RPQ (s, E, y). Let it correspond to the leaf containing position k in the syntax tree of $\hat{}E$. By Glushkov’s construction, this translates into a condition on the node of G we may be at when we reach the NFA state k.

In general, there can be a set S_k of graph nodes that are *forbidden* when the NFA is at each state k. The sets S_k can be just a handful of forbidden nodes, their complement (i.e., we specify a handful of permitted nodes), or in general any set of nodes that can be computed prior to running the RPQ.

In order to forbid the sets S_k, we reuse the array S of Section 4.2, where $S[s]$ marks the active NFA states with which we already reached node s. Since our mechanism avoids considering those nodes with states already in $S[s]$, it suffices that we modify S so that, instead of being initialized at $S[s] = 0$, it is initialized with the kth bit already set for all the states $s \in S_k$, for every k. Those initializations for states s must be extended to initializing the entries $S[v]$ for the wavelet tree nodes v, with the intersection of the entries of their children, just as in Section 4.2.

7.2 Powersets of predicates

The bit-parallel Glushkov simulation also efficiently handles classes of symbols labeling the NFA edges (like (`l1`|`l2`|`l5`), or negated labels), without building unnecessarily large NFAs. This can be useful for evaluating LCRs (see Section 2), or SPARQL's negated property sets [38], or to perform inference over RDF graphs (e.g., handling virtual disjunctions of inferred properties). Regarding the latter point, for example, while the Wikidata query service [46] does not support automated inference, the graph does include axioms such as that `mother` and `father` are sub-properties of `parent`, that `child` is the inverse of `parent`, and so forth, where a corresponding RPQ to capture ancestry on Wikidata may thus require a disjunction like (`parent`|`father`|`mother`|`^child`).

In general, we can permit the kth leaf of the syntax tree of the inverted regular expression $\hat{}E$ to denote a set P_k of symbols instead of a single symbol p_k. In order to handle the search with our bit-parallel simulation of Section 3.5, instead of setting the kth right-to-left bit of $B[p_k]$ to 1 (recall Figs. 2 and 3), we set to 1 the kth right-to-left bit of $B[p]$ for every $p \in P_k$. We also set those bits in all the ancestors v of p in the wavelet tree of B.

This requires a preprocessing time of $O(M \log |P|)$, where $M = \sum_{k=1}^{m} |P_k|$, but does not affect the search time. In terms of Theorem 1, we can now detach the number m of positions in the regular expression (or symbol-labeled leaves in its syntax tree) from the number M. We can then handle sets of predicates in time $O(2^m + M \log |P| + |A'_{\mathcal{M}}| \log |V| + alt(G'_{\mathcal{M}}, |P|)$; all the other complexities and conditions stay the same.

A classic solution that converts each set $P_k = \{p_k^1, \ldots, p_k^i\}$ into a subexpression $(p_k^1| \cdots |p_k^i)$, instead, corresponds to setting $m = \Theta(M)$, which has obvious undesired consequences: note M can approach $m|P|$, for example if the P_k are complements of single predicates p_k.

8 Conclusions

We have shown how the ring [7], a compact representation of labeled graphs, can be used to efficiently evaluate RPQs by combining, in a unique way, the capabilities of (1) the wavelet trees, to process ranges of graph nodes or labels, and (2) the bit-parallel simulation of Glushkov automata, to handle various NFA states simultaneously, in order to solve regular path queries (RPQs) on the graph. We prove that the algorithm we design to traverse the product graph induced by the query is optimal under alternation complexity [11], but our technique is even faster because it is able to process groups of nodes and labels simultaneously. Wavelet trees also enable various heuristic optimizations. As a result, our index uses 3–5 times less space than the alternatives, while outperforming them in many cases. On average, our index is the fastest, outperforming the next best (Blazegraph) by a factor of 5.1; a faster variant using 2–3 times less space than the alternatives is 6.1 times faster than Blazegraph.

