Evaluating Regular Path Queries on Compressed Adjacency Matrices*

Diego Arroyuelo[1,2][0000−0002−2509−8097],
Adrián Gómez-Brandón[1,3][0000−0002−1216−2176], and
Gonzalo Navarro[1,4][0000−0002−2286−741X]

[1] Millennium Institute for Foundational Research on Data (IMFD)
[2] Department of Computer Science, Pontificia Universidad Católica de Chile, Chile
[3] CITIC Research Center, Universidade da Coruña, Spain
[4] Department of Computer Science, University of Chile, Chile

Abstract. Regular Path Queries (RPQs), which are essentially regular expressions to be matched against the labels of paths in labeled graphs, are at the core of graph database query languages like SPARQL. A way to solve RPQs is to translate them into a sequence of operations on the adjacency matrices of each label. We design and implement a Boolean algebra on sparse matrix representations and, as an application, use them to handle RPQs. Our baseline representation uses the same space as the previously most compact index for RPQs and excels in handling the hardest types of queries. Our more succinct structure, based on k^2-trees, is 4 times smaller and still solves complex RPQs in reasonable time.

1 Introduction and Related Work

Graph databases have emerged as a crucial tool in several applications such as web and social networks analysis, the semantic web, and modeling knowledge, among others. We are interested in labeled graph databases, where the graph edges have labels. An important kind of queries in such databases are the *regular path queries* (RPQs, for short), which search for paths of arbitrary length matching a regular expression on their edge labels [3]. For example, in the simple RDF model [23], one can represent points of interest in New York City as nodes in a graph, and have edges such as $x \xrightarrow{\mathsf{walk}} y$ indicating that x is within a short walking distance of y, as well as edges of the form $x \xrightarrow{\mathsf{L}} y$ if subway stations x and y are connected directly by subway line L. Then the RPQ 'Central Park $\mathsf{walk}/(\mathsf{N}|\mathsf{Q}|\mathsf{R})^+/\mathsf{walk}$?y', asks for all sites ?y of interest that are reachable from Central Park by using subway lines N, Q, or R, through one or more stations and allowing a short walk before and after using the subway.

* Supported by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1-230755, Fondecyt Grant 1221926; CITIC is funded by Xunta de Galicia and CIGUS; GAIN/Xunta de Galicia Grant ED431C 2021/53 (GRC); Xunta de Galicia/FEDER-UE Grant IN852D 2021/3; MCIN/AEI and NextGenerationEU/PRTR Grants [PID2020-114635RB-I00, TED2021-129245B-C21]

RPQs are at the core of current graph database query languages, extending their expressiveness. In particular, the SPARQL 1.1 standard includes the support for *property paths*, that is, RPQs extended with inverse paths (known as two-way RPQs, or 2RPQs for short) and negated label sets. As SPARQL has been adopted by several systems, RPQs have become a popular feature [3]: out of 208 million SPARQL queries in the public logs from the Wikidata Query Service [22], about 24% use at least one RPQ feature [9]. Further developments like PGQL [28], Cypher [18], G-CORE [2], TigerGraph [15], and GQL [14], to name some of the most popular ones, also support RPQ-like features.

Handling (2)RPQs can be computationally expensive to evaluate as they usually involve a large number of paths [24], mostly for regular expressions using Kleene stars. There are two main algorithmic approaches to support them [33]: (1) to represent the regular expression of the 2RPQ using a finite automaton, which is then used to search over the so-called product graph [25]; and (2) to extend the relational algebra to support computing the transitive closure of binary relations to evaluate regular expressions having Kleene stars [21]. Although most theoretical results on 2RPQs have followed the first approach, property path evaluation in SPARQL has followed the second one [33].

Recent research introduced not only time- but also space-efficient solutions for evaluating graph joins [5, 6, 10]. With the big graphs available today, this is an important step towards in-memory processing of graph queries. In particular, the Ring data structure [6] is able to represent a labeled graph in space close to its plain representation, while supporting worst-case optimal joins (used, as we said, for BGP queries). Moreover, by using little extra space the Ring can be used to support 2RPQs efficiently [4], using the product-graph approach [25].

In this paper, we introduce a space-efficient approach for evaluating 2RPQs that, essentially, represents the subgraph corresponding to each graph label p using a sparse representation of its Boolean adjacency matrix M_p. We evaluate 2RPQs by translating them into classic operations on Boolean matrices [21]. This approach is typically disregarded because matrix sizes are quadratic on the number of graph nodes, but we exploit the sparsity of those matrices to represent them efficiently, using k^2-trees [11]. The use of k^2-trees to represent each RDF predicate is not new, for example it has been used to handle triple matching and binary joins [1] and full BGPs [5], but not 2RPQs. We show how to translate 2RPQs into matrix operations and how to handle the particularities of 2RPQs.

The result is the most space-efficient graph database representation (nearly 4 bytes per graph edge on a Wikidata graph, 4 times less than the previously most compact representation—the Ring [4]—and 14–21 times smaller than classical systems). In exchange, our structure is on average 5 times slower than the Ring, though it still solves most complex 2RPQs in a few seconds. We also implement an uncompressed baseline for sparse matrices based on the CSR and CSC formats [29, Sec. 3.4]. Its space matches that of the Ring and it excels on the most expensive 2RPQs, namely those where no graph node is specified. It is only outperformed by Blazegraph, which uses 5.5 times more space. Our new matrix-algebra-based approach stands out in the space-time tradeoff map.

