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

Evaluating Regular Path Queries on Compressed Adjacency Matrices

Diego Arroyuelo · Adrián Gómez-Brandón · Gonzalo Navarro

Received: date / Accepted: date

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 and GQL. 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 and time as the previously most compact index for RPQs, outperforming it on the hardest types of queries—those where both RPQ endpoints are unspecified. Our more succinct structure, based on k^2-trees, is 4 times smaller than any existing representation that handles RPQs. While slower, it still solves complex RPQs in a few seconds and slightly outperforms the smallest previous structure on the hardest RPQs. Our new sparse-matrix-based solutions dominate a good portion of the space/time tradeoff map, being outperformed only by representations that use much more space. They also implement an algebra of Boolean matrices that is of independent interest beyond solving RPQs.

Keywords Regular path queries on graph databases · Compact data structures for adjacency matrices · Sparse matrices · Sparse Boolean matrices

This work was supported by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1-230755; 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]. A preliminary version of this paper appears in *Proc. SPIRE 2023*.

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

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

✉ G. Navarro
IMFD & DCC, University of Chile, Santiago, Chile
E-mail: gnavarro@dcc.uchile.cl

1 Introduction and Related Work

Graph databases have emerged as a crucial tool in several applications such as web and social network analysis, the semantic web, and modeling knowledge, among others. We are interested in labeled graph databases, where the graph edges have labels. One particular way of querying graph databases is by means of basic graph patterns (BGPs, for short), which are small subgraphs with constant or variable nodes and edge labels that are to be matched homomorphically in the graph database. BGPs are strongly related to relational database joins [46]. Another important kind of queries that are more exclusive of graph databases are the *regular path queries* (RPQs, for short), which search for paths of arbitrary length matching a regular expression on their edge labels [7]. For example, in the simple RDF model [54], 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 '$\mathsf{Central\ Park\ walk/(O|R)^+/walk\ ?y}$', asks for all sites ?y of interest that are reachable from Central Park by using subway lines One ("O") or R, through one or more stations and allowing a short walk before and after using the subway.

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 [7]: out of 208 million SPARQL queries in the public logs from the Wikidata Query Service [53], about 24% use at least one RPQ feature [24]. Further developments like PGQL [66], Cypher [41], G-CORE [8], TigerGraph [33], and GQL [34], to name some of the most popular ones, also support RPQ-like features.

This paper focuses on the problem of, given a 2RPQ on a labeled directed graph, returning all the pairs of nodes connected by a path whose labels form a string matching the 2RPQ. We assume the graph is indexed beforehand to solve this query, and focus on indexed representations that use little space.

Solving (2)RPQs can be computationally expensive as they usually involve a large number of paths [55], mostly for regular expressions using Kleene stars. There are two main algorithmic approaches to support them [79]: (1) to represent the regular expression of the 2RPQ using a finite automaton, which is then used to search over the so-called product between the automaton and the database graph [56]; and (2) to extend the relational algebra to support computing the transitive closure of binary relations in order to evaluate regular expressions having Kleene stars [1, 52]. Although most theoretical results on 2RPQs have followed the first approach, property path evaluation in SPARQL has followed the second one [79].

Recent research introduced not only time- but also space-efficient solutions for evaluating graph joins [13, 26, 15]. With the big graphs available today, this is an important step towards in-memory processing of graph queries. In particular, the Ring data structure [15] 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 [16], using the product-graph approach [56].

Our contribution. In this paper, we introduce space-efficient approaches for evaluating 2RPQs that, essentially, represent 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 [52]. While this is in the line of mapping the syntax tree of the regular expression into another algebra [1], the use of the Boolean algebra as the target is typically disregarded because matrix sizes are quadratic on the number of graph nodes. We exploit, however, the sparsity of those matrices to represent them efficiently with two approaches:

1. We use k^2-trees [27] to represent each RDF predicate in compressed form. Although k^2-trees have been already used to handle triple matching and binary joins [5] and full BGPs [13], their use for supporting 2RPQs is new and requires novel algorithms. We show how to translate 2RPQs into matrix operations, particularly to Boolean sums, multiplications, and transitive closures, among other particularities of 2RPQs. We improve and extend known algorithms for the Boolean sum on k^2-trees [65], and develop new ones for sparse matrix multiplication (following a natural recursive strategy) and transitive closure (for which we develop a non-trivial strategy with the same time complexity as matrix multiplication).
2. We also adapt and implement an uncompressed baseline for sparse Boolean matrices based on the CSR and CSC formats [67, Sec. 3.4]. Our baseline implements state-of-the-art algorithms for sparse matrices, like Schoor's multiplication [69] adapted to the Boolean case, and an algorithm based on finding strongly connected components [64, 72] for the transitive closure. We implement those algorithms with special care on minimizing the working space.

Our main results can be summarized as follows:

- Our k^2-tree based representation (item 1 above) is the most space-efficient graph database representation so far. It uses nearly 4 bytes per graph edge on a Wikidata graph, which is 4 times less than the previously most compact representation—the Ring [16]—, 6.5 times less than Ring_{AB}—a larger and faster Ring variant—, and 14–22 times smaller than classical systems. In exchange, our structure is about 3 times slower than the Ring and 8 times slower than Ring_{AB}, though it still solves most 2RPQs within a few seconds. On the harder queries, however—those featuring both variable extremes—, our structure is slightly faster than the Ring and just about 3 times slower than Ring_{AB}. This implementation should be preferred when saving space is at premium.
- Our baseline (item 2 above) uses about 4 times the space of our k^2-tree based structure and is considerably faster. Its space matches that of the

Ring (the smallest previously existing structure), being slower than it by a factor of 1.6 on the simpler 2RPQs but outperforming it by a factor of 2.3 on the harder ones—on which it is still 1.5 times slower than Ring_{AB}. It should be preferred over the Ring when one expects to handle complex queries, as those drive the highest response times.

In general, our matrix-based implementations dominate the space/time tradeoff map of structures solving RPQs, yielding only to Ring_{AB} (which uses 6.5 times more space than k^2-trees and 1.7 times more than our baseline). A byproduct of our work yields a third relevant contribution:

- Both sparse Boolean matrix algebra implementations—from items 1 and 2 above—are of independent interest, and can be used in many other applications where operations like Boolean sums (and others like conjunction, difference, exclusive-or, etc.), multiplications, and transitive closures are of use. The matrices that result from operations use the same format of the input, and thus the system is compositional (i.e., the results can be further operated like the original matrices). We provide public implementations of both [17].

Some limitations of our solution follow. We discuss workouts for all of them in the Conclusions.

- Both data structures are static, that is, we can create new matrices as outputs of operations but cannot directly modify the matrices efficiently.
- The data structures are designed for use in main memory. The main point of the k^2-tree is to allow representing larger datasets in main memory, but it would not perform well on disk if that was necessary. The baseline solution, instead, would be disk-friendly with its current deployment.
- We do not support matrix complementation, which is useful to support negated label sets in property path queries. We left those negations aside because they are seldom used [25].

Compared to an early conference version of this paper [14], the present article includes improved algorithms for sum-like operations, multiplications, and especially transitive closures, on both the baseline and the k^2-tree based representations, a multithreaded implementation of the k^2-tree based algorithms, complete time complexity analyses of all the algorithms, and improved and extended experimental results.

2 Basic Concepts

2.1 Labeled Graphs and Regular Path Queries (RPQs)

Let $\mathscr{U}$ be a totally ordered, countably infinite set of *symbols* or *constants*, which we call the *universe*. A *directed edge-labeled graph* $G \subseteq \mathscr{U}^3$ is a finite set of triples $(s,p,o) \in \mathscr{U}^3$ encoding the graph edges $s \xrightarrow{p} o$ from vertex s to vertex o with edge label p. In the RDF model [54] (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 $P = \{p \mid \exists\ s,o \text{ such that } (s,p,o) \in G\}$. Similarly, let $V = \{x \mid \exists y,z \text{ such that } (x,y,z) \in G \vee (z,y,x) \in G\}$ 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 by $\mathsf{word}(\rho) = p_1 \cdots p_n$ the string labeling path ρ. Two-way RPQs (2RPQs) 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 rules:

1. ε is a 2RE.
2. If $c \in P^{\leftrightarrow}$, then c is a 2RE.
3. If E, E_1 and E_2 are 2REs, so are E^* (Kleene star), E_1/E_2 (concatenation), and $E_1 \mid E_2$ (disjunction).

We also abbreviate E^*/E as E^+ and $\varepsilon|E$ as $E^?$.

The *language* $L(E)$ of E is defined 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 $\mathsf{word}(\rho) \in L(E)$.

Let ϕ denote a set of variables, $\mu : \phi \to \mathscr{U}$ denote a partial mapping from variables to constants in $\mathscr{U}$, and $\mathsf{dom}(\mu)$ denote the set of variables for which μ is defined. If E is a 2RE, $s \in \phi \cup \mathscr{U}$ and $o \in \phi \cup \mathscr{U}$, then (s,E,o) is a *two-way regular path query*, or 2RPQ. 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 $G^{\leftrightarrow}$ as

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

In other words, the result of evaluating a 2RPQ (s,E,o) on $G^{\leftrightarrow}$ is the set of all pairs of constants (s_μ, o_μ) for which there exists a path $\rho = s_\mu p_1 \cdots p_n o_\mu$ in $G^{\leftrightarrow}$ such that $\mathsf{word}(\rho) \in L(E)$. Figure 1 illustrates our example RPQ on a small graph; the result of its evaluation is

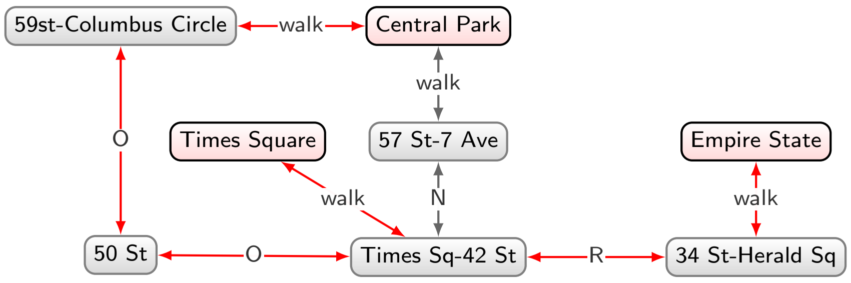


Fig. 1 Graph with points of interest and subway line connections across New York City. Red edges represent the paths that hold RPQ ‘Central Park walk/(O|R)$^+$/walk ?y’ and the red nodes are its solutions.

$\{$(Central Park, Times Square), (Central Park, Central Park), (CentralPark, EmpireState)$\}$.

2.2 Solving RPQs on Graph Databases

The most popular algorithmic approach to solve RPQs (s,E,o) on labeled directed graphs G is to traverse the so-called *product graph* [56, 20, 79], whose nodes are the cross-product between the automaton of the regular expression E and the nodes of the graph G. There is an edge labeled a from node (q,u) to node (p,v) in the product graph iff (1) one can go from state q to state p in the automaton by consuming an a, and (2) there is an edge labeled a from node u to node v in G.

The product graph is not built explicitly, but traversed in virtual form. If s is a constant node and o is a variable, the traversal starts from the node (q_0,s) of the product graph, where q_0 is the initial automaton state. One recursively visits all the neighbors of that node, avoiding to fall in loops of the product graph, and reporting every pair $(s_\mu,o_\mu)=(s,v)$ where we reach a node (f,v) in the product graph such that f is a final automaton state. If s is a variable and o is a graph node, we proceed analogously on the reversed edges. If both are graph nodes, we start from (q_0,s) and only report (s,o), stopping as soon as we find it. Finally, if both s and o are variables, we start from (q_0,s) for every possible node s in G.

Many real systems opt for other alternatives to solve RPQs [79], such as extending the relational algebra with transitive closures of binary relations so as to evaluate Kleene stars [1, 52]. The syntax tree of the expression E is then translated into the syntax tree of an expression in this extended relational algebra, for which various techniques of query optimization and query plan generation are developed. In this paper we follow a path closer to this line, using the Boolean instead of the Relational algebra to translate RPQs.

2.3 An Algebra on Boolean Matrices

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

- **Transpose**: A^T, where $a^T_{i,j}=a_{j,i}$.
- **Sum**: $A+B=C=(c_{i,j})$, where $c_{i,j}=a_{i,j}\vee b_{i,j}$.
- Other sum-like operations like $A\cap B$ (where $\vee$ above is replaced by $\wedge$), $A-B$ (where $\vee$ is replaced by $\wedge\neg$), and $A\oplus B$ (where $\vee$ is replaced by exclusive-or).
- **Product**: $A\times B=C$, for $c_{i,j}=\bigvee_{1\le k\le n} a_{i,k}\wedge b_{k,j}$.
- **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 restriction**: $\langle r\rangle A$, a matrix whose row r equals row r of A, the remaining cells are 0.
- **Column restriction**: $A\langle c\rangle$, a matrix whose column c equals column c of A, the remaining cells are 0.
- **Cell restriction**: $\langle r\rangle A\langle c\rangle$, a matrix whose cell (r,c) equals entry $A[r][c]$; the other cells are 0.

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

2.4 Boolean Matrix Multiplication and Transitive Closure

The multiplication of two $n\times n$ Boolean matrices A and B, of a and b non-zero entries, respectively, is one of the most important operations of the Boolean-matrix algebra, because of its applications in context-free parsing [75], context-free path queries on labeled graphs [18], triangle detection in graphs [48, 81], and on computing the transitive closure of Boolean matrices [40, 58, 43]. To illustrate its importance in the context of directed graphs, if A is a Boolean matrix representing the adjacency matrix of the graph, then $A^2 = A \times A$ is such that $A^2[i][j] = 1$ iff there is a path of length exactly 2 between nodes i and j. Also, by computing $(I + A)^2$ one obtains the Boolean matrix indicating the pairs of nodes (i, j) such that there is a path of length at most 2 between them. This can be generalized to any positive k-th power [80].

The most efficient algorithms for matrix multiplication work on algebraic rings, whereas $(0, 1, \vee, \wedge)$, the Boolean case, is just a semiring as there is no additive inverse. For instance, Strassen's algorithm [71] needs subtraction. A natural solution for the Boolean case is, however, to take the two values as integers, to then apply some fast multiplication algorithm. The result is then translated back to a Boolean matrix by replacing any non-zero value by a 1, whereas 0s remain unchanged. The fastest known matrix multiplication algorithm, by Coppersmith and Winograd, runs in time $O(n^{2.373})$ [30, 78]. Very recent advances [39] suggest that this exponent can be further pushed towards the lower bound $\Omega(n^2)$. Another approach is that of *combinatorial algorithms*, which use combinatorial properties of Boolean matrices to improve computation time. A typical example of this line is the (original) Four-Russians approach by Arlazarov et al. [10], which runs in time $O(n^3/\log^2 n)$ on a word RAM of $\Theta(\log n)$ bits [81]. After several progressive improvements, Yu [81] introduced an algorithm that runs in time $O(n^3 \mathrm{poly}(\log\log n)/\log^4 n)$.

