Time- and Space-Efficient Regular Path Queries

Diego Arroyuelo
Univ. Técnica Federico Santa María
IMFD
Santiago, Chile
darroyue@inf.utfsm.cl

Aidan Hogan
DCC, University of Chile
IMFD
Santiago, Chile
ahogan@dcc.uchile.cl

Gonzalo Navarro
DCC, University of Chile
IMFD
Santiago, Chile
gnavarro@dcc.uchile.cl

Javiel Rojas-Ledesma
DCC, University of Chile
IMFD
Santiago, Chile
jrojas@dcc.uchile.cl

***Abstract*—We introduce a time- and space-efficient technique to solve regular path queries over labeled (RDF) graphs. We combine a bit-parallel simulation of the Glushkov automaton of the regular expression with the ring index introduced by Arroyuelo et al., exploiting its wavelet tree representation in order to efficiently reach relevant states of the product graph. Our algorithm is able to simultaneously process several automaton states, as well as several graph nodes/labels. Our experiments show that our approach uses 3–5 times less space than existing state-of-the-art systems, while generally outperforming them in query times (nearly 3 times faster than the next best, on average).**

***Index Terms*—Regular path queries, Glushkov automaton, ring index, succinct data structures**

I. Introduction

A key feature of graph databases is the ability to query paths of arbitrary length [1], often supported as *regular path queries* (*RPQs*) [2], which specify a regular expression that constrains matching paths. Consider the graph of Fig. 1 describing transport within a city. Edges are directed and labeled with the type of transportation (l1, l2 and l5 denote three metro lines). An RPQ $x \xrightarrow{(l1|l2|l5)^+} y$ finds pairs of locations reachable by metro, where x and y are node variables, while the regular expression $(l1|l2|l5)^+$ will match paths of length one-or-more such that each edge has the label l1, l2 or l5. We may also fix nodes in an RPQ, for example Baquedano $\xrightarrow{(l1|l2|l5)^+} y$ finds nodes reachable from Baquedano by metro.

While regular path queries have long been studied in theoretical works [2, 3], more recently they have been included in practical query languages for graphs. SPARQL 1.1 introduced *property paths* for RDF graphs, which extend RPQs with inverse labels and negated edge labels [4, 5]. Other graph query languages would later add RPQ support [6, 1, 7]. RPQs are frequently used in practice: of 208 million SPARQL queries issued to the Wikidata Query Service [8], 24% use at least one RPQ/property path feature [9].

The problem of efficiently evaluating RPQs has been gaining increasing attention in recent years [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35]. The traditional algorithm – used, for example, in the theoretical literature to prove complexity bounds – is based on representing the regular expression of the RPQ as an automaton, defining the product graph of the data graph and the automaton, and then applying graph search (BFS, DFS, etc.) on the product graph [3]. While the product graph is potentially large, algorithms can expand it lazily during navigation. Other more recent approaches propose the use of recursive queries [15, 20, 30], parallel [28] and distributed [18, 19, 25, 27, 33] frameworks, indexing techniques [14, 17, 34, 35], multi-query optimization [21], approximation [29], just-in-time compilation [32], etc., to optimize RPQs. These works have mainly focused on improving efficiency in terms of time, but not space.

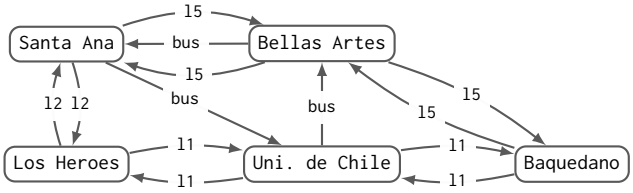

Fig. 1. Santiago metro stations with metro lines and buses

Our Contribution: We introduce a novel technique to evaluate RPQs (with inverses, aka. *2RPQs*) over a compressed representation of the graph called a ring [36]. The ring was introduced for handling join queries in worst-case optimal time while using almost the same space as a plain representation of all the triples (s, p, o) denoting directed, labeled edges $s \xrightarrow{p} o$. The ring converts the graph into a sequence based on the Burrows–Wheeler Transform (BWT) [37], and encodes that sequence using a wavelet tree data structure [38]. Our technique combines (1) the backward search capabilities of the BWT, (2) the ability of the wavelet trees to efficiently work on ranges of nodes or edge labels, and (3) the regularity of the Glushkov automaton [39] of the regular expression and the versatility of its bit-parallel simulation [40]. As a result, we are able to search over several paths in the product graph simultaneously. Theorem 1 shows that we spend logarithmic time per node and edge of the product subgraph induced by the query. Our approach uses about twice the space of a compact data representation (since we duplicate edges to handle reversed edges for 2RPQs). Experiments show that we use 3–5 times less space than competing graph databases that handle RPQs, while also offering better times overall (on average 2.8 times faster than the nearest system: Blazegraph).

This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt grant 1-200038.

II. Related Work

We present related work on efficiently evaluating RPQs and related expression types, such as property paths in SPARQL.

Evaluating path queries: Earlier works on evaluating path queries focused on shortest paths [10, 11]. Works later began to focus on RPQs; these can be loosely divided into navigational and relational approaches.

Regarding navigational approaches, Koschmieder and Leser [12] 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 [18] 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. [19] 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 [24] 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. [29] compute approximate RPQ results using bidirectional random walks from the source and target node.

Other relational approaches evaluate RPQs using recursive joins (or queries). Dey et al. [13] evaluate RPQs using either Datalog rules or recursive SQL queries; they further return *provenance* in the form of all edges involved in some or all matching paths. Yakovets et al. [15] translate property paths into recursive SQL queries, but note that the resulting queries can be complex. Jachiet et al. [30] propose an extended relational algebra with a transitivity/fixpoint operator, and describe how RPQs (more specifically, unions of conjunctive RPQs) can be translated to this algebra. Fionda et al. [26] propose *extended property paths*, which include difference and intersection over paths, as well constraints on nodes along the path; some expressions require a recursion over SPARQL.

Combining both navigational/automata and recursive/relational approaches, Yakovets et al. [20] propose hybrid "wave-plans" that can mix operators from both algebras and thus can express novel query plans. Abul-Basher [21] propose a related framework called "swarmguide" for optimizing multiple RPQs at once, based on reusing a maximum common sub-automaton.

Recent approaches leverage software or hardware acceleration techniques. Miura et al. [28] evaluate RPQs on top of field programmable gate arrays (FPGAs) to enable parallelism. Tetzel et al. [32] use just-in-time compilation to generate native C++ code that directly evaluates the RPQ on the graph.

Indexing: Custom indexes have also been proposed for RPQs. Gubichev et al. [14] extend RDF-3X with support for property paths using an indexing technique called FERRARI [41], which encodes the transitive closure of edges with a given label using intervals of node ids. Wang et al. [16] propose a predicate-based indexing scheme to evaluate RPQs over RDF graphs. Fletcher et al. [17] propose a *k-path index* that indexes all paths of length up to k in a B$^+$-tree. Kuijpers et al. [34] use k-path indexes to optimize RPQs over Neo4j. Liu et al. [35] populate k-path indexes with frequent paths.

RPQ fragments: Some works have focused on fragments of RPQs. One such fragment is that of *label-constrained reachability queries* (*LCRs*) [42], which match paths of arbitrary length such that each edge label on the path is in a given set $\{p_1, \ldots, p_n\}$. LCRs correspond to a fragment of RPQs of the form $(p_1|\ldots|p_n)^*$ [43] that has been shown to be common in practice [9, 44]. Works on efficiently evaluating LCRs have primarily explored specialized indexes over the labeled transitive closure of the graph, i.e., over precomputed tuples of the form (u, v, L) such that node v is reachable from node u in the graph via a path whose edges only use labels from the set L (and, typically, where there does not exist (u, v, L') such that $L' \subset L$) [42, 45, 43, 46, 47]. Given that the number of such tuples can be prohibitively large, approaches focus on ways to reduce index sizes while keeping query runtimes low.

