Uplifting the Superpowers of Worst-Case-Optimal Join Algorithms*

Adrián Gómez-Brandón[1,3], Aidan Hogan[2,3], and Gonzalo Navarro[2,3]

[1] Laboratorio de Bases de Datos, Universidade da Coruña, Spain
[2] Deptartment of Computer Science, University of Chile, Chile
[3] IMFD — Millennium Institute for Foundational Research on Data, Chile

Abstract. Worst-case-optimal (wco) join algorithms have demonstrated their power – in both theory and practice – to efficiently solve complex Basic Graph Patterns (BGPs). Modern graph query languages, such as SPARQL and GQL, have BGPs at their core, but also have a wide range of other features, including filters (aka. selections). Such conditions are typically handled via pre- or post-filtering, before or after processing the BGPs. In this paper we show how to uplift wco join algorithms so as to incorporate such filtering natively, improving efficiency. We demonstrate the superiority of this approach by extending the *Ring* – a compact index that provides wco resolution of BGPs within almost no extra space on top of the graph – so as to handle property graphs using our new techniques while retaining compactness. We implement this extension and experimentally show that it outperforms various baseline systems.

1 Introduction

There is an ever-growing demand for increasingly-efficient systems for querying knowledge graphs. In the context of open knowledge graphs, RDF and SPARQL dominate [32]. In the context of enterprise knowledge graphs, property graphs, query languages such as Cypher [23] and GQL [17], and commercial engines such as Neo4j [52], are a popular alternative. Despite these seemingly disparate graph models and query languages, the same key primitives – in particular, graph patterns [2] – underlie both RDF/SPARQL and property graph query engines. Techniques for querying one form of knowledge graph model can often be applied for the other, with various systems [12, 48, 51] natively supporting both.

A promising recent line of research for efficiency evaluating graph queries is to combine compact data structures and worst-case optimal joins [6, 7, 13, 30, 4]. Such works have focused on evaluating BGPs over triple-based models such as RDF. We extend these works in two ways. First, we show how techniques originally proposed for RDF-like models – specifically the Ring [6] – can be adapted to the property graph model. Second, we further add support for selections, aka. *filters*, which are among the most widely-used query operators in practice:

* Funded by ANID – Millennium Science Initiative Program – Code ICN17_002, and Fondecyt Grant 1260080, ANID, Chile

Bonifati et al. [14] find, for example, that 40% of SPARQL queries in real-world logs use `FILTER`, second only to `SELECT`. While filters can be evaluated over the results of a BGP, one of the most fundamental query optimizations is to "push down" filters, using them to eliminate intermediate results as soon as possible. We propose techniques to enable the push-down of filters by translating them into efficient operations directly over a custom compact data structure.

Paper structure: Section 2 discusses related works on compact data structures and wco joins for graphs. Section 3 presents preliminaries. Section 4 introduces the core technical abstraction of "leaptors": iterators enabling wco execution for graph queries with filters. Section 5 proposes the PG-Ring for the concrete setting of property graphs and GQL-like queries, describing the implementation of a static, in-memory database engine using leaptors. Section 6 presents experimental results for two benchmarks comparing PG-Ring with internal and external baselines. Section 7 wraps up our contributions and presents future work.

2 Related Work

In the context of efficiently evaluating graph patterns, recent years have seen two promising lines of research emerge.

The first line of research refers to *worst-case optimal* (wco) joins [43], which provide – in the context of querying graphs – theoretical and practical guarantees on the maximum cost of enumerating all solutions for a basic graph pattern (BGP). Such algorithms have been shown empirically to provide notable speedups over traditional join algorithms, upholding their theoretical underpinnings for RDF/SPARQL [28, 13, 51] and similar triple-based graph models [36]. However, the benefits of wco joins in terms of time often come at the cost of space [28, 36, 51] as they often require storing additional index orders.

The second line of research explores compressed data structures for indexing and querying graphs. In the RDF/SPARQL world, formats such as HDT [21], COTTAS [3], etc., propose compressed RDF file formats that can evaluate triple patterns, which can be used to build more complex query engines [47].

More recent works have combined both lines of research by investigating succinct data structures that support wco joins. These in-memory structures include the Ring (based on wavelet trees) [6], QDags (based on quadtrees) [7], Hypertries (based on bit tensors) [13], RDF-TDAA (based on Directly Addressable Arrays) [30], and RDFCSA (based on compressed suffix arrays) [4]. These works support wco queries while addressing their key limitation: space.

However, to the best of our knowledge, no work has investigated succinct data structures that support wco joins and filters (beyond equality). Furthermore, we are not aware of works applying wco joins over property graphs.

3 Graph Databases and Worst-Case-Optimal Algorithms

3.1 RDF and Basic Graph Patterns

In the RDF model [16], graph databases are directed labeled multigraphs; edges $s \xrightarrow{p} o$ are also regarded as *triples* (s, p, o) and thus the database is seen as the set of those triples. Node s is called the *subject*, label p the *predicate*, and node o the *object* of the triple. *Basic Graph Patterns (BGPs)* are the core feature of most graph query languages [2], consisting of a set of *triple patterns* (x, y, z), where any component can be a constant or a variable symbol. A *solution* to a BGP is a function that assigns a (node or label) value to each variable of the BGP so that, replacing variables by their values, all the triple patterns become graph triples in the database. *Solving* a BGP consists of enumerating all the solutions.

Let a graph G have N triples and Q be a BGP. We call $Q(G)$ the solution of Q over G, and $Q^* \geq |Q(G)|$ the maximum possible size of a solution of Q over any graph of N triples. An algorithm that solves Q is said to be *worst-case-optimal (wco)* [8, 28] if it works in time $\tilde{O}(Q^*)$, where $\tilde{O}$ hides factors that are polylogarithmic on N, or independent of it (such as $|Q|$). The wco concept and the development of practical wco algorithms have had an enormous impact on the resolution of complex and cyclic BGPs. In particular, wco algorithms outperform every algorithm based on enforcing the triple pattern conditions *consecutively* (as classic database join plans do); they enforce all the conditions *simultaneously*. The classical example is the triangle query $R(a, b) \bowtie S(b, c) \bowtie T(c, a)$, which if the tables have N pairs takes $\Omega(N^2)$ worst-case time with any consecutive-enforcing algorithm, whereas wco algorithms solve it in time $\tilde{O}(N^{3/2})$.

3.2 Leapfrog TrieJoin

Leapfrog Triejoin (LTJ) [49] is a seminal wco join algorithm. It precalculates six tries where the graph triples are inserted as strings of length 3 in all different orders, $\mathcal{T} = \{\mathtt{SPO}, \mathtt{SOP}, \mathtt{POS}, \mathtt{PSO}, \mathtt{OSP}, \mathtt{OPS}\}$. It chooses an order to handle the variables, which is called a *variable elimination order (VEO)*, and then assigns each triple pattern $t \in Q$ to a trie $\tau_t \in \mathcal{T}$, such that (i) the constants of t appear first in the order of τ_t, and (ii) the variables of t appear in τ_t in the same order of the VEO. For each triple pattern $t \in Q$, LTJ first descends by its constants on τ_t, and the node $v_t \in \tau_t$ arrived at is the *locus* of t. Next, LTJ takes one variable x after the other in the VEO. Let $T_x = \{t \in Q,\ x \text{ appears in } t\}$. For each $t \in T_x$, the possible values of x are the children of v_t. LTJ then intersects the children of all the loci $\{v_t, t \in T_x\}$. For each value c in the intersection, it branches with the *binding* $x := c$ and descends by value c from all the involved loci, $v_t \leftarrow child(v_t, c)$ for all $v_t \in T_x$. By the time all the variables are bound in this way, the set of bindings in that branch form a new solution to report. Fig. 1 shows a toy graph and its trie `POS` (ignore property `age` of nodes for now).

A way to implement LTJ intersections is to take the smallest child value c of one locus $v_t \in T_x$, and search the children of the next locus $v_{t'} \in T_x$ for the smallest child value $c' \geq c$; this operation is called *leap*, that is, $c' \leftarrow leap(c)$.

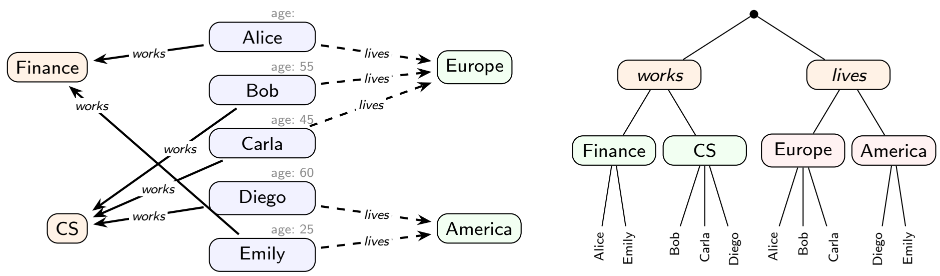

Fig. 1: On the left, a graph database of where people live and area they work in; person nodes have also property `age`. On the right, the trie `POS` for the graph.

We then assign $c \leftarrow c'$ and go on with the third locus $v_{t''} \in T_x$, and so on. We continue cycling over the loci until a whole pass over T_x retains the same c value, giving the next element in the intersection. After branching on $x := c$, we restart the process with the next child value in some loci. This algorithm is optimal in the sense that its cost is $\tilde{O}(alt)$, where alt is the *alternation complexity* [11] of the sets to intersect. This is the least number of switches from one sorted sequence of values to another needed to cover them all; for example intersecting $\{2i, 1 \leq i \leq n\} \cap \{2i-1, 1 \leq i \leq n\} = \emptyset$ is hard because we must switch at every new value ($alt = 2n-1$), whereas intersecting $\{i, 1 \leq i \leq n\} \cap \{i, n < i \leq 2n\} = \emptyset$ is easy ($alt = 1$; it suffices to compare two elements to certify the output). Other intersection techniques used in some LTJ implementations, like probing the elements of the smallest set in the others, do not yield this guarantee.