For sparse matrices, Yuster and Zwick [82] introduce an algorithm that carries out $O(m^{0.7}n^{1.2} + n^{2+o(1)})$ algebraic operations, where $m = \max(a, b)$. As noticed by Yuster and Zwick, their algorithm runs in almost optimal $O(n^{2+o(1)})$ time when $m \leq n^{1.14}$, and it outperforms Coppersmith and Winograd's algorithm when $m \leq n^{1.68}$. These algorithms are impractical in general because of big constants hidden in the asymptotic notation. A more practical one, by Amossen and Pagh [6], has output-sensitive time complexity $O(n^{2/3}z^{2/3} + n^{0.862}z^{0.408})$, where z is the number of 1s in the output matrix. In our baseline, we implement the algorithm of Schoor [69], which seems to be the most practical one. It takes $O(ab/n)$ time on average if the 1s are uniformly distributed, using $O(a + b)$ space to represent the matrices. It intersects the nonempty columns of A with the nonempty rows of B, and adds to the result the Cartesian product of all the cells in the matching columns and rows.

Regarding the transitive closure A^+ of a Boolean matrix A (again, with a non-zero entries), a classic result by Warshall [77] achieves $O(n^3)$ time, just like a naive matrix multiplication. Although $A^+ = A + A^2 + \cdots + A^n$, Furman [43] showed that only $O(\log n)$ steps of the following process are needed. First, define $A_1 = A$, and then $A_{2k} = A_k + A_k^2$, for $k = 1, 2, \ldots, \lceil \log_2 n \rceil$. By embedding the Boolean matrix into a ring, one can then achieve time $O(n^{\omega} \log n)$, where $\omega \geq 2$ is the smallest value for which there exists a $O(n^{\omega})$-time matrix multiplication algorithm. Munro [58] and Fischer and Meyer [40] showed that Boolean matrix multiplication and transitive closure have essentially the same complexity, meaning that only one matrix multiplication is enough to compute the transitive closure. Hence, all running times we gave for matrix multiplication are also valid for transitive closure.

A key idea for sparse matrices, which we implement in our baseline, is to detect the strongly connected components (scc) of the graph represented by the matrix, which can be done in $O(a)$ time [3, 70, 72, 35]. Every node can reach every other within each component, and the graph of the components (where we collapse all the vertices of each component into one) is acyclic, so reachability is easily computed on it. Purdom [64] introduced such an algorithm based on computing the scc, which runs in $O(a + \mu n)$ time, where μ is the number of scc. Munro's algorithm [58] also computes the scc, yet it uses matrix multiplication to compute the transitive closure on the scc adjacency matrix. Nuutila [61] introduces an improved algorithm based on the same approach, which has good practical performance. Penn [63] introduces a sparse-matrix representation called Zero-Counting by Quadrants (ZCQ) and then shows how to use it to carry out matrix multiplication to compute the transitive closure, as in Munro's algorithm. The approach is shown to be competitive in practice [63]. The particular matrix multiplication algorithm used by Penn mimics the one that we will later see in Eq. (3), and we use it as inspiration to develop a novel transitive closure algorithm on k^2-trees.

Regarding its application to database management systems, several practical ideas have been proposed, such as the least-fixed point approach by Aho and Ullman [2] (and further improvements, see the excellent description by Nuutila [62, Ch. 2]), graph traversals [80, 73], and hybrid approaches [49] mixing several of the above approaches. Amossen and Pagh [6] use Boolean matrix multiplication to efficiently handle join-project queries, outperforming classical approaches in most cases.

2.5 K^2-trees

A k^2-tree [27] is a data structure able to space-efficiently represent binary relations, point grids, and graphs. For $k = 2$ it is, indeed, a compact representation of MX-Quadtrees [68, Sec. 1.4.2.1], a variant of classical Quadtrees [57]. We will use it in this paper with $k = 2$ to represent Boolean matrices, as follows. Let A be a $v \times v$ Boolean matrix, assuming v is a power of $k = 2$.[1] 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, or else when the submatrix is a single cell. Each node in this tree has $k^2 = 4$ children. Figure 2 shows the k^2-tree representation of a sample Boolean matrix. Nodes representing a non-empty submatrix are marked with a `1`, otherwise the mark is a `0`. 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 $\log_4 v^2 = \log_2 v$, and the leaves list the 1s of A in a left-to-right order imposed by the z-order. Concretely, the positions $A[i, j] = 1$ are listed by increasing value of $\mathsf{z\text{-}order}(i, j)$, which is defined as follows: since i and j are integers of $\log_2 v$ bits each, $\mathsf{z\text{-}order}(i, j)$ is the $(2\log_2 v)$-bit integer number obtained by interleaving the bits of the binary encodings of i and j.

To represent this tree space-efficiently, we traverse it 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 bitvector $L[1..4n]$, where n is the number of internal nodes in the tree. Each tree node is represented by the position of the first bit of its signature. Given a node i, its j-th child $(1 \leq j \leq 4)$ 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 [28, 59]. Figure 2 (below) illustrates the bitvector representation of a k^2-tree. For clarity, the three levels of bitvector L are shown separately; the actual representation is the concatenation of these bitvectors.

The k^2-tree representation is especially useful for sparse matrices. Let matrix A have a 1s. Then, in the worst case every 1 induces a node (i.e., a 4-bit signature) in every level of the k^2-tree, for a total of $4a\log_2 v$ bits. Not all those induced nodes can be different, however: 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 [22].

We note that constant-time `rank` is possible in the so-called transdichotomous word RAM model of computation, where we assume that the computer word holds $\Theta(\log v)$ bits in order to represent $\log_2 v$-bit coordinates in $O(1)$ words (and thus handle them in $O(1)$ time [42]). We assume this computation model as well.

Other compact graph representations. The worst-case entropy of directed graphs is high enough to make standard representations using adjacency arrays close to optimal in the worst case [60]; therefore representations like CSR and CSC [67, Sec. 3.4], which list the nonempty cells in row-wise or column-wise order, are acceptable for sparse graphs (our baseline representation builds on those). Different representations have been proposed to exploit further regularities the graphs may exhibit in applications like Web graphs and social networks. For example, while k^2-trees exploit clustering as explained; others exploit locality [9], similarity of adjacency lists [23], bicliques [44], and many others.

3 Evaluating RPQs using 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 $\mathscr{M}$ to translate 2RPQs into matrix operations, following Losemann and Martens' work [52]. We start with the base cases:

- $\mathscr{M}(\varepsilon) = I$, the identity matrix.
- $\mathscr{M}(p) = M_p$, for $p \in P$.
- $\mathscr{M}(\hat{\ }p) = M_p^T$, for $p \in P$.

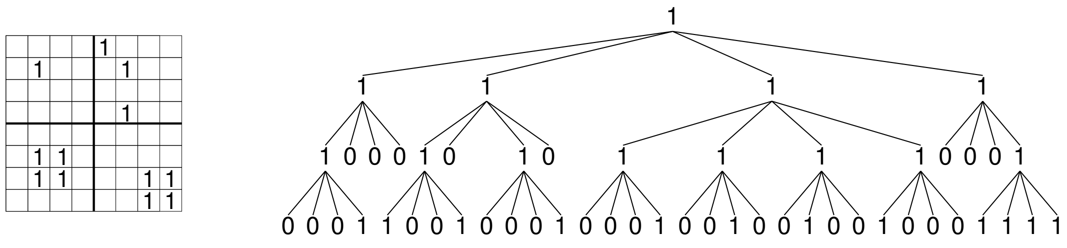
Next, let E_1 and E_2 be 2RPQs. We define the following recursive rules:

- $\mathscr{M}(E_1 \mid E_2) = \mathscr{M}(E_1) + \mathscr{M}(E_2)$
- $\mathscr{M}(E_1/E_2) = \mathscr{M}(E_1) \times \mathscr{M}(E_2)$
- $\mathscr{M}(E_1^+) = \mathscr{M}(E_1)^+$
- $\mathscr{M}(E_1^*) = I + \mathscr{M}(E_1)^+$, where I is the corresponding identity matrix.

Then, given a 2RPQ $R = (x, E, y)$, we extend $\mathscr{M}$ to evaluate it as follows:

1. If x and y are both variables, $\mathscr{M}(R) = \mathscr{M}(E)$

[1] 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.



```
1111

1000   1010   1111   0001

0001   1001   0001   0001   0010   0100   1000   1111
```

Fig. 2 A Boolean matrix (above, left), its corresponding k^2-tree representation (above, right), and the corresponding level-wise bitvector representation of the tree (below).

2. If x is a variable and y is a constant, $\mathscr{M}(R) = \mathscr{M}(E)\langle y\rangle$
3. If x is a constant and y is a variable, $\mathscr{M}(R) = \langle x\rangle\mathscr{M}(E)$
4. If x and y are both constant, $\mathscr{M}(R) = \langle x\rangle\mathscr{M}(E)\langle y\rangle$

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 $\log_2|V|$ levels, and each 1 in M_p induces at most $\log_2|V|$ 1s in its k^2-tree representation. As explained, per this representation we will assume $|V|$ is a power of 2. We will also use $v = |V|$, as well as $a = |A|$ and $b = |B|$ for the number of 1s in matrices A and B.

We implement k^2-trees, and thus bitvectors with rank support, in C. We store the bitvector as consecutive bits packed in a 64-bit-words array. To support rank we define a parameter s that allows us to trade space for time (and whose value we fix to $s = 4$ in our implementation). We then store the cumulative sum of 1s up to every sth cell. 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 make use of the primitive *popcount*, which counts the number of 1s in a word. This primitive can be implemented in constant time on 64-bit words [51, Sec. 7.1.3]. 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. As a result, we use $n/1024 + n/(4s)$ additional bits of space for storing a bitvector $B[1..n]$, and compute rank in time $O(s)$.

In the sequel we describe how the different operations of the Boolean algebra are implemented on this representation. Transpositions are described immediately because they are incorporated to the data structure rather than executed as an operation; later we describe how the operations handle matrices marked as transposed. For every operation we also consider two aspects: (1) how to incorporate parallelism and (2) how to handle restrictions.

Parallelism will be implemented with multithreading, but for simplicity it will be analyzed in the PRAM model of computation, assuming that the 1s are uniformly distributed on the matrices.[2] We will also assume that the number p of processors is small compared to the number of the 1s and to the side of the matrices, which is realistic in multicore architectures.

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.

We show next how we extend k^2-trees to implement transpositions. Then, in Sections 5 through 7 we implement and analyze the main operations, namely sum (and relatives), multiplication, and transitive closure, respectively, on the extended format.

Table 1 shows the simplified time complexities we will obtain, and compares them with those of the baseline we describe in Section 9 (which uses considerably more space). As explained in the caption, the baseline time complexities are always smaller (or equal).

[2] The actual multithreading adapts better to nonuniform distributions than our analysis under the PRAM model.

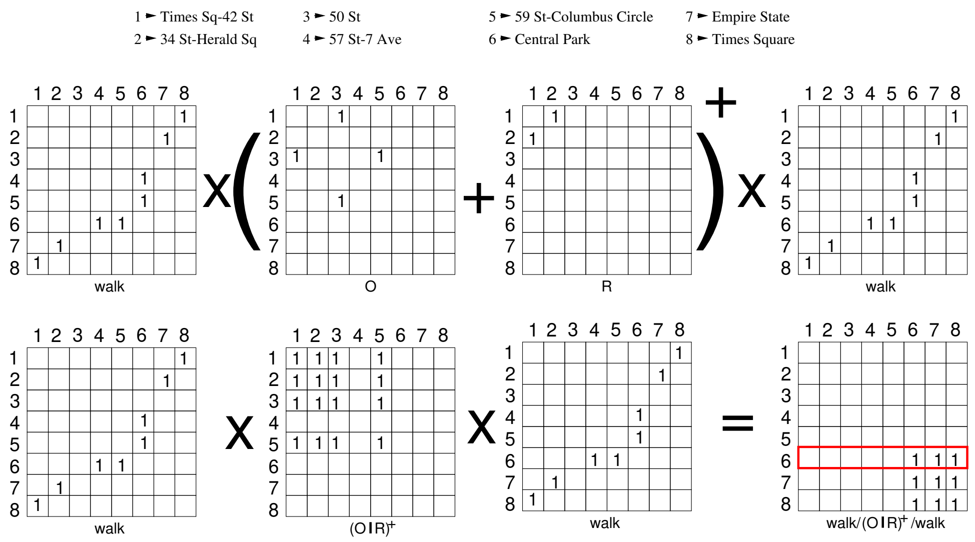


Fig. 3 Example of transforming the 2RPQ $R = (\mathsf{Central\ Park}, \mathsf{walk}/(\mathsf{O}|\mathsf{R})^+/\mathsf{walk}, ?\mathsf{y})$ into matrix operations. For readability we depict each node (station or point of interest) as an integer. On top, we can see the matrices and operations used to solve $E = \mathsf{walk}/(\mathsf{O}|\mathsf{R})^+/\mathsf{walk}$. Below, the second matrix represents $\mathcal{M}((\mathsf{O}|\mathsf{R})^+)$, which was obtained by computing the transitive closure on the sum of $\mathcal{M}(\mathsf{O})$ and $\mathcal{M}(\mathsf{R})$. The fourth matrix below is $\mathcal{M}(E)$. In order to solve R, as x is the node Central Park, we restrict to the sixth row of $\mathcal{M}(E)$ and the solutions are Central Park (6), Empire State (7) and Times Square (8).

Table 1 Simplified average time complexities for the main operations on $v \times v$ matrices with m uniformly distributed 1s. The transitive closure assumes the result has m^+ 1s. The PRAM time uses $p \le m$ processors. Note that, because $m \le v^2$, the baseline complexities are always smaller (or equal) than those of the k^2-tree.

Operation	Baseline time	k^2-tree time	k^2-tree parallel time
Transposition	1	1	1
Sum and relatives	m	$m \log v$	$(1/p)m \log v + m + \log v \log p$
Multiplication	$(m^2/v)\log v$	$m^{3/2}\log v$	$(1/p)m^{3/2}\log v + m^2/v + \log v \log p$
Transitive closure	$(m^+)\log v$	$(m^+)^{3/2}\log v$	$(1/p)(m^+)^{3/2}\log v + (m^+)^2/v + \log v^2 \log p$

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

$$A^T = \left(\begin{array}{c:c} A_0^T & A_2^T \\ \hdashline A_1^T & A_3^T \end{array}\right).$$

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.

5 Boolean Sum and Relatives

In this section we address the set-like Boolean operations, with special emphasis on the Boolean sum (or disjunction) for its impact on later operations like multiplication. We start from known techniques [65], and then improve the algorithms and adapt them to handle transpositions, parallelism, and restrictions. We also provide improved time complexity analyses.

If neither A or B is transposed, we can compute the sum $A+B$ with a simple sequential pass over both k^2-tree bitvectors [65], merging 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 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 on both root nodes, 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 pointed by the read-pointer (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 pointed by the read-pointers 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 any task. We do not append new tasks when the corresponding nodes are k^2-tree leaves. The process finishes when the queue becomes empty. Figure 4 illustrates the algorithm.

The total time is proportional to the sum of the number of nodes of both k^2-trees, $O(a\log(v^2/a)+b\log(v^2/b)) \subseteq O((a+b)\log v)$. We introduce a speedup that does not change the complexity but has a significant impact in practice: we do not append consecutive copy tasks for A or for B in the queue, but rather merge them into a single task that copies several signatures together, using a constant number of operations on computer words.

5.1 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. Instead, we handle the sum as any other set-like operation, see next.