Jin et al. [42] propose an LCR index that combines a spanning tree and an index of a partial transitive closure from which the full closure can be computed. Zou et al. [45] propose to decompose and build separate LCR indexes for each (strongly connected) component in order to improve scalability for graphs without large components. Valstar et al. [43] construct a partial LCR index that is complete for k "landmark" vertices with highest degree, applying a BFS variant that refers to the index when a landmark is encountered. Peng et al. [46] 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 via an intermediate node x such that (u, x, L_0) and (x, v, L_1) are indexed, and $L_0 \cup L_1 \subseteq L$. Other works have explored distance-aware indexes for variants of LCRs, including label-constrained shortest path queries (LCSPs) [48]; and label constrained k-reachability queries (LCKRs) [47].

These works on LCRs – and their variants – differ from ours in three main aspects: (1) they support a fragment of regular expressions; (2) they assume source and target nodes to be constant; (3) they use specialized index structures that occupy space additional to representing the graph.

Other settings: We focus on evaluating RPQs over a static graph on a single machine. However, other works have looked into evaluating RPQs over RDF graphs partitioned over multiple machines [18, 19, 25, 27, 33] or websites [22, 23]. Pacaci et al. [31] have recently explored the evaluation of RPQs over sliding windows of streaming graph data.

Novelty: We introduce a novel technique to evaluate (2)RPQs that is efficient both in time and space. While some indexing schemes explore a time–space trade-off, they occupy space *additional* to representing and indexing the graph [41, 43]. To the best of our knowledge, our approach is the first that can efficiently evaluate RPQs on a compressed representation of the graph, and the first to show key advantages of using Glushkov automata [39] in this setting: Not only does it enable a more space-efficient bit-parallel simulation of the automaton [40], its transitions exhibit a regularity that is crucial to efficiently evaluating RPQs. The combination of the backward search capabilities of the BWT [37], the ability of the wavelet trees [38] to work on ranges of nodes/labels, and the regularity of Glushvov's automaton, allow us to simulate

traversal of *only* the product subgraph induced by the RPQ. 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.

III. Key Concepts

A. Regular Path Queries

Let Σ denote a set of symbols. 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 the *nodes* of G as $V = \{x \mid \exists\, y, z,\ (x, y, z) \in G \vee (z, y, x) \in G\}$.

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. We call $\mathsf{word}(\rho) = p_1 \ldots p_n \in \Sigma^*$ the *word* of ρ.

We say that ε is a *regular expression*, and that any element of Σ is a regular expression. 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 $E^?$ as $\varepsilon|E$.

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

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

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 $\mathsf{dom}(\mu)$, which is 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:

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

If E is a two-way regular expression over Σ, $s \in V \cup \Phi$ and $o \in V \cup \Phi$, we call (s, E, o) a *two-way regular path query* (2RPQ). We define the evaluation of the 2RPQ (s, E, o) on G as the evaluation of the RPQ (s, E', o) on $G^{\leftrightarrow}$, where E' is the rewritten form of E using only atomic inverses.

Example. Take the graph G of Fig. 1 and the RPQ $(x, (\mathtt{11}|\mathtt{12}|\mathtt{15})^+, y)$, where $x, y \in \Phi$ are variables. Infinitely many paths in G match the expression $(\mathtt{11}|\mathtt{12}|\mathtt{15})^+$, including:

```
      UCh  l1  LH  l1  UCh
UCh  l1  LH  l1  UCh  l1  LH
```

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 $\mu(x) = \mathsf{UCh}$, $\mu(y) = \mathsf{UCh}$. □

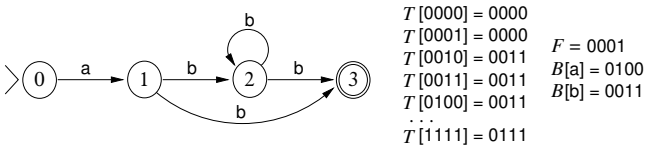

Fig. 2. The Gluskov automaton of the regular expression $a/b^*/b$, and its bit-parallel representation on the right.

B. Product Graph

A common approach for evaluating an RPQ (s, E, o) on G involves computing the *product graph* of G [3]. First we use Thompsons' construction to convert the regular expression E into a non-deterministic finite automaton (NFA) $M_E = (Q, \Sigma_E, \Delta, q_0, F)$, where Q denotes the set of states, $\Sigma_E \subseteq \Sigma$ the set of symbols used in E, Δ the transitions, q_0 the initial state, and F the set of accepting states. Letting V denote the nodes of G, the *product graph* $G_E \subseteq (V \times Q) \times (V \times Q)$ of G with respect to E is a directed graph defined as follows:

$$G_E = \{((x, q_x), (y, q_y)) \mid \text{there exists } p \in \Sigma_E \text{ such that } (x, p, y) \in G \text{ and } (q_x, p, q_y) \in \Delta\}.$$

The RPQ can then be evaluated using graph search (e.g., BFS, DFS, etc.) to find paths in the product graph G_E that start from some node $(x, q_0) \in V \times \{q_0\}$ and end in some node $(y, q_f) \in V \times F$ (where $x = s$ if $s \in \Sigma$, and $y = o$ if $o \in \Sigma$).

C. Bit-parallel Glushkov Automata

Consider a regular expression E on alphabet Σ with m occurrences of symbols in Σ. Compared to Thompson's construction of an NFA from E, Glushkov's [39, 49] has the disadvantage of generating $\Theta(m^2)$ edges in the worst case, and needing $O(m^2)$ construction time [50]. In exchange, it has various useful properties for our purposes:

1) The NFA has no ε-transitions.
2) The NFA has exactly $m + 1$ states, worst-case optimal.
3) All the transitions arriving at a state have the same label.

These properties imply the following important fact.

Fact 1. *In a Glushov 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. Fig. 2 (top) shows the Gluskov automaton for $a/b^*/b$. Take the states $X = \{0, 2\}$. The states reachable from X in one step via b are $\{2, 3\}$, i.e., 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. □

This property enables the *bit-parallel* simulation of the NFA [40]. This simulation represents NFA states as bits in a computer word, 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 all the states in parallel by using the classic arithmetical and logical operations on computer words. Assume for simplicity that the bits of the NFA states fit in a single computer word; we discuss the general case later. Further assume that the alphabet is an integer range $\Sigma = [1\,..\,\sigma]$. The simulation maintains the following variables:

- A computer word D holding $m+1$ bits tells, at every step, the active NFA states, as discussed. Assume the initial state corresponds to the highest bit.
- A table $B[1\,..\,\sigma]$ of computer words 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 computer word F marks with 1s the final NFA states.

The simulation is then carried out as follows:

1) We set $D \leftarrow 2^m$ to activate the initial state.
2) If $D \;\&\; F \neq 0$, then we have reached a final state and accept the word read (recall that '&' is the bitwise-and).
3) If $D = 0$, then we have run out of active states and reject.
4) For each input symbol c, we use Fact 1 to update D as follows, so the new active states are those that are reached from the current ones and also reached by symbol c:
$$D \;\leftarrow\; T[D] \;\&\; B[c], \tag{1}$$
5) Return to point 2.

Example. The Glushkov automaton of $a/b^*/b$ and its bit-parallel representation is shown in Fig. 2. 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. □

The space of the simulation is $O(2^m + \sigma)$, instead of the worst-case $O(2^m\sigma)$ of a classic DFA implementation. The tables are built in time $O(2^m)$ with lazy initialization for B.

A similar simulation can be used to read the text in reverse order [40] by building a table $T'[0\,..\,2^m - 1]$ where $T'[X]$ marks with 1s the states that can reach some state in X in one step, initializing $D \leftarrow F$ and, for each symbol c, updating

$$D \;\leftarrow\; T'[D \;\&\; B[c]], \tag{2}$$

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 $m + 1 > w$, 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 d-bit subtables $T_1, \ldots, T_{\lceil (m+1)/d \rceil}$, so that if we partition $X = X_1 \cdots X_{\lceil (m+1)/d \rceil}$, then $T[X] = T_1[X_1] \mid \cdots \mid T_{\lceil (m+1)/d \rceil}[X_{\lceil (m+1)/d \rceil}]$, where "|" denotes the bitwise-or. This reduces the space to $O((m/d)2^d + \sigma)$ and multiplies time by $O(m/d)$ instead of $O(m/w)$, for any desired $1 \leq d \leq \min(w, m+1)$ [40]. 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.

