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

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

[1] Universidade da Coruña, CITIC, Facultade de Informática, Spain
[2] Department 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.

Keywords: worst-case optimal · graph databases · filter · joins · GQL

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 [21] 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 [6,8,13,30,4]. Such works have focused on evaluating BGPs over triple-based models such as RDF.

* Funded by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile. AGB is funded by MCIN/AEI/10.13039/501100011033 and EU/ERDF "A way of making Europe" PID2022-141027NB-C21 (EARTHDL). CITIC is funded by the Xunta de Galicia and co-financed by the EU through FEDER Galicia: grant ED431G 2023/01. GN is funded by Fondecyt Grant 1260080, ANID, Chile.

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: 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 [44], 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 speed-ups 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 [19], 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) [8], 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

An RDF graph [16] G is a set of triples (s, p, o), each denoting an edge $s \xrightarrow{p} o$. Node s is called the *subject*, label p the *predicate*, and node o the *object*. A *Basic Graph Pattern (BGP)* Q is a set of *triple patterns* (x, y, z) where any element can be a constant or a variable. A *solution* to a BGP is a function that assigns a constant to each variable of the BGP so that all resulting triples occur in the graph. *Solving* a BGP Q consists of producing the set $Q(G)$ of all its solutions.

Let G have N triples. The *AGM bound* [9] of Q, $Q^* \geq |Q(G)|$, is the maximum size of a solution set of Q over any graph of N triples. An algorithm solving Q is *worst-case-optimal* (wco) [9,28] if it works in time $\tilde{O}(Q^*)$, where $\tilde{O}$ hides factors polylogarithmic on N, or independent of N (such as $|Q|$). The wco concept and wco algorithms have had notable impact on the resolution of complex and cyclic BGPs. Such algorithms outperform every algorithm that enforces the triple pattern conditions *consecutively* (as classic database join plans do) by enforcing 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, $\mathcal{T} = \{\texttt{SPO}, \texttt{SOP}, \texttt{POS}, \texttt{PSO}, \texttt{OSP}, \texttt{OPS}\}$. 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, $v_t \leftarrow \mathit{child}(v_t, c)$ 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 forms 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, $c' \leftarrow \mathit{leap}(c)$. 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, so c is 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

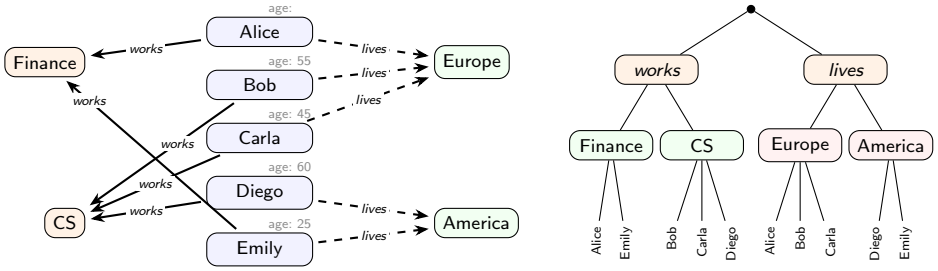

Fig. 1: On the left, a graph database of where people live and area they work in; person nodes have also property `age`. On the right, the trie `POS` for the graph.

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 [26], GQL [20], 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*. Queries may then specify that, to bind some variable x to a node or edge in the triple pattern, that node or edge must contain a property π and that its corresponding value $x.\pi$ must be within a range, say $a \leq x.\pi \leq b$ (where a and b can be constants or other property values, say $x.\pi \leq y.\pi'$). It is also customary to assign *labels* to nodes and edges, so that $x.\ell$ requires that variable x is assigned to an element having label ℓ. While SPARQL filters are applied directly on nodes and predicates in triples, we will 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 $\{(x, \textit{lives}, \mathsf{Europe}), (x, \textit{works}, \mathsf{CS})\}$ complemented with the condition $x.\mathsf{age} \geq 50$, aimed at finding senior people living in Europe and working in CS. LTJ will choose two loci, say in `POS` (it could also be `OPS`), by descending through $(\textit{lives}, \mathsf{Europe})$ and $(\textit{works}, \mathsf{CS})$, respectively. The intersection of the loci children yields all people x in Europe working in CS. 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 in Europe working in CS, 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 identifiers in the graph (we defer properties and labels on edges to Section 5, though they could be supported similarly), 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 values, and another of all the values in $\mathcal{A}_\pi$, we map actual values to integers with a binary search (in $O(\log N)$ time, only once per query), 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 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 [41]. This space is proportional to the database size, which includes one unit of space per value each property takes for each node. The leaptor time is then always in $O(\log N)$. See Fig. 2.

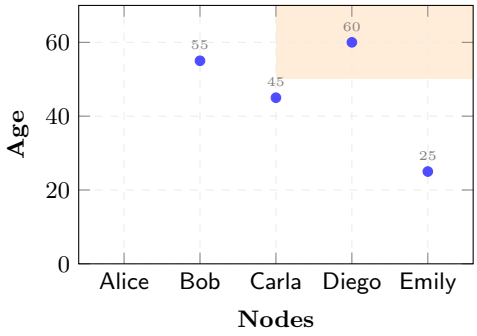

Fig. 2: The grid M_{age} for the graph of Fig. 1 (only showing person nodes). The shadowed area corresponds to the query $\mathit{leftmost}([\mathsf{Carla}, +\infty] \times [50, +\infty])$.

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 in Europe working in CS will be sought after c', not after c. In the example, the first candidate coming out of the intersection of children of trie loci $(\mathit{lives}, \mathsf{Europe})$ and $(\mathit{works}, \mathsf{CS})$ is Bob. The grid leaptor also returns Bob for the query $\mathit{leap}(\mathsf{Bob}) = \mathit{leftmost}([\mathsf{Bob}, +\infty] \times [50, +\infty])$, 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

We also define leaptors for labels. While labels can be handled as a special property (giving them value 0 when the label exists, and asking for $\mathit{leftmost}([c, +\infty] \times [0, 0])$ on their grid), a faster leaptor uses a *successor* data structure [45] storing the set D_ℓ of node ids having label ℓ. The structure can find the smallest integer $\geq c$ in D_ℓ (or return $+\infty$ if there is none) in time $O(\log\log|\mathcal{U}|)$, using $O(|D_\ell|)$ 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 nodes having a property π, independently of its value. The corresponding leaptor can, again, be simulated with query $\mathit{leftmost}([c, +\infty] \times [-\infty, +\infty])$ on M_π, but within linear space we can also store a successor data structure on the set D_π of the node ids having some value for property π, and run $\mathit{leap}(c)$ in time $O(\log\log|\mathcal{U}|)$.