3.3 Beyond RDF: Filtering by properties and labels

Most graph query languages, like SPARQL [27], GQL [22], and SQL/PGQ [17], have BGPs at their core, but also include filters. GQL and SQL/PGQ assume property graphs [1], which extend triples to allow nodes and edges to have *properties* with *values*: one may specify that some variable x, when assigned to a node or edge in the triple pattern, contains a property π and that its corresponding value $x.\pi$ is within a range, say $a \leq x.\pi \leq b$. It is also customary to assign *labels* to nodes and edges, requiring that variable x is assigned to an element having a label ℓ; let us denote this as $x.\ell$. While SPARQL filters are applied directly on nodes and predicates in triples, we allow zero-to-many values per property and zero-to-many labels, and hence our scheme is also flexible enough to capture features of RDF, as we will discuss in Section 4.3.

While the BGP core of the query can be solved efficiently with a wco algorithm, conditions on properties and labels have been typically handled outside of this wco algorithm by pre- or post-filtering: we can either filter all nodes and edge values v to leave for consideration only those holding $a \leq v.\pi \leq b$ (or where label $v.\ell$ exists), or first solve the BGP and filter every output, removing bindings of v that do not satisfy the conditions. If we use LTJ, a better solution is to adapt it so as to post-filter the values v as soon as they come out of the intersection of loci children, thereby pruning branches earlier.

However, as with non-wco algorithms, these approaches may force us to examine many more solutions than necessary. Consider, for the graph of Fig. 1, the BGP $\{(x, \textsf{\textit{lives}}, \textsf{Europe}), (x, \textsf{\textit{works}}, \textsf{CS})\}$ complemented with the condition $x.\textsf{age} \geq 50$, aimed at finding senior people working in CS in Europe. LTJ will choose two loci, say in `POS` (it could also be `OPS`), by descending through ($\textsf{\textit{lives}}, \textsf{Europe}$) and ($\textsf{\textit{works}}, \textsf{CS}$), respectively. The intersection of the loci children yields all people x working in CS in Europe. If we pre-filter by age on a large real graph of this form, we will generate a lot of senior people, very few of which will end up passing the BGP filter. If we post-filter, we will generate a lot of people working in CS in Europe, very few of which will pass the seniority filter. None of those methods meet the alternation complexity bound, which is never asymptotically larger than the minimum of both sets, and can be much smaller.

4 Leaptors: Incorporating Filters into LTJ

We generalize the logic of LTJ so as to handle not only triple patterns mapped to trie loci, but in general *leaptors* (these extend "iterators", which can only deliver the elements sequentially). A leaptor represents a set S with a total order and supports operation $\mathit{leap}(c) = \min\{c' \in S \cup \{+\infty\},\ c' \geq c\}$, in time polylogarithmic on $|S|$. An example leaptor is the algorithm that finds the child of a trie node (the locus) with the smallest value $c' \geq c$; this can be implemented, say, via exponential search on the increasing sequence of child values.

4.1 Properties and ranges of values

We now define a new leaptor to incorporate filters on property values. Let $\mathcal{U}$ be the set of node/edge identifiers in the graph, and $\mathcal{A}_\pi$ be the set of values a property π takes in the graph. Assume that both sets are integers; if not, we map them to integer ranges $[1, |\mathcal{U}|]$ and $[1, |\mathcal{A}_\pi|]$ while preserving order. For example, by storing an array of all the node/edge values, and another of all the values of $\mathcal{A}_\pi$, we can map from actual values to integer ranges with an $O(\log N)$-time binary search, and can map back any result in constant time.

We now define a *discrete grid* M_π of $[1, |\mathcal{U}|] \times [1, |\mathcal{A}_\pi|]$, and set a point at (v, a) in M_π for every node/edge v having property π with value a. This supports v having zero-to-many values for a property π. Query $\mathit{leap}(c)$ on the leaptor for $a \leq x.\pi \leq b$ is implemented on M_π via the orthogonal range query $\mathit{leftmost}([c, +\infty], [a^+, b^-])$, which finds a point with minimum x-value $c' \geq c$ whose y-value is within $[a^+, b^-]$ ($\mathit{leap}(c)$ returns $+\infty$ if this range is empty). Here, a^+ is the smallest value $a^+ \geq a$ in $\mathcal{A}_\pi$ and b^- is the largest value $b^- \leq b$ in $\mathcal{A}_\pi$; both are found in $O(\log |\mathcal{A}_\pi|) \subseteq O(\log N)$ time via binary searches (done only once per query). The geometric query itself is called an *orthogonal range successor query*, and can be solved in time $O(\log^\epsilon N)$ for any constant $\epsilon > 0$, using space linear in the number of points in M_π [40]. This space is proportional

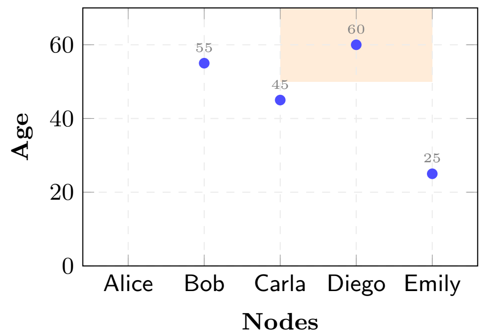

Fig. 2: The grid M_{age} for the graph of Fig. 1 (only showing person nodes). The shadowed area corresponds to the query *leftmost*([Carla, $+\infty$] $\times$ $[50, +\infty]$).

to the database size, which includes one unit of space per value each property has for each node/edge. The leaptor time is then always in $O(\log N)$. See Fig. 2.

Let us return to our example of Section 3.3 to illustrate the power of our method. Apart from the two leaptors implemented on tries for the triple patterns, we have the one implemented with grids for the condition $x.\mathsf{age} \geq 50$. Once a candidate c passes the filters of the two loci, we find the leftmost point (c', a) in $[c, +\infty] \times [a^+, b^-]$. This will yield the next senior person $c' \geq c$, and then the next person working in CS in Europe will be sought after c', not after c. In the example, the first candidate coming out of the intersection of children of trie loci (*lives*, Europe) and (*works*, CS) is Bob. The grid leaptor also returns Bob for the query *leap*(Bob) = *leftmost*([Bob, $+\infty$] $\times$ $[50, +\infty]$), and thus Bob is an output of the query. If we now start from the grid to get the next candidate, its leaptor returns Diego (see the shaded area in Fig. 2), which lets us discard Carla already.

This process achieves the alternation complexity, visiting fewer candidates than pre- or post-filtering, and potentially not fully traversing either set (those passing the BGP and those passing the age filter). For example, if the person ids happen to be clustered by country, occupation, or age, the alternation complexity can be much lower than the number of candidates processed by standard filtering.

4.2 Labels

Property graphs may also permit labels on nodes/edges, and queries may filter for elements having some label ℓ. While labels can be handled as a special property (giving them value 0 when the label exists, and asking for *leftmost*($[c, +\infty] \times [0, 0]$) on their grid), a faster leaptor is possible. Let D_ℓ index the node or edge ids with label ℓ. D_ℓ supports *successor* queries [45], which find the smallest integer $\geq c$ in a set of size N (or $+\infty$ if there is none) in time $O(\log \log N)$, using $O(N)$ space. The leaptor for label ℓ then simply implements $\mathit{leap}(c) = \mathit{successor}(D_\ell, c)$.

It is also generally possible to ask for elements having a property π, independently of its value. The corresponding leaptor can, again, be simulated with query *leftmost*($[c, +\infty] \times [-\infty, +\infty]$) on M_π, but within linear space we can also

SPO order		$L_{\mathtt{SP\underline{O}}}$	OSP order		$L_{\mathtt{OS\underline{P}}}$	POS order		$L_{\mathtt{PO\underline{S}}}$
Alice	*lives*	Europe	America	Diego	*lives*	*lives*	America	Diego
Alice	*works*	Finance	America	Emily	*lives*	*lives*	America	Emily
Bob	*lives*	Europe	CS	Bob	*works*	*lives*	Europe	Alice
Bob	*works*	CS	CS	Carla	*works*	*lives*	Europe	Bob
Carla	*lives*	Europe	CS	Diego	*works*	*lives*	Europe	Carla
Carla	*works*	CS	Europe	Alice	*lives*	*works*	CS	Bob
Diego	*lives*	America	Europe	Bob	*lives*	*works*	CS	Carla
Diego	*works*	CS	Europe	Carla	*lives*	*works*	CS	Diego
Emily	*lives*	America	Finance	Alice	*works*	*works*	Finance	Alice
Emily	*works*	Finance	Finance	Emily	*works*	*works*	Finance	Emily

Fig. 3: A Ring structure on the graph of Fig. 1, assuming the node id order is the lexicographic order of the strings. The Ring only stores the three shaded sequences. The arrows show how the triple (`Alice`, *lives*, `Europe`) is tracked across the three orders. The rectangle on `POS`, corresponding to the child of the trie root by *works*, descends by `Carla` to the rectangle on `SPO`.

store a successor data structure D_π with the node or edge ids having some value for property π, and run $leap(c)$ in time $O(\log\log N)$.

4.3 Extending the Ring Index

The Ring [6] is a *succinct* index that handles BGPs in wco time on RDF graphs. An index is said to be succinct if, including the data, it uses $\mathcal{S} + o(\mathcal{S})$ space, where $\mathcal{S}$ is the size of a plain representation of the data. The Ring represents the set of triples (s, p, o) with three sequences, $L_{\mathtt{SP\underline{O}}}$, $L_{\mathtt{OS\underline{P}}}$, and $L_{\mathtt{PO\underline{S}}}$. Sequence $L_{\mathtt{SP\underline{O}}}$ stores the objects of the N triples sorted by the order `SPO`, that is, triples (s, p, o) are sorted lexicographically and $L_{\mathtt{SP\underline{O}}}$ collects the sequence of their o components. Analogously, sequence $L_{\mathtt{OS\underline{P}}}$ collects the N predicates p in the order `OSP` and sequence $L_{\mathtt{PO\underline{S}}}$ collects the N subjects s in the order `POS`. Overall, letting $\mathcal{U}$ be the universe of nodes and predicates, the Ring uses $3N \log_2 |\mathcal{U}|(1 + o(1))$ bits, which is the space needed to store the N triples (s, p, o) in plain form, plus a sublinear redundant space. See Fig. 3.