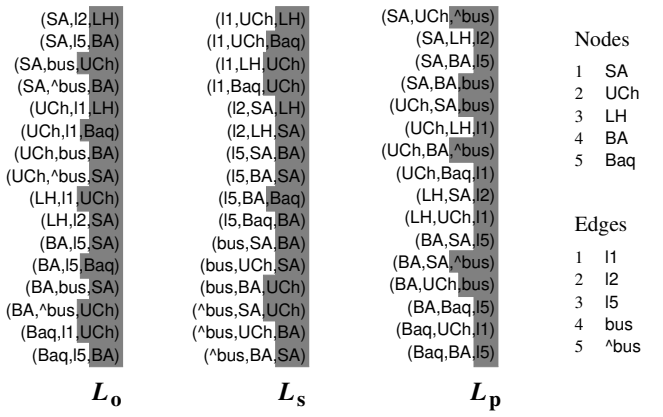

Fig. 3. The triples representing the completion of the graph of Fig. 1, adding a reverse edge labeled ˆbus for each edge labeled bus (l1, l2 and l5 are considered bidirectional). The triples are presented in three rotations; the last column in each rotation defines a component of the ring.

D. The Ring

The *ring* [36] is a novel representation for a set of triples (s, p, o), supporting worst-case optimal joins [51]. It regards triples with different rotations, (s, p, o), (p, o, s), or (o, s, p). What the ring actually stores are those objects, subjects, and predicates separated in three sequences, $L_{\rm o}$, $L_{\rm s}$, and $L_{\rm p}$, respectively, as follows (where n is the number of triples):

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

The concatenation $L_{\rm o} \cdot L_{\rm s} \cdot L_{\rm p}$ is akin to the Burrows–Wheeler Transform (BWT) [37] of the concatenation of all triples [36].

Example. Fig. 3 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 integers, and use this integer order for sorting. The leftmost rotation lists triples sorted by (s, p, o); the last column (o) of this rotation forms the sequence $L_{\rm o}$. The middle rotation sorts triples by (p, o, s), and its last column (s) forms the sequence $L_{\rm s}$. Finally, the

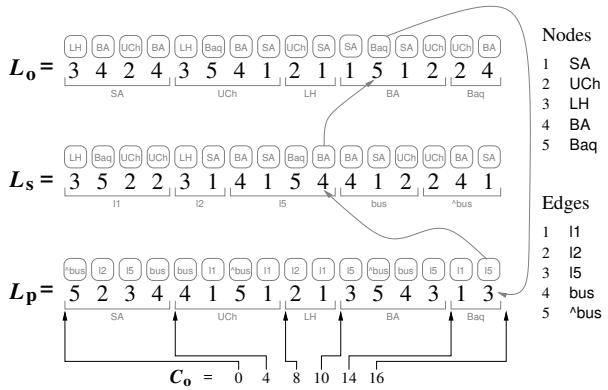

Fig. 4. The ring structure for the completion of the graph of Fig. 1, adding a reverse edge labeled ˆbus for each edge labeled bus (l1, l2 and l5 are bidirectional). We also show how the last triple in L_p is tracked.

rightmost rotation sorts the triples by (o, s, p) and its last column (p) forms L_p. The ring is then the concatenation 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). In particular, 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). Note that, in the three strings, the range $[1\,..\,n]$ represents all the triples and a range of size 1 represents an individual triple.

Example. In Fig. 3, 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 so-called *LF-steps*, defined on array L_p (and analogously on L_s and L_o), as follows:

$$\mathsf{LF}_\mathrm{p}(i) \;=\; C_\mathrm{p}[c] + \mathsf{rank}_c(L_\mathrm{p}, i), \tag{3}$$

where $c = L_\mathrm{p}[i]$, $C_\mathrm{p}[c]$ counts the occurrences of symbols smaller than c in L_p, and $\mathsf{rank}_c(L_\mathrm{p}, i)$ counts the occurrences of c in $L_\mathrm{p}[1\,..\,i]$. The subject of the triple for $L_\mathrm{p}[i]$ is $L_\mathrm{s}[i']$ for $i' = \mathsf{LF}_\mathrm{p}(i)$, and the object is $L_\mathrm{o}[i'']$ for $i'' = \mathsf{LF}_\mathrm{s}(i')$. It further holds that $i = \mathsf{LF}_\mathrm{o}(i'')$, where the predicate is at $L_\mathrm{p}[i]$.

Example. Fig. 4 shows the ring of Fig. 3, now directly as sequences L_o, L_s, and L_p of integers. We still 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 the triple referenced from $L_\mathrm{p}[16]$. The value $L_\mathrm{p}[16] = 3$ (l5) gives the predicate. It refers to the object 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 the corresponding subject, we note that this is the fourth 3 (l5) in L_p. Then, if we go to the fourth position in the area of l5 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 full triple is $\mathsf{BA} \xrightarrow{\mathsf{l5}} \mathsf{Baq}$. 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, $\mathsf{LF}_\mathrm{o}(12) = 16$. □

The key to solving multijoins with the ring is the so-called *backward search*, which computes in batch all the LF-steps in a range. Consider a range $L_\mathrm{p}[b_o\,..\,e_o]$ listing, say, all the triples with a specific object o (i.e., all the triples (o, s, p) for any s and p). The backward search by some specific predicate p gives the range $L_\mathrm{s}[b_p\,..\,e_p]$ corresponding to all the triples with object o and predicate p (i.e., all the triples (p, o, s) for any s). This is computed with the following formula, which extends the LF-steps (Eq. (3)) to ranges [52, 36]:

$$b_p \;=\; C_\mathrm{p}[p] + \mathsf{rank}_p(L_\mathrm{p}, b_o - 1) + 1, \tag{4}$$

$$e_p \;=\; C_\mathrm{p}[p] + \mathsf{rank}_p(L_\mathrm{p}, e_o). \tag{5}$$

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. Continuing our example, assume we wish to find all subjects for triples with predicate 3 (l5) 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 (l5) using Eqs. (4) and (5), we obtain $L_\mathrm{s}[b_s\,..\,e_s] = L_\mathrm{s}[8..9] = \langle 1, 5\rangle$, showing that we arrive at BA by l5 from sources $L_\mathrm{s}[8] = 1$ (SA) and $L_\mathrm{s}[9] = 5$ (Baq). □

The ring uses a data structure called a *wavelet tree* [38], described next, to index each of the sequences L_o, L_s, and L_p. This representation implements 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 [53, 36].

E. Wavelet trees

The wavelet tree represents a string $L[1\,..\,n]$ over an alphabet $[1\,..\,\sigma]$ as a perfect binary tree with σ leaves, one per symbol, so 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 $S_{\langle c_s, c_e\rangle}[i]$ descends by the left child of

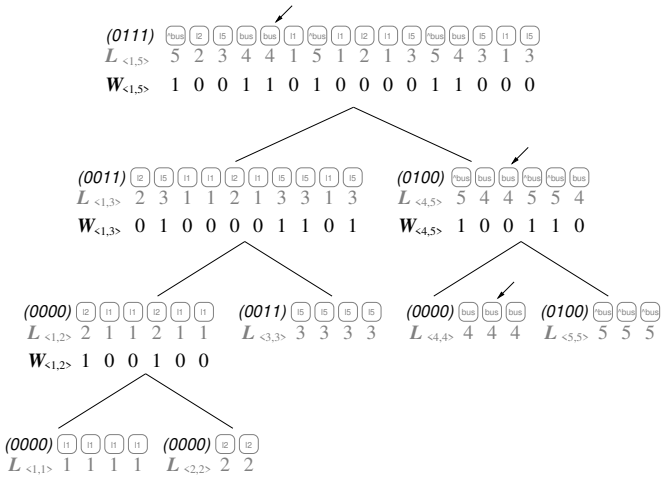

Fig. 5. The wavelet tree of the sequence L_p of Fig. 4. The short diagonal arrows track $L[5]$. The slanted bitvectors on the nodes refer to the B entries of the automaton of Fig. 6.

v. The leaves are conceptual and not stored. All the bitvectors stored at the internal wavelet tree nodes amount to $n \log \sigma$ bits (base 2), i.e., the same as a plain representation of L.