5.2 Set-like operations

Several other operations of the Boolean algebra have the same structure of the sum $A+B=A\cup B$ (i.e., Boolean "or" of the 1s): intersection $A\cap B$ (Boolean "and" of the 1s), difference $A-B$ (Boolean "and not" of the 1s), and symmetric difference $A\oplus B$ (Boolean "exclusive or" of the 1s). In general, those cannot be solved with the merge-like algorithm we described for the sum because they lack the key property that the signature of the resulting k^2-tree root is a function of the signatures of the k^2-tree roots of the operands (in the case of the sum, it is the bitwise-or of the signatures). Further, they may require skipping large submatrices of the operands. Instead, we must first operate the submatrices and only then define the signature of the result based on which are nonempty. We then resort to a recursive algorithm of the form

$$A\circ B=\left(\begin{array}{c:c} A_0\circ B_0 & A_1\circ B_1 \\ \hdashline A_2\circ B_2 & A_3\circ B_3 \end{array}\right), \qquad (1)$$

where $A=\binom{A_0\ A_1}{A_2\ A_3}$ and $B=\binom{B_0\ B_1}{B_2\ B_3}$ are the submatrices into which the k^2-tree representation splits A and B, and $\circ\in\{\cup,\cap,-,\oplus\}$. That is, we recursively operate the submatrices A_i and B_i, for $0\le i<4$, obtaining up to 4 submatrices $A_i\circ B_i$ represented as k^2-trees. Instead of producing the k^2-trees and later concatenate them levelwise, we prepare the memory space for the output separated by levels, so that the recursive calls directly append their results in the corresponding levels [65]. This works because a recursive tree traversal corresponds to a left-to-right traversal within each level. Figure 5 illustrates the algorithm for the sum (or union), to allow contrasting with the previous algorithm in Figure 4.

An important improvement we make on top of the basic recursive algorithms [65] is that, when one of the two arguments is an empty submatrix, we may have to copy the other argument to the output. This occurs for both A and B in $A\cup B$ and $A\oplus B$, and for A in $A-B$. Instead of carrying out this copy node by node of the k^2-trees, we perform a levelwise copy. In this copy we work $O(1)$ time per computed word copied, which in the transdichotomous RAM model of computation stores $\Theta(\log v)$ nodes of the k^2-tree. For example, copying a whole k^2-tree of a leaves (and $O(a\log v)$ nodes) takes time $O(a+\log v)$, not $O(a\log v)$ (the second additive term stands for the $O(1)$-time overheads at each level). The impact of this improvement is made clear soon in the analysis and later in the experimental results.

Transpositions. When solving operations in this way, transpositions are handled easily by exchanging the meaning of M_1 and M_2 in every node of the k^2-tree bitvector, if $M=\binom{M_0\ M_1}{M_2\ M_3}$ is transposed. As explained before, we use this technique for the sum when one matrix is transposed and the other is not. Otherwise, both the merge-like and the recursive algorithm can be used. The next analysis and later the experiments shed light on which algorithm is to be preferred depending on the case.

Analysis. Just as for the merge-like algorithm for $A+B$, the time complexity of the recursive algorithms is in $O(a\log(v^2/a)+b\log(v^2/b))\subseteq O((a+b)\log v)$, for all the operations, as we work at most $O(1)$ time per node of the input and output k^2-trees. This analysis can be refined, however.

Let us start with the intersection, $C=A\cap B$. A first refinement is that there are at most $\min(a,b)$ elements carried to the output, not $a+b$, thus its time complexity is in $O(\min(a,b)\log v)$ because the algorithm traverses only the k^2-tree nodes below which both A and B have leaves. We can prove even more refined adaptive bounds by relating this problem to the adaptive intersection of integer sets [19, 32], in particular with the trie approach by Arroyuelo and Castillo [11]. Let $c=|C|\le\min(a,b)$ be the output

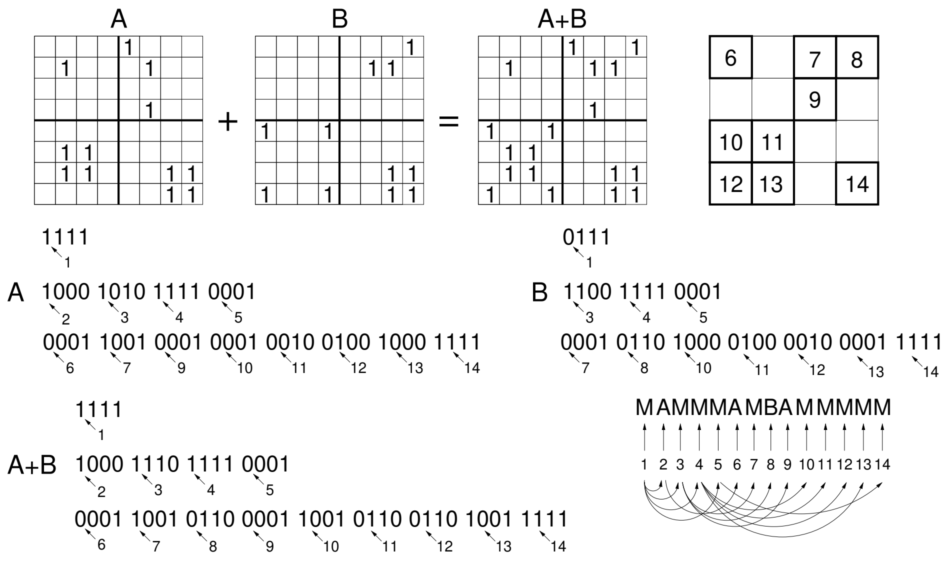


Fig. 4 Our sequential algorithm when summing matrices A and B shown on the top left. The k^2-tree representations of A, B, and $A+B$ are shown below the matrices, artificially separating the three levels in three lines. On the bottom left, the elements inserted in the queue along time, using M for "merge" and A/B for "copy A/B". The algorithm runs along 14 steps, one per element in the queue. Each such number in the queue has forward arcs towards the elements its step inserts; for example step 1, which corresponds to the whole matrix, inserts the elements 2 to 5 corresponding to the four quadrants. The scheme on the top right shows to which 2×2 submatrices do the steps 6–14 correspond. The diagonal arrows in the k^2-tree representations of A and B show the position of the read-pointers at each step, and those on $A+B$ show the corresponding write-pointers. For example, in step 1, we read 1111 from A and 0111 from B, thus we insert a "copy A" and three "merge" elements in the queue. Step 2 reads "copy A" from the queue and thus copies 1000 from the sequence of A and appends it to that of $A+B$. Step 3 reads "merge" and thus merges 1010 from A, 1100 from B, and writes 1110 to $A+B$. And so on.

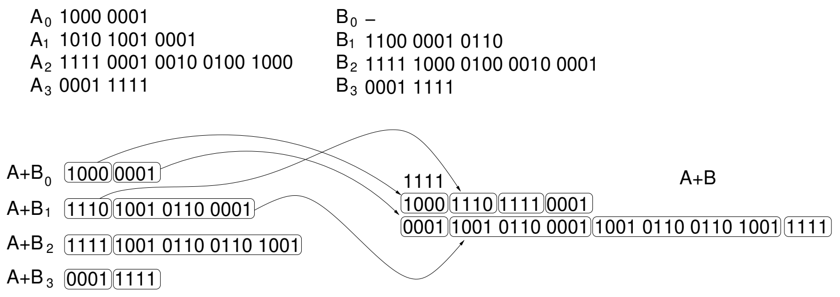


Fig. 5 The recursive algorithm to solve the same sum $A+B$ as in Figure 4. We set up the output space (bottom right of the figure) and then the k^2-tree bitvectors resulting from each quadrant are written directly to the corresponding levels as they are generated. The arrows show how this works for the quadrants 0 and 1. Because no quadrant is empty, the signature of the root of $A+B$ is 1111.

size. We define conceptual integer sets S_A and S_B, which represent the set of positions with 1s in the matrices A and B, respectively. Therefore, the set $S_A \cap S_B$ represents the set of 1s in the matrix $A \cap B$. The sets S_A and S_B are subsets of the universe $[0..v^2)$, concretely $S_A = \{\text{z-order}(i,j) \mid A[i,j] = 1\}$, and analogously for S_B (recall the definition of $\text{z-order}(i,j)$ from Section 2.5). As a consequence, the order of the elements in S_A corresponds to a left-to-right traversal of the corresponding k^2-tree leaves, and therefore the algorithm we propose to compute $A \cap B$ mimics the set intersection algorithm proposed by Arroyuelo and Castillo [11]. The only difference is that this time the sets are represented as 4-ary tries (i.e., k^2-trees) rather than binary tries, but the properties needed to prove their adaptive bound still hold. Then, the time for computing $A \cap B$ is $O(\delta \log(v^2/\delta))$, where δ is the *alternation measure* of S_A and S_B defined by Barbay and Kenyon [19], which in particular satisfies $c \leq \delta \leq \min(a,b)$. To this time, which measures the number of k^2-tree nodes traversed, we should add the $O(c \log(v^2/c)) \subseteq O(c \log v)$ cost to copy the nodes in the paths toward the c resulting points, but this is subsumed by $O(\delta \log(v^2/\delta)) \subseteq O(\delta \log v)$.

Measure δ enables a finer analysis of the intersection time. It measures the number of times we *need* to switch between S_A and S_B in order to collect all the $S_A \cup S_B$ integers, along a left-to-right traversal of both ordered sequences. For example, if all the a 1s of A are in the first quadrant, and all the b 1s of B are in the third quadrant, then their corresponding measure is $\delta = O(1)$ because all the values in S_A precede those in S_B, and thus we need $O(1)$ switches to collect them all. This yields an upper bound of $O(\log v)$ for our intersection. In this case the bound is pessimistic because our algorithm actually runs in $O(1)$ time. It would be tight if the two clusters would split only in the same subgrid of depth $\Theta(\log v)$ of A and B.

For the set difference, $C = A - B$, we note that $c \leq a$, thus the time is in $O(a\log(v^2/a)) \subseteq O(a\log v)$. Further, since $A - B = A \cap \overline{B}$, we can reuse the analysis of the intersection to obtain a finer measure. Let $|\overline{B}| = \overline{b} = v^2 - b$ denote the number of elements in $\overline{B}$. The analysis proceeds as before, obtaining time $O(\overline{\delta}\log(v^2/\overline{\delta}))$, where $\overline{\delta}$ is the alternation measure of S_A and $S_{\overline{B}}$, which satisfies $c \leq \overline{\delta} \leq \min\{a, \overline{b}\}$.

Thanks to our improved algorithm to copy whole submatrices, we can also use the alternation measure to refine the time complexity $O((a+b)\log v)$ of the union and symmetric difference. Returning to our example where all the 1s of A are in the first quadrant and all those of B are in the third, our algorithms run in time $O(a+b+\log v)$. In general, all the s integers of S_A and S_B that lie between two consecutive switches between S_A and S_B can be copied computer-word-wise, in time $O(s/\log v + 1)$. As we have to traverse $O(\delta\log(v^2/\delta))$ nodes and to copy $O(c\log(v^2/c))$ nodes (where this time δ can be smaller than $c \leq a+b$), we have a total time of $O(\delta\log(v^2/\delta) + c\log(v^2/c)/\log v) \subseteq O(a+b+\delta\log v)$ for the whole process.

We expect our refined analysis to show up in practice when the matrix densities are very different or they distribute non-uniformly.

5.3 Parallelism

Our basic merge-based summation algorithm described at the beginning of the section is difficult to parallelize, because we do not know where to start copying each of the summands at each level. The standard parallel merging algorithms, which first use a parallel-prefix scheme to find the cumulative sums of the arrays to merge and then copy each array independently, cannot be used here because our merge is not disjoint: there are some 1s at the same positions in A and B. In order to determine where to copy the next subtrees, we must actually merge the current ones, and thus must proceed in sequence.

The generic recursive algorithm for operation $\circ$ of Eq. (1) is more amenable to parallelization. We can proceed in parallel for each $A_i \circ B_i$, this time writing each result as an independent k^2-tree, and then concatenate them sequentially at the end. Writing the results directly to the output is not possible in the parallel context because, again, we do not know in advance where to write. We speed up the concatenations by maintaining, for each k^2-tree bitvector, $O(\log v)$ counters of the sizes of the k^2-tree levels. Figure 6 illustrates the algorithm we parallelize.

For all the operators, when one of the submatrices is zero, the result is either zero or the other submatrix, which helps speed up the computation. In particular, we share the submatrix in the second case instead of generating a new copy of it. We also do not need to build the rank data structures until the end of the whole operation, because the concatenation operates sequentially over the matrices.

We implement a multithreaded version of this algorithm, where the recursive calls keep opening new threads for a few levels to avoid saturating the system with many more processes than processors.

Analysis. Let us first analyze the sequential cost of this algorithm. In general, the concatenation time is proportional to the number of nodes of the operated k^2-trees, which as explained add up to $O((a+b)\log v)$. However, we do not concatenate the bitvectors bit by bit, but rather by whole computer words. Because of bit alignment issues, copying a computer word requires up to two read and two write operations, which in the transdichotomous RAM model contains $\Theta(\log v)$ bits. As a result, the total copying time is $O(a+b+\log v)$, the last term accounting again for a constant additive penalty per k^2-tree level copied.

Consider now the whole operation time. In the worst case, there are $a+b$ points across all submatrices in each level of the recurrence, thus the $a+b$ term above adds up to $O((a+b)\log v)$ across the $O(\log v)$ levels of the recursion. The terms $\log v$, instead, add up to $O((a+b)\log^2 v)$ in the worst case, in which each of the $a+b$ points are isolated and copied individually across all the levels.

In order to avoid the additive penalties leading the cost, we will use a special concatenation method that is $O(a)$ time when we merge one submatrix of a points with other three empty submatrices. Note that the resulting k^2-tree is identical to that of the nonempty submatrix preceded with a signature for the new root. We first write the new root signature, which marks the nonempty submatrix, and then concatenate the bitvector of that submatrix. The submatrix has $O(a\log v)$ bits but, as explained, we copy it by chunks of $\Theta(\log v)$ bits, so the total time is $O(a)$. Further, the $O(\log v)$ level counters of the nonempty submatrix are not copied, but shared and extended with one further entry. We call this a *trivial concatenation*, and say that it poses an additive penalty of $O(1)$ (as

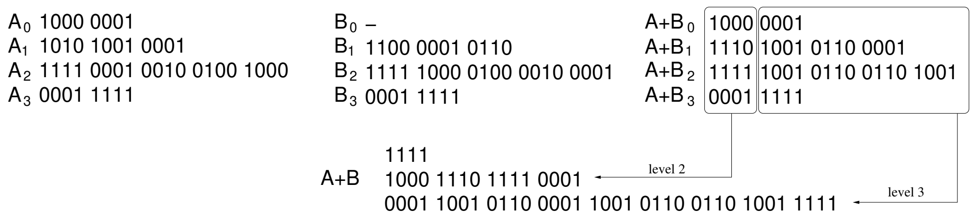


Fig. 6 The parallelizable recursive algorithm to solve the same sum $A+B$ as in Figure 4. We show the k^2-tree bitvectors of all the intervening quadrants as single sequences, and that of the result with one level per line. The recursive calls were called in parallel and have already produced the four quadrants of the result, $(A+B)_j$ for $0 \leq j < 4$. Because the four are nonempty, the signature of the root of $A+B$ is 1111. Now we copy the first levels of $(A+B)_j$ into the second level of the result, 1000 1110 1111 0001. Finally, we copy the second levels of $(A+B)_j$ into the third level of the result.

opposed to the $O(\log v)$ penalty posed by the nontrivial concatenations).

As said, we can have $O((a+b)\log v)$ concatenations along the process, but only $O(a+b)$ of those are nontrivial (those can be regarded as the branching nodes of a 4-ary tree with $a+b$ leaves). Nontrivial concatenations pose the $O(\log v)$ additive penalty, but that of trivial ones is only $O(1)$. Overall, the total time spent on additive penalties is $O((a+b)\log v)$, and this is also the total time complexity of the operations.