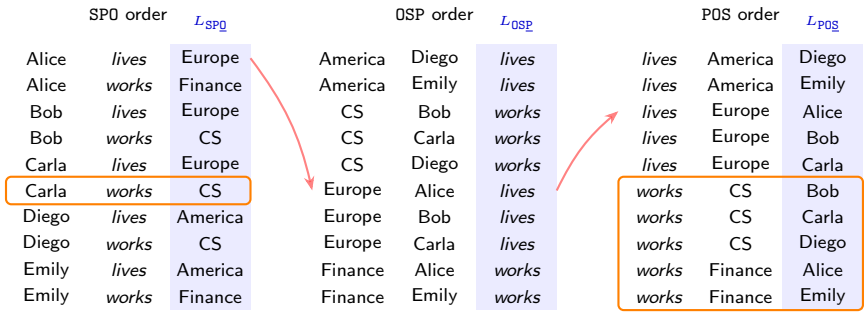

Fig. 3: A Ring structure on the graph of Fig. 1, assuming the node id order is the lexicographic order of the strings. The Ring only stores the three shaded sequences. The arrows show how the triple (`Alice`, *lives*, `Europe`) is tracked across the three orders. The rectangle on `POS`, corresponding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

4.3 ERing: A succinct wco index handling properties and labels

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, $L_{\mathtt{SPO}}$, $L_{\mathtt{OSP}}$, and $L_{\mathtt{POS}}$. Sequence $L_{\mathtt{SPO}}$ stores the objects of the N triples sorted by the order `SPO`, that is, triples (s, p, o) are sorted lexicographically and $L_{\mathtt{SPO}}$ collects the sequence of their o components. Analogously, sequence $L_{\mathtt{OSP}}$ collects the N predicates p in the order `OSP` and sequence $L_{\mathtt{POS}}$ collects the N subjects s in the order `POS`. Overall, letting $\mathcal{U}$ be the universe of nodes and $\mathcal{L}$ that of labels (or predicates), the Ring uses $(2N \log_2 |\mathcal{U}| + N \log_2 |\mathcal{L}|)(1 + o(1))$ 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* [25,38], which on a universe Σ (where Σ is $\mathcal{U}$ or $\mathcal{L}$) uses $N \log_2 |\Sigma|(1 + o(1))$ bits and in particular implements the operation $rank_c(L_*, i)$, the number of times c occurs in $L_*[1, i]$, in time $O(\log |\Sigma|)$. We also represent arrays $A_{\mathtt{O}}$, $A_{\mathtt{P}}$, and $A_{\mathtt{S}}$, where $A_*[1, |\Sigma|]$ holds in $A_*[e]$ the number of positions in L_* with values less than e. Those arrays can be represented with 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_*. Bitvector representations solving $select_b$ in constant time [15,37] require $N + o(N)$ bits of space, which fits within the sublinear extra space of the Ring.

With those structures we can *track* triples across the arrays L_*: the triple at $L_{\mathtt{SPO}}[i]$ has object $o = L_{\mathtt{SPO}}[i]$, and it corresponds to $L_{\mathtt{OSP}}[i']$, where $i' = A_{\mathtt{O}}[o] + rank_o(L_{\mathtt{SPO}}, i)$. The triple then has predicate $p = L_{\mathtt{OSP}}[i']$, and it corresponds to $L_{\mathtt{POS}}[i'']$, where $i'' = A_{\mathtt{P}}[p] + rank_p(L_{\mathtt{OSP}}, i')$. The subject of the triple is then $s = L_{\mathtt{POS}}[i'']$ and if we compute $A_{\mathtt{S}}[s] + rank_s(L_{\mathtt{POS}}, i'')$ we return to $L_{\mathtt{SPO}}[i]$.

Trie nodes correspond to ranges in some sequence L_*. A locus in SOP or SPO, reached by descending by s, corresponds to the range $L_{\underline{\texttt{SPO}}}[i,j] = L_{\underline{\texttt{SPO}}}[A_{\texttt{S}}[s]+1, A_{\texttt{S}}[s+1]]$ 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 $L_{\underline{\texttt{OSP}}}$. This is $L_{\underline{\texttt{OSP}}}[i',j']$, where $i' = A_{\texttt{O}}[o]+rank_o(L_{\underline{\texttt{SPO}}}, i-1)+1$ and $j' = A_{\texttt{O}}[o]+rank_o(L_{\underline{\texttt{SPO}}}, j)$. If, instead, we descend by p after s in SPO, we stay in the same sequence $L_{\underline{\texttt{SPO}}}$, 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 $L_{\underline{\texttt{POS}}}[i'',j'']$ descending by s to $L_{\underline{\texttt{SPO}}}[i''',j''']$, with $i''' = A_{\texttt{S}}[s] + rank_s(L_{\underline{\texttt{POS}}}, i''-1)+1$ and $j''' = A_{\texttt{S}}[s] + rank_s(L_{\underline{\texttt{POS}}}, j'')$. 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 $L_{\underline{\texttt{SPO}}}[i,j]$ after descending by s in trie SOP. To implement *leap*(c) on the values of o, we must find the smallest value in $L_{\underline{\texttt{SPO}}}[i,j]$ that is $\geq c$. This operation, called *range next value*, is supported in $O(\log|\Sigma|)$ time by the wavelet tree [22]. If, instead, we are in trie SPO and want to find the smallest value of p that is $\geq c$, we start at range $L_{\underline{\texttt{POS}}}[A_{\texttt{P}}[c]+1, N]$ (the range of all predicates $\geq p$ in order POS), and compute $L_{\underline{\texttt{SPO}}}[i'] = A_{\texttt{S}}[s] + rank_s(L_{\underline{\texttt{POS}}}, A_{\texttt{P}}[c]) + 1$. Tracking $L_{\underline{\texttt{SPO}}}[i']$ to $L_{\underline{\texttt{OSP}}}$ yields the desired value of p. The Ring then implements the leaptors in time $O(\log|\Sigma|)$, yielding a total query time complexity of $O(Q^* \cdot |Q| \log|\Sigma|) \subseteq \tilde{O}(Q^*)$.

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 $S_\pi[1,N_\pi]$ 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).

The bitvector representation of B_π [15,37] 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. 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 $select_0(B_{\mathsf{age}}, 3) - 3 = 4 = \mathsf{Diego}$.

Queries on labels were handled with generic successor data structures in Section 4.2. While using linear space, this 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 those m elements, that is, $\log_2 \binom{n}{m} = m \log \frac{n}{m} + O(m)$ bits. To match that space, we store one bitvector B_ℓ for each label ℓ, with one bit per node, marking with 1 the nodes having label ℓ. A *sparse* bitvector representation [39, Sec. 4.4], if m out of n bits are 1, uses $m \log \frac{n}{m} + 2m$ bits, and supports $rank_b$ and $select_0$ in time $O(\log m)$ and $select_1$ in constant time. Operation $leap(c)$ for the condition $x.\ell$ is then implemented as $select_1(B_\ell, rank_1(B_\ell, c-1)+1)$ with this leaptor, in time $O(\log m)$ (which is logarithmic on the data size, and thus $\tilde{O}(1)$). This representation also provides a leaptor for the next element *not* having a label ℓ, with $leap(c) = select_0(B_\ell, rank_0(B_\ell, c-1)+1)$, also in time $O(\log m)$.

The ERing. The resulting structure, which we call *Extended Ring (ERing)*, is succinct on the triples, properties and labels it represents. It implements LTJ and supports leaptors for filtering by the existence of node properties and labels, and by restricting the values of node properties to ranges. The range limits can be constants or other property values. When a range involves several variables, as in $x.\mathsf{age} \leq y.\mathsf{age}$, the leaptor is enabled only when binding the last variable (x or y in our example); by then, the condition has become a range with constants.