2 Basic Concepts

2.1 Labeled Graphs and Regular Path Queries (RPQs)

Let $\mathcal{U}$ be a totally ordered, countably infinite set of *symbols* or *constants*, which we call the *universe*. A *directed edge-labeled graph* $G \subseteq \mathcal{U}^3$ is a finite set of triples $(s, p, o) \in \mathcal{U}^3$ encoding the graph edges $s \xrightarrow{p} o$ from vertex s to vertex o with edge label p. In the RDF model [23] (which has gained popularity in representing directed edge-labeled graphs), s is called a *subject*, p a *predicate*, and o an *object*.

For a graph G, we define its set of edge labels as $P = \{p \mid \exists\, s, o, (s, p, o) \in G\}$. Similarly, let $V = \{x \mid \exists\, y, z,\ (x, y, z) \in G \vee (z, y, x) \in G\}$ be the set of graph nodes. We assume that the graph nodes have been mapped to integers in the range $[1\,..\,|V|]$. A path ρ from a node x_0 to node x_n in a graph G is a string $x_0 p_1 x_1 \cdots x_{n-1} p_n x_n$ such that $(x_{i-1}, p_i, x_i) \in G$ for $1 \leq i \leq n$. Given a path ρ, we denote $\mathsf{word}(\rho) = p_1 \cdots p_n$ the string labeling path ρ. Two-way RPQs (2RPQs) also allow traversing reversed edges. Hence, we define the set of inverse labels as $\hat{}P = \{\hat{}p \mid p \in P\}$, and $P^{\leftrightarrow} = P \cup \hat{}P$ the set of predicates and their inverses. We define the *inverse graph* as $\hat{}G = \{(y, \hat{}p, x) \mid (x, p, y) \in G\}$, and its *completion* as $G^{\leftrightarrow} = G \cup \hat{}G$. A *two-way regular expression* (2RE) is then formed from the following rules: ε is a 2RE; if $c \in P^{\leftrightarrow}$, then c is a 2RE; if E, E_1 and E_2 are 2REs, then so are E^* (Kleene star), E_1/E_2 (concatenation), and $E_1 \mid E_2$ (disjunction). If E is a 2RE, we also abbreviate E^*/E as E^+ and $\varepsilon|E$ as $E^?$.

The *language* $L(E)$ of E is defined exactly as that of the regular expressions over the alphabet $P^{\leftrightarrow}$ of terminals, and we say that a path ρ *matches* a 2RE E iff $\mathsf{word}(\rho) \in L(E)$. A *two-way regular path query*, or 2RPQ for short, is a query of the form (x, E, y), which looks for all the pairs of nodes (s, o) such that there exists a path $\rho = s p_1 \cdots p_n o$ in $G^{\leftrightarrow}$ where $\mathsf{word}(\rho) \in L(E)$; x and/or y can be constants (thus fixing the value of s and/or o, respectively), or variables.

2.2 An Algebra on Boolean Matrices

Let $A = (a_{i,j})_{1 \leq i,j \leq n}$ and $B = (b_{i,j})_{1 \leq i,j \leq n}$ be square $n \times n$ Boolean matrices. We define the following operations of interest for our work:

- **Transpose**: A^T, where $a^T_{i,j} = a_{j,i}$, for $1 \leq i, j \leq n$.
- **Sum**: $A + B = C = (c_{i,j})$, where $c_{i,j} = a_{i,j} \vee b_{i,j}$, for $1 \leq i, j \leq n$.
- **Product**: $A \times B = C$, where for $1 \leq i, j \leq n$ we have $c_{i,j} = \bigvee_{1 \leq k \leq n} a_{i,k} \wedge b_{k,j}$.
- **Exponentiation**: $A^k = \prod_{i=1}^{k} A$, that is, $A \times \cdots \times A$, writing A k times.
- **Transitive closure**: $A^+ = A + A^2 + \cdots + A^n$.
- **Reflexive-transitive closure**: $A^* = I + A^+$, where I is the identity matrix.
- **Row/column restrictions**: $\langle r \rangle A$, a matrix whose row r equals row r of A; $A\langle c \rangle$, a matrix whose column c equals column c of A; and $\langle r \rangle A \langle c \rangle$, a matrix whose cell (r, c) equals entry $A[r][c]$. The remaining cells are 0.

The implementation of these operations on sparse matrix representations is relatively straightforward, except for the multiplication and transitive closures.

2.3 K^2-trees

A k^2-tree [11] is a data structure able to space-efficiently represent binary relations, point grids, and graphs. We will use it in this paper to represent Boolean matrices, as follows. Let A be a $v \times v$ Boolean matrix, assuming v is a power of 2.[5] The root node of the k^2-tree represents the whole matrix A. Then, A is divided into 4 equally-sized quadrants, $A = \binom{A_0\ A_1}{A_2\ A_3}$, such that submatrix A_0 is represented recursively by the first child of the root, A_1 by the second child, and so on. The process stops as soon as one gets into an empty submatrix, which is represented by a leaf node. Each node in this tree has 4 children (in general, k^2 children, yet we use $k = 2$). This order in which quadrants are represented (i.e., top-left, top-right, bottom-left, and bottom-right) is known as z-order. The resulting tree height is $\log_4 v^2 = \log_2 v$.