Finally, our ability to work on ranges of nodes of the product graph allows for reporting ranges of results, instead of necessarily enumerating them one by one. This enables a compact representation of the set of answers from which any concrete answer can be efficiently extracted. This avenue has been seldom explored in previous work.

In terms of future work, we believe that it would be interesting to implement and evaluate the extensions proposed in Section 7 in order to capture a broader class of path queries. Given the ability of our algorithm to work efficiently with ranges of nodes, we also think that it would be interesting to explore other types of node ordering in order to increase the size of these ranges, for example, based on the types associated with the nodes. The Ring structure is currently read-optimized, where it would be of interest to explore methods for supporting updates. We also plan to look into the evaluation of C2RPQs, i.e., the evaluation of basic graph patterns where predicates can be (2)RPQs.

Competing interests: The authors declare no competing interests.

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. VLDB*, pages 411–422, 2007.
2. Z. Abul-Basher. Multiple-query optimization of regular path queries. In *Proc. ICDE*, pages 1426–1430, 2017.
3. F. Alkhateeb and J. Euzenat. Constrained regular expressions for answering RDF-path queries modulo RDFS. *Int. J. Web Inf. Syst.*, 10(1):24–50, 2014.
4. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

5. R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutiérrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, and H. Voigt. G-CORE: a core for future graph query languages. In *Proc. SIGMOD*, pages 1421–1432, 2018.
6. M. Arenas, S. Conca, and J. Pérez. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *Proc. WWW*, pages 629–638, 2012.
7. D. Arroyuelo, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto. Worst-case optimal graph joins in almost no space. In *Proc. SIGMOD*, pages 102–114, 2021.
8. D. Arroyuelo, A. Hogan, G. Navarro, and J. Rojas-Ledesma. Time- and space-efficient regular path queries. In *Proc. ICDE*, pages 3091–3105, 2022.
9. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comp.*, 42(4):1737–1767, 2013.
10. J. A. Baier, D. Daroch, J. L. Reutter, and D. Vrgoc. Evaluating navigational RDF queries over the Web. In *Proc. ACM HT*, pages 165–174, 2017.
11. J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Alg.*, 4(1):1–18, 2008.
12. G. Berry and R. Sethi. From regular expression to deterministic automata. *Theor. Comp. Sci.*, 48(1):117–126, 1986.
13. J. Biega, E. Kuzey, and F. M. Suchanek. Inside YAGO2s: A transparent information extraction architecture. In *Proc. WWW*, pages 325–328, 2013.
14. F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen. Distance oracles in edge-labeled graphs. In *Proc. EDBT*, pages 547–558, 2014.
15. A. Bonifati, W. Martens, and T. Timm. Navigating the maze of Wikidata query logs. In *Proc. WWW*, pages 127–138, 2019.
16. A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.
17. A. Brüggemann-Klein. Regular expressions into finite automata. *Theor. Comp. Sci.*, 120(2):197–213, 1993.
18. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
19. D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.
20. F. Claude, G. Navarro, and A. Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Inf. Syst.*, 47:15–32, 2015.
21. D. Colazzo, V. Mecca, M. Nolé, and C. Sartiani. PathGraph: Querying and exploring big data graphs. In *Proc. SSDBM*, pages 29:1–29:4, 2018.
22. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proc. SIGMOD*, pages 323–330, 1987.