4.4 Worst-case optimality

The concept of worst-case optimality for BGPs (i.e., intersections of triple patterns) is well-defined for the RDF model. Node properties π can be expressed in RDF by creating a node for each literal value $a \in \mathcal{A} = \cup_\pi \mathcal{A}_\pi$, and represent $v.\pi = a$ with a triple (v, π, a). For labels we can use a reserved property lab and represent $v.\ell$ with (v, lab, ℓ). Conditions like $x.\pi = a$ or $x.\ell$ are then expressed as triple patterns (x, π, a) or (x, lab, ℓ). Range conditions $a \leq x.\pi \leq b$ can be translated into $\{(x, \pi, z), R(a, z), R(z, b)\}$ with variables x and z. The (virtual) table $R(i, j)$ contains all the pairs $i, j \in \mathcal{A}$ such that $i \leq j$. The range condition is then simulated by making node ids respect the order of the literals in $\mathcal{A}$ and using a simple leaptor for z: $leap(c) = \max(a, c)$ if $c \leq b$ and $+\infty$ otherwise.

Mapping properties and labels into plain RDF and using the standard Ring [6] is then wco, but not succinct: it creates one triple per node label or property value. We show that our ERing, instead, is a succinct-space wco solution.

Definition 1. *A solution to BGPs extended with clauses for the existence of properties and labels, and comparing property values, is* worst-case optimal *if its time complexity is within the AGM bound of the corresponding query mapped onto the RDF model, as described above.*

Theorem 1. *The ERing is worst-case-optimal when solving extended BGPs.*

Proof. It suffices to observe that the ERing's time complexity is not worse than the standard Ring's [6] on the corresponding RDF graph, with a certain VEO: for each range condition $a \leq x.\pi \leq b$, translated into $\{(x, \pi, z), R(a, z), R(z, b)\}$, we bind z immediately after x. In the ERing, the potential values of x and z are filtered simultaneously, which per the alternation complexity is never worse than

filtering one after the other. With the standard Ring, each potential value of x will be instantiated and only then filtered by z. The conditions on the existence of properties $x.\pi$ are particular cases, $-\infty \leq x.\pi \leq +\infty$.

Simpler conditions like (x, π, a) or (x, lab, ℓ) are filtered optimally by the Ring, by finding the locus of the path (π, a) or (lab, ℓ) in the trie POS: later, when binding x, this locus filters the corresponding values. ERing achieves the same by adding, at the time of binding x, the leaptor $\mathit{leftmost}([c, +\infty], [a, a])$ on S_π for properties or the leaptor $\mathit{select}_1(B_\ell, \mathit{rank}_1(B_\ell, c-1)+1)$ for labels.

Conditions that compare properties, like $x.\pi \leq y.\pi'$, are translated into $\{(x, \pi, z), (y, \pi', z'), R(z, z')\}$. In this case, the ERing eliminates (say) x before y, and only during the elimination of y it enforces $z \leq z'$ (where $z = x.\pi$ is by then a constant). This corresponds to, and is never worse than, eliminating variables in the order x, z, y, z' with the standard Ring.

Since the ERing solution is not worse than the standard Ring solution with some VEO, and it implements all leaps in time $\tilde{O}(1)$, it is wco. □

The ERing can be better than the (standard) Ring (mapped to RDF): it can filter simultaneously what the Ring can only filter sequentially, which is equivalent to pre- or post-filtering. On the other hand, the Ring can use VEOs that are not available to the ERing: consider conveniently translating the clause $x.\pi = y.\pi$ to $\{(x, \pi, z), (y, \pi, z)\}$, where binding z first might be advantageous.

Conversely, one could represent RDF triples using properties and labels to enable simultaneous filtering over RDF: triples (u, p, v) where v is a literal could map to properties on u ($u.p = v$), while triples (u, type, v) could map to labels on u ($u.\ell = v$). Our model already supports zero-or-more labels and property values per node, but would require some amendments to cover RDF properties mixing datatype and node values, joins on predicates, and other such details.

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 properties and labels. Let $\mathcal{G}(\mathcal{N}, \mathcal{E})$ be a property graph with $\mathcal{N} \cap \mathcal{E} = \emptyset$. Let $\mathcal{P}$ be a set of properties, and $\mathcal{A}$ a set of atomic values properties can take (e.g., integers, dates, strings). Each node or edge has zero or more properties, but a given property has at most one value for a given node or edge. Nodes have zero or more labels in $\mathcal{L}_N$ and edges have exactly one label in $\mathcal{L}_E$ ($\mathcal{L}_N \cap \mathcal{L}_E = \emptyset$).

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 [20]; their common graph pattern matching language is called GPML [17]. A GPML query is of the form `MATCH patterns`, where `patterns` 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 [17,1,20]; the formal syntax and semantics are given in Appendix A.

The most basic edge patterns are of the form $(\mathtt{s}) \rightarrow (\mathtt{o})$, 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* $\in \mathcal{L}_N$. Conditions with no variable names, like `(:person)` $\rightarrow$ `(y)`, are also permitted. 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 only the nodes with both 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 `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 projects the variable or property values to return. We use *bag semantics*, so repetitions caused by projection are not removed.

The following query, for example, looks for names and ages of senior people in Europe working in CS 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 ERing described in Section 4.3 to define PG-Ring, which indexes property graphs within succinct space.

For compatibility with the ERing, we create new variables where edge patterns omits them; those are naturally projected out by the `RETURN` clauses.

Since the edges of property graphs have exactly one label in $\mathcal{L}_E$, we treat edges $e = s \to o$ with label ℓ as triples (s, ℓ, o) in order to create the sequences L_*. The symbols of $L_{\underline{\texttt{POS}}}$ and $L_{\underline{\texttt{SPO}}}$ then range over $\mathcal{N}$ and those of $L_{\underline{\texttt{OSP}}}$ range over $\mathcal{L}_E$. The wavelet tree of $L_{\underline{\texttt{OSP}}}$ uses $N \log_2 |\mathcal{L}_E|(1 + o(1))$ bits, which is succinct.

Storing the additional edge identifiers of property graphs would not be succinct, 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 $[A_{\texttt{P}}[\ell] + 1, A_{\texttt{P}}[\ell + 1]]$, and edge patterns of the form `(s)` $-[:\ell]\to$ `(o)` are treated as triple patterns with a constant predicate, instead of using a leaptor.

Label expressions L on edges, `(s)` $-[:L]\to$ `(o)`, are converted into a set of $O(L)$ disjoint ranges: the complement of ℓ becomes $\{[1, A_{\texttt{P}}[\ell]], [A_{\texttt{P}}[\ell+1]+1, N]\}$, disjunction becomes union of ranges, and conjunction becomes intersection. We then maintain, in general, a set of disjoint ranges during query processing, not just one. For simplicity we consider a single range in the discussion that 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 their identifiers are not explicit in any sequence L_*, but implicit as indices in $L_{\underline{\texttt{POS}}}$ (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 $\mathit{leap}(c)$ differently depending on which sequence L_* holds the range that represents the locus v_t. See Appendix B for details.

Leaptors for node properties and labels. Some simplifications with respect to the treatment in Section 4.3 are possible (see Appendix B). For label expressions on nodes, we can handle the complement of a single label ℓ with the leaptor for not having a label we described in Section 4.3. A leaptor for a label expression L builds a syntax tree where negations are pushed down to the leaves and $\mathit{leap}(c)$ is solved by recursively calling $\mathit{leap}(c)$ 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 our intersection algorithm. 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 are used in the same way as bitvectors B_ℓ of labels, to filter nodes having or not having property π.

Boolean leaptors. We use a similar syntax tree, with multiary conjunction nodes, to process together the `MATCH` edge patterns and the `WHERE` clauses. This generalizes our set intersections by allowing disjunction nodes (to represent `OR` in `WHERE` clauses), which leap by taking the least of their children's leap values. Negations are pushed to the leaves and applied 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)`). See Fig. 4, and Appendix B for details.