We now give a PRAM-based analysis assuming that $p \leq \min(a,b)$ and that the a 1s in A and the b 1s in B distribute uniformly. Let $S(v^2, a+b, p)$ be the parallel time on $v \times v$ matrices, $a+b$ points in total, and p processors. Assuming we assign $p/4$ processors to build each submatrix $A_i \circ B_i$, the recurrence for the parallel time is

$$S(v^2, a+b, p) = S(v^2/4, (a+b)/4, p/4) + (a+b+\log_4(v^2)),$$

where the second term (using convenient constants) stands for the cost to sequentially concatenate the 4 resulting submatrices. The recursion continues up to the level $\ell' = \log_4 p$, where $4^{\ell'} \geq p$. At this level, the a/p and b/p points in each pair of submatrices are sequentially merged, in time $S(v^2/p, (a+b)/p, 1) = O(\frac{1}{p}(a+b)\log v)$. The whole recurrence then solves as follows:[3]

$$S(v^2, a+b, p) = \qquad (2)$$
$$S(v^2/p, (a+b)/p, 1) + \sum_{\ell=0}^{\log_4 p - 1} \left(\frac{a+b}{4^\ell} + \log_4 \frac{v^2}{4^\ell} \right)$$
$$= O\left(\frac{1}{p}(a+b)\log v + (a+b) + \log v \log p \right).$$

Although the analysis is simplified, it suggests we can expect a nearly perfect speedup, at least for small enough $p = O(\log v)$, and disregarding the small $O(\log v \log p)$ additive penalty.

[3] Note that all the concatenations up to level $\log_4 p$ are nontrivial, per our assumptions on the number of points in the submatrices.

5.4 Restrictions

For $\langle r\rangle(A+B)\langle c\rangle$ (where it may be that only $\langle r\rangle$ or only $\langle c\rangle$ are present), 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 $\langle r\rangle A\langle c\rangle + \langle r\rangle B\langle c\rangle$. The merge-like algorithm for the sum cannot be used, however, because just as it happened for the generic operations $\circ$, we do not know beforehand whether a submatrix (or the merge of two) will be nonempty after restricting it to some row/column, even if it intersects the row/column.

We then implement all the restricted operations, including the sum, using the recursive algorithms. The only difference is that, when the row and/or column are restricted, only two of the four submatrices will be nonempty, and when both are restricted, only one submatrix will be nonempty.

6 Boolean Multiplication

For the multiplication $A \times B$ we use the following classic divide-and-conquer recursive procedure. Letting $A = \binom{A_0\ A_1}{A_2\ A_3}$ and $B = \binom{B_0\ B_1}{B_2\ B_3}$ as before, we recursively compute 8 products of those submatrices in order to produce

$$A \times B = \qquad (3)$$
$$\left(\begin{array}{c:c} A_0 \times B_0 + A_1 \times B_2 & A_0 \times B_1 + A_1 \times B_3 \\ \hdashline A_2 \times B_0 + A_3 \times B_2 & A_2 \times B_1 + A_3 \times B_3 \end{array} \right).$$

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, as explained.

Once the k^2-tree bitvectors of the four submatrices are recursively obtained, we concatenate them levelwise, as for our improved recursive sum operation. There is no need to build the `rank` data structures for this concatenation because it proceeds left-to-right in each level. We also maintain the $O(\log v)$ level counters in each k^2-tree to speed up concatenations.

Transpositions are again handled by exchanging the meaning of M_1 and M_2 in every node of the transposed matrices $M = \binom{M_0\ M_1}{M_2\ M_3}$.

6.1 Complexity

One part of the multiplication cost is given by the number of recursive calls. We distinguish three stages to analyze a scenario with sparse matrices.

1. In the first stage, all the submatrices are nonempty. Since there are 4^ℓ submatrices in level ℓ, 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, up to level $\ell_1 = \log_4 \min(a,b)$. In this stage the cost follows the recurrence $T(v^2) = 8 \cdot T(v^2/4)$, therefore the cost up to level ℓ_1 is $8^{\ell_1} = \min(a,b)^{3/2}$.
2. In the second stage, the worst case is that the emptier matrix has only one point in its submatrices while the fuller has $\max(a,b)/\min(a,b)$ evenly distributed points. This continues for $\ell_2 = \log_4 \frac{\max(a,b)}{\min(a,b)}$ further levels, in which 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. From each of the 8^{ℓ_1} submatrices where stage 1 ends, we have then a cost of $2^{\ell_2} = \sqrt{\max(a,b)/\min(a,b)}$, which multiplied by 8^{ℓ_1} yields the cost $\min(a,b)\sqrt{\max(a,b)}$ up to the end of stage 2.
3. In the third stage, we have just one point in each of the submatrices, so the cost is $\log_2 v - \ell_1 - \ell_2 = \log_4(v^2/\max(a,b))$ to track a single point along both submatrices. This is done from each of the $8^{\ell_1}2^{\ell_2} = \min(a,b)\sqrt{\max(a,b)}$ submatrices where stage 2 ends, leading to the final cost
$$O(\min(a,b)\sqrt{\max(a,b)}\log(v^2/\max(a,b))). \quad (4)$$

The second part of the multiplication cost is that of summing pairs of partial submatrices, recall Eq. (3). In the worst case, those matrices may add up to $a \cdot b$ points at across every level of the recursion. Just as in Section 5.2, where we had $a+b$ points in every level, the total merging cost of the partial results is $O(ab\log^2 v)$. With the technique of the trivial concatenations, this can be reduced to $O(ab\log v)$, where we pay $O(\log v)$ time only on the branching nodes of the resulting k^2-tree, which has ab leaves at most.

Note that this term dominates the cost of the first part. We can show that the average time, on matrices with uniformly distributed 1s, is better. 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 $b/4^\ell$ 1s in B. Every such $a_{ik} = 1$ will pair with every such $b_{k'j} = 1$ iff $k = k'$, which occurs with probability $1/(v/2^\ell)$, so on average there will be $8^\ell(a/4^\ell)(b/4^\ell)(2^\ell/v) = ab/v$. This leads to a total average time of $O((ab/v)\log v)$ for the second part. Since $ab/v = \min(a,b)\max(a,b)/v \leq \min(a,b)\sqrt{\max(a,b)}$ because $\max(a,b) \leq v^2$, the bound $O(\min(a,b)\sqrt{\max(a,b)}\log v)$ of the first part of the cost dominates on the average.

6.2 Parallelism

A further advantage of Eq. (3) is that it is easily parallelized, as it features 8 independent multiplications and 4 sums, each sum depending only on the result of two multiplications. A multithreaded version assigns a thread to each of the 8 multiplications and to each of the 4 sums, forcing sequential execution of each sum after its two corresponding multiplications. The recursive calls are further parallelized for a few levels to avoid having many more processes than processors, as explained.

We give a PRAM-based analysis of this process. Let $T(v^2, p)$ be the time of the algorithm on a $v \times v$ matrix and p processors, where we assume that $p \leq \min(a,b)$ and that the 1s distribute uniformly across the matrices. This implies that, with respect to the number of recursive calls, the parallelism is confined inside the stage 1 of the analysis in Section 6.1.

We allocate $p/8$ processors to each of the 8 multiplications. Then we will have more than one available processor per recursive call up to level $\ell_0 = \log_8 p$. From that level, each of the $8^{\ell_0} = p$ parallel calls start to run sequentially, for $\ell_1 - \ell_0$ further levels. Each processor then runs $\ell_1 - \ell_0$ levels of stage 1, then ℓ_2 levels of the stage 2, and then the rest of the levels of stage 3. The total time spent by each processor is then $8^{\ell_1-\ell_0}2^{\ell_2}\log(v^2/\max(a,b))$, which is exactly the sequential cost of the recursive calls (Eq. (4)) divided by $8^{\ell_0} = p$. In addition, we have the $O(\ell_0) = O(\log p)$ time spent in the first ℓ_0 levels.

To anayze the 4 sums we can use Eq. (2), replacing $a+b$ by ab/v. The total parallel time is then

$$T(v^2,p) = \qquad (5)$$
$$O\left(\frac{1}{p}\min(a,b)\sqrt{\max(a,b)}\log v + \frac{1}{p}\frac{ab}{v}\log v + \frac{ab}{v} + \log v \log p\right) =$$
$$O\left(\frac{1}{p}\min(a,b)\sqrt{\max(a,b)}\log v + \frac{ab}{v} + \log v \log p\right)$$

which, compared to the sequential time, again suggests we can expect a nearly perfect speedup in our multithreaded implementation. If $p = O((v\log v)/\sqrt{\max(a,b)})$, in particular, the speedup is perfect except for the small additive term $O(\log v \log p)$.

6.3 Restrictions

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.

Having one restriction (row or column) ensures that at most 6 or the 8 multiplications in Eq. (3) are nonzero, thereby modifying the recurrence of the number of multiplications to $T(v^2) = 6 \cdot T(v^2/4)$, which solves to $T(v^2) = O((\min(a,b)^{\log_4 6})$. Multiplied by the $2^{\ell_2}\log_4(v^2/\max(a,b))$ cost of stages 2 and 3 (where the restrictions yield no better upper bounds) we obtain the final bound, $O(\min(a,b)^{\log_4 3}\sqrt{\max(a,b)}\log v)$; $\log_4 3 < 0.8$.

For the sums, we note that on average only one out of v of the a 1s in A or the b 1s in B satisfy the row or column restriction, so the average number of points to sum per level is ab/v^2. The time of the second part then becomes $O((ab/v^2)\log v)$.

Having both row and column restrictions yields $T(v^2) = 4 \cdot T(v^2/4)$, which solves to $T(v^2) = O(\min(a,b))$ and to $O(\sqrt{ab}\log v)$ for the three stages, plus just $O(\log^2 v)$ time for the sums.

7 Transitive Closure

A simple positive transitive closure algorithm obtains A^+ by iteratively computing $A \leftarrow A + A \times A$ until no change occurs in A [43]. This occurs at most after $\log_2 v$ iterations, so the time complexity is $O(\log v)$ times that of multiplying A by itself. The reflexive-transitive closure is computed as $A^* = I + A^+$, where I is the identity matrix. Transposed matrices can be operated as is and the result would be transposed as well.

Since the number a of 1s in A grows in every iteration until reaching $a^+ = |A^+|$, we can use Eq. (4) with $a = b = a^+$ to obtain an average time of $O((a^+)^{3/2}\log^2 v)$ along the $O(\log v)$ matrix multiplications, assuming a uniform distribution of the 1s. The non-positive closure adds $O((a^+ + v)\log v)$ further time for the final sum.

We now introduce a more efficient algorithm, which obtains A^+ at the cost of $O(1)$ multiplications. Inspired by Warshall's algorithm [77] (and, in a way, in the ZCQ decomposition [63]), we compute A^+ in two steps. Let $A = \binom{A_0\ A_1}{A_2\ A_3}$ be a $v \times v$ matrix. In the first step we obtain the matrix A', where $a'_{i,j} = 1$ iff we can go from node i to node j through a single edge or using only intermediate nodes in $[1\,..\,v/2]$. This is computed as

$$A' = \left(\begin{array}{c:c} A'_0 & A'_1 \\ \hdashline A'_2 & A'_3 \end{array}\right) = \qquad (6)$$
$$\left(\begin{array}{c:c} A_0^+ & A_1 + A'_0 \times A_1 \\ \hdashline A_2 + A_2 \times A'_0 & A_3 + A_2 \times A'_1 \end{array}\right).$$

(where A'_3 can also be computed as $A_3 + A'_2 \times A_1$). So we first compute $A'_0 = A_0^+$ recursively, then A'_1 and A'_2 (which depend on A'_0), and finally A'_3 (which depends on A'_1, or on A'_2 in its alternative formulation).

In the second step, we also permit the paths to go through nodes in $[v/2+1\,..\,v]$, thereby completing the closure. The resulting matrix is computed as

$$A^+ = A'' = \left(\begin{array}{c:c} A''_0 & A''_1 \\ \hdashline A''_2 & A''_3 \end{array}\right) = \qquad (7)$$
$$\left(\begin{array}{c:c} A'_0 + A'_1 \times A''_2 & A'_1 + A'_1 \times A''_3 \\ \hdashline A'_2 + A''_3 \times A'_2 & (A'_3)^+ \end{array}\right).$$

(where A''_0 can also be computed as $A'_0 + A''_1 \times A'_2$). This time we start by computing $A''_3 = (A'_3)^+$ recursively, then compute A''_2 and A''_1 (which depend on A''_3), and finally A''_0 (which depends on A''_2 or on A''_1).

7.1 Correctness

We call $X = [1\,..\,v/2]$ and $Y = [v/2+1\,..\,v]$ and use the notation $[M]_{i,j} = m_{i,j}$ for any matrix M. In A', we must show that $[A']_{i,j} = 1$ iff there exists a (nonempty) path from node i to node j whose sequence of intermediate nodes is in X^* (the Kleene closure of X). It is easily seen that $A'_0 = A_0^+$, because the allowed intermediate nodes are precisely all those in A_0. We inductively assume that $A'_0 = A_0^+$ is computed correctly.

The cells $[A']_{i,j}$ that fall in A'_1 satisfy that $i \in X$ and $j \in Y$. Every path from i to j whose intermediate nodes are in X corresponds to a path of zero or more edges starting at i and ending in some $k \in X$, plus a final edge from k to j. Therefore, either $[A]_{i,j} = 1$ (i.e., $k = i$), or $[A']_{i,k} = 1$ and $[A]_{k,j} = 1$. This is equivalent to $[A_1 + A'_0 \times A_1]_{i,j-n/2} = 1$. The case of A'_2 is analogous: any path from $i \in Y$ to $j \in X$ through a sequence of nodes in X^* starts with an edge from i to some $k \in X$ and follows with a path (of length zero or more) from k to j, thus it corresponds to $[A_2 + A_2 \times A'_0]_{i-n/2,j} = 1$.

Finally, a path from $i \in Y$ to $j \in Y$ that can have a sequence of intermediate nodes in X^* can be either a direct edge from i to j with no intermediate nodes (for which we must have $[A_3]_{i-n/2,j-n/2} = 1$), or it can be formed by an edge from i to some $k \in X$ followed by a path from $k \in X$ to $j \in Y$ using intermediate nodes in X (for which we must have $[A_2 \times A'_1]_{i-n/2,j-n/2} = 1$). Thus we obtain $A'_3 = A_3 + A_2 \times A'_1$. We can analogously derive the equivalent formula $A'_3 = A_3 + A'_2 \times A_1$.

For the second step, we start by computing $A''_3 = (A'_3)^+$ recursively, and inductively assume its computation is correct. This corresponds to a concatenation of paths that start and end in Y, going through zero or more nodes of X between each pair of nodes in Y, and where the final node in Y of each path is the initial node of the next. The intermediate nodes then form a sequence $x_1 y_1 x_2 y_2 \dots y_m x_{m+1}$, where $y_r \in Y$ and $x_r \in X^*$. The set of all those sequences is then $(X^*Y)^*X^*$. Therefore, $[(A'_3)^+]_{i-n/2,j-n/2} = 1$ whenever there is a path between i and j whose intermediate sequence of nodes is in $(X^*Y)^*X^*$. But then, note that $(X^*Y)^*X^* = (X|Y)^*$ is an easy to prove equality between regular languages. Thus, the path between i and j can use any number of intermediate nodes in $[1..n]$, and then $(A'_3)^+ = A''_3 = (A^+)_3$.