The sequences $L_*[1, N]$ are represented with a data structure called a *wavelet tree* [26, 38], which uses $N \log_2 |\mathcal{U}|(1 + o(1))$ bits and in particular implements the operation $rank_c(L_*, i)$, the number of times c occurs in $L_*[1, i]$, in time $O(\log |\mathcal{U}|)$. We also represent arrays $A_{\mathtt{O}}$, $A_{\mathtt{P}}$, and $A_{\mathtt{S}}$, where $A_*[1, |\mathcal{U}|]$ holds in $A_*[e]$ the number of positions in L_* with values less than e. Those arrays can be represented with sparse bitvectors $B_*[1, N]$ where $B_*[A_*[e]] = 1$ for all e and 0 otherwise, so $A_*[e] = select_1(B_*, e)$, where $select_b(B_*, e)$ is the position of the eth bit b in B_*. Sparse bitvectors [44, 18, 20] use $|\mathcal{U}| \log_2(N/|U|) + 2|\mathcal{U}| \in O(N)$ bits, which fits within the sublinear extra space of the Ring, and answer $select_1$ queries in constant time.

With those structures we can *track* triples across the arrays L_*: the triple at $L_{\mathtt{SP\underline{O}}}[i]$ has object $o = L_{\mathtt{SP\underline{O}}}[i]$, and it corresponds to $L_{\mathtt{OS\underline{P}}}[i']$, where $i' = A_{\mathtt{O}}[o] +$

$rank_o(L_{\mathtt{SP\underline{O}}}, i)$. The triple then has predicate $p = L_{\mathtt{OS\underline{P}}}[i']$, and it corresponds to $L_{\mathtt{PO\underline{S}}}[i'']$, where $i'' = A_{\mathtt{P}}[p] + rank_p(L_{\mathtt{OS\underline{P}}}, i')$. The subject of the triple is then $s = L_{\mathtt{PO\underline{S}}}[i'']$ and if we compute $A_{\mathtt{S}}[s] + rank_s(L_{\mathtt{PO\underline{S}}}, i'')$ we return to $L_{\mathtt{SP\underline{O}}}[i]$.

Trie nodes correspond to ranges in some sequence L_*. A locus in SOP or SPO, reached by descending by s, corresponds to the range $L_{\mathtt{SP\underline{O}}}[i, j] = L_{\mathtt{SP\underline{O}}}[A_{\mathtt{S}}[s] + 1, A_{\mathtt{S}}[s + 1]]$ of the triples $(s, *, *)$. If we now descend by o in SOP, the resulting locus corresponds to range $(s, o, *)$ in the order SOP, which we do not represent, but also to range $(o, s, *)$ in the order OSP, which we do represent in $L_{\mathtt{OS\underline{P}}}$. This is $L_{\mathtt{OS\underline{P}}}[i', j']$, where $i' = A_{\mathtt{O}}[o]+rank_o(L_{\mathtt{SP\underline{O}}}, i-1)+1$ and $j' = A_{\mathtt{O}}[o]+rank_o(L_{\mathtt{SP\underline{O}}}, j)$. If, instead, we descend by p after s in SPO, we stay in the same sequence $L_{\mathtt{SP\underline{O}}}$, which represents that trie, now restricting the range to the triples $(s, p, *)$. The new range is computed by descending from the root of POS with p, and from the resulting range $L_{\mathtt{PO\underline{S}}}[i'', j'']$ descending by s to $L_{\mathtt{SP\underline{O}}}[i''', j''']$, with $i''' = A_{\mathtt{S}}[s] + rank_s(L_{\mathtt{PO\underline{S}}}, i'' - 1) + 1$ and $j''' = A_{\mathtt{S}}[s] + rank_s(L_{\mathtt{PO\underline{S}}}, j'')$. We will still call those operations "descending" by a value, even if they are mapped to ranges in L_*. See again Fig. 3 for examples of tracking and descending.

Descending operations allow the Ring to simulate the LTJ algorithm by mapping loci to ranges in L_*. The Ring leaptor is implemented as follows. Say we are again in $L_{\mathtt{SP\underline{O}}}[i, j]$ after descending by s in trie SOP. To implement $leap(c)$ on the values of o, we must find the smallest value in $L_{\mathtt{SP\underline{O}}}[i, j]$ that is $\geq c$. This operation, called *range next value*, is supported in $O(\log |\mathcal{U}|)$ time by the wavelet tree [24]. If, instead, we are in trie SPO and want to find the smallest value of p that is $\geq c$, we start at range $L_{\mathtt{PO\underline{S}}}[A_{\mathtt{P}}[c] + 1, N]$ (the range of all predicates $\geq p$ in order POS), and compute $L_{\mathtt{SP\underline{O}}}[i'] = A_{\mathtt{S}}[s] + rank_s(L_{\mathtt{PO\underline{S}}}, A_{\mathtt{P}}[c]) + 1$. Tracking $L_{\mathtt{SP\underline{O}}}[i']$ to $L_{\mathtt{OS\underline{P}}}$ yields the desired value of p. The Ring then implements the leaptors in time $O(\log |\mathcal{U}|)$, yielding a total query time complexity of $O(Q^* \cdot |Q| \log |\mathcal{U}|) \subseteq \tilde{O}(Q^*)$.

Incorporating properties and labels. We add to the Ring succinct representations of grids M_π that use $N_\pi \log_2 |\mathcal{A}_\pi|(1 + o(1))$ bits, where N_π is the sum of the number of values property π has over all nodes. This is again the size of the data plus a sublinear term. Such a succinct representation is the wavelet tree of the sequence S_π of values property π takes, sorted by increasing identifiers of the nodes. We also represent a bitvector $B_\pi[|\mathcal{U}| + N_\pi]$ where, for each node identifier v, we append a 1 and then as many 0s as values v has for property π; note that positions in S_π align with 0s in B_π. The grid of Fig. 2, for example, would be represented with the sequence $S_{\mathsf{age}} = \langle 55, 45, 60, 25 \rangle$ and the bitvector $B_{\mathsf{age}} = 110101010$ (considering only the node ids that appear in the figure).

For B_π we use a dense bitvector representation, which uses $(|\mathcal{U}| + N_\pi)(1 + o(1))$ bits of space (thus fitting within the sublinear extra space of the index) and supports queries $select_b$ in constant time [15, 37]. We thus compute with $c' \leftarrow select_1(B_\pi, c) - c + 1$ the first position in S_π of node identifiers $\geq c$. The wavelet tree then solves the orthogonal range successor query from $S_\pi[c'..]$, in time $O(\log |\mathcal{U}|)$ [10]. The answer c'' is finally converted back to an identifier with $select_0(B_\pi, c'') - c''$. In our example, to compute $leftmost([\mathsf{Carla}, +\infty] \times [50, +\infty])$ on M_{age}, where $\mathsf{Carla} = 3$, we start with $c' \leftarrow select_1(B_{\mathsf{age}}, 3) - 3 + 1 = 2$, so we find the first value ≥ 50 on $S_{\mathsf{age}}[2..]$. This is at $S_{\mathsf{age}}[3] = 60$. We finally convert that position, $c'' = 3$, back to the identifier $\mathit{select}_0(B_{\mathsf{age}}, 3) - 3 = 4 = \mathsf{Diego}$.

Queries on labels are handled with generic successor data structures in Section 4.2. While using linear space, such a representation is not succinct: If a label ℓ is assigned to m elements out of n, a succinct representation uses space close to the bits required to specify which are those m elements, that is, $\log_2 \binom{n}{m} = m \log \frac{n}{m} + O(m)$ bits. We thus use instead a sparse bitvector B_ℓ for each label ℓ, with one bit per node, marking with a 1 the elements having label ℓ. A sparse bitvector [44, 18, 20] represents n bits, m of which are 1, using at most $m \log \frac{n}{m} + 2m$ bits, and provides operations rank_b and select_0 in time $O(\log m)$ and (as already mentioned) select_1 in constant time. Operation $\mathit{leap}(c)$ for the condition $x.\ell$ is then implemented as $\mathit{select}_1(B_\ell, \mathit{rank}_1(B_\ell, c-1)+1)$ with this leaptor, in time $O(\log m) \subseteq O(\log N)$. This representation also provides a leaptor for the next element *not* having a label ℓ, with $\mathit{leap}(c) = \mathit{select}_0(B_\ell, \mathit{rank}_0(B_\ell, c-1)+1)$, also in time $O(\log m)$.

Section 5.2 describes a similar mechanism for edge properties and labels. This extended Ring is succinct on the triples, properties and labels it represents. It implements LTJ and supports leaptors for filtering. If we count the filters on property values and Boolean expressions on labels as part of the query size $|Q|$, the overhead for query time stays $O(|Q| \log |\mathcal{U}|) \subseteq \tilde{O}(1)$.

RDF model. Node properties π can be expressed in RDF by creating nodes with all the literal values a that property π can yield, and represent $v.\pi = a$ with a triple (v, π, a). Labels can be given a reserved property $\mathsf{\mathit{lab}}$. Conditions like $x.\pi = a$ can then be expressed as a triple pattern (x, π, a), whereas range conditions like $a \leq x.\pi \leq b$ can be translated into $\{(x, \pi, y), a \leq y \leq b\}$ with variables x and y. The second clause is a range condition on (literal) node identifiers, which can be implemented by ensuring that node identifiers respect the order of the literals and using a leaptor for y that defines $\mathit{leap}(c) = \max(a, c)$ if $c \leq b$ and $+\infty$ otherwise. Properties of edges $u \xrightarrow{p} v$ can be modeled by reification: create a node e representing the edge, with properties $e.\mathit{from} = u$, $e.\mathit{to} = v$, $e.\mathit{label} = p$, set further properties $e.\pi = a$ on the edge, and express them all as RDF triples.