To represent this tree space-efficiently, we traverse the tree in level order. At each node, we write its 4-bit signature (which represents the node) indicating whether each of the 4 children represents an empty submatrix or not. For instance, the signature `0110` indicates that quadrants 0 and 3 of the submatrix represented by the current node are empty, whereas A_1 and A_2 (second and third children) are non-empty. The result is a bit vector $L[1\,..\,4n]$, where n is the number of internal nodes in the tree. Each tree node is represented by the first bit of its signature. Given a node i, its j-th child ($0 \leq j \leq 3$) is represented at position $4 \cdot \mathsf{rank}_1(L, i) + 1$, where $\mathsf{rank}(L, i)$ counts the number of 1s in $L[1\,..\,i]$ in $O(1)$ time using $o(n)$ additional bits of space [12, 26].

The k^2-tree representation is especially useful for representing sparse matrices. Let matrix A have a 1s. Then, in the worst case every 1 induces a 4-bit signature in every level of the k^2-tree, for a total of $4a \log_2 v$ bits. The actual upper bound is lower because not all those signatures can be different: in the worst case all the k^2-tree nodes up to level $\lfloor \log_4 a \rfloor$ exist, and from there on each 1 of A has its own path; this adds up to $4a \log_4(v^2/a) + 4a/3 + O(1)$ bits. The figures further improve when the 1s are clustered in A [7].

3 Evaluating RPQs through the Boolean Matrix Algebra

For a given directed edge-labeled graph G of n edges, let P be the corresponding set of graph labels as defined in Section 2.1. In our approach, for every $p \in P$ we define a $|V| \times |V|$ Boolean matrix M_p, such that $M_p[x][y] = 1$ iff $(x, p, y) \in G$. We translate an RPQ into operations on those matrices, so that the resulting Boolean matrix contains all pairs (x, y) that match the regular expression. We define next the recursive formulas $\mathcal{M}$ to translate 2RPQs into matrix operations, following Losemann and Martens' work [21]. We start with the base cases: $\mathcal{M}(\varepsilon) = I$, the identity matrix; $\mathcal{M}(p) = M_p$, for $p \in P$; $\mathcal{M}(\hat{}p) = M_p^T$, for $p \in P$.

Next, let E_1 and E_2 be 2RPQs. We define the following recursive rules: $\mathcal{M}(E_1 \mid E_2) = \mathcal{M}(E_1) + \mathcal{M}(E_2)$; $\mathcal{M}(E_1/E_2) = \mathcal{M}(E_1) \times \mathcal{M}(E_2)$; $\mathcal{M}(E_1^+) = \mathcal{M}(E_1)^+$; $\mathcal{M}(E_1^*) = I + \mathcal{M}(E_1)^+$, where I is the corresponding identity matrix.

[5] If v is not a power of 2 we round it up to the next power, leaving the extended cells empty. This imposes almost no extra overhead on the k^2-tree representation.

Then, given a 2RPQ $R = (x, E, y)$, we evaluate it as follows: (1) if x and y are both variables, $\mathcal{R}(R) = \mathcal{M}(E)$; (2) If x is a variable and y is a constant, $\mathcal{R}(R) = \mathcal{M}(E)\langle y\rangle$; (3) If x is a constant and y is a variable, $\mathcal{R}(R) = \langle x\rangle\mathcal{M}(E)$; (4) If x and y are both constant, $\mathcal{R}(R) = \langle x\rangle\mathcal{M}(E)\langle y\rangle$.

4 Implementation of the Boolean Matrix Algebra

We now describe how the Boolean-matrix operations are carried out. To analyze the corresponding algorithms, we use $|M_p|$ as the number of 1s in the matrix, which is the number of edges with label p in graph G. We represent each matrix M_p using a k^2-tree of $O(\log |V|)$ levels, and each 1 in M_p induces $O(\log |V|)$ 1s in its k^2-tree representation. We will use $v = |V|$, as well as $a = |A|$ and $b = |B|$ for the number of 1s in matrices A and B. We assume $|V| = 2^i$, for $i \geq 0$.

We implement k^2-trees, and thus bitvectors with rank support, from scratch. We store the bitvector as consecutive bits packed in a 64-bit-words array. To support rank we store the cumulative sum of 1s up to every sth cell of the array. To save space, full 64-bit integers store the full sum only every 2^{16} bits, and the others are stored in relative form using 16-bit integers. To compute rank we start from the last recorded sum and use *popcount* on the full words until reaching the desired one, and a partial *popcount* on the desired word. Here s is a space-time tradeoff parameter: we use $n/1024 + n/(4s)$ additional bits of space for storing a bitvector $B[1\,..\,n]$, and compute rank in time $O(s)$. We use $s = 4$.

4.1 Transposition

Transposition is used to implement reversed edges, as seen in Section 3. Instead of materializing the transposed matrix as a k^2-tree, we note that $A^T = \left(\begin{smallmatrix} A_0^T & A_2^T \\ A_1^T & A_3^T \end{smallmatrix}\right)$. So, the k^2-tree for A^T can be obtained by interchanging the roles of the second and third children of every node. We do not materialize this interchange, but associate a *transposed* flag to every matrix, so we simply have to toggle it in order to transpose the matrix in $O(1)$ time.

4.2 Boolean Sum

The easier case to implement $A + B$ arises when no matrix is transposed. In this case we can perform a sequential pass over both k^2-tree bitvectors, so as to merge their corresponding nodes levelwise, without need of any rank operation.

We implement this traversal with a queue of tasks, which are of two types. (*1*) A *copy* task indicates just to copy the next node from A or B; and (*2*) a *merge* task indicates merging the next nodes of A and B. The queue is initialized with a merge task, the read-pointers (which indicate the next k^2-tree node to be read) at the beginning of the bitvectors of A and B, and the write-pointer at the beginning of the output k^2-tree bitvector.