Let us now consider the computation of A''_2. It should hold $[A''_2]_{i-n/2,j} = 1$ iff there is a path from $i \in Y$ to $j \in X$ going through a sequence of zero or more intermediate nodes in X or Y. If there are no nodes of Y in such sequence, then it belongs to X^* and it must hold that $[A'_2]_{i-n/2,j} = 1$. Otherwise, let y be the last node belonging to Y in the sequence. The sequence is then of the form zyx, where $z \in (X|Y)^*$ and $x \in X^*$. Then it must hold that $[A''_3]_{i-n/2,y-n/2} = 1$ and $[A'_2]_{y-n/2,j} = 1$, and consequently $[A''_3 \times A'_2]_{i-n/2,j-n/2} = 1$. The formula $A''_2 = A'_2 + A''_3 \times A'_2$ is then proved. The case $A''_1 = A'_1 + A'_1 \times A''_3$ is analogous.

The final case, A''_0, is also analogous. It must hold that $[A''_0]_{i,j} = 1$ iff there is a path from i to j with intermediate nodes in X or Y. If this sequence has only nodes in X, then it belongs to X^* and it must hold that $[A'_0]_{i,j} = 1$. Otherwise, we can partition the sequence as xyz, where $x \in X^*$ and y is now the first occurrence of an element in Y. Then it must be that $[A'_1]_{i,y-n/2} = 1$ and $[A''_2]_{y-n/2,j} = 1$, and as a consequence it must hold that $[A'_1 \times A''_3]_{i,j} = 1$.

7.2 Complexity

Let $C(v^2)$ be the number of recursive calls to compute the closure of a $v \times v$ matrix, and $T(v)$ the number of recursive calls to multiply two $v \times v$ matrices. Our computation in Eqs. (6) and (7) follows the recurrence

$$C(v^2) = 2 \cdot C(v^2/4) + 6 \cdot T(v^2/4).$$

Note that, if we replace $C(v^2)$ by $T(v^2)$, we obtain the same recurrence of Section 6.1. Therefore, we can prove by induction on v that $C(v^2) = T(v^2)$, and thus the number of calls in our closure algorithms is the same as in a multiplication. The number of elements in the matrices we multiply, however, can be as high as a^+. Using Eq. (4) we obtain $O((a^+)^{3/2} \log(v^2/a^+))$ recursive calls.

For the cost of the sums (both the ones done inside the multiplications and those of Eqs. (6) and (7)), we again assume that there are $(a^+)^2/v$ elements on average in every level. Since $a^+ \leq v^2$, the total average cost of our transitive closure algorithm is

$$O\left((a^+)^{3/2} \log v\right). \tag{8}$$

This is $\log v$ times less than the cost of the standard technique, and corresponds to multiplying two uniformly distributed matrices with a^+ 1s.

7.3 Parallelism

It can be seen in Eqs. (6) and (7) that the computation of A'_2 can be carried out in parallel with those of A'_1 and A'_3, though A'_3 must be computed after A'_1 (and all must be computed after A'_0). Analogously, A''_1 can be computed in parallel with A''_2 and A''_0, all after A''_3. In addition, we can use parallelism to perform each isolated multiplication.

To analyze this process, let us call $T(v^2, a^+, p)$ the time for multiplying two random matrices with a^+ 1s on a $v \times v$ submatrix with p processors. Combining Eqs. (5) and (8) and ignoring big-O notation, we get

$$T(v^2, a^+, p) = \frac{1}{p}(a^+)^{3/2} \log v + (a^+)^2/v + \log v \log p.$$

Similarly, let $C(v^2, a^+, p)$ be the time to compute the closure on a $v \times v$ matrix A, assuming that there are already $a^+ = |A^+|$ uniformly distributed 1s in A, and with p processors. Since we perform, on $v/2 \times v/2$ submatrices having on average $a^+/4$ elements, a sequence of two recursive calls and two multiplications, plus other two pairs of multiplications in parallel (i.e., those of A'_1 with A'_2 and those of A''_1 with A''_2), the recurrence for C is

$$\begin{aligned}
& C(v^2, a^+, p) \; = \\
& 2 \cdot C(v^2/4, a^+/4, p) + 2 \cdot T(v^2/4, a^+/4, p) + \\
& 2 \cdot T(v^2/4, a^+/4, p/2) \; \leq \\
& 2 \cdot C(v^2/4, a^+/4, p) + \frac{3}{4} \cdot \frac{1}{p}(a^+)^{3/2} \log v + \\
& \frac{1}{2}(a^+)^2/v + 4 \log v \log p \; \leq \\
& \frac{1}{p}(a^+)^{3/2} \log v + \frac{2}{3} \cdot (a^+)^2/v + O(\log^2 v \log p),
\end{aligned}$$

where the first two terms are obtained by unrolling the recurrence into exponentially decreasing terms, and the latter one is obtained by noting that the recursion ends at the level ℓ where $a^+/4^\ell = 1$, and bounding $\log a^+ = O(\log v)$. The result is $O(T(v^2, a^+, p) + \log^2 v \log p)$, with the same constant in the leading term. This suggests that we can expect a parallel time proportional to that of multiplying the resulting matrix, though this time the additive penalty may become noticeable.

7.4 Restrictions

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, unlike the standard algorithm [43], 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 solution that is much faster than our new transitive closure algorithm.

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.

8 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 each leaf 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 [47] of the matrices $A_i = \mathscr{M}(E_i)$ using $|A_i|$ as their weight (see Section 3 for the definition of $\mathscr{M}(E_i)$). 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 the concatenation $E_1 \,/\, \cdots \,/\, E_m$ is carried out. We cannot apply the well-known optimal algorithm to choose the order for dense matrices [31, 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 $A_1, \ldots, A_m = \mathscr{M}(E_1), \ldots, \mathscr{M}(E_m)$ 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.

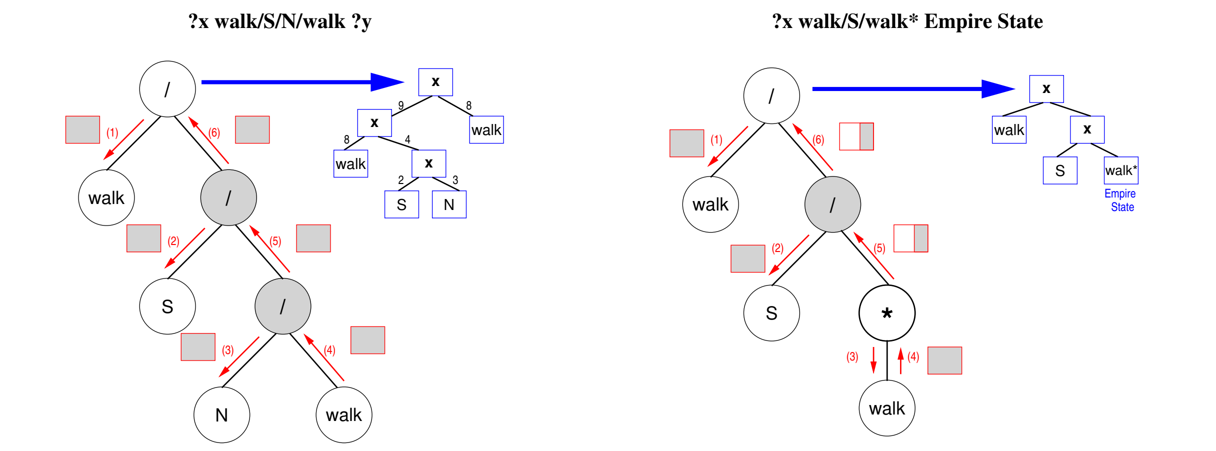


Fig. 7 Two examples for our query plan algorithm. For each query, the left part represents the syntax tree of the regular expression. The gray nodes depict delayed operations. The full- and partial-filled rectangles mean non-restricted and restricted operations, respectively. The tree pointed by the root represents the order of operations that the root has to compute, where the edges denote the size of the matrix representation. In ?x walk/S/N/walk ?y, the query is transformed into a sequence of multiplications on the matrices of the predicates without restrictions. Therefore, those multiplications are delayed until the root. In the root, we firstly choose the multiplication $\mathscr{M}(\mathsf{S}) \times \mathscr{M}(\mathsf{N})$ because the sum of their sizes is 5, smaller than the sum of the first (10) and last pair (11). Then $\mathscr{M}(\mathsf{S})$ and $\mathscr{M}(\mathsf{N})$ are replaced by their product, of size 4, and the procedure continues recursively. Regarding ?x walk/S/walk* Empire State, we can restrict the column of our root. The complete right branch of the tree inherits that restriction, except the node walk due to the Kleene star operator. Once $\mathscr{M}(\mathsf{walk}^*)$ is computed, we restrict it to the corresponding column. Since the previous operations are multiplications, $\mathscr{M}(\mathsf{S}) \times \mathscr{M}(\mathsf{walk}^*)\langle\mathsf{Empire\ State}\rangle$ is delayed until the root. In contrast to the previous example, since the last matrix is restricted by a column, the algorithm computes the multiplication right to left.

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 as described. 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 as described above.

In the restriction on products, we obtain a sequence $\langle r\rangle A_1 \times \cdots \times A_m\langle c\rangle$ (where it could be that only $\langle r\rangle$ or only $\langle c\rangle$ is 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 continue left-to-right in the sequence with normal matrix multiplications, $A' \times A_3$, and so on. Alternatively, once A' is obtained, we could proceed as with an unrestricted multiplication. While this broadens the optimization space, it is more complex, and still most likely to always choose the left-to-right order anyway, because the leftmost matrix (initially A') is likely to be much sparser than the others.

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 done with the multiplication algorithm that enforces the restriction on the other end.

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 satisfies $\langle r\rangle(A \times B)\langle c\rangle = (\langle r\rangle A) \times (B\langle c\rangle)$, 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 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 $\langle r\rangle A \times B^*$ and $\langle r\rangle A \times B^+$ are handled analogously, as well as the cases with both restrictions. The parser is enhanced to detect those cases.

Figure 7 illustrates two relevant cases.

9 A Baseline

We could not find an established software for computations with sparse Boolean matrices, for example to implement transitive closures. We then implemented a baseline representation of sparse matrices, which combines (and adapts to the Boolean case) the well-known CSR and CSC formats [67, Sec. 3.4] in order to speed up multiplications. We store 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.

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 $O(a+b)$. The algorithm is also cache-friendly, as it makes a single left-to-right pass over the input and output arrays. Further, it uses native memory-copy operations when copying whole rows/columns that occur in only one of the matrices, which is faster than merging (despite both operations being linear in the output size). Figure 8 illustrates the sum operation.

For the Boolean multiplication $A \times B$, we use Schoor's algorithm [69], whose average time is $O(ab/v)$ if the 1s are uniformly distributed. Schoor's algorithm intersects the nonempty columns c_i of A with the nonempty rows r_j of B. For each pair $c_i = r_j$, it creates the Cartesian product of all the rows associated with c_i in A with all the columns associated with r_j in B. The result is the union of the pairs in all those Cartesian products. Our implementation, which is more space-efficient, takes $O(ab\log(v)/v)$ time: we first create the row-wise view of the matrix and at the end use it to generate the column-wise view. To create the row-wise view, we set up a priority queue of *tasks* $c_i = r_j$, pointing to the associated rows and columns of A and B and sorted by the smallest row associated with c_i. Once the set of all tasks is created, we extract the smallest row from the queue and append to the result all the columns associated with r_j in B—we may have to merge several column sequences if they are paired with the same minimum row value, and use another priority queue for that. The use of priority queues yields the $O(\log v)$ additional term in the time complexity. Figure 9 illustrates the algorithm.

Closures can be computed naively using $O(\log v)$ multiplications. We implement instead an advanced closure algorithm [64], which first computes the strongly connected components (scc) of the graph using Tarjan's algorithm [72], then creates the reduced and acyclic graph of the scc, computes reachability on the reduced graph in topological order, and finally expands the scc to their node sets. Implemented with the aim of using little working space, the whole algorithm takes time $O(|A^+|\log v)$.

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. Restricted closure operations are performed as for the k^2-tree based representation. The parser and its optimizations are also exactly the same.

10 Experimental Results

We implemented our scheme in C/C++11 and ran our experiments on an Intel(R) Xeon(R) CPU E5-2630 at 2.30GHz, with 6 cores containing 24 processors in total, 15 MB of cache, and 384 GB of RAM. We compiled using g++ with flags `-std=c++11`, `-O3`, and `-msse4.2`. We measure elapsed times. Our code is publicly available [17].

We first study the performance of the individual matrix operations, under various densities, for all our implementations. We then test our implementations in a real scenario where RPQs are solved.

10.1 Performance of Matrix Operations

In order to evaluate the performance of matrix operations, we created 80 squared matrices with v set to 1,000, where the 1s are uniformly distributed according to different densities: 10^{-1}, 10^{-2}, 10^{-3}, and 10^{-4}. Specifically, we generated 20 matrices for each density. We built those matrices with the k^2-tree and our baseline. On those systems, for each consecutive pair of matrices of the same density, we ran the operations sum, intersection, and multiplication without any kind of restriction. In addition, the transitive closure is computed on each matrix. Since we have sequential and parallel algorithms for each operation, we denote them as k^2-tree and k^2-tree-p, respectively. The averaged times of each type operation, separated by density, are shown in Figure 10. In Figure 11, instead, we operate matrices of different densities against a dense matrix (on top) and against a sparse matrix (on the bottom).

We test the sum (Boolean "or") and intersection (Boolean "and") operations as representatives of the other similar operations. For those, we include in the comparison the existing work that supports set operations [65] (called here Set-Ops), which also uses k^2-trees to represent the data. Supporting matrix multiplications and transitive closures on k^2-trees, instead, is a novelty of our approach. We have not

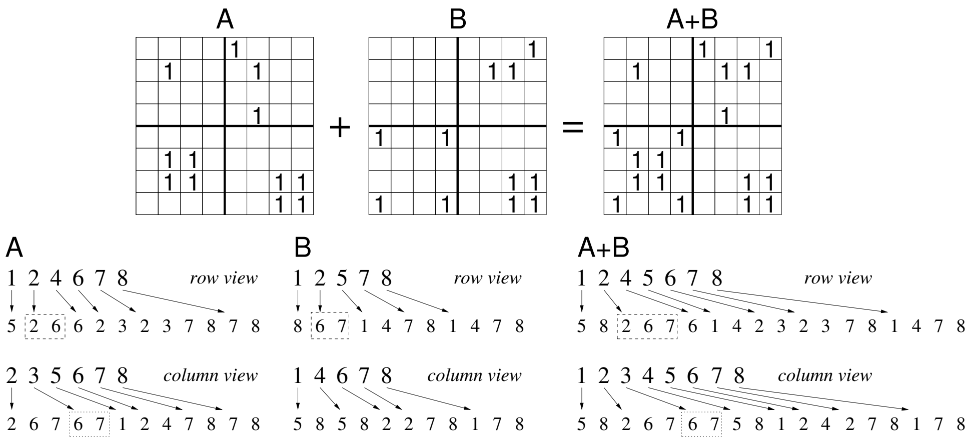


Fig. 8 The baseline format and summation algorithm. In both the row and column views, we traverse both sets of rows/columns, copy the unique rows/columns (as the dotted box corresponding to column 3 in A, whose rows 6 and 7 are copied to the output) and merge the repeated ones (as the dashed boxes corresponding to row 2, whose sets of columns, $\{2,6\}$ and $\{6,7\}$, are merged in the output).