The wavelet tree obtains $L[i]$ 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 ease of presentation). 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 [54, 55]. Therefore, in time $O(\log \sigma)$ we arrive at a leaf and determine $L[i]$. The total space of the wavelet tree is $n \log \sigma + o(n \log \sigma) + O(\sigma \log n)$ bits, the latter term being needed for the tree pointers. Note that $O(\sigma \log n)$ also absorbs the space of the arrays C_x used for backward search.

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. Furthermore, the number of leaf positions to the left of c is precisely $C[c]$, which directly gives the values of the LF and the backward search formulas (Eqs. (3) to (5)).

Example. Fig. 5 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 $\mathsf{rank}_4(L_\mathrm{p}, 5) = i = 2$. The lengths of all the leaves to the left add up to $C_\mathrm{p}[4] = 10$, so adding i we obtain position $12 = \mathsf{LF}_\mathrm{p}(5)$. □

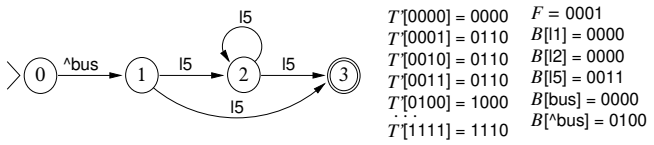

Fig. 6. The Glushkov automaton for the regular expression $\hat{}\mathsf{bus}/\mathsf{l5}^*/\mathsf{l5}$, its bitvector F and table B, and the transition table T' of its reversed automaton.

Wavelet trees can be used for many other purposes [56, 57]. We will indeed make use of their extended capabilities for our algorithm. A good warmup is the following algorithm to enumerate all distinct symbols in $L[b\,..\,e]$: We start at the root and descend to the 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 to the 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 (we later exemplify more complex variants of this algorithm). The total time is then $O(\log \sigma)$ per distinct symbol reported, irrespective of the total number of symbols.

IV. Our Approach

In order to evaluate RPQs, we will use part of the ring's structure to navigate backwards all the paths that match a given 2RPQ. More precisely, we use 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 $G^{\leftrightarrow}$.

We will first focus on 2RPQs 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 towards subjects. Since we use the NFA backwards, we will start from its final states, $D = F$, use the reverse Glushkov simulation of Eq. (2), and report a valid binding $x = s$ at every node $s \in V$ where the initial NFA state is activated. The navigation will start from the range of o in L_p.

This technique also handles 2RPQs 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 2RPQs.

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. Assume we are at station Baq and want to know what we can reach by following via line 5 (15) taking the bus once. The corresponding RPQ is $(\mathsf{Baq}, \mathsf{15}^+/\mathsf{bus}, y)$, and the reversed regular expression is $\hat{}E = \hat{}\mathsf{bus}/\mathsf{l5}^*/\mathsf{l5}$, equivalent to the example $a/b^*/b$ of Fig. 2. We have converted bus to

ˆbus to reverse the edge direction (we do not do this for 15, which is bidirectional). Fig. 6 shows the Glushkov automaton for this regular expression; note that $B[\text{ˆbus}]$ corresponds to $B[a]$ and $B[15]$ to $B[b]$ in Fig. 2, and that the alphabet of the regular expression is the set of predicates.

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

We will virtually traverse the relevant subgraph G'_E of the product graph G_E 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 $L_{\rm p}$ corresponding to the current object (initially, o), and is simulated in the following three parts:

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

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 also 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'_E. 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 loops of G'_E.

In terms of the product graph, visiting a node s of G with a set D of active NFA states corresponds to traversing *simultaneously* all the nodes of G'_E that combine s with an active state in D. Thus, bit-parallelism enables us to perform significantly less work than classical techniques that visit G'_E node by node. Furthermore, we will combine Fact 1 with the ability of wavelet trees to work on ranges of symbols to carry out steps 1 and 2 in a way that spends time *only on the resulting predicates and subjects*, thereby bounding our time complexity in terms of the subgraph of the size of G'_E, without spending any effort to discard edges that connect G'_E with other nodes of G_E. We now describe each part in detail.

A. Part one: Finding predicates from objects

The first part finds the distinct predicates p that lead to (i.e., precede in the (o, s, p) triples) the current range of objects. We will use the wavelet tree of $L_{\rm p}$ to discover all the distinct predicates p in $L_{\rm p}[b_o\,..\,e_o]$, as described at the end of Section III-E.

Next, we identify predicates p that lead to a currently active NFA state, i.e., such that $D \;\&\; B[p] \neq 0$ per Eq. (2). We will find them efficiently thanks to Fact 1, and an enhancement of the wavelet tree of $L_{\rm p}$, 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 $L_{\rm p}$: 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 enables us to confine the influence of p to the table B in the bit-parallel simulation.

Example. The $B[\cdot]$ entries for all the nodes of the wavelet tree of $L_{\rm p}$ are written as slanted bitvectors on the nodes in Fig. 5. Those on the leaves correspond to the entries in Fig. 6, and those on internal nodes to the bitwise-or of their children. □

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 $L_{\rm p}$.

With this extension of B, we proceed as follows. We start from the root v of the wavelet tree of $L_{\rm p}$, 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 $L_{\rm s}[b\,..\,e]$. 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. To start the search from $L_{\rm p}[14\,..\,15]$ 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. 5, with the range $L_{\langle 1,5\rangle}[14\,..\,15]$. 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 $\mathsf{rank}_0(W_{\langle 1,5\rangle}, 14-1)+1 = 9$ and $\mathsf{rank}_0(W_{\langle 1,5\rangle}, 15) = 10$). 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 11 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., 15) reaching our range (i.e., Baq). Its range is $L_{\langle 3,3\rangle}[4\,..\,4]$, which added to the number of leaves in 11 and 12 (equivalent to $C_{\rm p}[3] = 6$) yields the range $L_{\rm s}[10\,..\,10]$, completing the backward search step for symbol 15 (recall Eqs. (4) and (5)).

On the other hand, we do not descend from $v_{\langle 1,5\rangle}$ to its right child, $v_{\langle 4,5\rangle}$, because $B[v_{\langle 4,5\rangle}] \;\&\; D = 0100 \;\&\; 0001 = 0000$. Even if we did, we would obtain an empty interval in $L_{\langle 4,5\rangle}$

because there are no 4s or 5s in $L_{\langle 1,5\rangle}[15\,..\,16]$. □

Note that, if $D \,\&\, B[v] \neq 0$, then the same holds for at least one of the two children of v. As a consequence, all the wavelet tree nodes we traverse are ancestors of qualifying leaves. Since we spend constant time on each such ancestor, we can bound the total cost of this part by charging $O(\log |P|)$ to each useful predicate p, for which we report the interval $L_{\mathrm{s}}[b_p\,..\,e_p]$. We do not pay any extra cost on the useless predicates thanks to Fact 1, because we must intersect every $B[p]$ with the same set D of active states. In terms of the product graph traversal, where we are simultaneously processing all the nodes that combine o with the active states in D; this technique allows us to obtain all the distinct edges of G'_E that we can traverse from the current nodes of G'_E.