While correct, a wco solution to those translated range conditions is less efficient than the one we described, because they filter the variables *in sequence*, not *simultaneously*. Consider the clauses $\{(x, \pi, y), a \leq y \leq b\}$. If x is eliminated before y, LTJ will produce all the possible values of x, and only later, when y is eliminated, it will filter those values of x for which $a \leq x.\pi \leq b$, *one by one*. If y is eliminated before x, LTJ will bound it to *every* literal node within $[a, b]$, and only later, when x is eliminated, connect the y values, *one by one*, to actual values of x by label π. Both strategies correspond to what we have exemplified as pre- or post-filtering; our strategy using grids M_π performs the filtering *simultaneously* for the range values and the other conditions on x.

Rather than represent properties and labels in RDF, a dual idea is to represent RDF triples with properties and labels in order to support more efficient simultaneous filtering using grids in the RDF setting. For example, RDF triples of the type (x, p, y), where p is a datatype-property and y is a literal, could be represented as properties on x ($x.p = y$), while triples of the form (x, *type*, y) could be represented as labels on x ($x.\mathit{lab} = y$). This is feasible since our model supports zero-or-more labels and property values per node. However, our model would require some amendments to cover hybrid object/datatype properties (RDF properties with a mix of datatype and node values), joins on the predicate position, and other such details, as allowed in the base RDF/SPARQL case.

5 Property Graphs

In the previous section we described techniques for efficiently handling filters on properties and labels. We now apply these techniques for a concrete data model and query language, namely property graphs and a fragment of GQL. Some details of this concrete setting (e.g., edge identifiers, Boolean label expressions) require additional care, while others (e.g., one edge label, at most one value for a property on a given node or edge) provide opportunities to optimize further.

5.1 Data model and query language

A *property graph* is a directed labeled multigraph where any node or edge may feature a set of property-value pairs; properties have a unique value per node/edge. In addition, an edge is annotated with one label, and a node with zero or more labels. Unlike in RDF, edges have an identity beyond their label (e.g., there can be two edges with the same label connecting the same nodes).

Let $\mathcal{G}(\mathcal{N}, \mathcal{E})$ be a property graph, with $\mathcal{N} \cap \mathcal{E} = \emptyset$. Let $\mathcal{P}$ be the set of properties associated with any node or edge, and let properties have atomic values in $\mathcal{A}$ (e.g., integers, dates, strings). The nodes have zero or more labels in $\mathcal{L}_N$ and the edges have exactly one label in $\mathcal{L}_E$, with $\mathcal{L}_N \cap \mathcal{L}_E = \emptyset$.

Our graph of Fig. 1 is a property graph, with node property age and edge labels *lives* and *works*. Possible edge properties could be, for example, a date since (for edges with label *lives*) or an integer salary (for edges with label *works*). Possible labels for nodes (representing persons) could be *person* and *has-phd*.

Two main standards exist to query property graphs: SQL/PGQ and GQL [22]; their common graph pattern matching language is called GPML [17]. A GPML query is of the form `MATCH pattern`, where `pattern` specifies the subgraph to match as a set of *edge patterns*, optionally augmented with a `WHERE` clause to filter the results. We now describe the fragment we support [17, 1, 22].

The most basic edge patterns are of the form (`s`) $\rightarrow$ (`o`), where `s` and `o` can be constants in $\mathcal{N}$ or node variables from a set $\mathcal{V}_N$. Those are analogous to triple patterns but do not specify any label. Conditions on labels are associated with node variables, for example we can add a clause `(x:person)` to specify that $x \in \mathcal{V}_N$ can match only nodes having the label *person* $\in \mathcal{L}_N$. We can include these clauses in edge patterns, for example in `(x:person)` $\rightarrow$ `(y)`. Label conditions may consist of expressions including conjunctions (`&`), disjunctions (`|`), negations (`!`), and groups of labels. For example, a clause `(x:person&has-phd)` admits the nodes with labels *person* and *has-phd*.

Label conditions can also be given on edge patterns, as in $(\mathtt{s}) - [\mathtt{:works}] \rightarrow (\mathtt{o})$, which allows matching only edges whose label is *works* $\in \mathcal{L}_E$; expressions on labels can also be used. It is also possible to give variable names, from a set $\mathcal{V}_E$ disjoint from $\mathcal{V}_N$, to edges: $(\mathtt{s}) - [\mathtt{e}] \rightarrow (\mathtt{o})$ gives variable name $e \in \mathcal{V}_E$ to the edge; we can also incorporate expressions as in $(\mathtt{s}) - [\mathtt{e} : \mathtt{works}] \rightarrow (\mathtt{o})$.

The `WHERE` clause filters the results by restricting the existence or the values of some of their properties or by comparing the identifiers of edges/nodes (yet we cannot compare edge variables with node variables). Conditions can be combined with `AND`, `OR`, and `NOT`. Comparisons are of the form `(A op B)`, where `A` and `B` are constants or variable property values $x.\pi$ of the same type, and $\mathtt{op} \in \{=, \neq, <, \leq, \geq, >\}$. We can directly compare node or edge identifiers as if they were properties, as in `(x<=y) AND (y<=z)`, which avoids reporting the same tuples (x, y, z) in every possible order. One can also state that a variable must have, or not have, some property using `(x.since IS NOT NULL)` or `(x.since IS NULL)`.

A final `RETURN` clause indicates which variable values, or their property values, to return from each solution. This implies a projection of the tuples returned in the standard BGPs. Under *bag semantics* repetitions resulting from projections are not removed and the query algorithms can still be wco.

The following query, for example, looks for names and ages of senior people working in CS in Europe that started in their job before 2010 or have no phd:

```
MATCH (x:person)-[:lives]->(Europe), (x)-[e:works]->(CS)
WHERE (x.age >= 50) AND ((e.since < 01/01/2010) OR (!x.has-phd))
RETURN x.name, x.age
```

5.2 PG-Ring: A succinct index for property graphs

We build on the Ring extensions described in Section 4.3 to define PG-Ring, which indexes property graphs within succinct space.

Since the edges of property graphs have exactly one label in $\mathcal{L}_N$, we treat edges $e = s \rightarrow o$ as triples $(s, \lambda(e), o)$ in order to create the sequences L_*. The symbols of $L_{\mathtt{PO\underline{S}}}$ and $L_{\mathtt{SP\underline{O}}}$ then range over $\mathcal{N}$ and those of $L_{\mathtt{OS\underline{P}}}$ range over $\mathcal{L}_E$.

Storing the additional edge identifiers of property graphs would exceed the data size budget, so we rather exploit the fact that such identifiers are internal by defining the identifier of an edge as its index in the order `POS`. We choose this index order since all edge identifiers with some label ℓ are precisely in the range $[A_{\mathtt{P}}[\ell]+1, A_{\mathtt{P}}[\ell+1]]$, and edge patterns of the form $(\mathtt{s}) - [: \ell] \rightarrow (\mathtt{o})$ are treated as a constant predicate in a triple pattern, instead of using a leaptor.

Boolean expressions L on labels, $(\mathtt{s}) - [: L] \rightarrow (\mathtt{o})$, are converted into a set of $O(L)$ disjoint ranges: the complement of ℓ becomes $\{[1, A_{\mathtt{P}}[\ell]], [A_{\mathtt{P}}[\ell+1]+1, N]\}$, disjunction becomes union of ranges, and conjunction becomes intersection. Though we process a set of disjoint ranges during query processing, for simplicity we assume a single range in what follows.

Leaptors for edge variables. Variables in $\mathcal{V}_E$ can be assigned to edge identifiers. Those will be handled like any other variable in LTJ, with the particularity that

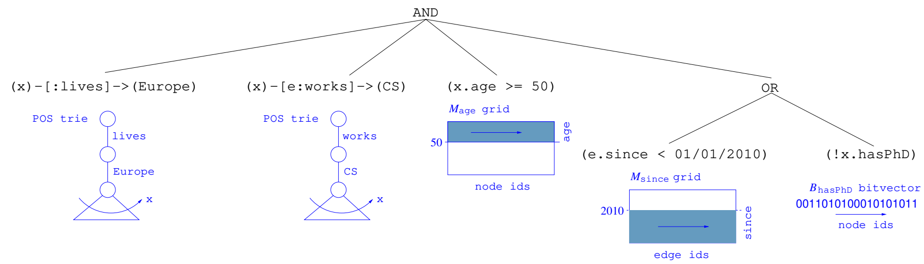

Fig. 4: The syntax tree of our example of Section 5.1. Each internal node implements a Boolean leaptor; leaf nodes implement trie, grid, and bitvector leaptors.

their identifiers are not explicit in any sequence L_*, but implicit as indices in $L_{\underline{\mathtt{POS}}}$ (i.e., on order `POS`). The leaptor for a variable $z \in \mathcal{V}_E$ – which is mentioned on an edge pattern t represented by a trie $\tau_t \in T_z$ – will thus implement $leap(c)$ differently depending on which sequence L_* holds the range that represents the locus v_t. See Appendix A for details [9].

Leaptors for node properties and labels. Since now each property π defined for a node v has exactly one value $v.\pi$, we can slightly simplify the scheme of Section 4.3, storing bitvectors $B_\pi[1, |\mathcal{N}|]$ telling at $B_\pi[v]$ whether node v has property π (in the example of Fig. 2, $B_{\mathsf{age}} = 01111$). If so, the wavelet tree sequence S_π that represents M_π has value $S_\pi[rank_1(B_\pi, v)] = v.\pi$, and the first position of S_π with node identifier $\geq c$ is $rank_1(B_\pi, c-1)+1$. See Appendix A for details [9].