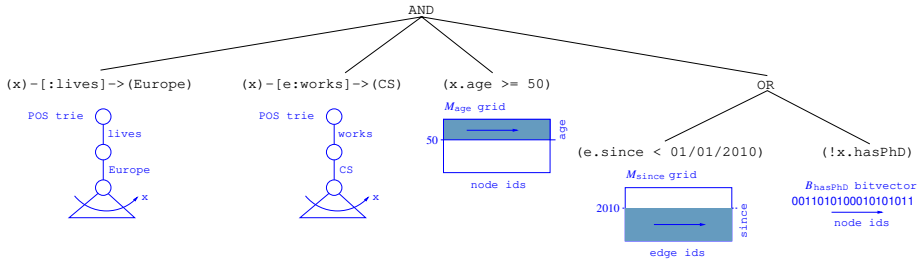

Fig. 4: The syntax tree of our example of Section 5.1. Each internal node implements a Boolean leaptor; leaf nodes implement trie, grid, and bitvector leaptors.

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 C. 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 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 G. 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 graph, nodes and edges do not have properties. We use the first six queries from the benchmark; the others are discarded because they require unsupported optional edges and negation. See Appendix D for details.

We compare systems supporting GQL (details in Appendix E): Umbra [43], DuckDB [46], Kùzu [29], Neo4j [42], Memgraph [34], MillDB [51], and our PG-Ring. Our figures show the space required by their data and indexes, not the extra needed for query resolution. This extra space is negligible for PG-Ring but not for others that produce intermediate results (we discuss an extreme case soon). We run the six queries on all the systems, limiting results to 1, 10, 100, 1,000, and 10,000. Appendix D shows all results; Fig. 5 summarizes some.

Fig. 5 (left) 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

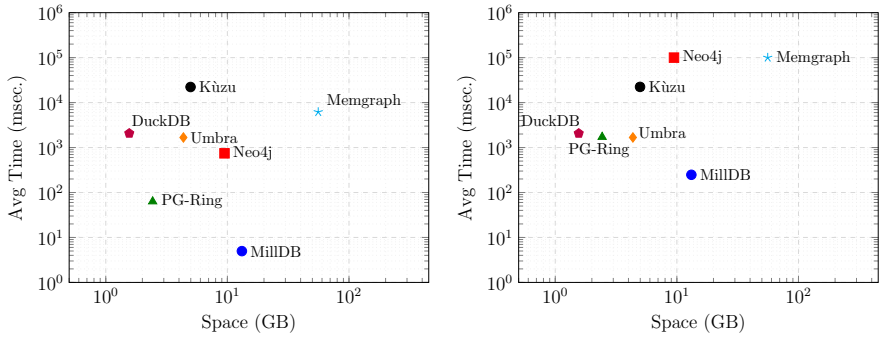

Fig. 5: The space–time trade-off of the evaluated systems on six queries of LSQB. The first plot is limited to 10 results, and the second to 1,000.

is 1.6 times smaller due to its columnar compression, but its average times are 12−2,700 times slower (see Appendix D). 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 D), 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.

Fig. 5 (right) shows the same evaluation when restricting the results to 1,000. In this scenario, PG-Ring performs competitively on cyclic queries, with 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 also shows that DuckDB and Umbra fare better for obtaining many results. These systems use hash-joins, which require processing the left-side of the join completely and indexing it for lookup before processing the join's right-side (looking up each such value). Even with an output limit, all joins except those on the right extreme of the plan are computed completely, where the last, rightmost table-scan (and the joins above it) can halt early once the limit is reached. Thus the number of intermediate results they process varies little with the limit. The consequence is that they may generate huge intermediate results, preventing their use at larger scale. For example, DuckDB, although using less space than PG-Ring in the static index, generated one billion intermediate results on the cyclic query Q3, which has only 15 million final results. Even on the relatively small LSQB dataset, it ran out of memory on Q1 and Q6 without limit.

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.

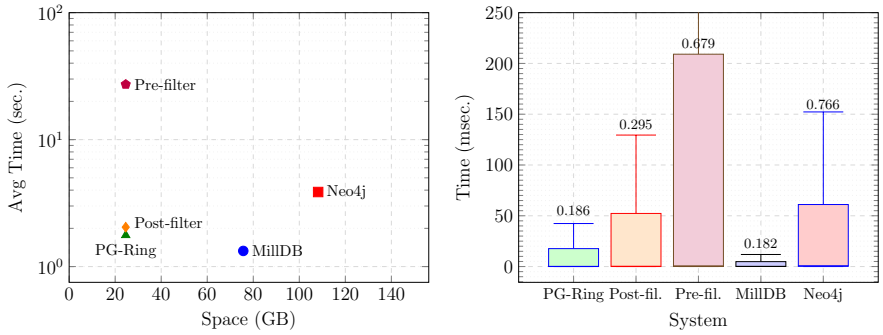

Fig. 6: On the left, the space–time trade-off of the systems evaluated on Wikidata. On the right, boxplots represent the distribution of the query execution times. The numbers on top indicate the median time for each system.

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 F for details, which includes discussions on the limitations of property graphs for representing Wikidata [27], in particular for its multi-valued and node-valued qualifiers/properties.

We exclude Umbra [43], 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 [42], 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 add two more 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 these specific 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-filtering 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 [6] supports this setting, an interesting question is when it is preferable to represent (e.g.) datatype values as property values versus graph nodes.

We only support fragments of languages like GQL and SPARQL, missing typical relational operators (`UNION`, `OPTIONAL`), as well as path queries. It would be of interest to explore pushing further query operators from GQL, SPARQL, etc., into low-level operations on the succinct data structures; for example, existing methods for evaluating paths on the Ring could be deployed [7].

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 [40], 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://github.com/adriangbrandon/PG-Ring`. The datasets are available from Zenodo at `https://zenodo.org/records/19818050`. Appendices are published on arXiv, at `https://arxiv.org/abs/2608.03840` [24].

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, A. Gómez-Brandón, A. Hogan, G. Navarro, and J. Rojas-Ledesma. Optimizing RPQs over a compact graph representation. *The VLDB Journal*, 33(2):349–374, 2024.
8. 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.
9. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.
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. 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.
19. 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.
20. 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.
21. 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.
22. 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.
23. 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.
24. A. Gómez-Brandón, A. Hogan, and G. Navarro. Uplifting the superpowers of worst-case-optimal join algorithms. arXiv:2608.03840, 2026. `https://arxiv.org/abs/2608.03840`.
25. 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.
26. S. Harris, A. Seaborne, and E. Prud'hommeaux. SPARQL 1.1 Query Language. W3C Recommendation, March 2013. `https://www.w3.org/TR/sparql11-query/`.
27. D. Hernández, A. Hogan, C. Riveros, C. Rojas, and E. Zerega. Querying Wikidata: Comparing SPARQL, relational and graph databases. In *Proc. 15th International Semantic Web Conference (ISWC), Part II*, pages 88–103, 2016.
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. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.
40. G. Navarro. Practical adaptive dynamic bitvectors. *Software Practice and Experience*, 55(9):1539–1559, 2025.
41. Y. Nekrich and G. Navarro. Sorted range reporting. In *Proc. 13th Scandinavian Symposium on Algorithmic Theory (SWAT)*, LNCS 7357, pages 271–282, 2012.
42. Neo4j, Inc. Neo4j graph database management system, 2024.
43. 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.
44. 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.
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 Formal Syntax and Semantics of our GQL Fragment