To process a copy task, we append the next signature (of A or B) to the output, and enqueue its (up to) 4 children, as copy tasks for A or B, respectively.

To process a merge task, we append to the output the bitwise-or of the next 4-bit signatures of A and B, and enqueue up to 4 new elements, as follows. For i from 1 to 4, if the ith bit of the signatures of both A and B are 1, we append a merge task. If only one of them is 1, we append a copy task for the corresponding matrix. If none is 1, we do not append a task. We do not append new tasks when the corresponding nodes are k^2-tree leaves. The process finishes when the queue becomes empty. The total time is proportional to the sum of the bitvector length of both matrices, $O(a \log(v^2/a) + b \log(v^2/b)) \subseteq O((a+b) \log v)$.

Handling transpositions. If both A and B are transposed, we just merge them as described and mark the result as transposed. When one is transposed and the other is not, we cannot anymore resort to a sequential traversal of both bitvectors. The transposed one must already have rank support built to enable k^2-tree traversals. We traverse sequentially the non-transposed k^2-tree, and include in the queue the corresponding node of the transposed one (as those nodes are not read in left-to-right order). To generate the new tasks, we must use the k^2-tree traversal operations to locate the corresponding nodes in the transposed k^2-tree.

While the time complexity is the same, summing a transposed with a non-transposed matrix is slower in practice. We always choose that the transposed matrix is the one with a shorter bitvector (we can because $A^T + B = (A + B^T)^T$), in order to minimize the non-local traversals.

4.3 Boolean Multiplication

For the multiplication $A \times B$ we use the following classic divide-and-conquer recursive procedure. Let $A = \left(\begin{smallmatrix} A_0 & A_1 \\ A_2 & A_3 \end{smallmatrix}\right)$ and $B = \left(\begin{smallmatrix} B_0 & B_1 \\ B_2 & B_3 \end{smallmatrix}\right)$ be the four submatrices into which the k^2-tree representation splits A and B. Then, we recursively compute 8 products of those submatrices in order to produce

$$A \times B = \left(\begin{array}{c:c} A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ \hdashline A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \end{array} \right). \tag{1}$$

A fortunate consequence of the k^2-tree representation is that, if any of those submatrices is empty (i.e., there is a 0 in the signature of the root of A or B), then we know that its product with any other submatrix is also zero. Further, summing a product $A_i \times B_j$ with a zero matrix does not even need to copy the product; we just reference it as the final result.

Once the k^2-tree bitvectors of the four submatrices are recursively obtained, we concatenate them levelwise. There is no need to build the rank data structures until we obtain the final matrix because the concatenation proceeds left-to-right in each level. We only take care of maintaining, for each bitvector, $O(\log v)$ pointers to the positions where the levels start.

Transpositions are handled easily, by exchanging the meaning of M_1 and M_2 in every node of the k^2-tree bitvector if $M = \left(\begin{smallmatrix} M_0 & M_1 \\ M_2 & M_3 \end{smallmatrix}\right)$is transposed.

A rough analysis. One term of the multiplication cost is given by the number of recursive calls, which follows the recurrence $T(v^2) = 8 \cdot T(v^2/4)$. Since our matrices are sparse, the worst case arises when every submatrix has points up to the level ℓ where we have $4^\ell \geq \min(a,b)$ submatrices, that is, $\ell = \log_4 \min(a,b)$. From this level, the worst case is that the $\max(a,b)/\min(a,b)$ points in the submatrices of the fuller matrix distribute uniformly for $\ell' = \log_4 \frac{\max(a,b)}{\min(a,b)}$ further levels. Between those levels, the recurrence becomes $T'(v^2) = 2 \cdot T'(v^2/4)$ because the single point in the emptier submatrix can make us enter into at most two submatrices of the other. This continues until, in level $\ell + \ell'$, both submatrices contain one point each, and from there on the cost is just $\log_2 v - \ell - \ell'$ to track a single point along both submatrices. The cost up to level ℓ is then $8^\ell = \min(a,b)^{3/2}$. From each of those 8^ℓ submatrices we have a cost of $2^{\ell'} = (\max(a,b)/\min(a,b))^{1/2}$, and from each of those $8^\ell 2^{\ell'} = \min(a,b)\sqrt{\max(a,b)}$ submatrices we have $O(\log(v^2/\max(a,b)))$ additional time. The total cost of recursive calls is then $O(\min(a,b)\sqrt{\max(a,b)}\log(v^2/\max(a,b)))$.

The second part of the cost is that of summing pairs of partial submatrices. In the worst case, those matrices may add up to $a \cdot b$ points at across every level ℓ of the recursion. Since summing submatrices in level ℓ costs $O(\ell)$ per element, the total cost of summing partial results is in $O(ab \log^2 v)$. Since this is an utterly pessimistic upper bound, we offer an average-case time analysis for matrices with uniformly distributed 1s. We multiply 8^ℓ pairs of $v/2^\ell \times v/2^\ell$ submatrices in level ℓ. On average, each has $a/4^\ell$ 1s in A and $b/4^\ell$ cells in B. Every such a_{ik} will pair with every such $b_{k'j}$ iff $k = k'$, which occurs with probability $1/(v/2^\ell)$, so on average there will be $8^\ell(a/4^\ell)(b/4^\ell)(2^\ell/v) = ab/v$ 1s to sum per level ℓ, with a maximum of v^2. This leads to a total average cost upper bounded by $O(\min(a,b)\sqrt{\max(a,b)}\log v + \min(v^2, (ab/v))\log^2 v)$.