23. A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Aggregation support for modern graph analytics in TigerGraph. In *Proc. SIGMOD*, pages 377–392, 2020.
24. A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke. Graph pattern matching in GQL and SQL/PGQ. In *Proc. SIGMOD*, pages 2246–2258, 2022.
25. S. C. Dey, V. Cuevas-Vicentín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. On implementing provenance-aware regular path queries with relational query engines. In *Proc. EDBT/ICDT*, pages 214–223, 2013.
26. O. Erling and I. Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*, pages 7–24. Springer, 2009.
27. P. Ferragina and G. Manzini. Indexing compressed texts. *J. ACM*, 52(4):552–581, 2005.
28. V. Fionda, G. Pirrò, and M. P. Consens. Querying knowledge graphs with extended property paths. *Semantic Web*, 10(6):1127–1168, 2019.
29. G. H. L. Fletcher, J. Peters, and A. Poulovassilis. Efficient regular path query evaluation using path indexes. In *Proc. EDBT*, pages 636–639, 2016.
30. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proc. SIGMOD*, pages 1433–1445, 2018.
31. T. Gagie, G. Navarro, and S. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci.*, 426-427:25–41, 2012.
32. T. Gagie, G. Navarro, and S. J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci.*, 426-427:25–41, 2012.
33. T. Gagie, J. Kärkkäinen, G. Navarro, and S. J. Puglisi. Colored range queries and document retrieval. *Theor. Comp. Sci.*, 483:36–50, 2013.
34. V.-M. Glushkov. The abstract theory of automata. *Russian Math. Surv.*, 16:1–53, 1961.
35. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.
36. A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: Fast property paths in RDF-3X. In *Proc. GRADES*, page 14, 2013.
37. X. Guo, H. Gao, and Z. Zou. Distributed processing of regular path queries in RDF graphs. *Knowl. Inf. Syst.*, 63(4):993–1027, 2021.
38. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, Mar. 2013. `http://www.w3.org/TR/sparql11-query/`.
39. O. Hartig and G. Pirrò. SPARQL with property paths on the Web. *Semantic Web*, 8(6):773–795, 2017.
40. L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda. On the optimization of recursive relational queries: Application to graph queries. In *Proc. SIGMOD*, pages 681–697, 2020.
41. R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *Proc. SIGMOD*, pages 123–134, 2010.
42. A. Koschmieder and U. Leser. Regular path queries on large graphs. In *Proc. SSDBM*, pages 177–194, 2012.
43. E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoc. SPARQL with property paths. In *Proc. ISWC*, pages 3–18, 2015.
44. J. Kuijpers, G. Fletcher, T. Lindaaker, and N. Yakovets. Path indexing in the Cypher query pipeline. In *Proc. EDBT*, pages 582–587, 2021.
45. B. Liu, X. Wang, P. Liu, S. Li, and X. Wang. PAIRPQ: An efficient path index for regular path queries on knowledge graphs. In *Proc. APWeb-WAIM*, pages 106–120, 2021.
46. S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In *Proc. ISWC*, pages 376–394, 2018.
47. M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical compressed string dictionaries. *Inf. Syst.*, 56:73–108, 2016.
48. Q. Mehmood, M. Saleem, R. Sahay, A. N. Ngomo, and M. d'Aquin. QPPDs: Querying property paths over distributed RDF datasets. *IEEE Access*, 7:101031–101045, 2019.
49. A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comp*, 24(6):1235–1258, 1995.
50. K. Miura, T. Amagasa, and H. Kitagawa. Accelerating regular path queries using FPGA. In R. Bordawekar and T. Lahiri, editors, *Proc. ADMS@VLDB*, pages 47–54, 2019.
51. J. I. Munro. Tables. In *Proc. FSTTCS*, pages 37–42, 1996.
52. J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theor. Comp. Sci.*, 438:74–88, 2012.

53. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.
54. G. Navarro. Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. *ACM Comput. Surv.*, 46(4): 52:1–52:47, 2013.
55. G. Navarro. Wavelet trees for all. *J. Discr. Alg.*, 25:2–20, 2014.
56. G. Navarro and M. Raffinot. New techniques for regular expression searching. *Algorithmica*, 41(2):89–116, 2005.
57. V. Nguyen and K. Kim. Efficient regular path query evaluation by splitting with unit-subquery cost matrix. *IEICE Trans. Inf. Syst.*, 100-D(10):2648–2652, 2017.
58. M. Nolé and C. Sartiani. Regular path queries on massive graphs. In *Proc. SSDBM*, pages 13:1–13:12, 2016.
59. A. Pacaci, A. Bonifati, and M. T. Özsu. Regular path query evaluation on streaming graphs. In *Proc. SIGMOD*, pages 1415–1430, 2020.
60. Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang. Answering billion-scale label-constrained reachability queries within microsecond. *PVLDB*, 13(6):812–825, 2020.
61. Y. Peng, X. Lin, Y. Zhang, W. Zhang, and L. Qin. Answering reachability and k-reach queries on large graphs with label constraints. *VLDB J.*, 31(1):101–127, 2022.
62. J. Pérez, M. Arenas, and C. Gutiérrez. nSPARQL: A navigational language for RDF. *J. Web Semant.*, 8(4):255–270, 2010.
63. S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *Proc. ICDE*, pages 1009–1020, 2013.
64. F. Tetzel, W. Lehner, and R. Kasperovics. Efficient compilation of regular path queries. *Datenbank-Spektrum*, 20(3):243–259, 2020.
65. B. B. Thompson, M. Personick, and M. Cutcher. The Bigdata®RDF Graph Database. In *Linked Data Management*, pages 193–237. Chapman and Hall/CRC, 2014.
66. L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida. Landmark indexing for evaluation of label-constrained reachability queries. In *Proc. SIGMOD*, pages 345–358, 2017.
67. O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *Proc. GRADES*, page 7, 2016.
68. T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. ICDT*, pages 96–106, 2014.
69. D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Comm. ACM*, 57(10):78–85, 2014.
70. S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, and S. Bedathur. Efficiently answering regular simple path queries on large labeled networks. In *Proc. SIGMOD*, pages 1463–1480, 2019.
71. X. Wang, G. Rao, L. Jiang, X. Lyu, Y. Yang, and Z. Feng. TraPath: Fast regular path query evaluation on large-scale RDF graphs. In *Proc. WAIM*, pages 372–383, 2014.
72. X. Wang, J. Wang, and X. Zhang. Efficient distributed regular path queries on RDF graphs using partial evaluation. In *Proc. CIKM*, pages 1933–1936, 2016.
73. N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In *Proc. AMW*, 2013.
74. N. Yakovets, P. Godfrey, and J. Gryz. Query planning for evaluating SPARQL property paths. In *Proc. SIGMOD*, pages 1875–1889, 2016.
75. L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. Efficient processing of label-constraint reachability queries in large graphs. *Inf. Syst.*, 40:47–66, 2014.

A Complete running example

Fig. 14 illustrates the whole process of matching the (2)RPQ (y,`15`$^+$/`^bus`,`Baq`), with the NFA of Fig. 8, in the graph of Fig. 1, as represented in Fig. 5. We use a top-down tree to show branches in the process; Fig. 9 shows the states of the product graph $G'_{\mathcal{M}}$ we traverse (backwards in BFS). The top nodes of the tree illustrate what we have already done in previous examples (edge $1 \rightarrow 2$ of $G'_{\mathcal{M}}$): starting from $L_{\mathrm{p}}[14..15]$ (`Baq`) and $D = 0001$, we identified the only edge label reaching that node, `15`, that is relevant in our NFA. The label `11` also appears in $L_{\mathrm{p}}[15..16]$ because a transition labeled `11` reaches `Baq`, but D & $B[$`11`$] = 0000$ because our NFA does not match it; this pruned branch is shown with a dashed arrow leading to an `X`. We have also seen that the only subject of those edges labeled `15` is `BA`, at $L_{\mathrm{s}}[10..10]$, where the NFA is active at states 0110. We move to `BA` because $S[$`BA`$] = S[4] = 0000$, so $D = 0110$ contains some unseen NFA states at this node. We mark $S[4] = 0110$ to indicate that we have already reached `BA` with those active NFA states. In part 3 of our process, we find the interval of L_{p} corresponding to $L_{\mathrm{s}}[10] = 4$ (`BA`), $L_{\mathrm{p}}[C_{\mathrm{o}}[4] + 1..C_{\mathrm{o}}[5]] = L_{\mathrm{p}}[11..14]$, completing one step.