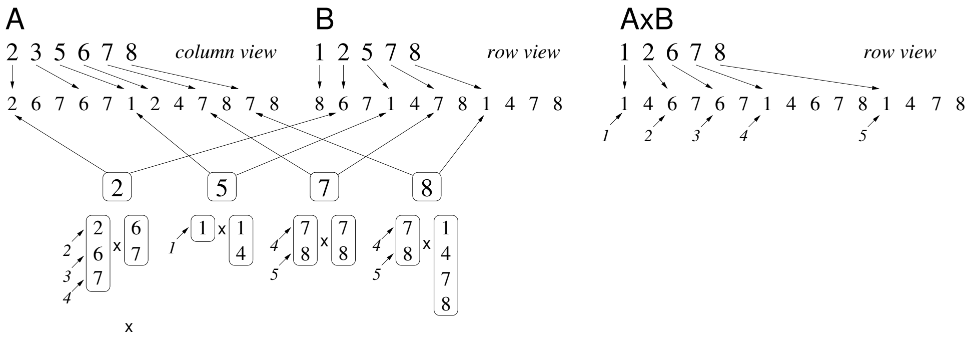


Fig. 9 Our implementation of Schoor's multiplication algorithm. The row view of $A \times B$ is produced from the column view of A and the row view of B. On the bottom-left, the priority queue of 4 tasks created from the intersecting elements $c_i = r_j \in \{2,5,7,8\}$, for each of which we store two pointers towards lists of rows in A and of columns in B. Their cartesian products are indicated explicitly on the bottom for illustrative purposes. The diagonal arrows show the 5 steps along which we create the 5 rows of the results, by choosing the smallest remaining row in each task. We first create row 1, with columns $\{1,4\}$, then row 2 with columns $\{6,7\}$, then row 6 with colums $\{6,7\}$. The next row, 7, appears in tasks 2, 7, and 8, so we merge their sets of columns, $\{6,7\}$, $\{7,8\}$, and $\{1,4,7,8\}$. The last row, 7, also requires merging from tasks 7 and 8.

found any other software to compare with, in particular a non-compressed representation of sparse Boolean matrices; this is why we created our baseline, which is also included in the comparison. As a sanity check, we introduce in the comparison a mature software for numeric (not Boolean) computations on sparse matrices: Octave [36], an open-source alternative to MatLab. Octave is compared only for sums and multiplications, whose complexities are similar for numeric versus Boolean matrices. No equivalent to intersections and transitive closures are supported in Octave.

K^2-trees. Figure 10 shows that our sequential algorithm described in Section 5 is consistently faster than the recursive one, by a factor of 1.1–4.2, and than the sequential algorithm implemented in Set-Ops, by a factor around 1.6–3.0. The former difference owes to the fact that the sequential algorithm is simpler than the recursive one; the latter difference owes to our improvement when copying submatrices by whole computer words. This is evident in Figure 11, where on top the size of the output is still large (a dense matrix), but its alternation measure δ decreases as the other matrix becomes sparser (recall our fine-grained analysis in Section 5). Both our sequential and recursive algorithms are adaptive to δ (the recursive algorithm even in terms of complexity) and consequently their times decrease with the size of the sparser matrix; Set-Ops, instead, stays proportional to the output size and independent of δ. On the bottom plots, the output size also decreases (as δ does) with the size of the denser matrix, and thus all the times decrease accordingly. Here the adaptiveness of our recursive algorithm makes it outperform our sequential algorithm when the difference in densities is maximal, yet its more complex nature makes it finally yield to Set-Ops' sequential algorithm as the difference in densities decreases.

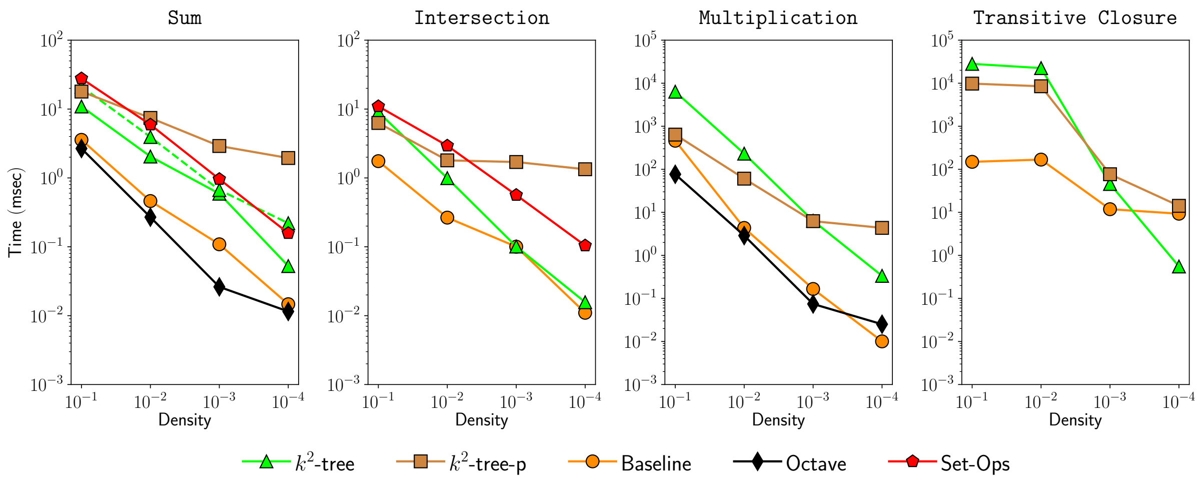


Fig. 10 Averaged times of k^2-tree versions and baselines on matrices with different operations and decreasing density. The dashed line denotes the recursive algorithm of the sum in the k^2-tree.

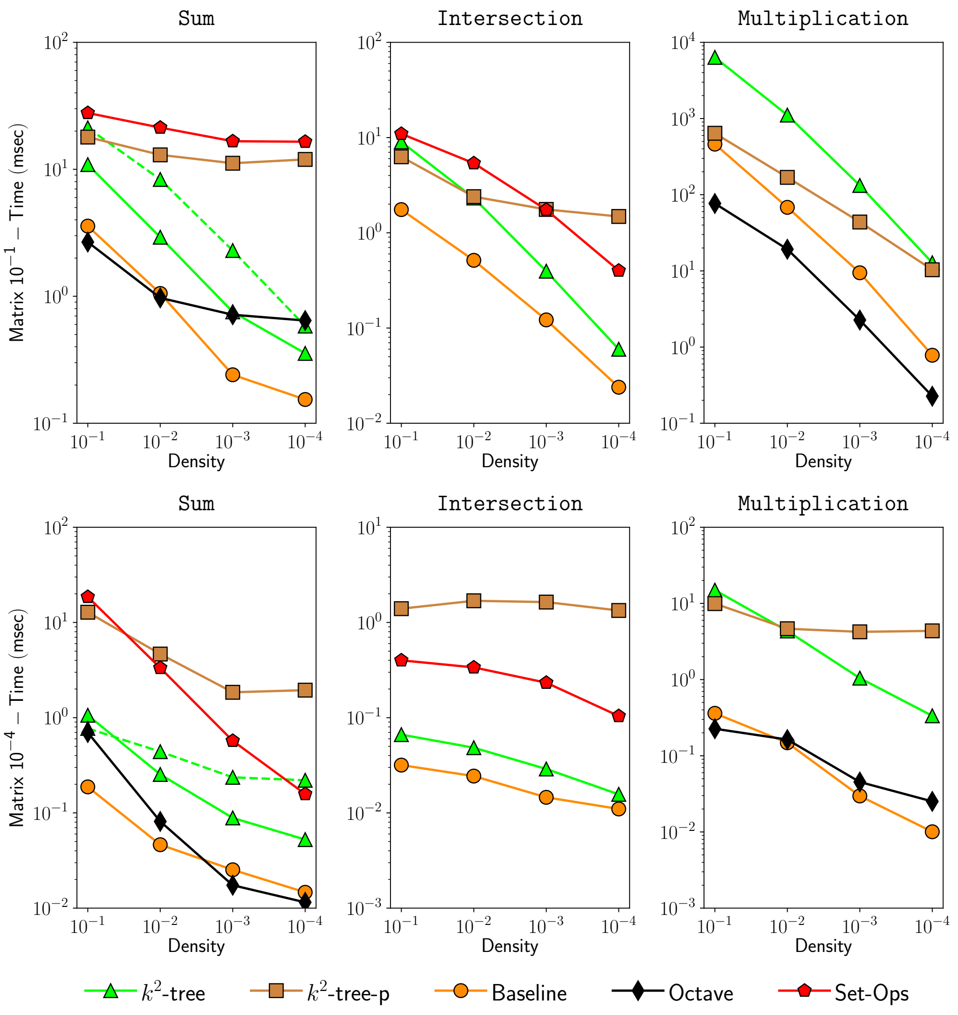


Fig. 11 Average times of k^2-tree versions and baselines on matrices with different densities: we compute A op B, where op $\in \{+, \cap, \times\}$, such that matrix A has density 10^{-1} on top and density 10^{-4} on the bottom, while the density of B varies on the x-axis. The dashed line denotes the recursive algorithm of the sum in the k^2-tree.

For the other sum-like operations, like the intersection, only a recursive version exists, as explained, both for our algorithm and for Set-Ops'. As expected from Table 1, the times are proportional to the number m of 1s in the matrices (i.e., to the densities). This is explained again by our fined-grained analysis in Section 5, which shows a dependence on the output size more than on the input size (in the case of the sum, both input and output sizes are of the same order); this can also be seen in Figure 11. Despite implementing the same algorithm, our code is faster than Set-Ops', with the difference broadening as densities decrease.

Multiplications are in Figure 10 around 3 orders of magnitude slower than sums on the denser matrices, which corresponds to comparing time complexities $m^{3/2}$ with m (see Table 1). As m decreases on the sparser matrices, the gap shrinks to one order of magnitude. Figure 11 shows a notable match with our time complexity analysis of Section 6.1, $O(\min(a,b)\sqrt{\max(a,b)}\log v)$: the slope on the top plot, where the x axis is the size of $\min(a,b)$, is about twice that of the bottom plot, where the x axis is the size of $\max(a,b)$ (the plots use logarithmic scale).

Finally, transitive closures are about an order of magnitude slower than multiplications, as their time complexity is proportional to $(m^+)^{3/2}$, where m^+ is the size of the output. This output size reaches a saturation point at density 10^{-2}, so the time for density 10^{-1} is not very different. The expected density of the transitive closure of a random matrix with density $d > 1/v$ converges to around a constant close to 1 [50], that is, m^+ approaches v^2. This is indeed the case of our densities 10^{-1} and 10^{-2}, but not of the sparser matrices.

Parallel k^2-trees. Table 1 predicts perfect speedups for k^2-tree-p, except for two overheads: one is proportional to the output size and the other is a polylog factor that grows for the transitive closure operation. In the case of the sum-like operations, the impact of the first overhead is comparable to the parallel time; in addition, the parallel algorithm must be recursive and cannot write the results directly to the output (recall Section 5.3). This combination makes parallelism achieve in Figure 10 only a moderate speedup of 1.8 over Set-Ops on dense matrices, and to always lose to both implementations of the sequential k^2-tree. The speedup vanishes, and parallelism becomes actually counterproductive, as densities decrease. This is related to an effect that does not show up in the PRAM analysis: the system overhead incurred when creating the threads reduces the impact of parallelism on the easier operations. The case of intersections is similar, though the k^2-tree-p outperforms the k^2-tree (by a very slight margin) on the densest matrices. The overhead incurred by parallelization is more visible in the case of intersections, where the resulting matrices are extremely sparse but k^2-tree-p is unable to reduce the times obtained for density 10^{-2}. Figure 11 also shows that the k^2-tree-p cannot exploit the extremely low densities of the output.

On multiplications, the PRAM overheads are less significant because they are of lower order than the amount of work to do—$O(m^2/v)$ versus $O(m^{3/2})$. Figure 10 shows that, on the denser matrices, the speedup is near 10—making the parallel k^2-tree approach our baseline—, but it decreases up to becoming counterproductive on densities below 10^{-3}. Figure 11 confirms that the speedup is significative when the density of the resulting matrix—which is vd_1d_2 on matrices with side v and densities d_1 and d_2—is 10^{-1} or higher (e.g., two matrices of density 10^{-2} in our case).

On transitive closures, perhaps due to the higher polylog overhead, the best speedup obtained by the k^2-tree-p is around 3 and decreases as for multiplications.

Baselines. Octave is 1.3–3.9 times faster than our baseline on sums. On multiplications, it is up to 6 times faster, but the gap decreases with lower densities, where the baseline finally catches up. In general, we can see that our baseline implementation achieves a reasonably competitive performance against much more mature implementations of sparse matrix operations, thereby providing a relevant implementation of Boolean operations on sparse matrices. Further, the top of Figure 11 shows that, on sums, Octave is not adaptive to the alternation complexity δ (which decreases with the size of the sparser matrix), but is instead proportional to the size of the input or the output. As a consequence, our baseline does outperform Octave—by a margin of up to 4.2—when the densities are very different.[4]

In all the operations, our baseline is considerably faster than the k^2-tree, as we can expect from its better time complexities in Table 1. For the sum-like operations, where the gap is just $O(\log v)$, the baseline is about 4–9 times faster than the recursive sum and the intersection (they get close in the intersection when there are very few points), and even 3–5 times faster than our sequential implementation of the sum. Figure 11 shows that the baseline is also adaptive to the lower densities, just as the k^2-tree, because it also processes faster the submatrices that must be copied directly to the output; recall Section 9.

[4] Further, Octave's times may differ significantly depending on the order in which the two matrices are multiplied ($A \times B$ or $B \times A$), even when they are random and the result has about the same cardinality. We chose the best of both times in the plots (although in real cases one cannot choose).

On the heavier operations, where the time complexities are farther apart (recall that m^2/v is always less than $m^{3/2}$), the baseline is 14–53 times faster for multiplications and 4–189 times faster for transitive closures. An exception is the lowest density, where the recursive algorithm we designed for the k^2-tree outperforms the algorithm based in strongly connected components we implemented for the baseline. This latter algorithm has some basic setup costs that possibly offset its benefits when there are very few 1s in the matrix.

As shown in the next section, the baseline uses in exchange about 4 times more space than the k^2-trees. The next section also shows that the large differences exposed here shrink considerably on the real-life application. The reason is that those matrices are far larger and sparser than those we tried here, and as we have seen the differences shrink as the density decreases.

10.2 Performance on Real-Life RPQs

We now use our Boolean sparse matrix algebra implementations to solve actual RPQs, from a query log posed to a real-world graph database. Besides comparing our implementations with existing solutions and recommending which one to use depending on the case, our experiments aim to answer two fundamental questions:

1. How *the strategy* of using Boolean matrices compares with the most commonly used ones, without necessarily reducing space. As we will see, the baseline uses about the same amount of memory as the Ring, which allows us comparing the strategies under equal conditions of memory usage.
2. What is the price of *reducing space*, from that of the baseline to a quarter of it, as used by k^2-trees, while employing the same strategy based on Boolean matrices. While k^2-trees will be slower than the baseline, they can be used in main memory on much larger datasets.