4.4 Closure

We opted for a simple transitive closure algorithm for now. The closure A^+ is obtained by iteratively computing $A \leftarrow A + A \times A$ until no change occurs in A [19]. This occurs at most after $\log_2 v$ iterations, so the time complexity is $O(\log v)$ times that of multiplying A by itself (note that a grows in every iteration, so the time complexity becomes bounded by $O(|A^+|^{3/2}\log^3 v)$). The transitive closure is computed as $A^* = I + A^+$, where I is the identity matrix.

Needless to say, unrestricted closure operations are the most expensive, both in time complexity and in practice, so we aim to avoid them as much as possible.

4.5 Restrictions

Restrictions indicate that we only want to retrieve a column or a row of the matrix after the operations, or even just a cell. A naive way to implement them is to first obtain the full matrix M and then traverse the desired row or column. Yet, restrictions give an important opportunity of optimizing all the other operations.

Sums. For $\langle r\rangle(A+B)\langle c\rangle$ (where only $\langle r\rangle$ or only $\langle c\rangle$ could be present as well), we restrict the traversal of both matrices, acting as if the submatrices not intersecting the desired row and/or columm were empty. That is, we implement the restricted sum as $\langle r\rangle A\langle c\rangle + \langle r\rangle B\langle c\rangle$. We cannot, however, simply traverse both k^2-tree bitvectors and write the output left-to-right, as in Section 4.2, because now we do not know beforehand whether a submatrix (or the merge of two submatrices) will be nonempty after restricting it to some row/column, even if it intersects the row/column. Our solution is then recursive, similar to the multiplication algorithm (yet still considerably simpler).

Products. A restricted product $\langle r\rangle(A \times B)\langle c\rangle$ is handled as $(\langle r\rangle A) \times (B\langle c\rangle)$, where again only one of the restrictions may be present. We consider the column or row restrictions along the whole recursion, pretending that the submatrices that do not intersect the desired row or column are empty.

Closures. Operation $A^+\langle c\rangle$ is implemented as $S \leftarrow (E+A)\langle c\rangle$, where E is the empty matrix, and then repeatedly doing $P \leftarrow A \times S$ and $S \leftarrow S+P$ until S does not change. Note that the only nonzero column of P and S is c. To implement $A^*\langle c\rangle$ we start with $S = (I+A)\langle c\rangle$ instead. A row restriction $\langle r\rangle A^+$ is handled analogously, starting with $S = \langle r\rangle(A+E)$ and then iterating over $P \leftarrow S \times A$ and $S \leftarrow S+P$, or using the initial step $S \leftarrow \langle r\rangle(I+A)$ for $\langle r\rangle A^*$.

Note that this iteration does not make the path lengths grow exponentially for the transitive closure, but linearly. Therefore, we could need up to v iterations to compute the closure. In practice, the closure is reached much sooner and the operations are significantly faster, leading to a much better solution.

When both row and column are restricted, we only want a cell of the transitive closure. We then choose the row/column with fewer elements in A and run a row-restricted or column-restricted closure, whichever is emptier. At each step, we check if the desired cell is full, stopping immediately if so.

4.6 Query Plan

We first build the syntax tree of the 2RE E of the 2RPQ (x, E, y). In principle, we can simply traverse the syntax tree and solve it in postorder in the standard way, interpreting the leaves p as the matrix M_p, $\hat{}p$ as M_p^T, and ε as I, and interpreting the internal nodes as the corresponding operations on the matrices resulting from their children, according to the translations of Section 3. Our particular application, however, enables some relevant optimizations.

Let us first assume that both x and y are variables. A first simple optimization is that the closures are idempotent, so a sequence of closures is reduced to one. More precisely, $(A^*)^* = (A^*)^+ = (A^+)^* = A^*$ and $(A^+)^+ = A^+$. Sums and products yield more important optimizations, though.

Sums. We exploit the fact that the Boolean sum is commutative and associative to carry out a sequence of consecutive sums, $E_1 \mid \ldots \mid E_m$, in the best possible order. Since the cost of computing $A+B$ is proportional to $|A|+|B|$, if it were the

case that $|A+B| = |A|+|B|$, the best possible order would be given by building the Huffman tree [20] of the matrices $A_i = \mathcal{M}(E_i)$ using $|A_i|$ as their weight. Since, instead, it holds that $\max(|A|, |B|) \leq |A + B| \leq |A| + |B|$, we opt for a heuristic that simulates Huffman's algorithm on the actual size of the matrices as they are produced. Concretely, we start with $\{A_1, \ldots, A_m\}$ and iteratively remove from the set the two matrices A_i and A_j with the smallest sizes, sum them, and return $A_i + A_j$ to the set, until it has a single matrix.

Products. Matrix multiplication is not commutative but still associative, so we can decide the order in which the sequence of multiplications to compute $E_1 \,/ \cdots / \, E_m$ is carried out. We cannot apply the well-known optimal algorithm to choose the order for dense matrices [13, Sec. 15.2] because the time complexity of our sparse matrix multiplications depends on the number of 1s in the matrices. Further, this number of 1s can increase or decrease after a multiplication. We then opt for a heuristic analogous to the one we use for sums: we start from the sequence $A_1, \ldots, A_m = \mathcal{M}(E_1), \ldots, \mathcal{M}(E_m)$ and iteratively choose the consecutive pair A_i, A_{i+1} that minimizes $|A_i| + |A_{i+1}|$, multiply them, and replace the pair by $A_i \times A_{i+1}$, until the sequence has a single element.