Three symbols appear on $L_{\mathrm{p}}[11..14]$ (i.e., three edge labels reach `BA`), but only `15` (left child) and `^bus` (right child) match our NFA. By `15` we reach $L_{\mathrm{s}}[8..9]$ using backward search. In this interval we find two sources that, by `15`, reach `BA`: `SA` (left child) and `Baq` (right child), both with NFA state $D = 0110$ (the same as before). Conversely, by `^bus`, we reach $L_{\mathrm{s}}[16..16]$ using backward search. There we find the only source, `SA`, that reaches `BA`, with NFA state $D = 1000$. We process the three sources in BFS order, left to right:

1. By `15` we reach `BA` from `SA` (leftmost tree node in this level). We accept going to `SA` because $S[$`SA`$] = S[1] = 0000$ and $D = 0110$ has new states, so we set $S[1] = 0110$. In part 3 we obtain the interval $L_{\mathrm{p}}[1..4]$ for `SA`. This is transition $2 \rightarrow 3$ in the product graph.
2. By `15` we reach `BA` from `Baq` (middle tree node in this level). Although we had already seen `Baq`, it was only with states $S[$`Baq`$] = S[5] = 0001$, so the current state $D = 0110$ has some unvisited NFA states; we set $S[5] = 0111$ and part 3 leads us to $L_{\mathrm{p}}[15..16]$. This is transition $2 \rightarrow 4$ in the product graph.
3. By `^bus` we reach `BA` from `SA` as well (rightmost tree node in this level). Since $S[$`SA`$] = S[1] = 0110$ and $D = 1000$, we have new states and we accept going to `SA`, setting $S[1] = 1110$. The NFA state is still 1000 (transition $2 \rightarrow 5$ in the product graph), which contains the initial state, so we report node `SA` as a solution to our 2RPQ. We then continue from it, reaching $L_{\mathrm{p}}[1..4]$ using part 3.

Our BFS traversal now branches from each of the tree nodes identified above:

1. From $L_{\mathrm{p}}[1..4]$ (`SA`) with $D = 0110$, we find edges labeled `11`, `15`, and `^bus` leading to it:
   (a) Our NFA cannot process `12` (D & $B[$`12`$] = 0000$), so we abandon that edge.