Systems. We compared our implementations with the following systems:

- *Ring*: A compact data structure that supports RPQs in labeled graphs [16]. The variant *Ring$_{AB}$* uses more space but is much faster.
- *Virtuoso*: A popular graph database that hosts the public DBpedia endpoint, among others [38].
- *Blazegraph*: The graph database system [74] hosting the official Wikidata Query Service [53].
- *Jena*: A reference implementation of the SPARQL standard.

The Ring (and Ring$_{\text{AB}}$) are paradigmatic examples of the technique that solves RPQs based on the product graph; recall Section 2.2. The Ring$_{\text{AB}}$ variant optimizes the search by splitting the regular expression at a convenient edge, whose label is infrequent in the database graph, and running two opposite searches from those edges. Jena also relies on graph traversal in BFS order to find shortest paths that match RPQs.

Virtuoso and Blazegraph, instead, extend the relational algebra. They reduce RPQs to joins and unions as much as possible, and use different mechanisms to handle the transitive closures (see, e.g., [4]).

Datasets and queries. We used a Wikidata graph [76] 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 32-bit integers, the data set requires 8.5 GB.

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 [53]. 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 1,567 unique queries.

We ran the queries in each system with a timeout limit of 60 seconds. On the k^2-tree, we ran the single- and multi-threaded versions of our algorithms.

General results. Table 2 summarizes the space usage and time performance of all the systems. Notably, our k^2-tree based approach yields the most compact structure, requiring only 4.33 bytes per triple (bpt). This is nearly half the space of the described plain representation of the raw data, and about a fourth of the space used by the next smallest representations that support 2RPQs (Ring and our baseline). Classical systems use 14–22 times more space than our k^2-trees. Note also that the k^2-tree representation is 1–2 orders of magnitude faster to build than the others.

This reduced space is paid in terms of time performance. Our sequential k^2-tree is on average 2.6 times slower than our baseline, 2.7 times slower than the Ring, 7.9 times slower than the fastest system (Ring$_{\text{AB}}$), and 1.6 times slower than the fastest classical system (Virtuoso). Ring$_{\text{AB}}$ and Virtuoso stand out for their stability—just one timeout. Still, the k^2-tree solves those 2RPQs in less than 4 seconds on average, and are competitive with established systems like Blazegraph and Jena. The median k^2-tree time is 1.7–11.7 times higher than the others (excluding the baseline). Our k^2-tree-p does not compete in general with the sequential k^2-tree, which is to be expected in

Table 2 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.

	k^2-tree	k^2-tree-p	Baseline	Ring	Ring_{AB}	Virtuoso	Blazegraph	Jena
Index space (bpt)	4.33	4.33	16.45	16.41	27.93	60.07	90.79	95.83
Indexing time (hs)	0.3	0.3	5.5	7.5	8.3	3.0	39.4	37.4
Average (sec)	3.25	3.46	1.27	1.19	0.41	2.08	3.23	4.51
Median (sec)	0.33	0.38	0.004	0.09	0.03	0.13	0.13	0.21
Timeouts	39	44	14	9	1	1	41	84
Average c (sec)	2.83	3.09	1.07	0.65	0.25	1.79	3.24	3.62
Median c (sec)	0.33	0.38	0.004	0.08	0.03	0.11	0.13	0.19
Timeouts c	30	33	12	2	0	1	39	58
Average ¬c (sec)	11.92	11.21	5.45	12.43	3.66	8.17	2.98	22.83
Median ¬c (sec)	0.87	0.79	0.01	2.09	0.93	3.89	0.14	1.57
Timeouts ¬c	9	11	2	7	1	0	6	26

principle because the matrices are very sparse in this application (we show later a case where it does).

Our baseline, on the other hand, uses almost the same space as the Ring, and it is on average 7% slower. It is 2.6 times faster than k^2-trees, and 3.1 times slower than the Ring_{AB} (which uses 1.7 times more space, however). While using 3.7 times less space than Virtuoso, it is 1.6 times faster. While the baseline performs similarly to the Ring in those aspects, it solves many easy queries much faster than all the systems—its median is an order of magnitude lower. Yet, it produces more timeouts than the Ring, Ring_{AB}, and Virtuoso, thereby displaying less stability. The other systems time out on many more queries, though.

The situation turns more against our matrix-based methods on the easier 2RPQs—those containing some constant. On those, the single-threaded k^2-tree is on average 4.4 times slower than the Ring, 11.3 times slower than the Ring_{AB}, and 1.6 times slower than Virtuoso, yet still outperforming Blazegraph and Jena. The baseline is here 1.6 times slower than the Ring and 4.3 times slower than the Ring_{AB}, yet it is still 1.7 times faster than Virtuoso, and it is still an order of magnitude faster on the median.

On the harder queries, with no constant extreme, the relative performance of matrix-based methods is much better: the k^2-tree and k^2-tree-p are 5% and 11% faster than the Ring, respectively, still using 4 times less space. This time the parallel version reduces the median times, as most queries are hard enough to benefit from parallelism. The baseline, still using about the same space as the Ring, is 2.3 times faster. In these queries, however, the fastest system is Blazegraph, which is 1.8 times faster than our baseline and 3.8 times faster than k^2-tree-p. The Ring_{AB}, using not as much space, is 1.5 times faster than our baseline and 3.1 times faster than the k^2-tree-p. Yet, this comes at the expense of using 1.7 times more space than the baseline and 6.5 times more space than k^2-trees. The baseline still has 1–2 orders of magnitude faster median times.

Figure 12 shows the query time distributions using violin plots [45] (these show a symmetric histogram of values along the y axis) together with averages (higher segments) and medians (lower segments). The violin plots are placed, along the x coordinate, corresponding approximately to the space usage of the different structures—see Table 2 for the detailed space usage. The left plot refers to all the queries, whereas the right one considers only the harder 2RPQs (with both variable extremes). To avoid cluttering, we leave out the k^2-tree-p, as it looks identical to the k^2-tree.

In the general case, the k^2-tree, the baseline, and both Ring variants are the dominant representations. The first one offers a decent solution with low space (slightly over 4 bpt), solving 90% of the queries in less than 5.3 seconds. Using 4 times that space (about 16 bpt), the Ring and the baseline distribute similarly and are noticeably faster than k^2-trees, though the baseline has a much lower median. Finally, the Ring_{AB} uses about twice that space (28 bpt) and is considerably faster and more stable than the baseline and the Ring. All classical systems are outperformed in general by the Ring_{AB} in both time and space.

When it comes to handling the harder 2RPQs, the right plot shows that the baseline becomes noticeably faster than the Ring—which distributes even worse than k^2-trees and thus becomes not competitive. The baseline's distribution is only outperformed—by a dis-

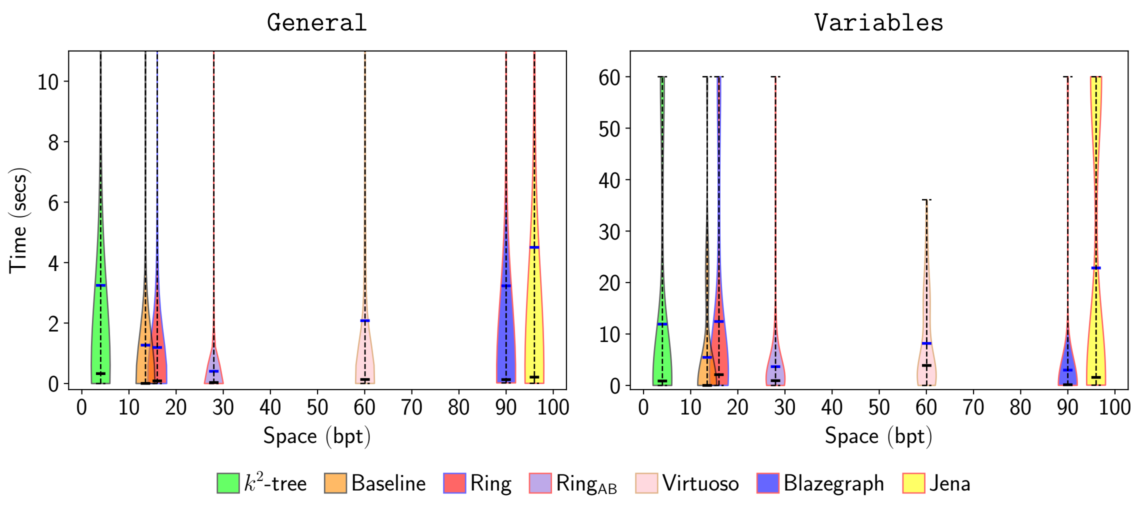


Fig. 12 Space and query time distribution of the systems in general (left) and for the 2RPQs with no constants (right). The baseline and the Ring use almost the same space, but we separate them for readability.

crete margin—by that of Ring_{AB} and Blazegraph, yet using much more space (28 and 90 bpt, respectively).

Finally, beyond stating which systems perform better depending on the case, let us return to our two general questions:

1. When using the same amount of space (i.e., baseline versus Ring), the technique of using Boolean matrices makes little difference in the overall query times, being slower than product-graph traversal on the easy queries and faster on the harder ones (i.e., when both extremes of the query are variable). This can be expected from the fact that, on the simpler queries, the Ring starts the product graph traversal from just one node, whereas the Boolean matrix strategy computes a global result (even if it can restrict the operations to start from a specific row or column). On the harder queries, the Ring loses that advantage and yields to the baseline. Indeed, the improved version Ring_{AB} recovers the possibility of starting the traversals from a few selected edges, and thus performs close to the baseline, yet it requires almost twice its space to do so. In turn, using much less space than classic solutions, Ring_{AB} outperforms them in most cases (except Blazegraph on the harder queries, which is only slightly better).
2. When the space usage is reduced to a quarter within matrix-based strategies (i.e., baseline versus k^2-tree), average times increase uniformly by a factor around 2.6. Still, the k^2-tree handles most queries within a few seconds and even slightly outperforms the Ring on the harder queries. This makes it convenient when there is no enough main memory to hold the baseline data structure, or when it is convenient to free main memory for other data structures in a larger system.

Discussion by query types. Figure 13 showcases the performance of the systems across different types of queries. For instance, queries of form `x (a|b)* y` with variable `x` and constant `y` are represented as '`v (|)* c`': `v` indicates a variable, `c` indicates a constant, and the middle section represents the pattern of the regular expression. We show the most popular query types that appear in the log.

We note that the baseline is in all cases the best or close to the best, with the notable exception of `v /* v`, where it is forced to compute a full transitive closure.

The k^2-tree is in general much slower, which is the price it pays for its compactness, but still has a performance comparable with the classical systems (Virtuoso, Blazegraph, Jena) in various queries, and excells in some like `c * v`, `c /* v`, `v | v`, and `v + v`. In the first two cases, it optimizes the query by restricting a matrix to a row; for example the query `c /* v`, corresponds to the case $\langle r \rangle A \times B^*$, where we avoid to compute the transitive closure of B. The difference between `c * v` and `v * c` lies in the number of results reported, averaging 2,930 and 53,052, respectively; the case of `c /* v` and `v /* c` is analogous. This shows how much the actual data influences performance, despite the symmetry of the queries. The query `v | v` corresponds to the Boolean sum, which is particularly fast to implement in our matrix algebra.

As expected from the discussion in Section 10.1, our parallel version rarely outperforms the sequential one in this application, because the matrices are extremely sparse. It is even counterproductive in very

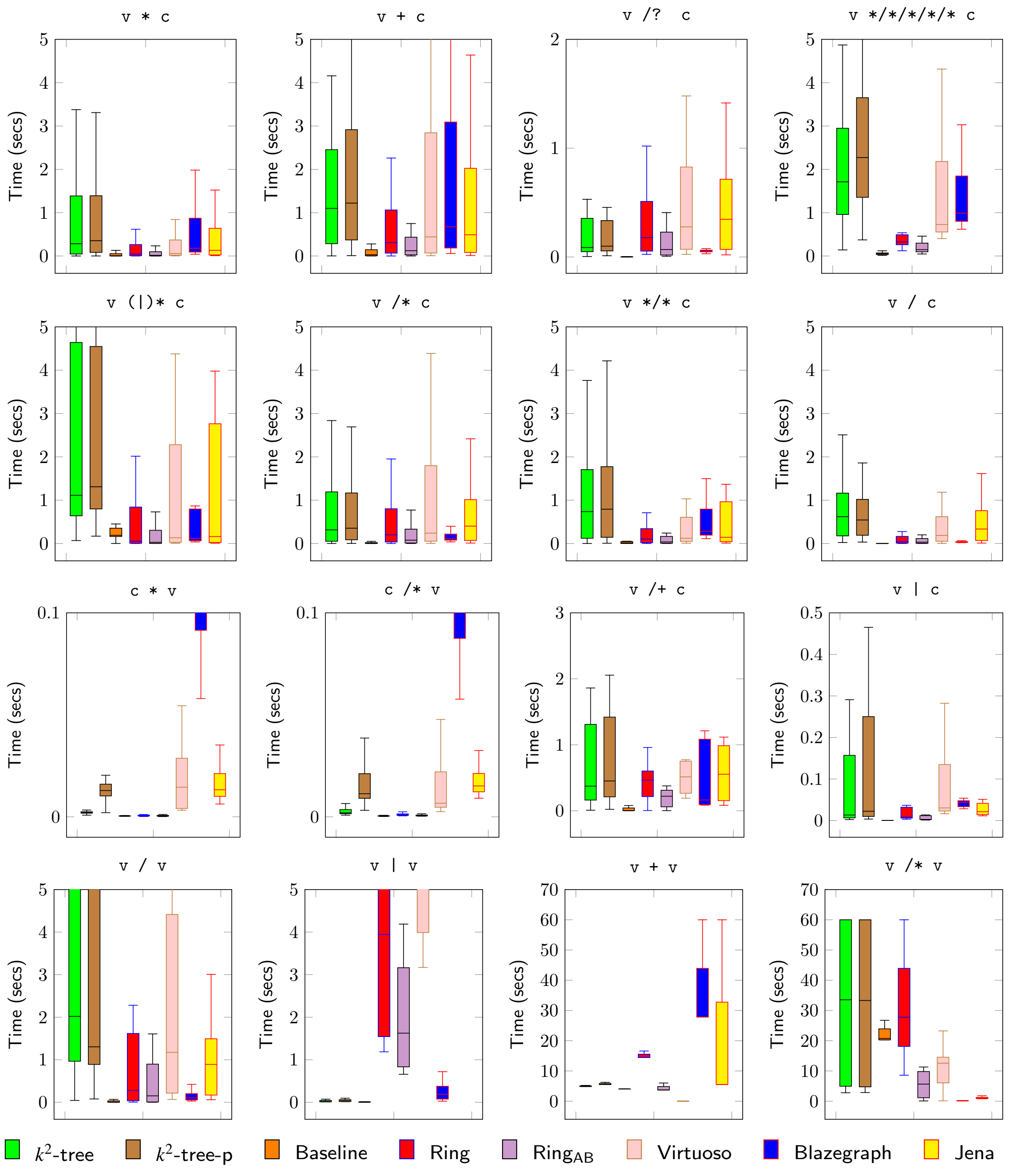


Fig. 13 Boxplots of the distribution of query times. Some boxplots are completely above the vertical limit.

easy queries like c * v and c /* v, where the cost of creating the threads outweights the advantage of a parallel execution. A relevant case where parallelism helps is v / v, because it boils down to matrix multiplication, which is the case where parallelism obtains the best speedups. This shows that parallelism could be beneficial in fixed path queries of the form v /.../ v.