Handling restrictions. When x (resp., y) is a constant we are restricting a row (resp., column) of the matrix after the operations. For efficiency, then, we apply the restricted operations of Section 4.5. Regarding the sums, because $\langle r\rangle(A + B)\langle c\rangle = \langle r\rangle A\langle c\rangle + \langle r\rangle B\langle c\rangle$, we can restrict all the involved matrices at the same time. Consequently, the sum can be computed in any order, and the plan still focuses on looking for the best order based on Huffman's algorithm. In the restriction on products, we obtain a sequence $\langle r\rangle A_1 \times \cdots \times A_m\langle c\rangle$ (where only $\langle r\rangle$ or only $\langle c\rangle$ could be present). Consider the case $\langle r\rangle A_1 \times \cdots \times A_m$. The number of 1s reduces faster when multiplying the pair that contains the restricted matrix, so we compute $A' = \langle r\rangle A_1 \times A_2$. The matrix A' already has all zeros except in row r, so we can continue left-to-right in the sequence with normal matrix multiplications, $A' \times A_3$, and so on. The case $A_1 \times \cdots \times A_m\langle c\rangle$ is analogous, starting with $A' = A_{m-1} \times A_m\langle c\rangle$ and then completing the multiplications right to left. When both restrictions are present, we choose an end and proceed as explained until the final multiplication, $\langle r\rangle A' \times A''\langle c\rangle$, which is carried out with the restricted multiplication algorithm to enforce the other restriction.

Some restrictions can be inherited by the operands of a node, which speeds up processing. Since $\langle r\rangle(A + B)\langle c\rangle = \langle r\rangle A\langle c\rangle + \langle r\rangle B\langle c\rangle$, both children of a sum inherit the same restrictions. Instead, the product $\langle r\rangle(A \times B)\langle c\rangle = (\langle r\rangle A) \times (B\langle c\rangle)$, thus only the left child inherits a row restriction and only the right child inherits a column restriction. Closures do not inherit their restrictions to their operand, because $\langle r\rangle A^*\langle c\rangle \neq (\langle r\rangle A\langle c\rangle)^*$ and $\langle r\rangle A^+\langle c\rangle \neq (\langle r\rangle A\langle c\rangle)^+$. Restrictions are not inherited to leaves of the syntax tree, however, because internal operands handle them more efficiently than leaves. On the other hand, they are removed from parents when inherited to children because the nonrestricted operands run faster than those of Section 4.5 when their operands have already been restricted.

Finally, we create a special implementation for the case $A^+ \times B\langle c\rangle$ that avoids computing the full closure A^+, as a kind of restricted positive closure that starts instead with $S \leftarrow A \times B\langle c\rangle$. To handle $A^* \times B\langle c\rangle$ we start with $S \leftarrow (E + B)\langle c\rangle$. The cases $\langle r\rangle A \times B^{*/+}$ are handled analogously, as well as the cases with both restrictions. The parser is enhanced to detect those cases.

5 Experimental Results

We implemented our scheme in C++11 and 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. We compiled using `g++` with flags `-std=c++11`, `-O3`, and `-msse4.2`.

5.1 A Baseline

We implemented a baseline representation of sparse matrices, which combines (and adapts to the Boolean case) the well-known CSR and CSC formats [29, Sec. 3.4] in order to speed up multiplications. It stores a vector of nonempty row numbers and a similar vector of their starting positions in a third, larger, vector. This third vector stores, for each nonempty row, the increasing sequence of the columns of its nonempty cells. Similar (redundant) vectors are stored for the column-wise view of the matrix.

Transpositions are carried out in $O(1)$ time by just exchanging the row-view and the column-view vectors. The Boolean sum $A + B$ merges the nonempty rows, and when the same row appears in both matrices it merges their nonempty columns. The column-view is computed analogously, thus the sum takes time $O(a + b)$. For the Boolean multiplication $A \times B$, we use Schoor's algorithm [30], whose average time is $O(ab/v)$ if the 1s are uniformly distributed. Our implementation, which is more space-efficient, takes $O(ab \log(v)/v)$ time.

Row and/or column restrictions are handled by restricting the above algorithms to the given row/column; note that finding the desired rows/columns takes just $O(\log v)$ time with the baseline format. Closure operations and their restrictions are performed as for the k^2-tree based representation. The parser and its optimizations are also exactly the same.

5.2 Benchmark

We used a Wikidata graph [32] of $n = 958{,}844{,}164$ edges, $v = 348{,}945{,}080$ nodes, and 5,419 predicates. Separating the edges by predicate and representing the two nodes of each edge as two 32-bit integers, the data set requires 8.5 GB.

We compared our implementations with the following systems:

- *Ring*: A compact data structure that supports RPQs in labeled graphs [4].
- *Jena*: A reference implementation of the SPARQL standard.
- *Virtuoso*: A popular graph database that hosts the public DBpedia endpoint, among others [17].

	k^2-tree	Baseline	Ring	Jena	Virtuoso	Blazegraph
Index space	4.33	16.45	16.41	95.83	60.07	90.79
Index time	0.3	5.5	7.5	37.4	3.0	39.4
Average	8.40	5.67	1.68	5.26	3.87	3.58
Median	1.38	2.46	0.08	0.20	0.14	0.13
Timeout	83	48	22	105	55	46
Average c	7.47	5.37	0.65	3.83	2.98	3.30
Median c	1.32	2.48	0.08	0.17	0.11	0.13
Timeout c	57	37	2	63	37	39
Average ¬c	24.19	10.75	19.22	29.59	18.95	8.35
Median ¬c	13.52	0.63	5.53	4.50	7.98	0.19
Timeout ¬c	26	11	20	42	18	7

Table 1. Index space (in bytes per triple), 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. 2RPQs with some constant node are indicated by c, and without by ¬c.