B. 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} to find all the distinct subjects s in that range, mapping them 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. (2)), 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. Once we obtain the range $L_{\mathrm{s}}[10\,..\,10] = 4$ (BA) from edge label 3 (15), identifying 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. 6). This new state D is independent of the subject we arrived at. □

We need to prevent falling into loops, 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 implement this filter efficiently, we will again exploit Fact 1 and enhance the wavelet tree of L_{s}.

For each subject s we store a bitvector $D[s]$ with all the active NFA states we already reached s with. This adds $O(|V|)$ working space, but can be zeroed in constant time with lazy initialization. If we arrive at s and $D \mid D[s] = D[s]$, then s can be skipped; otherwise we set $D \leftarrow D \,\&\, {\sim}D[s]$ and then $D[s] \leftarrow D \mid D[s]$, where "$\sim$" is bitwise-not. This leaves only the NFA states that are new to s in D, and adds to $D[s]$ the new active states we have arrived at s with. We initially mark the states F on the node o where we start the search.

We use the same technique of storing $D[\cdot]$ entries at wavelet tree nodes 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, $D[v]$ must be the intersection of those $D[s]$ cells below v. If $D \mid D[v] = D[v]$, we prune the wavelet tree traversal at node v, otherwise we set $D[v] \leftarrow D \mid D[v]$ and continue by both left and right children. Just as for predicates, there is always a useful descendant leaf s from the nodes v we traverse, and so the total cost is $O(\log |V|)$ per useful subject arrived at.

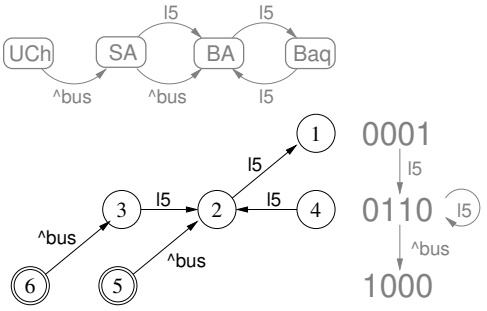

Fig. 7. The visited part of the product graph to match the DFA of Fig. 6 in our graph of Figs. 1 and 4. We show in the rows the DFA states (i.e., sets D of NFA states) we visit, and label the graph edges for readability.

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.

C. Part three: Mapping subjects back to objects

In this 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 easily 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. Fig. 7 shows the traversed part of the product graph for our running example, considering the DFA states. The numbers in the nodes represent the steps. The processes illustrated along the preceding examples aim to the (reverse) traversal $2 \xrightarrow{\text{l5}} 1$. □

D. Other kinds of RPQs

The algorithm we have described reports all the subjects (i.e., nodes) $s \in V$ for which there is a path matching E towards some object in the range we started with. If we start with the L_{p} range for a single object o (i.e., solving (x, E, o) with $x \in \Phi$ and $o \in V$), then the answers to the query are all the pairs (s, o). We can use this same algorithm for solving (s, E, y), where $s \in V$ and $y \in \Phi$, by converting it into $(y, \hat{}E, s)$ (as we did in our running example).

We can also handle RPQs where both $s, o \in V$ are fixed by starting from o and processing $\hat{}E$, stopping when we reach s or we 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 (x, E, o), one per possible object o, but this would be very inefficient if many objects do not lead to answers (s, o). In particular, we could work on edges that are not in G'_E. Instead, we will use the ability of the BWT and of wavelet trees to work on ranges of symbols, not only on individual ones. Instead of starting with each specific object o, we will start with the full range in L_p. 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. The edges traversed are in G'_E because each such subject s does reach some object o via 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. Since we run the queries (s, E, y) only from subjects that will produce some result, this second traversal also visits edges in G'_E. Alternatively, we can first find the objects o that are reachable with E from some subject, and then run only the useful queries (x, E, o).

The first step of this solution, which starts from the full L_p range, uses the power of the ring to handle a range of nodes simultaneously, traversing in one step a set of nodes of G'_E that relate a number of nodes of G with the same set of NFA states. This provides a second speedup over a classical node-wise traversal of G'_E. The union of the queries (s, E, y) we perform amounts to a second traversal of G'_E.

E. Time complexity

The following theorem shows that the cost of our algorithm is essentially bounded by the size of the subgraph G'_E of the product graph G_E induced by the query.

Theorem 1. *Let G be a directed labeled graph over nodes V and an alphabet P of edge labels. Consider an RPQ (x, E, y) where x or y are variables, E has m literals, and the computer word holds $O(m)$ bits. The ring representation of G can return all the matching pairs (s, o) for the RPQ in time $O(2^m + m \log |P| + |G'_E| \log |G|)$, where G'_E is the subgraph of the product graph G_E of G and Glushkov's automaton A of E, induced by all the paths ρ from any node (s_μ, i) to any node (o_μ, f), where μ is an evaluation of (x, E, y), and i and f are initial and final states of A. The working space of the query is $O(m(2^m + |P| + |V| + \max_\rho))$ bits, where $\max_\rho$ is the length of the longest path ρ. For any desired parameter $1 \leq d \leq m$, we can multiply the above times by d and reduce all 2^m terms in space and time to $2^{m/d}$; in particular d must be chosen so that the computer word holds $O(m/d)$ bits.*

Proof. The algorithm virtually visits the nodes of G'_E in reverse order. Let us first consider the query (x, E, o) for $o \in V$. The algorithm starts simultaneously from all the nodes $(o, f) \in G'_E$, for any final NFA state $f \in F$ (let us 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 $(v, d) \in G'_E$ for any NFA state $d \in D$. Each transition to new states (v', d') is done in three parts. In the first part, it finds in $O(\log |P|)$ time every distinct label $p \in P$ of edges in G'_E that lead to some state (v, d) for $d \in D$. This cost can be charged to the edges of G'_E that lead to some state (v, d), because there is at least one edge per resulting label. In the second part, for each label p found, it finds in $O(\log |V|)$ time every distinct node $v' \in V$ such that we reach some current node (v, d) from some unvisited node (v', d') in G'_E via label p (it obtains simultaneously the set D' of all those states d'). We can then charge the cost to the nodes (v', d') of G'_E. The third part takes $O(1)$ time per node arrived at, which becomes the current node in the next iteration.

The total cost is then $O(|G'_E| \log |G|)$ for the traversal. Glushkov's construction takes $O(m^2)$ time to mark all the bits in $B[1 \,..\, |P|]$ after the constant-time lazy initialization of B. The construction of the bit-parallel tables takes time $O(2^m)$, dominated by table T. Computing the cells B for the internal wavelet tree nodes of L_p adds $O(m \log |P|)$ time, again using lazy initialization, because only $O(m)$ wavelet tree leaves have nonzero cells in B. The lazy initialization of D for the wavelet tree nodes of L_s adds $O(1)$ time.

The working space is $O(m2^m)$ bits for the bit-parallel simulation of the NFA, $O(m(|P| + |V|))$ bits for the tables B/D 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 the tables B/D (see App. C, [58]), and $O(m \max_\rho)$ bits for the recursive path traversals carrying the active states D.

If the computer word cannot hold $O(m)$ bits, we split the words into d words of $O(m/d)$ bits each and operate on each word sequentially, thereby multiplying times by $O(d)$. This reduces every term 2^m in space and time to $2^{m/d}$, and can be done whenever it is convenient.

All the other types of queries (x, E, y) are reduced to the query (x, E, o) we have considered: in case (s, E, y), where $s \in V$ and $y \in \Phi$, we just reverse the query; in case (x, E, y) where $x, y \in \Phi$, we perform many queries (s, E, y), which subsume the cost of the initial query that finds the relevant nodes s from any o. Here G'_E is the union of the graphs G'_E for every s. The only case where we may visit more nodes than those in G'_E is the query (s, E, o) with both $s, e \in V$, because we visit nodes that may not lead to s. This is why this case is left out of the theorem. □

The theorem does not reflect that we can process several NFA states d simultaneously when traversing the nodes (v, d) of G'_E, thanks to the bit-parallel simulation of the NFA. On the other extreme, we can choose $d = m$ (i.e., simulate the automaton in nondeterministic form) so as to completely remove the exponential dependence on m; we then obtain a time complexity of $O(m^2 \log |P| + m|G'_E| \log |G|)$. This version still non-trivially manages to traverse just G'_E. 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.