A.1 Property graph

A property graph is a tuple $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \rho, \mathcal{L}_N, \mathcal{L}_E, \lambda, \mathcal{P}, \mathcal{A})$ where:

1. $\mathcal{N}$ is the set of nodes and $\mathcal{E}$ is a set of directed edge identifiers, $\mathcal{N} \cap \mathcal{E} = \emptyset$.
2. $\rho : \mathcal{E} \to \mathcal{N} \times \mathcal{N}$ is a total function that associates each edge identifier in $\mathcal{E}$ with its source and target nodes in $\mathcal{N}$, in that order.
3. $\mathcal{L}_N$ and $\mathcal{L}_E$ are sets of node and edge labels, respectively, with $\mathcal{L}_N \cap \mathcal{L}_E = \emptyset$.
4. $\lambda : (\mathcal{N} \to 2^{\mathcal{L}_N}) \cup (\mathcal{E} \to 2^{\mathcal{L}_E})$ is a total function that associates nodes with sets of labels in $\mathcal{L}_N$ and edges with sets of labels in $\mathcal{L}_E$. It must hold that $|\lambda(e)| = 1$ if $e \in \mathcal{E}$, that is, edges have exactly one label.
5. $\mathcal{P}$ is a set of properties and $\mathcal{A}$ is a universe of values (literals). Each $\pi \in \mathcal{P}$ is a partial function $\pi : \mathcal{N} \cup \mathcal{E} \to \mathcal{A}$.

A.2 Query syntax

Queries use variables drawn from sets $\mathcal{V}_N$ (to be matched to nodes) and $\mathcal{V}_E$ (to be matched to edges). Those are disjoint from nodes, edges, and among them: $\mathcal{V}_N \cap (\mathcal{N} \cup \mathcal{E} \cup \mathcal{V}_E) = \emptyset$ and $\mathcal{V}_E \cap (\mathcal{N} \cup \mathcal{E} \cup \mathcal{V}_N) = \emptyset$. Queries have the form

```
MATCH patterns
WHERE clauses
RETURN outputs
```

where the `WHERE` part is optional. The patterns describe the subgraph to match, as a list of conditioned edges:

$$
\begin{aligned}
\texttt{patterns} &\longrightarrow \texttt{pattern}^+ \\
\texttt{pattern} &\longrightarrow \texttt{(node-expr) -> (node-expr)} \\
\texttt{pattern} &\longrightarrow \texttt{(node-expr) -[edge-expr]-> (node-expr)} \\
\texttt{node-expr} &\longrightarrow \texttt{node-base} \mid \texttt{: label-expr}(\mathcal{L}_N) \mid \texttt{node-cond} \\
\texttt{node-base} &\longrightarrow \texttt{node} \in \mathcal{N} \mid \texttt{node-var} \in \mathcal{V}_N \\
\texttt{node-cond} &\longrightarrow \texttt{node-base : label-expr}(\mathcal{L}_N) \\
\texttt{edge-expr} &\longrightarrow \texttt{edge-base} \mid \texttt{: label-expr}(\mathcal{L}_E) \mid \texttt{edge-cond} \\
\texttt{edge-base} &\longrightarrow \texttt{edge-var} \in \mathcal{V}_E \\
\texttt{edge-cond} &\longrightarrow \texttt{edge-base : label-expr}(\mathcal{L}_E) \\
\texttt{label-expr}(\mathcal{L}) &\longrightarrow (\texttt{label} \in \mathcal{L})^+ \\
\texttt{label-expr}(\mathcal{L}) &\longrightarrow \texttt{label-expr}(\mathcal{L}) \texttt{ \& label-expr}(\mathcal{L}) \\
\texttt{label-expr}(\mathcal{L}) &\longrightarrow \texttt{label-expr}(\mathcal{L}) \texttt{ | label-expr}(\mathcal{L}) \\
\texttt{label-expr}(\mathcal{L}) &\longrightarrow \texttt{! label-expr}(\mathcal{L}) \\
\texttt{label-expr}(\mathcal{L}) &\longrightarrow \texttt{(label-expr}(\mathcal{L}) \texttt{)}
\end{aligned}
$$

The clauses set additional conditions on the subgraph matches:

$$
\begin{aligned}
\texttt{clauses} &\longrightarrow \texttt{clause}^{+} \\
\texttt{clause} &\longrightarrow \texttt{node-cond} \mid \texttt{edge-cond} \\
\texttt{clause} &\longrightarrow \texttt{node-base op node-base} \mid \texttt{edge-base op edge-base} \\
\texttt{clause} &\longrightarrow \texttt{node-base.}\pi \in \mathcal{P}\ \texttt{IS NULL} \mid \texttt{node-base.}\pi \in \mathcal{P}\ \texttt{IS NOT NULL} \\
\texttt{clause} &\longrightarrow \texttt{edge-base.}\pi \in \mathcal{P}\ \texttt{IS NULL} \mid \texttt{edge-base.}\pi \in \mathcal{P}\ \texttt{IS NOT NULL} \\
\texttt{clause} &\longrightarrow \texttt{node-prop op node-prop} \mid \texttt{edge-prop op edge-prop} \\
\texttt{clause} &\longrightarrow \texttt{clause AND clause} \mid \texttt{clause OR clause} \mid \texttt{NOT clause} \mid \texttt{(clause)} \\
\texttt{node-prop} &\longrightarrow a \in \mathcal{A} \mid \texttt{node-base.}\pi \in \mathcal{P} \\
\texttt{edge-prop} &\longrightarrow a \in \mathcal{A} \mid \texttt{edge-base.}\pi \in \mathcal{P} \\
\texttt{op} &\longrightarrow \texttt{=} \mid \neq \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}
\end{aligned}
$$

Finally, the outputs indicate what to return from the subgraph matches

$$
\begin{aligned}
\texttt{outputs} &\longrightarrow \texttt{output}^{+} \\
\texttt{output} &\longrightarrow \texttt{node-base} \mid \texttt{edge-base} \\
\texttt{output} &\longrightarrow \texttt{node-base.}\pi \in \mathcal{P} \mid \texttt{edge-base.}\pi \in \mathcal{P}
\end{aligned}
$$

A.3 Query semantics

To define the semantics of a query, we map each edge pattern to a first-order predicate on node and edge variables. Those are not the same variables of the query, but come from another set $\mathcal{V}$. The predicates N_v resulting from nodes and E_v resulting from edges will connect the variables $v \in \mathcal{V}$ with the constants and the variables of the query, from $\mathcal{V}_N$ or $\mathcal{V}_E$. We translate the edge patterns of the `MATCH` component into predicates with a function P, as follows:

$$
\begin{aligned}
P(\texttt{pattern}_1, \ldots, \texttt{pattern}_k) &= P(\texttt{pattern}_1) \wedge \cdots \wedge P(\texttt{pattern}_k) \\
P(\texttt{(node-expr}_1\texttt{)->(node-expr}_2\texttt{)}) &= \exists e \in \mathcal{E}, u, v \in \mathcal{N}, \rho(e) = (u, v) \\
&\quad \wedge N_u(\texttt{node-expr}_1) \wedge N_v(\texttt{node-expr}_2) \\
P(\texttt{(node-expr}_1\texttt{)-[edge-expr]->(node-expr}_2\texttt{)}) &= \exists e \in \mathcal{E}, u, v \in \mathcal{N}, \rho(e) = (u, v) \\
&\quad \wedge N_u(\texttt{node-expr}_1) \wedge N_v(\texttt{node-expr}_2) \\
&\quad \wedge E_e(\texttt{edge-expr})
\end{aligned}
$$