The general case: Boolean leaptors. The `WHERE` clause features a general Boolean expression, not just a conjunction. We use the same syntax tree mechanism to define a *Boolean leaptor* for the whole expression. The use of a syntax tree generalizes the set intersections discussed so far by also allowing disjunction nodes, which leap by taking the least of the leap values of both children. Negations must be removed by pushing them to the leaves and applying them to the atomic condition, both for properties (e.g., `NOT (x < y)` becomes `(x >= y)`) and for labels (e.g. `NOT (x.person)` becomes `(!x.person)`). We further integrate into the syntax tree the edge patterns of the `MATCH` clause as a conjunction with those of the `WHERE` clause. See Fig. 4, and Appendix A for more details [9].

6 Experiments

We implement PG-Ring in C++11 on top of the succinct data structures library, *SDSL*, as described in more detail in Appendix B [9]. We compare it with various state-of-the-art alternatives that support property graphs and GQL-like languages for two different benchmarks: LSQB and Wikidata. We run a set of queries with a time limit of 600 seconds. Following the LSQB benchmark [35], for each query, we report the average runtime over 5 runs. That measurement is

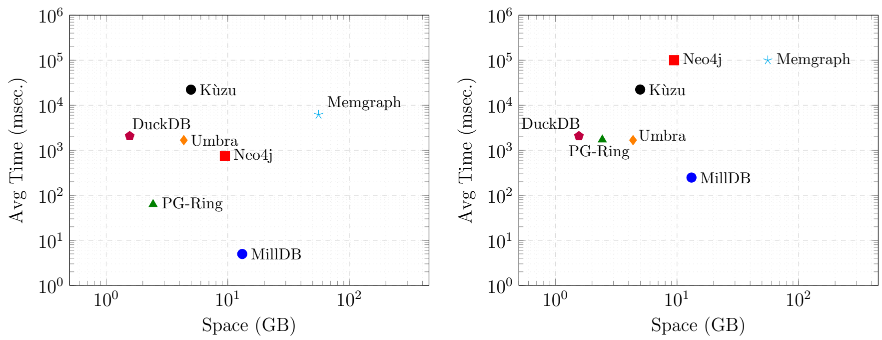

Fig. 5: The space-time trade-off of the evaluated systems on the six queries of LSQB. The first plot is limited to 10 results, and the second to 1,000.

computed after a first warm-up to mitigate the advantage of in-memory systems. Disk activity was monitored using `iostat` at one-second intervals. In LSQB, a single pass over the queries was enough to avoid disk reads, whereas in Wikidata, additional warm-up queries (e.g., reporting all the nodes) were required beforehand. For both benchmarks, the effect on time performance of a *cold cache* scenario is reported in Appendix F [9]. All experiments were conducted on an AMD Ryzen 9 9900X at 4.4GHz, with 24 cores, 32 MB cache, and 249 GB RAM.

6.1 Labelled Subgraph Query Benchmark

We used the Labelled Subgraph Query Benchmark (LSQB) [35] to generate a graph with 35.5 million nodes and 219.4 million edges. Each node can be given one of 14 possible labels, whereas each edge can be given one of 15 possible labels. In this benchmark, nodes and edges do not have properties. We also choose six queries to evaluate from the benchmark. See Appendix C [9] for further details.

We compare systems supporting GQL (details in Appendix D [9]): Umbra [42], DuckDB [46], Kùzu [29], Neo4j [41], Memgraph [34], MillDB [51], and our PG-Ring. We run the six queries on all the systems, limiting results to 10 and 1,000. Appendix C shows detailed results [9]; Fig. 5 summarizes them.

The left plot shows the space and average time of each system with the number of results limited to 10. We can see that PG-Ring achieves a good balance between space and time. In space, PG-Ring is dominated only by DuckDB, which is 1.6 times smaller due to its columnar compression, but its average times are 12−2,700 times slower (see Appendix C [9]). The next competitor in space is Umbra, which uses 1.8 times more space than PG-Ring. In time performance, the PG-Ring wins in three cyclic queries (Q2, Q3, and Q6, see Appendix C [9]), showing the effect of using the LTJ algorithm to solve these queries. In the other queries, the clear winner is MillDB. In fact, MillDB is very competitive in all queries, but it requires around 5.4 times more space than PG-Ring.

The right plot shows the same evaluation when restricting the results to 1,000. In this scenario, PG-Ring performs competitively on cyclic queries, with

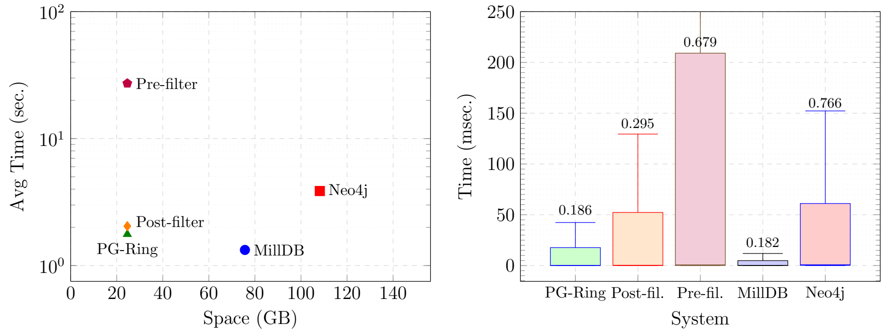

Fig. 6: On the left, the space-time trade-off of the evaluated systems. On the right, boxplots represent the distribution of query execution times. The numbers on top indicate the median time for each system.

the exception of Q2, where Neo4j performs best. In contrast, Neo4j shows weaker performance in Q3, where it exceeds the time limit, and Q6, where it is 11 times slower. In Q1, Memgraph stands out by being 1.8 times faster than MillDB, but it is the most space demanding structure and times out in Q3. For all remaining queries, MillDB is the top performer, taking less than 4 milliseconds per query. The plot shows that DuckDB and Umbra fare better for obtaining many results.

In conclusion, PG-Ring offers a good space-time trade-off, it is highly competitive for cyclic queries, highlighting the impact of the wco join algorithm; and ranks second in space usage thanks to the use of compact data structures.

6.2 Wikidata Benchmark

We use the Wikidata knowledge graph [50] to evaluate our performance in a large scale graph with properties (via qualifiers). The Wikidata graph is represented using the RDF model; therefore, we transformed it into a property graph. The resulting graph consists of 109 million nodes, which can take 103 thousand distinct labels and have 10,569 different properties. In total, there are 719.5 million edges, with 1,640 possible edge labels and 603 distinct properties. From the Wikidata SPARQL query logs, we extracted a set of 19.6 thousand queries and converted them to GQL syntax. See Appendix E for details [9].

We exclude Umbra [42], DuckDB [46], and Kùzu [29] because Wikidata nodes can have multiple labels (classes), but these systems assume a single type/table per entity. Memgraph [34] was also excluded due to its high memory usage and the impracticality of loading the Wikidata graph. In Neo4j [41], we create indexes on the entity identifiers and on the properties used in the queries.

To study the effect of incorporating the filters directly to the LTJ algorithm, we include two variants of PG-Ring: The *pre-filter* baseline first applies the property and label filters, and only then evaluates the edge patterns. The *post-filter* baseline evaluates the edge patterns first and subsequently applies filters.

Fig. 6 shows the performance in space and time of different systems on the selected queries, limiting the output to 1,000,000 results. PG-Ring achieves the most favorable space-time trade-off: it requires only 32% of the storage used by MillDB, which is on average just 32% faster than PG-Ring. MillDB shows, however, more stability across all queries, with its third quartile being at around 5 milliseconds, while the other systems exceed 10 milliseconds. Conversely, Neo4j is slower than PG-Ring and consumes significantly more space, exceeding 100 GB: in a real-world scenario Neo4j will consume much more space because we are restricting the indexes to just those properties involved in the queries.

The experiment also shows that pushing the filters directly into the LTJ algorithm leads to superior performance. On average, PG-Ring achieves a 16% improvement over the *post-filter* baseline and is 15 times faster than *pre-filter*. The boxplots show a more marked difference in distributions, for example PG-Ring is almost 60% faster than *post-filter* in the median, and much more stable in general. As an example to explain this difference, consider one of the queries:

```
MATCH (v:Q5)-[e:P21]->(u)
WHERE (v.P569 >= 1554-01-01) AND (v.P569 < 1555-01-01)
RETURN v, e, u
```

which gets *the gender (`P21`) of humans (`Q5`) born in (`P569`) 1554*, for which PG-Ring achieves a speedup over 5,000 times compared to the *post-filter* baseline, which needs to check for every human if they were born in 1554. Conversely, the *pre-filter* is twice as slow as PG-Ring because it first retrieves all entities born in 1554 and only then verifies whether they are humans. Note that Wikidata can also store birth dates for non-human entities (e.g., fictional characters, animals).

7 Conclusions

We proposed succinct data structures that support wco joins and filters. We show the advantages of pushing the filtering within the wco processing of the joins, thereby uplifting the superpowers of those join processing algorithms, and outperforming the classic pre- and post-filtration methods that handle the wco joins as a black box. We adapt and optimize these structures for the concrete setting of evaluating the corresponding fragment of GQL queries over property graphs, giving rise to PG-Ring. Experiments on two property graphs show that although existing systems sometimes outperform PG-Ring in terms of time (MillDB), or space (DuckDB), PG-Ring provides a strong time–space trade-off in this setting.

For future work, we plan to implement PG-Ring for RDF/SPARQL, optimize it specifically for this setting, and compare it with leading systems. Though the Ring supports this setting, an interesting question arises from when it is preferable to represent (e.g.) datatype values as property values versus graph nodes. We also wish to explore pushing further query operators from GQL, SPARQL, etc., into low-level operations on the succinct data structures.