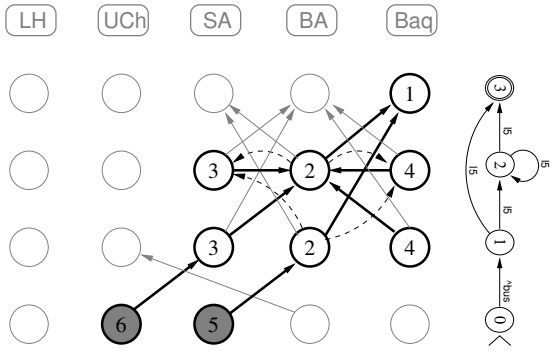

Fig. 8. The product graph G_E of the graph of Fig. 1 and the NFA of Fig. 6, highlighting in black the subgraph G'_E that is traversed (backwards) in Fig. 7.

Example. Fig. 8 shows the product graph G_E of our running example, with the nodes and edges of G'_E in bold. The rows now correspond to the NFA states, by definition. The dashed edges also belong to G'_E but we avoid them to prevent loops. The shaded nodes correspond to reported results. Theorem 1 proves that we spend, at worst, logarithmic time per node and edge of G'_E. Comparing the figure with Fig. 7, however, one can see that our simulation processes the nodes of the second and third rows of G_E simultaneously (in row 0110 of Fig. 7). We maintain in Fig. 8 the numbering of the nodes of Fig. 7, which helps see the nodes we visit simultaneously. □

V. Implementation and Experiments

We implemented our scheme in C++11 using the succinct data structures library (SDSL, https://github.com/simongog/sdsl-lite). We ran our experiments on an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores, 15 MB of cache, and 96 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/darroyue/Ring-RPQ.

Benchmark.: We evaluate our approach on a real-world benchmark based on a Wikidata graph [59] 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 10.7 GB in plain form (with 32-bit integers for each triple component, 12 bytes per tuple) and 7.9 GB in packed form (i.e., using $\lceil \log |S| \rceil + \lceil \log |P| \rceil + \lceil \log |O| \rceil$ bits, or 8.63 bytes per tuple).

We compare 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 [60].

Blazegraph: The graph database system [61] hosting the official Wikidata Query Service [8].

Datalog: A Datalog-based engine that we cannot identify due to licensing terms.

[1]To the best of our knowledge, ArangoDB, Neo4j, OrientDB and TigerGraph do not support RPQs declaratively with the standard semantics as we define, though Neo4j and TigerGraph do provide RPQ-like features.

Systems are then configured per vendor recommendations, as in previous work [36]. Jena, Virtuoso and Blazegraph all implement RPQs per the semantics of property paths in SPARQL 1.1, whereby fixed-length 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. Jena and Blazegraph implement a navigational BFS-style function called ALP (Arbitrary Length Paths) defined by the SPARQL standard [4], while Virtuoso uses a transitive closure operator implemented over its relational database engine.

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 apply 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 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, as well as an index on nodes.

In order to test on challenging, real-world RPQs, we extracted all non-trivial RPQs (i.e., not a simple label) from the code-`500` (timeout) sections of all seven intervals of the Wikidata Query Logs [8]. 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. All queries are run with a timeout of 60 seconds under set semantics (using `DISTINCT` in SPARQL). 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

TABLE I
RPQ PATTERNS WITH MORE THAN 10 QUERIES IN OUR LOG.

1st–5th	#
v/*c	450
v*c	421
v+c	107
c*v	98
c/*v	95

6th–10th	#
v/c	48
v*/*c	30
v|*c	30
v*/*/*/*/*c	28
v/v	20

11th–13th	#
v/?c	20
v^v	14
v|v	11

constant (c) or variable (v). Table I shows the patterns with more than 10 RPQs in our log.

Index construction: The Ring works with a dictionary-encoded version of the graph as described in Section IV, 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 [36]. We represent L_s and L_p using wavelet matrices [62], 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 2.3 hours, using 64.75 GB of RAM. Prior dictionary encoding takes 5.2 additional hours.

Implementing queries: We use our generic query algorithm of Section IV, 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 origin 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 [56]. 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 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 IV-A) 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 zeroing them again after running the query. Array D, on the other hand, is implemented using a compact lazy initialization structure (App. C, [58]), which uses $O(|V|)$ extra bits on top of D. We use 16-bit cells for D, as queries in our log have fewer than 16 predicates (with some few exceptions that use operator |, which are handled differently as explained).

TABLE II
INDEX SPACE (IN BYTES PER EDGE), INDEXING TIME (IN HOURS), AND SOME STATISTICS ON THE QUERY TIMES (IN SECONDS). ROW "TIMEOUTS" COUNTS THE QUERIES TAKING OVER 60 SECONDS OR REJECTED BY THE PLANNER FOR BEING TOO COSTLY. RPQS WITH SOME CONSTANT NODE ARE INDICATED BY c, AND WITHOUT BY ¬c.

	Ring	Jena	Virtuoso	Blazegraph	Datalog
Index space	16.41	95.83	60.07	90.79	78.32
Index time	7.5	37.4	3.0	39.4	6.0
Average	1.27	5.26	3.87	3.58	11.05
Median	0.07	0.20	0.14	0.13	2.71
Timeout	11	105	55	46	198
Average c	0.52	3.83	2.98	3.30	10.60
Median c	0.06	0.17	0.11	0.13	2.68
Timeout c	1	63	37	39	178
Average ¬c	14.02	29.59	18.95	8.35	18.84
Median ¬c	3.93	4.50	7.98	0.19	6.65
Timeout ¬c	10	42	18	7	20

a) Space and query time: Table II compares the space usage, the time taken for indexing, and the query times of the systems tested. The Ring is the smallest index, using 16.41 bytes per triple. This is about twice the space of the compact representation of the data, consistent with the fact that we duplicate all the edges. Array D, needed at query time, uses 3.09 additional bytes per triple, whereas B uses 9.04×10^{-5} bytes per triple. The total working space usage at query time is 19.50 bytes per triple, 1/3–1/5 of the space used by the other indexes (not considering their extra working space).

Virtuoso has the fastest indexing time, taking around 3 hours. For Datalog and Ring, which take 6.0 and 7.5 hours respectively, we include dictionary encoding, which took the bulk of time (5.2 hours). Jena (37.4 hours) and Blazegraph (39.4 hours) took much longer to index.

The Ring offers the fastest query times on average, being 2.8 times faster than Blazegraph, the next best performer. The Ring is also the system with fewest timeouts. On the queries where some node is a constant ("c" in the table, 94.5% of the log), the Ring is on average 5.7 times faster than Virtuoso, the next best competitor for this query type. 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 (in second place). Regarding medians, the Ring is about twice as fast as the next best performer overall, while Blazegraph greatly outperforms other systems in the median case for RPQs with two variables. Datalog is outperformed by the other query engines, which

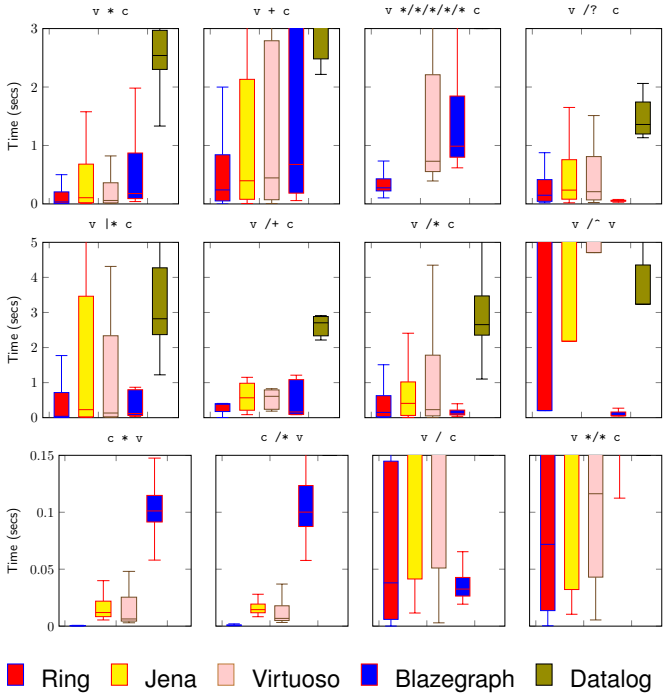

Fig. 9. Boxplots for the distribution of query times. Note that in the case of $\mathtt{c} * \mathtt{v}$ and $\mathtt{c}/ * \mathtt{v}$, the results for Ring hug the x-axis close to zero

offer native support and planning for RPQs.