Functions N_v and E_v map `node-expr` and `edge-expr`, respectively, to predicates on the variable $v \in V$. Note that the connection with the constants and variables of the query are made via N_v(`node`), N_v(`node-var`), and E_v(`edge-var`). Similarly, functions L_v map `label-expr`:

$$
\begin{aligned}
N_v(\texttt{node} \in \mathcal{N}) &= (v = \texttt{node}) \\
N_v(\texttt{node-var} \in \mathcal{V}_N) &= (v = \texttt{node-var}) \\
N_v(\texttt{: label-expr}) &= L_v(\texttt{label-expr}) \\
N_v(\texttt{node-base : label-expr}) &= N_v(\texttt{node-base}) \wedge L_v(\texttt{label-expr}) \\
E_v(\texttt{edge-var} \in \mathcal{V}_E) &= (v = \texttt{edge-var}) \\
E_v(\texttt{: label-expr}) &= L_v(\texttt{label-expr}) \\
E_v(\texttt{edge-base : label-expr}) &= E_v(\texttt{edge-base}) \wedge L_v(\texttt{label-expr}) \\
L_v(\texttt{label}_1, \ldots, \texttt{label}_k) &= (\texttt{label}_1 \in \lambda(v)) \wedge \cdots \wedge (\texttt{label}_k \in \lambda(v)) \\
L_v(\texttt{label-expr}_1 \texttt{ \& label-expr}_2) &= L_v(\texttt{label-expr}_1) \wedge L_v(\texttt{label-expr}_2) \\
L_v(\texttt{label-expr}_1 \texttt{ | label-expr}_2) &= L_v(\texttt{label-expr}_1) \vee L_v(\texttt{label-expr}_2) \\
L_v(\texttt{!label-expr}) &= \neg L_v(\texttt{label-expr}) \\
L_v(\texttt{((label-expr))}) &= L_v(\texttt{label-expr})
\end{aligned}
$$

Clauses set further conditions on `MATCH` via variables in $\mathcal{V}_N \cup \mathcal{V}_E$. Function C connects these conditions by using predicates N and E directly on the variables of $\mathcal{V}_N \cup \mathcal{V}_E$.

$$
\begin{aligned}
C(\texttt{clause}_1, \ldots, \texttt{clause}_k) &= C(\texttt{clause}_1) \wedge \cdots \wedge C(\texttt{clause}_k) \\
C(\texttt{node} \in \mathcal{N} \texttt{ : label-expr}) &= N_{\texttt{node}}(\texttt{label-expr}) \\
C(\texttt{node-var} \in \mathcal{V}_N \texttt{ : label-expr}) &= N_{\texttt{node-var}}(\texttt{label-expr}) \\
C(\texttt{edge-var} \in \mathcal{V}_E \texttt{ : label-expr}) &= E_{\texttt{edge-var}}(\texttt{label-expr}) \\
C(\texttt{node-base}_1 \texttt{ op node-base}_2) &= \texttt{node-base}_1 \texttt{ op node-base}_2 \\
C(\texttt{edge-base}_1 \texttt{ op edge-base}_2) &= \texttt{edge-base}_1 \texttt{ op edge-base}_2 \\
C(\texttt{node-base.}\pi \in \mathcal{P} \texttt{ IS/IS NOT NULL}) &= (\texttt{node-base} \notin / \in \mathrm{Dom}(\pi)) \\
C(\texttt{edge-base.}\pi \in \mathcal{P} \texttt{ IS/IS NOT NULL}) &= (\texttt{edge-base} \notin / \in \mathrm{Dom}(\pi)) \\
C(\texttt{node-prop}_1 \texttt{ op node-prop}_2) &= V(\texttt{node-prop}_1) \texttt{ op } V(\texttt{node-prop}_2) \\
C(\texttt{edge-prop}_1 \texttt{ op edge-prop}_2) &= V(\texttt{edge-prop}_1) \texttt{ op } V(\texttt{edge-prop}_2) \\
C(\texttt{clause}_1 \texttt{ AND clause}_2) &= C(\texttt{clause}_1) \wedge C(\texttt{clause}_2) \\
C(\texttt{clause}_1 \texttt{ OR clause}_2) &= C(\texttt{clause}_1) \vee C(\texttt{clause}_2) \\
C(\texttt{NOT clause}) &= \neg C(\texttt{clause}) \\
C(\texttt{((clause))}) &= C(\texttt{clause}) \\
V(a \in \mathcal{A}) &= a \\
V(\texttt{node-base.}\pi \in \mathcal{P}) &= \pi(\texttt{node-base}) \\
V(\texttt{edge-base.}\pi \in \mathcal{P}) &= \pi(\texttt{edge-base})
\end{aligned}
$$

At this point, the expression $P(\texttt{patterns}) \wedge C(\texttt{clauses})$ is a predicate whose free variables are those of $\mathcal{V}_N \cup \mathcal{V}_E$ used by the query. We will form, from the

`RETURN` clause, the tuple of values to be returned:

$$\begin{aligned} O(\texttt{output}_1, \ldots, \texttt{output}_k) &= (O(\texttt{output}_1), \ldots, O(\texttt{output}_k)) \\ O(\texttt{node-base}) &= \texttt{node-base} \\ O(\texttt{edge-base}) &= \texttt{edge-base} \\ O(\texttt{node-base}.\pi \in \mathcal{P}) &= \pi(\texttt{node-base}) \\ O(\texttt{edge-base}.\pi \in \mathcal{P}) &= \pi(\texttt{edge-base}) \end{aligned}$$

Finally, the semantics of the `MATCH-WHERE-RETURN` query is (with bag semantics)

$$\{O(\texttt{outputs}) \mid P(\texttt{patterns}) \wedge C(\texttt{clauses})\}.$$

For illustration, the semantics of the example query given at the end of Section 5.1 is as follows:

$$\begin{aligned} \{(\texttt{name}(x), \texttt{age}(x)) \mid\ & \exists e_1 \in \mathcal{E}, u_1, v_1 \in \mathcal{N}, \rho(e_1) = (u_1, v_1) \\ & \wedge \exists e_2 \in \mathcal{E}, u_2, v_2 \in \mathcal{N}, \rho(e_2) = (u_2, v_2) \\ & \wedge u_1 = x \ \wedge\ \texttt{person} \in \lambda(u_1) \ \wedge\ v_1 = \texttt{Europe} \ \wedge\ \texttt{lives} \in \lambda(e_1) \\ & \wedge u_2 = x \ \wedge\ v_2 = \texttt{CS} \ \wedge\ e_2 = e \ \wedge\ \texttt{works} \in \lambda(e_2) \\ & \wedge \texttt{age}(x) \geq 50 \ \wedge (\texttt{since}(e) < 01/01/2010 \ \vee\ \neg\texttt{has-phd} \in \lambda(x))\} \end{aligned}$$

Here, x and e are free variables used in the query, where the values for two properties of x are projected, while e is not projected.