– *Blazegraph*: The graph database system [31] hosting the official Wikidata Query Service [22].

To evaluate complex real-world 2RPQs, we extracted all 2RPQs that were not simple labels, from the code-500 (timeout) sections of the seven intervals of the Wikidata Query Logs [22]. We then normalized variable names and removed disrupting queries: duplicated queries and queries producing more than 10^6 results for compatibility with Virtuoso. The result was 1,589 unique queries.

We ran the queries in each system with a timeout limit of 60 seconds. Table 1 summarizes the space usage and time performance of all the systems. Notably, our approach, using k^2-trees, yields the most compact structure, requiring only 4.33 bytes per triple (bpt). This is less than half the space of the described plain representation of the raw data, and nearly a fourth of the space used by the next smallest representations that support 2RPQs (Ring and our baseline). Classical systems use 14–21 times more space than our k^2-trees. Note also that the k^2-tree representation is orders of magnitude faster to build than the others.

Our reduced space is paid in terms of time performance. Our structure is around 5 times slower than the Ring, on average, and 1.5–2.5 times slower than the classical systems. Still, we solve these complex 2RPQs in less than 10 seconds on average. Among our matrix-based methods, our structure is 50% slower than the baseline, which uses 4 times more space.

On 2RPQs with some constant, our structure is 11.5 times slower than the Ring and 2.0–2.5 times slower than the classical systems. The gap is considerably narrowed, however, on 2RPQs with both variables, where our structure is just 25% slower than the Ring. Blazegraph is the fastest system in this case, being around 3 times faster than our structure, yet this comes at the expense of using

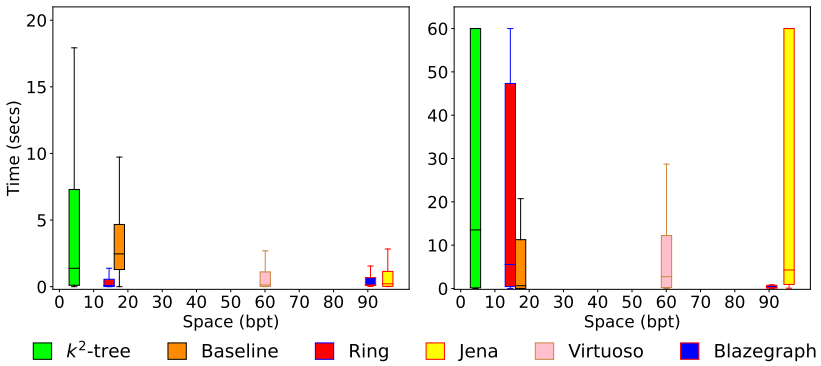

Fig. 1. Space and query time distribution of the systems in general (left) and for the 2RPQs with no constants (right). The baseline and the Ring use almost the same space.

using 21 times more space. Our baseline yields the best tradeoff for these queries, as it uses 5.5 times less space than Blazegraph and is only 30% slower.

Figure 1 displays the space and query time distribution of all the systems. It can be seen that the k^2-trees and the Ring are the dominant representations in general. When it comes to handling the hardest types of 2RPQs (i.e., without constants), the dominant representations are the two matrix-algebra-based solutions we have introduced and Blazegraph.

6 Conclusions

We have explored the use of a matrix algebra to implement Regular Path Queries (RPQs) on graph databases. This path is usually disregarded because the matrix sizes are quadratic on the number of graph nodes, but we exploit their sparsity to sidestep this issue. Our experiments show that even our baseline (i.e., uncompressed) sparse matrix representation uses the same space of the most compact among previous representations, and outperforms them on the most difficult RPQs (i.e., those with no constant ends). We also develop a more compressed sparse matrix representation based on k^2-trees, which is four times smaller than the baseline and, although slower, it still handles the RPQs within a few seconds.

Immediate extensions to our work are the implementation of negated labels, which require a nonexpensive way to represent and handle submatrices full of 1s. Such extensions of k^2-trees have been proposed [8], but they have not been adapted to handle Boolean matrix operations. We also plan to implement more efficient transitive closure algorithms [27]. Finally, we plan to strenghten our query optimizer in order to detect common subexpressions and exploit a number of identities of the Boolean algebra we have disregarded for now.

This work can be combined with Qdags [6] to support multijoins as well. We also plan to extend it to a complete algebra for sparse matrices, Boolean and possibly numeric [29]. Such matrices arise, for example, in ML applications [16].

An extended version can be found in https://arxiv.org/abs/2307.14930.