A limitation of PG-Ring is that it is currently read-only, but it can be adapted to support updates. The original Ring article [6, Sec. 8 & App. E] discusses in detail how to handle changes on nodes and edges. The PG-Ring represents property values and labels via wavelet trees and bitvectors, for which dynamic versions exist [31] while retaining succinctness and enabling updates in time $O(\log^2 N)$, but at the cost of an $O(\log N)$ time penalty factor on queries. This can be alleviated with adaptive dynamic bitvectors [39], a recent development that supports dynamism with a much smaller penalty when the update-to-query ratio is very low, something proven to work well for Wikidata-style workloads [5].

Supplemental Material Statement: Source code and query sets are available from Github at https://anonymous.4open.science/r/PG-Ring-B1F8. The datasets are available from Zenodo at https://zenodo.org/records/19818050. Appendices are submitted as an extended version on EasyChair, and will be published on arXiv.

Declaration of use of Generative AI: Generative AI was used only as a support tool for code completion, for assisting with the parsing and preprocessing of some data, and to produce Figs. 1–3 in tikz from textual descriptions. All suggestions were reviewed, tested, and adapted before being included in the final work.

References

1. R. Angles. The Property Graph Database Model. In *Proc. Alberto Mendelzon Workshop (AMW)*, 2018.
2. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40, 2017.
3. J. Arenas-Guerrero and S. Ferrada. COTTAS: columnar triple table storage for efficient and compressed RDF management. In *Proc. 24th International Semantic Web Conference (ISWC), part II*, pages 313–331, 2025.
4. D. Arroyuelo, F. Barisione, A. Fariña, A. Gómez-Brandón, and G. Navarro. New compressed indices for multijoins on graph databases. *Information Systems*, 137:article 102647, 2026.
5. D. Arroyuelo, D. Campos, A. Gómez-Brandón, Y. Linker, G. Navarro, C. Rojas, and D. Vrgoc. CompactLTJ: Space & time efficient Leapfrog Triejoin on graph databases. *The Very Large Databases Journal*, 34:article 67, 2025.
6. D. Arroyuelo, A. Gómez-Brandón, A. Hogan, G. Navarro, J. Reutter, J. Rojas-Ledesma, and A. Soto. The Ring: Worst-case optimal joins in graph databases using (almost) no extra space. *ACM Transactions on Database Systems*, 49(2):1–45, 2024.
7. D. Arroyuelo, G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal joins using compressed quadtrees. *ACM Transactions on Database Systems*, 47(2):8:1–8:53, 2022.
8. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
9. Anonymous Authors. Extended Version (with Appendices A–F), Uplifting the Superpowers of Worst-Case-Optimal Join Algorithms. Uploaded to EasyChair for review, to be published on arXiv, 2026.
10. J. Barbay, F. Claude, and G. Navarro. Compact binary relation representations with rich functionality. *Information and Computation*, 232:19–37, 2013.

11. J. Barbay and C. Kenyon. Deterministic algorithm for the t-threshold set problem. In *Proc. 14th ISAAC*, pages 575–584, 2003.
12. B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, S. Rondelli, A. Ryazanov, M. Schmidt, K. Sengupta, B. B. Thompson, D. Vaidya, and S. Wang. Amazon Neptune: Graph Data Management in the Cloud. In *Proc. ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks*, 2018.
13. A. Bigerl, F. Conrads, C. Behning, M. Ahmed Sherif, M. Saleem, and A.-C. Ngonga Ngomo. Tentris - a tensor-based triple store. In *Proc. 19th International Semantic Web Conference (ISWC), part I*, pages 56–73, 2020.
14. A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *The VLDB Journal*, 29(2-3):655–679, 2020.
15. D. R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.
16. R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, February 2014. https://www.w3.org/TR/rdf11-concepts/.
17. A. Deutsch, N. Francis, A. Green, K. W. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke. Graph pattern matching in GQL and SQL/PGQ. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 2246–2258, 2022.
18. P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974.
19. O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 619–630, 2015.
20. R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.
21. Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutierrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *Journal of Web Semantics*, 19:22–41, 2013.
22. N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoc. A Researcher's Digest of GQL. In *Proc. 26th International Conference on Database Theory (ICDT)*, pages 1:1–1:22, 2023.
23. N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proc. International Conference on Management of Data (SIGMOD)*, pages 1433–1445, 2018.
24. T. Gagie, G. Navarro, and S.J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25–41, 2012.
25. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA)*, pages 326–337, 2014.
26. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
27. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, March 2013. https://www.w3.org/TR/sparql11-query/.

28. A. Hogan, C. Riveros, C. Rojas, and A. Soto. A worst-case optimal join algorithm for SPARQL. In *Proc. 18th International Semantic Web Conference (ISWC)*, pages 258–275, 2019.
29. G. Jin, X. Feng, Z. Chen, C. Liu, and S. Salihoglu. Kùzu graph database management system. In *Proc. 13th Conference on Innovative Data Systems Research (CIDR)*, 2023.
30. Y. Liu, Y. Lin, X. Peng, Y. Li, and J. Zhang. RDF-TDAA: Optimizing RDF indexing and querying with a trie based on directly addressable arrays and a path-based strategy. *Expert Systems and Applications*, 279:article 127384, 2025.
31. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.
32. S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of Wikidata: Semantic technology usage in Wikipedia's Knowledge Graph. In *Proc. 17th International Semantic Web Conference (ISWC), part II*, pages 376–394, 2018.
33. M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro. Practical compressed string dictionaries. *Information Systems*, 56:73–108, 2016.
34. Memgraph Ltd. Memgraph: An in-memory graph database, 2024.
35. A. Mhedhbi, M. Lissandrini, L. Kuiper, J. Waudby, and G. Szárnyas. LSQB: A large-scale subgraph query benchmark. In *Proc. 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–11, 2021.
36. A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endowment*, 12(11):1692–1704, 2019.
37. J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.
38. G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.
39. G. Navarro. Practical adaptive dynamic bitvectors. *Software Practice and Experience*, 55(9):1539–1559, 2025.
40. Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT)*, LNCS 7357, pages 271–282, 2012.
41. Neo4j, Inc. Neo4j graph database management system, 2024.
42. T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *Proc. 10th Conference on Innovative Data Systems Research, (CIDR)*, 2020.
43. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *Journal of the ACM*, 65(3):16:1–16:40, 2018.
44. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.
45. M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.
46. M. Raasveldt and H. Mühleisen. DuckDB: An embeddable analytical database. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 1981–1984, 2019.
47. R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh. Comunica: A modular SPARQL query engine for the web. In *Proc. 17th International Semantic Web Conference (ISWC), part II*, pages 239–255, 2018.
48. W. van Leeuwen, T. Mulder, B. Van De Wall, G. Fletcher, and N. Yakovets. Avantgraph query processing engine. *Proc. VLDB Endowment*, 15(12):3698–3701, 2022.

49. T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 96–106, 2014.
50. D. Vrandecic and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.
51. D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil-Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero. MillenniumDB: An open-source graph database system. *Data Intelligence*, 5(3):560–610, 2023.
52. J. Webber. A programmatic introduction to Neo4j. In *Proc. ACM Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 217–218. ACM, 2012.

A PG-Ring: Additional Details

A.1 Leaptors for edge variables

We implement *leap*(c) differently depending on which sequence L_* holds the range that represents the locus v_t:

– $L_{\texttt{PO}\underline{\texttt{S}}}[i,j]$: this is the easiest case because the order POS coincides with that of the edge identifiers. We simply return the first element of $[i,j] \cap [c,\infty]$, that is, $\max(i,c)$ if it is $\leq j$, or else $+\infty$.
– $L_{\texttt{SP}\underline{\texttt{O}}}[i,j]$: this case occurs when a value s was previously used to descend from a range $L_{\texttt{PO}\underline{\texttt{S}}}[i',j']$. We then restrict the range in $L_{\texttt{SP}\underline{\texttt{O}}}$ by starting from range $[i',j'] \cap [c,N]$ in $L_{\texttt{PO}\underline{\texttt{S}}}$ and descending by s towards $L_{\texttt{SP}\underline{\texttt{O}}}[i'',j'']$, as described in Section 4.3 when binding p after s. If $i'' > j''$ (i.e., the range is empty) we return $+\infty$, otherwise we track $L_{\texttt{SP}\underline{\texttt{O}}}[i'']$ to $L_{\texttt{OS}\underline{\texttt{P}}}$ as also described in Section 4.3 and return the resulting position.
– $L_{\texttt{OS}\underline{\texttt{P}}}[i,j]$: there are two subcases.
  - If both s and o were bound, we had been in a range $L_{\texttt{PO}\underline{\texttt{S}}}[i',j']$, descended by s to $L_{\texttt{SP}\underline{\texttt{O}}}[i'',j'']$, and finally descended by o to $L_{\texttt{OS}\underline{\texttt{P}}}[i,j]$. Generalizing the previous case, we retraverse the path starting from interval $[i',j'] \cap [c,N]$ in $L_{\texttt{PO}\underline{\texttt{S}}}$, and upon obtaining the final range $L_{\texttt{OS}\underline{\texttt{P}}}[i''',j''']$, return i''' if $i''' \leq j'''$, or else $+\infty$.
  - If s was not bound, then p was not bound either (or we would be in order POS). In this case we first find p such that $A_{\texttt{P}}[p] < c \leq A_{\texttt{P}}[p+1]$, which covers position c in order POS. We then find the smallest value $p' \geq p$ in $L_{\texttt{OS}\underline{\texttt{P}}}[A_{\texttt{O}}[o]+1, A_{\texttt{O}}[o+1]]$ (using operation *range next value* on the wavelet tree of $L_{\texttt{OS}\underline{\texttt{P}}}$). Next, we descend by p' towards $L_{\texttt{PO}\underline{\texttt{S}}}[i',j']$. Finally, we return i' if $i' \leq j'$, or else $+\infty$.