Fig. 9 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 bottom row of plots. Also, in the case of $\mathtt{c} * \mathtt{v}$ and $\mathtt{c}/ * \mathtt{v}$, the results for Ring are so close to the x-axis that they may be difficult to see. We can observe that the Ring tends to outperform other systems in most patterns involving Kleene star ($*$) or Kleene plus ($+$). However, other systems sometimes outperform the Ring for RPQs matching paths of fixed length 1 or 2. Fixed-length paths can be solved as join queries, with more efficient algorithms that, for example, can start in the "middle" of the path, and work outwards (if deemed to be more efficient based, e.g., on cardinality estimates).

Our system and Datalog work on integer-encoded triples, whereas others work on the original strings. As shown in previous work [36], we can encode the strings of this benchmark within just 3 additional bytes per triple, incurring around 3 extra milliseconds per query, in order to decode the answers. These numbers do not alter our general conclusions.

VI. Conclusions

We have shown how the ring [36], 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 cost of the resulting algorithm is proportional to the subgraph of the product graph induced by the query, but our technique is even faster because it is able to process groups of nodes and labels simultaneously. As a result, our index uses 3–5 times less space than the alternatives, while matching or exceeding their performance (on average, our index is the fastest, outperforming the next best by a factor of 2.8).

We have not yet explored strategies for partitioning the NFA at edges with labels that appear infrequently in the graph and then joining the results, as do several techniques described in Section II. Our techniques do permit running the NFA forwards or backwards from those labels, so this could be explored in future. Furthermore, the wavelet tree offers powerful operations that provide on-the-fly selectivity statistics, which can be used for even more sophisticated query planning. For example, by roughly doubling the space, we can compute in logarithmic time the amount of distinct predicates labeling edges towards a given range of objects, or distinct subjects that are sources of a given range of predicates [63].

Our technique is particularly well-suited to integrate RPQs in SPARQL multijoin queries solved with Leapfrog Triejoin, reusing the same ring data structure [36]. In this case, in addition to the triples of the basic graph patterns, there will be triples of the form (x, E, y), where E is a regular expression. By treating E as any other relation, the Leapfrog algorithm will choose to first instantiate x (resp., y), and thus will ask for the smallest $x \geq x_0$ (resp., $y \geq y_0$) that has a solution for some y (resp., x). Later, it will instantiate y (resp., x) and will ask for the smallest $y \geq y_0$ (resp, $x \geq x_0$) that has a solution for a concrete value of x (resp., y). The capability of wavelet trees to work on ranges of symbols allows us to find those smallest $x \geq x_0$ or $y \geq y_0$ values efficiently, for example by successive binary partitioning of the range of candidates.

Other requirements are also efficiently met with our data structures. For example, we can easily enforce visiting specific nodes within the regular expression, or that those nodes have specific attribute values, by marking the non-complying nodes as already visited with the NFA states that enforce those conditions, so they will be avoided in our traversal. The bit-parallel Glushkov simulation also efficiently handles classes of symbols labeling the NFA edges (like $(11|12|15)$, or negated labels), without building unnecessarily large NFAs; this could be used to support negated property sets defined in SPARQL property paths, or even inference over RDF graphs (e.g., handling virtual disjunctions of inferred properties).

Finally, our ability to work on ranges of nodes of the product graph can be strengthened so as to maintain those ranges along our step 3. This could reduce processing times and allow for reporting ranges of results, instead of necessarily enumerating them one by one. This ability also 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, and might even break classical lower bounds based on the output size of the queries.

References

[1] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc, "Foundations of Modern Query Languages for Graph Databases," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 68:1–68:40, 2017. [Online]. Available: https://doi.org/10.1145/3104031

[2] I. F. Cruz, A. O. Mendelzon, and P. T. Wood, "A Graphical Query Language Supporting Recursion," in *SIGMOD International Conference on Management of Data*. ACM Press, 1987, pp. 323–330.

[3] A. O. Mendelzon and P. T. Wood, "Finding regular simple paths in graph databases," *SIAM Journal on Computing*, vol. 24, no. 6, pp. 1235–1258, 1995.

[4] S. Harris, A. Seaborne, and E. Prud'hommeaux, "SPARQL 1.1 Query Language," W3C Recommendation, Mar. 2013, http://www.w3.org/TR/sparql11-query/.

[5] E. V. Kostylev, J. L. Reutter, M. Romero, and D. Vrgoc, "SPARQL with Property Paths," in *International Semantic Web Conference (ISWC)*, ser. LNCS, vol. 9366. Springer, 2015, pp. 3–18.

[6] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "PGQL: a property graph query language," in *International Workshop on Graph Data Management: Experiences and Systems (GRADES)*. ACM, 2016, p. 7.

[7] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Aggregation Support for Modern Graph Analytics in TigerGraph," in *SIGMOD International Conference on Management of Data*. ACM, 2020, pp. 377–392.

[8] 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 *International Semantic Web Conference (ISWC)*, 2018, pp. 376–394.

[9] A. Bonifati, W. Martens, and T. Timm, "Navigating the Maze of Wikidata Query Logs," in *The World Wide Web Conference (WWW)*. ACM, 2019, pp. 127–138.

[10] Z. Miao, D. C. Stefanescu, and A. Thomo, "Grid-Aware Evaluation of Regular Path Queries on Spatial Networks," in *International Conference on Advanced Information Networking and Applications (AINA)*. IEEE Computer Society, 2007, pp. 158–165.

[11] A. Gubichev and T. Neumann, "Path Query Processing on Very Large RDF Graphs," in *International Workshop on the Web and Databases (WebDB)*, 2011.

[12] A. Koschmieder and U. Leser, "Regular Path Queries on Large Graphs," in *International Conference on Scientific and Statistical Database Management (SSDBM)*, ser. LNCS, vol. 7338. Springer, 2012, pp. 177–194.

[13] S. C. Dey, V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher, "On implementing provenance-aware regular path queries with relational query engines," in *Joint 2013 EDBT/ICDT Conferences*. ACM, 2013, pp. 214–223.

[14] A. Gubichev, S. J. Bedathur, and S. Seufert, "Sparqling kleene: fast property paths in RDF-3X," in *International Workshop on Graph Data Management Experiences and Systems (GRADES)*. CWI/ACM, 2013, p. 14.

[15] N. Yakovets, P. Godfrey, and J. Gryz, "Evaluation of SPARQL Property Paths via Recursive SQL," in *Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, ser. CEUR Workshop Proceedings, vol. 1087. CEUR-WS.org, 2013.

[16] 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 *Web-Age Information Management (WAIM)*, ser. LNCS, vol. 8485. Springer, 2014, pp. 372–383.

[17] G. H. L. Fletcher, J. Peters, and A. Poulovassilis, "Efficient regular path query evaluation using path indexes," in *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2016, pp. 636–639.

[18] M. Nolé and C. Sartiani, "Regular Path Queries on Massive Graphs," in *SIGMOD International Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, 2016, pp. 13:1–13:12.

[19] X. Wang, J. Wang, and X. Zhang, "Efficient Distributed Regular Path Queries on RDF Graphs Using Partial Evaluation," in *International Conference on Information and Knowledge Management (CIKM)*. ACM, 2016, pp. 1933–1936.