B PG-Ring: Additional Details

B.1 Leaptors for edge variables

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

- $L_{\underline{\texttt{POS}}}[i, j]$: this is the easiest case because the order `POS` coincides with that of the edge identifiers. We simply return the first element of $[i, j] \cap [c, N]$, that is, $\max(i, c)$ if it is $\leq j$, or else $+\infty$.
- $L_{\underline{\texttt{SPO}}}[i, j]$: this case occurs when a value s was previously used to descend from a range $L_{\underline{\texttt{POS}}}[i', j']$. We then restrict the range in $L_{\underline{\texttt{SPO}}}$ by starting from range $[i', j'] \cap [c, N]$ in $L_{\underline{\texttt{POS}}}$ and descending by s towards $L_{\underline{\texttt{SPO}}}[i'', j'']$, 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 $L_{\underline{\texttt{SPO}}}[i'']$ to $L_{\underline{\texttt{POS}}}$ as also described in Section 4.3, and return the resulting position.
- $L_{\underline{\texttt{OSP}}}[i, j]$: we first find p such that $A_{\texttt{P}}[p] < c \leq A_{\texttt{P}}[p + 1]$, that is, p covers position c in the order `POS`. Now there are two subcases.
  1. The answer is in the range $[c, A_{\texttt{P}}[p + 1]]$, meaning that its corresponding label is still p. To detect this case, we compute the number $k = c - A_{\texttt{P}}[p]$ of occurrences of p up to the one in $L_{\underline{\texttt{POS}}}[c] = p$, and determine if the kth occurrence of p, of a posterior one, occurs in $L_{\underline{\texttt{OSP}}}[i, j]$: we map $L_{\underline{\texttt{OSP}}}[i, j]$ by p to $L_{\underline{\texttt{POS}}}[i', j']$, and if $[i', j'] \cap [c, N]$ is not empty, we return $\max(i', c)$.

2. Otherwise, the answer is past the area of label p. We then find the smallest value $p' > p$ in $L_{\texttt{OSP}}[i, j]$ (using operation *range next value* on the wavelet tree of $L_{\texttt{OSP}}$). If no such p' exists, we return $+\infty$. Otherwise, we descend by p' towards $L_{\texttt{POS}}[i'', j'']$ and return i''.

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 $L_{\texttt{POS}}[e]$ and track it to the sequence that contains the current range, as explained in Section 4.3.

B.2 Leaptors for node properties

Now that each property π defined for a node v has exactly one value $v.\pi$, we can simplify the scheme of Section 4.3, storing bitvectors $B_\pi[1, |\mathcal{N}|]$ that tell 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 $rank_1(B_\pi, c-1) + 1$.

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 $c'' \leftarrow leftmost([c', +\infty], [a', b'])$ 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.

B.3 Boolean leaptors for the `MATCH` and `WHERE` clauses

We create a single syntax tree with the conditions from the `MATCH` triples and the `WHERE` clauses, if any. Unlike the syntax trees of label expressions, some leaves of this syntax tree may not apply for each current variable x being eliminated, because x is not mentioned in the condition. Similarly, leaves representing edge patterns do not apply if they do not contain the variable x being 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 $x.\mathsf{hasPhD}$ holds for the bound value of x) or that

node ANDed with the leaf for `e.since < 01/01/2010` (if not). We say that the reduced syntax tree is *reduced for* x.

C Implementation

We implement PG-Ring in C++11 on top of the succinct data structures library, *SDSL* (`https://github.com/simongog/sdsl-lite`) [23]. The grids of properties M_π are implemented with wavelet matrices (`wm_int`) and B_π as plain bitvectors (`bit_vector`). The bitvectors B_ℓ for labels are represented using sparse bitvectors (`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 *bag semantics* (allowing repeated results) and supports *homomorphic pattern matching* (allowing two edge patterns match the same graph edge). When the system receives a query, we use the dictionaries and functions to get the integer identifiers. Once the solutions are obtained, their 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 in the `WHERE` clause are not implemented.

C.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 binding the current one, and considering the new resulting range lengths (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 $(x.\pi = y.\pi')$, 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., $x.\pi > a$ for a constant a, or y is bound and $y.\pi' = a$ 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, $(\max_\pi - a)/(\max_\pi - \min_\pi + 1)$, 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, $[1, +\infty] \times [a+1, \infty]$) 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 (which we implemented and found to be slower than using the dynamic VEO, so we show experiments only on the latter).

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 the range $[1, n]$ that satisfies the condition.

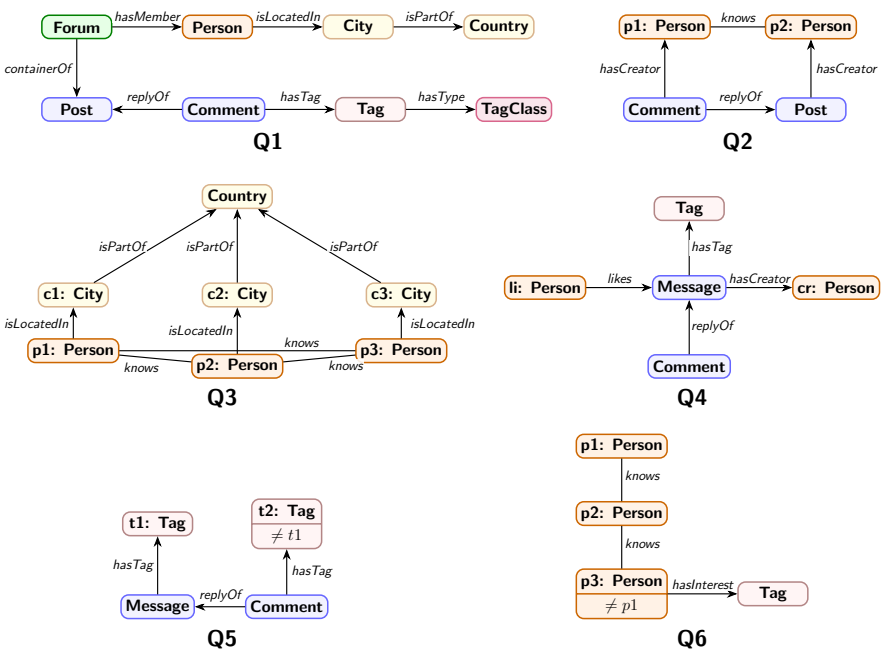

Fig. 7: The first six queries from the Labelled Subgraph Queries Benchmark.

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 reduced 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.

D 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 social network generator [18] 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 Table 1 the average query times obtained.

E Systems Compared

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

- Umbra [43]: 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.
- Neo4j [42]: 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 adding them in both directions.

F 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

Table 1: Average times (in msec) of queries, with different limits on the number of results reported. TO stands for timeouts, set at 600,000 msec.