When it comes to binding the value of an edge variable $z := e$, the resulting range will contain only one entry. We can start from $L_{\texttt{PO}\underline{\texttt{S}}}[e]$ and track it to the sequence that contains the current range, as explained in Section 4.3.

A.2 Leaptors for node properties and labels

A leaptor for *leap*(c) with ranges $a^+ \leq x.\pi \leq b^-$, for constants a^+ and b^-, is implemented as follows. We first compute $c' = rank_1(B_\pi, c-1)+1$. Second, we compute $c'' \leftarrow \mathit{leftmost}([c',+\infty],[a',b'])$ with the wavelet tree of S_π. Finally, if there is no answer we return $+\infty$ and otherwise we return $select_1(B_\pi, c'')$.

For range comparisons between properties of two variables, say $x.\pi \leq y.\pi'$, we proceed as follows for *leap*(c) on variable x: if y is not yet bound, we compute c' as in the previous paragraph and return $select_1(B_\pi, c')$, the first node $\geq c$ having property π. Otherwise, we proceed as in the previous paragraph, treating $y.\pi'$ as a constant (which is found at $S_{\pi'}[rank_1(B_{\pi'}, y)]$).

We similarly handle comparisons between node identifiers, like $x < y$, treating them in the same way as property comparisons $x.id < y.id$ for a special property

id, with $x.id = x$ (the numeric identifier of $x \in \mathcal{N}$). That is, when processing x, we return $leap(c) = +\infty$ if y is bound and $c \geq y$, and $leap(c) = c$ otherwise.

Node labels are handled as in Section 4.3. For Boolean conditions on labels, we can handle the complement of a single label ℓ with the leaptor for not having a label we described in there. A leaptor for a general Boolean expression L on labels can push negations down to the leaves of the syntax tree and implement $leap(c)$ by recursively solving the queries on both children of internal nodes, taking the minimum returned value c' on disjunctions. For conjunctions, we create multiary nodes containing all the subexpressions that descend by conjunction nodes, and intersect all children as a set, just as for an intersection algorithm. Overall, this matches the alternation complexity of the problem with an extra penalty factor of $O(|L| \log N) \subseteq \tilde{O}(1)$. Bitvectors B_π of properties can be used in the same way as bitvectors B_ℓ of labels, to filter by nodes having or not having property π.

A.3 The general case: Boolean leaptors

Unlike for labels, some leaves of the syntax tree may not apply for the current variable x being eliminated, either because x is not mentioned in the condition or because x is mentioned but there are other unbound variables. Similarly, leaves representing edge patterns do not apply if they do not contain the variable x to be eliminated. When eliminating x, those leaves become *trivial* leaptors $leap(c) = c$. For each new variable x to eliminate, we first simplify the syntax tree for x, with the rules *trivial* `AND` $X = X$ and *trivial* `OR` $X =$ *trivial*. This, in particular, removes edge patterns that do not participate in the elimination of x. In Fig. 4, for example, the `OR` node disappears when eliminating x. Similarly, bound variables may render conditions directly true or false (such as `!x.hasPhD` in our example once we eliminate x) and those are simplified in the syntax tree for the elimination of further variables: when eliminating variable e, the syntax tree is either just the trie for the edge variable e (if x.hasPhD holds for the bound value of x) or that node `AND`ed with the leaf for `e.since < 01/01/2010` (if not). We say that the reduced syntax tree is *reduced for x*.

B Implementation

We implement PG-Ring in C++11 on top of the succinct data structures library, *SDSL* (https://github.com/simongog/sdsl-lite) [25]. The grids of properties M_π are implemented with wavelet matrices (`wm_int`) and B_π as plain bitvectors (`bit_vector`). Regarding the labels, the bitvector B_ℓ is represented using the sparse bitvector (`sd_vector`).

Generally, datasets contain node, edge, and label identifiers as strings, but PG-Ring needs to deal with them as integers. As they are static identifiers, for each of them, we store a dictionary that allows the mapping between strings and integers. We use a dictionary based on Plain-Front Coding from the compressed string dictionaries library, *libCSD* (https://github.com/migumar2/libCSD) [33]. Similarly, the PG-Ring represents each property value as an integer, so we need to map the original values to an integer depending on the property type (e.g. integers, floats, dates). To this end, we transform each value with a function that associates the original value with an integer identifier. This function is type-dependent and ensures that (i) the original value can be recovered from its identifier, and (ii) the identifier can be compared with other identifiers of the same type in exactly the same way as the original values are compared.

Our system works under the *bag semantics* and supports *homomorphic pattern matching*, allowing repetitions in nodes and relationships. When the system receives a query, we use the dictionaries and functions to get the integer identifiers. Once the query is solved, the identifiers are mapped to the original data. We describe next how we choose appropriate variable elimination orders in the presence of clauses on properties and labels. In this prototype, disjunctions on `WHERE` clause are not implemented.

B.1 Variable elimination orders

The order in which variables are eliminated can have an important impact on practical efficiency [49]. Good practices for LTJ on graph databases [28] are (i) leave *lonely variables* (those appearing only once in the BGP) to the end, (ii) eliminate first the variables that will produce fewer bindings, and (iii) avoid if possible to bind variables not connected to anyone already bound. For the Ring, it was shown [6] that taking the minimum length of the L_* ranges of all the loci in T_x was a good predictor to choose the variable with fewest bindings. They also showed that *adaptive* VEOs, which choose the next variable to eliminate only after eliminating the current one, and considering the range lengths after previous bindings (which differ in each branch) outperforms *static* VEOs, which define a fixed variable ordering from the beginning.

To choose a good VEO for the PG-Ring, we classify the variables as follows:

1. *Non-lonely node variables*: variables in $\mathcal{V}_N$ that appear in two or more edge patterns.
2. *Non-lonely edge variables*: variables in $\mathcal{V}_E$ that appear in two or more edge patterns.
3. *Lonely variables*: node or edge variables that occur in just one edge pattern.

The non-lonely node variables are better choices to eliminate first: binding one of them fixes a node across multiple branches, while fixing an edge constrains only one branch. Accordingly, we first bind the non-lonely node variables, then the non-lonely edge variables, and finally the lonely variables. Within each set, we follow policies (ii) and (iii). For (ii), we estimate the *selectivity* of the bindings, both for the edge patterns and for the `WHERE` clauses, to bind first variables with lower selectivity. Selectivity will be the expected fraction of matches for a given variable x and a node of the reduced syntax tree for x, assuming uniformity and independence. The selectivity of `AND` nodes is then the product of the selectivities of their children, and that of an `OR` node with children selectivities s_1 and s_2 is $1-(1-s_1)(1-s_2)$. For edge pattern leaves, selectivity is estimated as the length of the range in the corresponding L_* sequence divided by N (i.e., the fraction of edges x can match). For the conditions, we proceed as follows.

Comparisons of property values. Consider a clause comparing $x.\pi$ with $y.\pi'$. We know from the data the ranges $[\min_\pi, \max_\pi]$ for $x.\pi$ and $[\min_{\pi'}, \max_{\pi'}]$ for $y.\pi'$. Selectivity is estimated as the probability that a point from the two-dimensional rectangle $[\min_\pi, \max_\pi] \times [\min_{\pi'}, \max_{\pi'}]$ satisfies the comparison. For example, $x.\pi > y.\pi'$ corresponds to the probability that a point lies above the diagonal line $(x = y)$, assuming uniformity. Further, assuming independence, we multiply this probability by the fraction of 1s in B_π and by the fraction of 1s in $B_{\pi'}$, which are the fractions of node pairs having properties π and π', respectively.

In case one of the values is fixed (e.g., $x.\pi > v$ for a constant v, or y is bound and $y.\pi' = v$ in $x.\pi > y.\pi'$), the selectivity estimation can be refined to the fraction of the range $[\min_\pi, \max_\pi]$ that satisfies that condition, $(\max_\pi - v)/(\max_\pi - \min_\pi + 1)$, times the fraction of 1s in B_π. A more refined option is to explicitly count the number of points in M_π within the range defined by the comparison (in our case, $[1, +\infty] \times [v + 1, \infty]$) and divide it by n (where $n = |\mathcal{N}|$ if $x \in \mathcal{V}_N$ and $n = N$ if $x \in \mathcal{V}_E$). This counting takes $O(\log |\mathcal{A}_\pi|)$ time with wavelet trees, which is significant if we have to apply it on every branch. This option is thus applied only on the static VEO.

Comparisons of identifiers For comparisons involving the identifiers of two variables, the estimation is analogous but easier. As the values of both variables are in the same range $[1, n]$, equality comparisons are assigned a selectivity of $1/n$, inequality comparisons a selectivity of $1 - 1/n$, and all other types of comparisons a default selectivity of 0.5. In case one of the identifiers is fixed (e.g., $x > v$ for a constant v, or y is bound to v in $x > y$), we estimate it by the fraction of range $[1, n]$ that satisfies the condition.

Expressions on labels. We similarly use the fraction f_ℓ of 1s in B_ℓ to estimate the selectivity of a query $x.\ell$. In case of negations, the correct estimation is $1 - f_\ell$. For more complex Boolean formulas, we do as with general Boolean expressions.

Internal query optimization. Beyond choosing VEOs, we can optimize the Boolean syntax tree that results for each variable x to eliminate: The children of `AND` nodes are probed left to right, restarting with the leftmost one after each value c in the intersection is found; therefore `AND` children should be sorted by most selective conditions first (edge patterns and filters can be mixed in this order). Further, range conditions on the same variable x or a property $x.\pi$ should be combined into one (like $x.\pi \geq a$ `AND` $x.\pi \leq b$ into $a \leq x.\pi \leq b$), because the grid query is more efficient than the alternation complexity.