We note that some queries, like v / c and v / v, are also basic graph patterns (BGPs), that is, fixed subgraphs to be found in the data. The Ring solves them by resorting to existing algorithms for BGPs that run on the same data structure it uses [15]. We could optimize our k^2-tree algorithms in the same way, as there also exist efficient algorithms for BGPs that run on the very same k^2-tree structures [13]. A more complete integration of RPQs and BGPs is discussed in the Conclusions.

11 Conclusions

We have explored the use of a Boolean 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 most RPQs within a few seconds and (slightly) outperforms the most compact previous representations on the difficult queries. We have implemented and adapted state-of-the-art algorithms for sparse matrices to implement our baseline, and designed new ones for the k^2-trees.

Our implementations offer a full composable Boolean matrix algebra, as the result of the operations is given in the same representation of the arguments. Such implementations, which are publicly available [17], are of interest beyond solving RPQs, as their use arises in other situations, such as ML applications [37]. We additionally developed a multithreaded version of the k^2-tree implementation that is shown to significantly outperform the sequential version for matrix multiplications when the density of the resulting matrix is high enough. While graph databases are much sparser, the multithreaded version can be useful in other applications that handle denser matrices.

Future work. We first discuss how to address the limitations discussed in the Introduction.

One extension to the current work is to implement our algorithms over *dynamic* versions of the k^2-tree [12, 29] which would permit adding and removing nodes and edges from the graphs in addition to creating new matrices via matrix algebra operations. Experiments on those dynamic k^2-trees show that, while retaining essentially the same space usage of the static version, updates can be performed in a few microseconds, while static times to traverse the matrix increase by at most 50%. Those update times are very competitive: the construction time we reported for k^2-trees is 1.1 microseconds per edge, comparable to the cost of building them via successive insertions on the existing dynamic k^2-tree representations. It is also possible to support updates in the baseline representation, by extending its current simple arrays to lists of blocks of varying size.

An advantage of one of the k^2-tree implementations that support dynamism [12] is that, in addition, it features more locality of reference. It partitions the tree into connected components, which are compactly represented in contiguous blocks of memory, with actual pointers among them. Deploying those blocks onto disk pages would yield a disk-friendly representation. The baseline representation, instead, is naturally disk-friendly, as all the operations boil down to sequential traversals on its arrays.

Our third limitation is the support of negated labels, which in the case of k^2-trees require a non-expensive way to represent and handle submatrices full of 1s. Such extensions of k^2-trees have been proposed [21, 65], but they have not been adapted to handle the most complex Boolean matrix operations (multiplication and transitive closure). The baseline can also be extended so as to mark matrices as negated and simulating the complement of the represented matrices when performing the operations.

We can also strenghten our RPQ optimizer in order to detect common subexpressions and exploit a number of identities of the Boolean algebra we have disregarded for now, for example $A+A, A \times I, A+0, I^*$, and others that may arise naturally.

Another path of future work is to integrate RPQs with BGPs, the other main segment of most graph query languages. In those combined queries, some triple patterns (x, p, y) refer to predicates p and others may be 2RPQs of the form (x, E, y). We can then use our matrix algebra to materialize those RPQs into a resulting matrix, which acts as a new predicate p_E. The result is a simple BGP, which can then be solved with Qdags [13], an existing solution BGPs that is also based on representing each predicate as a k^2-tree. In this way we would not need any extra space, since both indices use exactly the same data structures. A particularly intriguing question is whether we can extend the idea of *Lazy Qdags* [13] to RPQs, so as to deliver the output gradually, as required, instead of producing it eagerly in full.

Acknowledgements: The authors thank the reviewers for their insightful comments.

Competing interests: The authors declare no competing interests.

References

1. Agrawal R (1988) Alpha: An extension of relational algebra to express a class of recursive

queries. IEEE Transactions on Software Engineering 14(7):879—885

2. Aho AV, Ullman JD (1979) The universality of data retrieval languages. In: Proc. 6th POPL, pp 110–120
3. Aho AV, Hopcroft JE, Ullman JD (1974) Data Structures and Algorithms. Addison-Wesley
4. Aimonier-Davat J, Skaf-Molli H, Molli P, Dang M, Nédelec B (2023) Join ordering of SPARQL property path queries. In: Proc. 20th ESWC, pp 38—54
5. Álvarez-García S, Brisaboa NR, Fernández J, Martínez-Prieto M, Navarro G (2015) Compressed vertical partitioning for efficient RDF management. Knowledge and Information Systems 44(2):439–474
6. Amossen RR, Pagh R (2009) Faster join-projects and sparse matrix multiplications. In: Proc. 12th ICDT, pp 121–126
7. Angles R, Arenas M, Barceló P, Hogan A, Reutter JL, Vrgoc D (2017) Foundations of modern query languages for graph databases. ACM Computing Surveys 50(5):68:1–68:40
8. Angles R, Arenas M, Barceló P, Boncz PA, Fletcher GHL, Gutiérrez C, Lindaaker T, Paradies M, Plantikow S, Sequeda JF, van Rest O, Voigt H (2018) G-CORE: A core for future graph query languages. In: Proc. SIGMOD, pp 1421–1432
9. Apostolico A, Drovandi G (2009) Graph compression by BFS. Algorithms 2(3):1031–1044
10. Arlazarov V, Dinic E, Kronrod M, Faradžev I (1970) On economical construction of the transitive closure of a directed graph. Dokl Akad Nauk SSSR 194(11):487–488, in Russian. English translation in Soviet Math. Dokl. 11:5, 1209–1210
11. Arroyuelo D, Castillo JP (2023) Trie-compressed adaptive set intersection. In: Proc. 34th CPM, pp 1:1–1:19
12. Arroyuelo D, de Bernardo G, Gagie T, Navarro G (2019) Faster dynamic compressed d-ary relations. In: Proc. 26th SPIRE, LNCS 11811, pp 419–433
13. Arroyuelo D, Navarro G, Reutter JL, Rojas-Ledesma J (2022) Optimal joins using compressed quadtrees. ACM Transactions on Database Systems 47(2):article 8
14. Arroyuelo D, Gómez-Brandón A, Navarro G (2023) Evaluating regular path queries on compressed adjacency matrices. In: Proc. 30th SPIRE, pp 35–48
15. Arroyuelo D, Gómez-Brandón A, Hogan A, Navarro G, Reutter JL, Rojas-Ledesma J, Soto A (2024) The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. ACM Transactions on Database Systems 49(2):article 5
16. Arroyuelo D, Gómez-Brandón A, Hogan A, Navarro G, Rojas-Ledesma J (2024) Optimizing RPQs over a compact graph representation. The Very Large Databases Journal 33:349–374
17. Arroyuelo D, Gómez-Brandón A, Navarro G (2024) Sparse Boolean Matrix Algebra. https://github.com/adriangbrandon/rpq-matrix
18. Azimov R, Epelbaum I, Grigorev SV (2021) Context-free path querying with all-path semantics by matrix multiplication. In: Proc. 4th GRADES-NDA, pp 4:1–4:7
19. Barbay J, Kenyon C (2008) Alternation and redundancy analysis of the intersection problem. ACM Transactions on Algorithms 4(1):4:1–4:18
20. Barceló P (2013) Querying graph databases. In: Proc. 32nd PODS, pp 175–188
21. de Bernardo G, Álvarez-García S, Brisaboa NR, Navarro G, Pedreira O (2013) Compact querieable representations of raster data. In: Proc. 20th SPIRE, pp 96–108
22. de Bernardo G, Gagie T, Ladra S, Navarro G, Seco D (2023) Faster compressed quadtrees. Journal of Computer and System Sciences 131:86–104
23. Boldi P, Rosa M, Santini M, Vigna S (2011) Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Proc. 20th WWW, pp 587–596
24. Bonifati A, Martens W, Timm T (2019) Navigating the maze of Wikidata query logs. In: Proc. WWW, pp 127–138
25. Bonifati A, Martens W, Timm T (2020) An analytical study of large SPARQL query logs. The VLDB Journal 2–3:655–679
26. Brisaboa N, Cerdeira-Pena A, de Bernardo G, Fariña A, Navarro G (2023) Space/time-efficient rdf stores based on circular suffix sorting. The Journal of Supercomputing 79:5643–5683
27. Brisaboa NR, Ladra S, Navarro G (2014) Compact representation of Web graphs with extended functionality. Information Systems 39(1):152–174
28. Clark DR (1996) Compact PAT trees. PhD thesis, University of Waterloo, Canada
29. Coimbra ME, Hrotkó J, Francisco AP, Russo LMS, de Bernardo G, Ladra S, Navarro G (2022) A practical succinct dynamic graph representation. Information and Computation 285B:article 104,862
30. Coppersmith D, Winograd S (1990) Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9(3):251–280

31. Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to Algorithms, 3rd edn. MIT Press
32. Demaine ED, López-Ortiz A, Munro JI (2000) Adaptive set intersections, unions, and differences. In: Proc. 11th SODA, pp 743–752
33. Deutsch A, Xu Y, Wu M, Lee VE (2020) Aggregation Support for Modern Graph Analytics in TigerGraph. In: Proc. SIGMOD, pp 377–392
34. 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 (2022) Graph pattern matching in GQL and SQL/PGQ. In: Proc. SIGMOD, pp 2246–2258
35. Dijkstra E (1976) A Discipline of Programming. Prentice Hall, chapter 25
36. Eaton JW, Bateman D, Hauberg S, Wehbring R (2021) GNU Octave version 6.3.0 manual: a high-level interactive language for numerical computations
37. Elgohary A, Boehm M, Haas PJ, Reiss FR, Reinwald B (2019) Compressed linear algebra for declarative large-scale machine learning. Communications of the ACM 62(524):83–91
38. Erling O, Mikhailov I (2009) RDF support in the Virtuoso DBMS. In: Networked Knowledge – Networked Media, Springer, pp 7–24
39. Fawzi A, Balog M, Huang A, Hubert T, Romera-Paredes B, Barekatain M, Novikov A, Ruiz FJR, Schrittwieser J, Swirszcz G, Silver D, Hassabis D, Kohli P (2022) Discovering faster matrix multiplication algorithms with reinforcement learning. Nature 610:47–53
40. Fischer MJ, Meyer AR (1971) Boolean matrix multiplication and transitive closure. In: Proc. 12th SWAT, pp 129–131
41. Francis N, Green A, Guagliardo P, Libkin L, Lindaaker T, Marsault V, Plantikow S, Rydberg M, Selmer P, Taylor A (2018) Cypher: An Evolving Query Language for Property Graphs. In: Proc. SIGMOD, pp 1433–1445
42. Fredman ML, Willard DE (1993) Surpassing the information theoretic bound with fusion trees. Journal of Computer and System Sciences 47(3):424–436
43. Furman ME (1970) 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
44. Hernández C, Navarro G (2014) Compressed representations for Web and social graphs. Knowledge and Information Systems 40(2):279–313
45. Hintze JL, Nelson RD (1998) Violin plots: A box plot-density trace synergism. The American Statistician 52(2):181—184
46. Hogan A, Riveros C, Rojas C, Soto A (2019) A worst-case optimal join algorithm for SPARQL. In: Proc. 18th ISWC, pp 258–275
47. Huffman DA (1952) A method for the construction of minimum-redundancy codes. Proc Institute of Electrical and Radio Engineers 40(9):1098–1101
48. Itai A, Rodeh M (1978) Finding a minimum circuit in a graph. SIAM Journal of Computing 7(4):413–423
49. Jakobsson H (1991) Mixed-approach algorithms for transitive closure (extended abstract). In: Proc. 10th PODS, pp 199–205
50. Karp RM (1990) The transitive closure of a random digraph. Random Structures and Algorithms 1(1):73–94
51. Knuth DE (2009) The Art of Computer Programming, volume 4: Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams. Addison-Wesley Professional
52. Losemann K, Martens W (2012) The complexity of evaluating path expressions in SPARQL. In: Proc. 31st PODS, pp 101–112
53. Malyshev S, Krötzsch M, González L, Gonsior J, Bielefeldt A (2018) Getting the most out of Wikidata: Semantic technology usage in Wikipedia's knowledge graph. In: Proc. ISWC, pp 376–394
54. Manola F, Miller E (2004) RDF Primer. W3C Recommendation, `http://www.w3.org/TR/rdf-primer/`
55. Martens W, Niewerth M, Popp T, Rojas C, Vansummeren S, Vrgoc D (2023) Representing paths in graph database pattern matching. Proc VLDB Endowment 16(7):1790–1803
56. Mendelzon AO, Wood PT (1995) Finding regular simple paths in graph databases. SIAM Journal on Computing 24(6):1235–1258
57. Morton GM (1966) A computer oriented geodetic data base; and a new technique in file sequencing. Tech. rep., IBM Ltd.
58. Munro JI (1971) Efficient determination of the transitive closure of a directed graph. Information Processing Letters 1(2):56–58
59. Munro JI (1996) Tables. In: Proc. 16th FSTTCS, pp 37–42
60. Navarro G (2016) Compact Data Structures – A practical approach. Cambridge University Press
61. Nuutila E (1994) An efficient transitive closure algorithm for cyclic digraphs. Information Processing Letters 52(4):207–213

62. Nuutila E (1995) Efficient transitive closure computation in large digraphs. PhD thesis, Finnish Academy of Technology, Finland
63. Penn G (2006) Efficient transitive closure of sparse matrices over closed semirings. Theoretical Computer Science 354(1):72–81
64. Purdom PW (1970) A transitive closure algorithm. BIT 10:76–94
65. Quijada-Fuentes C, Penabad MR, Ladra S, Gutiérrez G (2019) Set operations over compressed binary relations. Information Systems 80:76–90
66. van Rest O, Hong S, Kim J, Meng X, Chafi H (2016) PGQL: A property graph query language. In: Proc. GRADES, p 7
67. Saad Y (2003) Iterative Methods for Sparse Linear Systems. SIAM
68. Samet H (2006) Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann
69. Schoor A (1982) Fast algorithm for sparse matrix multiplication. Information Processing Letters 15(2):87–89
70. Sharir M (1981) A strong-connectivity algorithm and its applications to data flow analysis. Computers and Mathematics with Applications 7(1):67–72
71. Strassen V (1969) Gaussian elimination is not optimal. Numerische Mathematik 13:354—356
72. Tarjan RE (1972) Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2):146–160
73. Tetzel F, Kasperovics R, Lehner W (2019) Graph traversals for regular path queries. In: Proc. 2nd GRADES-NDA, pp 5:1–5:8
74. Thompson BB, Personick M, Cutcher M (2014) The Bigdata®RDF Graph Database. In: Linked Data Management, Chapman and Hall/CRC, pp 193–237
75. Valiant LG (1975) General context-free recognition in less than cubic time. Journal of Computer and Systems Sciences 10(2):308–315
76. Vrandecic D, Krötzsch M (2014) Wikidata: A free collaborative knowledgebase. Communications of the ACM 57(10):78–85
77. Warshall S (1962) A theorem on boolean matrices. Journal of the ACM 9(1):11–12
78. Williams VV (2012) Multiplying matrices faster than Coppersmith-Winograd. In: Proc. 44th STOC, pp 887–898
79. Yakovets N, Godfrey P, Gryz J (2016) Query Planning for Evaluating SPARQL Property Paths. In: Proc. SIGMOD, pp 1875–1889
80. Yannakakis M (1990) Graph-theoretic methods in database theory. In: Proc. 9th PODS, pp 230–242
81. Yu H (2018) An improved combinatorial algorithm for boolean matrix multiplication. Inf Comput 261:240–247
82. Yuster R, Zwick U (2005) Fast sparse matrix multiplication. ACM Transactions on Algorithms 1(1):2–13