References

1. Álvarez-García, S., Brisaboa, N.R., Fernández, J., Martínez-Prieto, M., Navarro, G.: Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems **44**(2), 439–474 (2015)
2. Angles, R., Arenas, M., Barceló, P., Boncz, P.A., Fletcher, G.H.L., Gutiérrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J.F., van Rest, O., Voigt, H.: G-CORE: A core for future graph query languages. In: SIGMOD International Conference on Management of Data. pp. 1421–1432. ACM (2018). https://doi.org/10.1145/3183713.3190654
3. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoc, D.: Foundations of Modern Query Languages for Graph Databases. ACM Computing Surveys **50**(5), 68:1–68:40 (2017). https://doi.org/10.1145/3104031
4. Arroyuelo, D., Hogan, A., Navarro, G., Rojas-Ledesma, J.: Time- and space-efficient regular path queries. In: Proc. 38th IEEE International Conference on Data Engineering (ICDE). pp. 3091–3105 (2022)
5. Arroyuelo, D., Navarro, G., Reutter, J.L., Rojas-Ledesma, J.: Optimal joins using compressed quadtrees. ACM Transactions on Database Systems **47**(2), article 8 (2022)
6. Arroyuelo, D., Hogan, A., Navarro, G., Reutter, J., Rojas-Ledesma, J., Soto, A.: Worst-case optimal graph joins in almost no space. In: ACM International Conference on Management of Data (SIGMOD). pp. 102–114 (2021)
7. de Bernardo, G., Gagie, T., Ladra, S., Navarro, G., Seco, D.: Faster compressed quadtrees. Journal of Computer and System Sciences **131**, 86–104 (2023)
8. de Bernardo, G., Álvarez-García, S., Brisaboa, N.R., Navarro, G., Pedreira, O.: Compact querieable representations of raster data. In: Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 96–108 (2013)
9. Bonifati, A., Martens, W., Timm, T.: Navigating the Maze of Wikidata Query Logs. In: The World Wide Web Conference (WWW). pp. 127–138. ACM (2019)
10. Brisaboa, N., Cerdeira-Pena, A., de Bernardo, G., Fariña, A., Navarro, G.: Space/time-efficient rdf stores based on circular suffix sorting. The Journal of Supercomputing **79**, 5643–5683 (2023)
11. Brisaboa, N.R., Ladra, S., Navarro, G.: Compact representation of Web graphs with extended functionality. Information Systems **39**(1), 152–174 (2014)
12. Clark, D.R.: Compact PAT Trees. Ph.D. thesis, University of Waterloo, Canada (1996)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, 3rd edn. (2009)
14. Deutsch, A., Francis, N., Green, A., Hare, K., Li, B., Libkin, L., Lindaaker, T., Marsault, V., Martens, W., Michels, J., Murlak, F., Plantikow, S., Selmer, P., van Rest, O., Voigt, H., Vrgoč, D., Wu, M., Zemke, F.: Graph pattern matching in GQL and SQL/PGQ. In: Proc. International Conference on Management of Data (SIGMOD). pp. 2246—-2258 (2022)
15. Deutsch, A., Xu, Y., Wu, M., Lee, V.E.: Aggregation Support for Modern Graph Analytics in TigerGraph. In: SIGMOD International Conference on Management of Data. pp. 377–392. ACM (2020). https://doi.org/10.1145/3318464.3386144
16. Elgohary, A., Boehm, M., Haas, P.J., Reiss, F.R., Reinwald, B.: Compressed linear algebra for declarative large-scale machine learning. Communications of the ACM **62**(524), 83–91 (2019)

17. Erling, O., Mikhailov, I.: RDF support in the Virtuoso DBMS. In: Networked Knowledge – Networked Media, pp. 7–24. Springer (2009)
18. Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., Taylor, A.: Cypher: An Evolving Query Language for Property Graphs. In: SIGMOD International Conference on Management of Data. pp. 1433–1445. ACM (2018)
19. Furman, M.E.: Application of a method of fast multiplication of matrices in the problem of Finding the transitive closure of a graph. Soviet Mathematical Doklady **11**(5), 1252 (1970)
20. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proceedings of the Institute of Electrical and Radio Engineers **40**(9), 1098–1101 (1952)
21. Losemann, K., Martens, W.: The Complexity of Evaluating Path Expressions in SPARQL. In: Proc. 31st Symposium on Principles of Database Systems (PODS). pp. 101–112. ACM (2012)
22. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In: International Semantic Web Conference (ISWC). pp. 376–394 (2018)
23. Manola, F., Miller, E.: RDF Primer. W3C Recommendation (2004), `http://www.w3.org/TR/rdf-primer/`
24. Martens, W., Niewerth, M., Popp, T., Rojas, C., Vansummeren, S., Vrgoc, D.: Representing paths in graph database pattern matching. Proc. VLDB Endow. **16**(7), 1790–1803 (2023), https://www.vldb.org/pvldb/vol16/p1790-martens.pdf
25. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM Journal on Computing **24**(6), 1235–1258 (1995)
26. Munro, J.I.: Tables. In: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). pp. 37–42. LNCS 1180 (1996)
27. Penn, G.: Efficient transitive closure of sparse matrices over closed semirings. Theoretical Computer Science **354**(1), 72–81 (2006)
28. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: a property graph query language. In: International Workshop on Graph Data Management: Experiences and Systems (GRADES). p. 7. ACM (2016)
29. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM (2003)
30. Schoor, A.: Fast algorithm for sparse matrix multiplication. Information Processing Letters **15**(2), 87–89 (1982)
31. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata®RDF Graph Database. In: Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014)
32. Vrandecic, D., Krötzsch, M.: Wikidata: A free collaborative knowledgebase. Communications of the ACM **57**(10), 78–85 (2014)
33. Yakovets, N., Godfrey, P., Gryz, J.: Query Planning for Evaluating SPARQL Property Paths. In: SIGMOD International Conference on Management of Data. pp. 1875–1889. ACM (2016)