C The LSQB Benchmark

The Labelled Subgraph Query Benchmark (LSQB) [35] is a subgraph matching benchmark designed to evaluate the performance of database management systems in graphs with labels in nodes and edges. The benchmark uses the LDBC

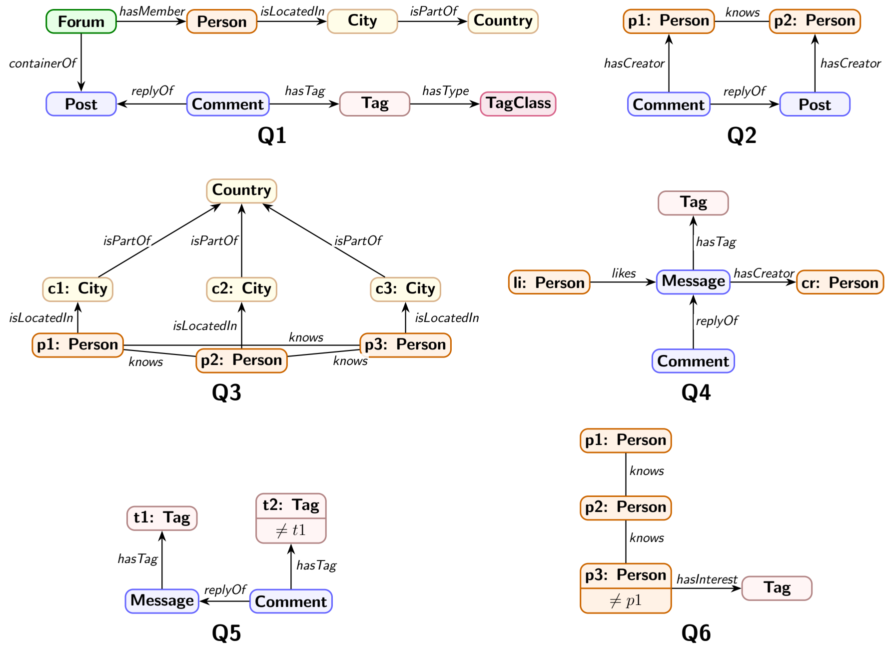

Fig. 7: The first six queries from the Labelled Subgraph Queries Benchmark.

social network generator [19] to create the dataset. In our case, we set the *scalability factor* to 10, obtaining 35,496,603 nodes and 219,425,671 edges. Each node can be given one of 14 possible labels, whereas each edge can be given one of 15 possible labels. In this benchmark, nodes and edges do not have properties.

The LSQB benchmark consists of nine queries, where the last three contain optional edges and negations of relationships. The PG-Ring does not support this kind of operators, thus we limit the benchmark to the first six queries. Fig. 7 shows the chosen queries and Fig. 8 the average query times obtained.

D Systems Compared

In the evaluation, we use these systems with the following settings:

- Umbra [42]: A system based on relational tables whose query plans use binary joins but may introduce wco plans for some sub-queries.
- DuckDB [46]: A non-wco relational database that stores data in columnar format. It uses vectorized and pipelined models to produce optimized binary join plans.
- Kùzu [29]: A graph query engine that stores property graphs using unsorted adjacency lists. It employs cost-based dynamic programming to generate execution plans that combine wco and pairwise joins.

System	Limit	Q1	Q2	Q3	Q4	Q5	Q6
Umbra	10	1,912.38	465.54	162.26	5,038.58	2,432.72	45.28
	1,000	1,913.06	474.24	162.04	5,028.82	2,437.34	46.62
DuckDB	10	3,071.52	446.38	299.24	5,138.54	3,199.64	137.22
	1,000	3,109.04	451.96	301.62	5,196.26	3,202.10	138.52
Kuzu	10	11,207.38	1,394.62	115,999.14	2,260.88	2,772.04	61.06
	1,000	11,440.90	1,406.14	116,053.84	2,279.18	2,784.96	65.10
Neo4j	10	8.92	3.66	4,439.06	3.76	5.82	2.26
	1,000	56.22	**32.08**	TO	35.88	33.96	27.28
Memgraph	10	**4.38**	37.88	65,980.02	36.84	8.00	1.72
	1,000	**17.26**	192.38	TO	50.88	9.46	4.50
MillDB	10	5.79	2.25	15.80	**2.70**	**1.47**	1.84
	1,000	31.34	77.91	1,368.47	**3.44**	**3.14**	**1.93**
PG-Ring	10	259.25	**1.45**	**0.66**	3.24	114.01	**0.05**
	1,000	8,810.63	173.70	**24.17**	150.87	1,135.47	2.48

Fig. 8: Average times (in msec) of queries, with limit of 10 and 1,000 results.

- Neo4j [41]: A widely used graph database that implements the property graph model. Its execution plans are based on index lookups, expand operators, and pairwise joins.
- Memgraph [34]: An in-memory graph database that stores the property graph model using adjacency structures. Query plans rely on traversal-based operators combined with pairwise joins.
- MillDB [51]: A graph database that supports the property graph model using a quad-based storage. The query plans combine wco joins and pairwise joins.
- PG-Ring: the system presented in this work.

We ensure that all systems run in a single-threaded execution. All the systems use non-repeated edge semantics, except MillDB and PG-Ring. For this reason, for those two systems the queries of LSQB are modified adding in the `WHERE` clause additional expressions to avoid repeated edges. As Kùzu, PG-Ring does not support undirected edges in the query pattern. To support all LSQB queries, we handle undirected edges by storing them in both directions.

E The Wikidata Benchmark

We convert a Wikidata knowledge graph [50] into a property graph as follows. Each Wikidata entity is modeled as a node, with the exception of those referenced by the *instance-of* property, which are converted into node labels. Those literals (e.g. strings, numbers, coordinates, dates) linked to a node are represented as properties of the node. The remaining RDF properties are modeled as relationships whose label is the RDF property itself. The edges store the temporal, numeric, and geospatial properties obtained from their qualifiers. In addition, we verify that the added values match the corresponding property type. The resulting graph consists of 108,882,974 nodes, which can take on 102,718 distinct labels and have 10,569 different properties. In total, there are 719,656,925 edges, featuring 1,640 possible edge labels and 603 distinct properties.

We obtained 19,605 queries from the Wikidata SPARQL query log by extracting BGPs with filters and applying a similar transformation to the BGPs (and filters) in order to match the transformed knowledge graph. Queries that are syntactically similar (e.g., same graph pattern, same filter types) are grouped together, regardless of constants and operands. After grouping them, we choose a representative sample of 1,000 queries, with a minimum of one per group and distributing the rest in proportion to the number of elements in each group.

F Cold Cache Scenario

For fairness to disk-based systems, we reported the averaged time in a *warm cache* scenario. We have also evaluated the systems in a *cold cache* setup: before the execution of each set of queries for each system, we clean the cache with the command `echo 3 | sudo tee /proc/sys/vm/drop_caches`. Then, we measure the running times of a single execution of each query.

Fig. 9 details the results on both benchmarks in the *cold cache* scenario. We can observe how the performance of disk-based systems degrades with respect to the *warm cache* scenario. This time, the outstanding systems in the LSQB benchmark are PG-Ring (mostly when limiting to 10 output tuples) and Memgraph (mostly when limiting to 1000 outputs). On the Wikidata benchmark, the cold cache setup affects sharply the performance of both MillDB and Neo4j, as they are disk-based.

System	Limit	Q1	Q2	Q3	Q4	Q5	Q6
Umbra	10	37,806.90	2,235.14	188.86	14,835.52	2,547.69	426.12
	1,000	37,875.42	2,233.34	195.78	14,842.23	2,537.86	423.26
DuckDB	10	12,321.13	884.21	297.54	11,795.84	4,263.62	182.53
	1,000	11,929.72	867.36	301.84	11,665.70	4,289.34	166.20
Kuzu	10	16,416.00	3,140.02	118,911.93	3,314.12	3,875.07	140.09
	1,000	16,404.10	3,174.60	118,559.12	3,358.14	3,798.43	152.60
Neo4j	10	2,316.14	760.80	170,786.27	407.12	14,725.46	113.13
	1,000	6,115.30	23,148.54	TO	4,135.32	24,139.87	442.88
Memgraph	10	**21.08**	36.92	41,272.58	24.26	**6.42**	1.64
	1,000	**28.14**	**113.78**	TO	**27.68**	**6.64**	**4.10**
MillDB	10	1,751.43	995.05	687.08	273.11	7.86	157.41
	1,000	6,687.00	33,693.54	2,287.22	465.04	71.40	165.09
PG-Ring	10	262.09	**3.04**	**1.22**	**1.46**	117.94	**0.13**
	1,000	8,804.49	180.28	**28.82**	157.19	1,136.67	4.37

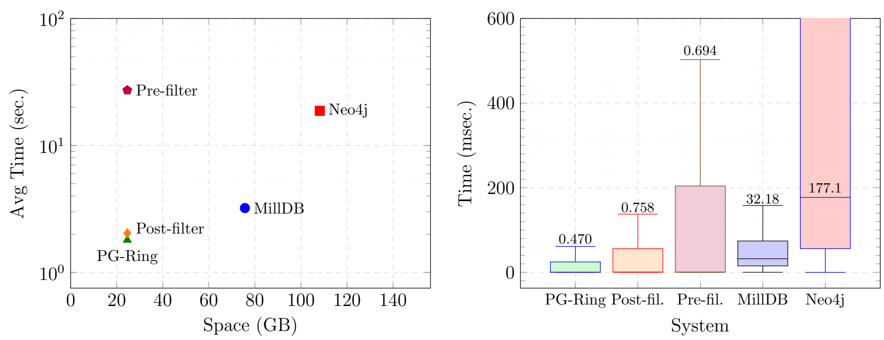

Fig. 9: Results in the *cold cache* scenario. On top the detailed execution times (in msec) for the six queries of LSQB. On the bottom, the space-time trade-off and times distribution on the Wikidata benchmark. The numbers of the boxplots indicates the median of execution times.