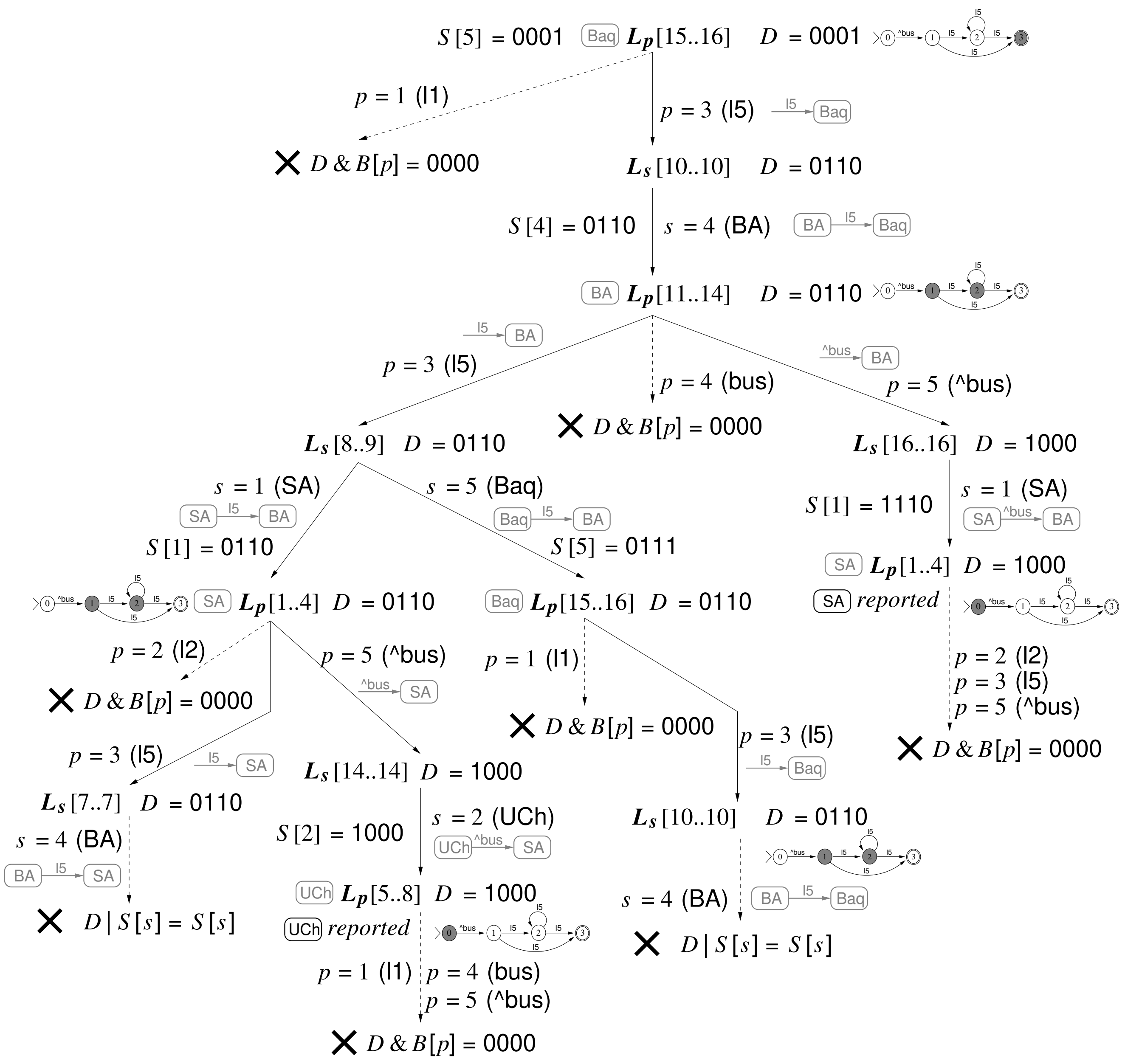


Fig. 14 The whole process to match the RPQ of Fig. 8 in our graph of Figs. 1 and 5.

(b) By l5 we find the source BA, but since $S[\mathsf{BA}] = S[4] = 0110$ and $D = 0110$, we have already visited BA with those active states, so we abandon this branch too, thus avoiding to fall into a cycle.

(c) By ˆbus we reach $L_s[14\,..\,14]$ with state $D = 1000$. The only source here is $L_s[14] = 2 =$ UCh. Since $S[\mathsf{UCh}] = S[2] = 0000$, we enter this state and set $S[2] = 1000$ (transition $3 \to 6$ in the product graph). Furthermore, since D contains the initial state, we report UCh as the second solution to the 2RPQ.

2. From $L_p[15\,..\,16]$ (Baq) with $D = 0110$, we find edges labeled l1 and l5 leading to it:

(a) Our NFA cannot process l1, so we abandon this branch.

(b) By l5 we reach BA again, and once again we prune the branch to avoid falling into cycles, because $S[\mathsf{BA}] = S[4] = 0110$.

3. Finally, from $L_p[1\,..\,4]$ (SA) and $D = 1000$, which we had reported, the NFA has nowhere to go, so we reject the three possible edge labels, l2, l5, and ˆbus. The same happens in the last tree level from $L_p[5\,..\,8]$ (UCh) after reporting it, so we finish.