[20] N. Yakovets, P. Godfrey, and J. Gryz, "Query Planning for Evaluating SPARQL Property Paths," in *SIGMOD International Conference on Management of Data*. ACM, 2016, pp. 1875–1889.

[21] Z. Abul-Basher, "Multiple-Query Optimization of Regular Path Queries," in *International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2017, pp. 1426–1430.

[22] J. A. Baier, D. Daroch, J. L. Reutter, and D. Vrgoc, "Evaluating Navigational RDF Queries over the Web," in *ACM Conference on Hypertext and Social Media (HT)*. ACM, 2017, pp. 165–174.

[23] O. Hartig and G. Pirrò, "SPARQL with property paths on the Web," *Semantic Web*, vol. 8, no. 6, pp. 773–795, 2017. [Online]. Available: https://doi.org/10.3233/SW-160237

[24] V. Nguyen and K. Kim, "Efficient Regular Path Query Evaluation by Splitting with Unit-Subquery Cost Matrix," *IEICE Trans. Inf. Syst.*, vol. 100-D, no. 10, pp. 2648–2652, 2017.

[25] D. Colazzo, V. Mecca, M. Nolé, and C. Sartiani, "PathGraph: querying and exploring big data graphs," in *International Conference on Scientific and Statistical Database Management (SSDBM)*. ACM, 2018, pp. 29:1–29:4.

[26] V. Fionda, G. Pirrò, and M. P. Consens, "Querying knowledge graphs with extended property paths," *Semantic Web*, vol. 10, no. 6, pp. 1127–1168, 2019.

[27] Q. Mehmood, M. Saleem, R. Sahay, A. N. Ngomo, and M. d'Aquin, "QPPDs: Querying Property Paths Over

Distributed RDF Datasets," *IEEE Access*, vol. 7, pp. 101 031–101 045, 2019.

[28] K. Miura, T. Amagasa, and H. Kitagawa, "Accelerating Regular Path Queries using FPGA," in *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB)*, R. Bordawekar and T. Lahiri, Eds., 2019, pp. 47–54.

[29] S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, and S. Bedathur, "Efficiently Answering Regular Simple Path Queries on Large Labeled Networks," in *SIGMOD International Conference on Management of Data*. ACM, 2019, pp. 1463–1480.

[30] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda, "On the Optimization of Recursive Relational Queries: Application to Graph Queries," in *SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2020, pp. 681–697.

[31] A. Pacaci, A. Bonifati, and M. T. Özsu, "Regular Path Query Evaluation on Streaming Graphs," in *SIGMOD International Conference on Management of Data*. ACM, 2020, pp. 1415–1430.

[32] F. Tetzel, W. Lehner, and R. Kasperovics, "Efficient Compilation of Regular Path Queries," *Datenbank-Spektrum*, vol. 20, no. 3, pp. 243–259, 2020.

[33] X. Guo, H. Gao, and Z. Zou, "Distributed processing of regular path queries in RDF graphs," *Knowl. Inf. Syst.*, vol. 63, no. 4, pp. 993–1027, 2021.

[34] J. Kuijpers, G. Fletcher, T. Lindaaker, and N. Yakovets, "Path Indexing in the Cypher Query Pipeline," in *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2021, pp. 582–587.

[35] B. Liu, X. Wang, P. Liu, S. Li, and X. Wang, "PAIRPQ: An Efficient Path Index for Regular Path Queries on Knowledge Graphs," in *International Joint Conference on Web and Big Data (APWeb-WAIM)*, ser. LNCS, vol. 12859. Springer, 2021, pp. 106–120.

[36] D. Arroyuelo, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto, "Worst-case optimal graph joins in almost no space," in *ACM International Conference on Management of Data (SIGMOD)*, 2021, pp. 102–114.

[37] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.

[38] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003, pp. 841–850.

[39] V.-M. Glushkov, "The abstract theory of automata," *Russian Mathematical Surveys*, vol. 16, pp. 1–53, 1961.

[40] G. Navarro and M. Raffinot, "New techniques for regular expression searching," *Algorithmica*, vol. 41, no. 2, pp. 89–116, 2005.

[41] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum, "FERRARI: Flexible and efficient reachability range assignment for graph indexing," in *International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2013, pp. 1009–1020.

[42] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang, "Computing label-constraint reachability in graph databases," in *SIGMOD International Conference on Management of Data*. ACM, 2010, pp. 123–134.

[43] L. D. J. Valstar, G. H. L. Fletcher, and Y. Yoshida, "Landmark Indexing for Evaluation of Label-Constrained Reachability Queries," in *SIGMOD International Conference on Management of Data*. ACM, 2017, pp. 345–358.

[44] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large SPARQL query logs," *VLDB J.*, vol. 29, no. 2-3, pp. 655–679, 2020.

[45] 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.*, vol. 40, pp. 47–66, 2014. [Online]. Available: https://doi.org/10.1016/j.is.2013.10.003

[46] Y. Peng, Y. Zhang, X. Lin, L. Qin, and W. Zhang, "Answering Billion-Scale Label-Constrained Reachability Queries within Microsecond," *PVLDB*, vol. 13, no. 6, pp. 812–825, 2020.

[47] 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.*, vol. 31, no. 1, pp. 101–127, 2022.

[48] F. Bonchi, A. Gionis, F. Gullo, and A. Ukkonen, "Distance oracles in edge-labeled graphs," in *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, 2014, pp. 547–558.

[49] G. Berry and R. Sethi, "From regular expression to deterministic automata," *Theoretical Computer Science*, vol. 48, no. 1, pp. 117–126, 1986.

[50] A. Brüggemann-Klein, "Regular expressions into finite automata," *Theoretical Computer Science*, vol. 120, no. 2, pp. 197–213, 1993.

[51] T. L. Veldhuizen, "Triejoin: A simple, worst-case optimal join algorithm," in *Proc. International Conference on Database Theory (ICDT)*, 2014, pp. 96–106.

[52] P. Ferragina and G. Manzini, "Indexing compressed texts," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[53] A. Atserias, M. Grohe, and D. Marx, "Size bounds and query plans for relational joins," *SIAM Journal on Computing*, vol. 42, no. 4, pp. 1737–1767, 2013.

[54] D. R. Clark, "Compact PAT trees," Ph.D. dissertation, University of Waterloo, Canada, 1996.

[55] J. I. Munro, "Tables," in *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, 1996, pp. 37–42.

[56] T. Gagie, G. Navarro, and S. J. Puglisi, "New algorithms on wavelet trees and applications to information retrieval," *Theoretical Computer Science*, vol. 426-427, pp. 25–41, 2012.

[57] G. Navarro, "Wavelet trees for all," *Journal of Discrete Algorithms*, vol. 25, pp. 2–20, 2014.

[58] G. Navarro, "Spaces, Trees, and Colors: The algorithmic landscape of document retrieval on sequences," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 52:1–52:47, 2013. [Online]. Available: https://doi.org/10.1145/2535933

[59] D. Vrandecic and M. Krötzsch, "Wikidata: A free collaborative knowledgebase," *Communications of the ACM*, vol. 57, no. 10, pp. 78–85, 2014.

[60] O. Erling and I. Mikhailov, "RDF support in the Virtuoso DBMS," in *Networked Knowledge – Networked Media*. Springer, 2009, pp. 7–24.

[61] B. B. Thompson, M. Personick, and M. Cutcher, "The Bigdata®RDF Graph Database," in *Linked Data Management*. Chapman and Hall/CRC, 2014, pp. 193–237.

[62] F. Claude, G. Navarro, and A. Ordóñez, "The wavelet matrix: An efficient wavelet tree for large alphabets," *Information Systems*, vol. 47, pp. 15–32, 2015.

[63] T. Gagie, J. Kärkkäinen, G. Navarro, and S. J. Puglisi, "Colored range queries and document retrieval," *Theoretical Computer Science*, vol. 483, pp. 36–50, 2013.