System	Limit	Q1	Q2	Q3	Q4	Q5	Q6
Umbra	1	1,912.18	462.28	159.64	5,031.74	2,435.16	45.62
	10	1,912.38	465.54	162.26	5,038.58	2,432.72	45.28
	100	1,942.86	574.16	161.16	5,061.36	2,453.38	45.68
	1,000	1,913.06	474.24	162.04	5,028.82	2,437.34	46.62
	10,000	1,939.22	509.38	188.14	5,091.54	2,489.56	48.70
DuckDB	1	3,111.74	454.40	300.04	5,222.78	3,198.34	131.96
	10	3,071.52	446.38	299.24	5,138.54	3,199.64	137.22
	100	3,148.88	453.56	304.50	5,236.04	3,247.80	138.80
	1,000	3,109.04	451.96	301.62	5,196.26	3,202.10	138.52
	10,000	3,128.20	**455.14**	311.40	5,229.64	3,222.50	140.64
Kuzu	1	11,352.10	1,403.96	115,401.98	2,277.40	2,796.80	60.84
	10	11,207.38	1,394.62	115,999.14	2,260.88	2,772.04	61.06
	100	11,372.08	1,392.02	116,026.98	2,271.80	2,777.86	63.20
	1,000	11,440.90	1,406.14	116,053.84	2,279.18	2,784.96	65.10
	10,000	11,201.86	1,432.66	115,931.40	2,291.42	2,799.98	80.84
Neo4j	1	7.56	2.60	2,165.30	3.14	4.40	1.86
	10	8.92	3.66	4,439.06	3.76	5.82	2.26
	100	12.24	**6.42**	TO	5.76	16.76	5.38
	1,000	56.22	**32.08**	TO	35.88	33.96	27.28
	10,000	479.56	2,449.34	TO	337.00	291.10	325.70
Memgraph	1	**5.04**	42.26	36,936.34	39.92	9.06	2.56
	10	**4.38**	37.88	65,980.02	36.84	8.00	1.72
	100	**5.18**	38.88	TO	35.32	10.70	1.10
	1,000	**17.26**	192.38	TO	50.88	9.46	4.50
	10,000	**122.20**	810.90	TO	86.12	52.16	36.02
MillDB	1	5.32	1.78	9.19	2.64	**1.39**	1.78
	10	5.79	2.25	15.80	**2.70**	**1.47**	1.84
	100	9.16	11.79	147.86	2.00	**1.47**	1.08
	1,000	31.34	77.91	1,368.47	**3.44**	**3.14**	**1.93**
	10,000	165.66	786.65	13,582.08	**15.98**	**21.62**	**9.59**
PG-Ring	1	127.37	**0.60**	**0.12**	2.07	103.46	**0.03**
	10	259.25	**1.45**	**0.66**	3.24	114.01	**0.05**
	100	1,120.93	17.00	**3.67**	18.62	211.50	**0.25**
	1,000	8,810.63	173.70	**24.17**	150.87	1,135.47	2.48
	10,000	56,581.03	1,601.01	275.35	902.16	10,086.07	19.99

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.

It is important to note that, per well-known limitations [27], we cannot directly model Wikidata completely with property graphs. Specifically, we cannot directly represent qualifiers that have items/nodes as values (we can add a string ID of the node, but this does not necessarily "join" with the node, unless we duplicate the ID as a node property) or multi-valued properties (which could be captured as arrays or lists where supported as datatypes). We omit such features for simplicity as they should not affect comparative performance.

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.

More precisely, from each SPARQL query in the Wikidata logs, we extract (maximal) subqueries corresponding to a BGP with (if present) filters. We skip queries with multiple such BGPS (ignoring SERVICE clauses), and remove BGPs: (1) with Cartesian products (not fully connected via variables), (2) with constants not in the graph, (3) with unsupported filters (beyond <, >, =, <=, =, !, &, |) or filters with variables not in the BGP, and (4) seen previously (modulo isomorphism). Per the data mapping, coordinates are split into two lat/long properties, constant values for P31 (instance of) are added as node labels, relations between items are added as edges with a fresh edge variable, datatype property values are added as properties to nodes, and datatype qualifier values are added as properties to edges. The resulting query is written to GQL syntax; if the BGP contains a non-projected variable on a datatype property Pn for subject s, we add "IS NOT NULL(s.Pn)" to ensure existence.

These queries are classified into different groups by computing a structural fingerprint of the graph pattern and a descriptor of the WHERE clause. We find 391 groups of queries partitioning by fingerprint. The top 5 are simple triple patterns without WHERE clause, covering almost 50% of the queries. We select 1,000 queries by sampling uniformly while ensuring at least one representative of each group.

G 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. 8 and Table 2 detail 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 Memgraph and PG-Ring. On the Wikidata bench-

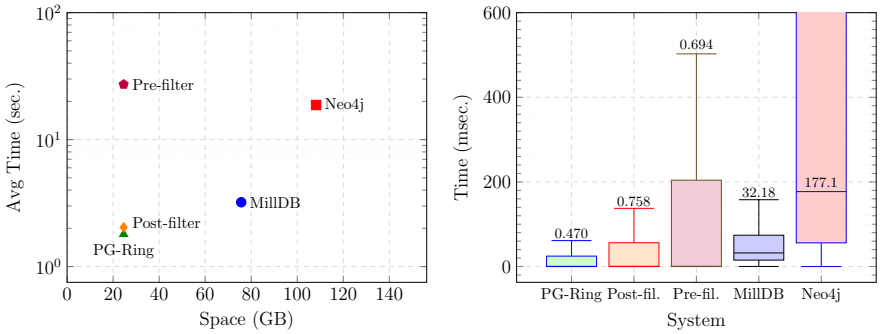

Fig. 8: The space-time trade-off and times distribution on the Wikidata benchmark in the *cold cache* scenario. The numbers in the boxplots indicate the median of execution times.

mark, the cold cache setup sharply affects the performance of both MillDB and Neo4j, as they are disk-based.

Table 2: Detailed execution times (in msec) for the six queries of LSQB in the *cold cache* scenario. TO stands for timeouts, set at 600,000 msec.

System	Limit	Q1	Q2	Q3	Q4	Q5	Q6
Umbra	1	37,701.30	2,261.50	190.40	14,710.50	2,512.00	421.70
	10	37,806.90	2,235.14	188.86	14,835.52	2,547.69	426.12
	100	37,858.30	2,234.60	188.70	14,812.40	2,504.10	417.30
	1,000	37,875.42	2,233.34	195.78	14,842.23	2,537.86	423.26
	10,000	38,079.30	2,277.30	**211.60**	14,841.10	2,520.60	426.90
DuckDB	1	12,018.00	807.70	295.90	11,489.20	4,370.30	187.40
	10	12,321.13	884.21	297.54	11,795.84	4,263.62	182.53
	100	12,164.50	854.20	293.10	11,637.20	4,254.40	155.80
	1,000	11,929.72	867.36	301.84	11,665.70	4,289.34	166.20
	10,000	12,234.70	854.30	301.30	11,430.60	4,447.40	191.30
Kuzu	1	15,634.50	4,772.30	116,282.50	3,556.90	3,742.70	243.80
	10	16,416.00	3,140.02	118,911.93	3,314.12	3,875.07	140.09
	100	16,518.40	3,170.80	117,082.60	3,258.20	3,877.30	136.80
	1,000	16,404.10	3,174.60	118,559.12	3,358.14	3,798.43	152.60
	10,000	15,722.20	3,235.60	115,890.00	3,348.20	3,788.40	172.40
Neo4j	1	1,324.90	572.60	173,106.40	268.50	13,181.40	145.30
	10	2,316.14	760.80	170,786.27	407.12	14,725.46	113.13
	100	4,290.70	3,185.90	TO	621.70	16,108.20	224.40
	1,000	6,115.30	23,148.54	TO	4,135.32	24,139.87	442.88
	10,000	7,307.90	142,413.20	TO	49,078.00	69,088.20	313.00
Memgraph	1	**21.50**	42.00	19,912.40	27.60	6.80	1.30
	10	**21.08**	36.92	41,272.58	24.26	**6.42**	1.64
	100	**24.60**	50.50	TO	34.40	**5.60**	2.00
	1,000	**28.14**	**113.78**	TO	**27.68**	**6.64**	**4.10**
	10,000	**99.10**	**423.80**	TO	**59.70**	**30.20**	23.60
MillDB	1	1,599.68	451.58	634.56	233.53	**2.35**	214.05
	10	1,751.43	995.05	687.08	273.11	7.86	157.41
	100	2,273.16	5,140.34	825.91	588.99	32.13	156.70
	1,000	6,687.00	33,693.54	2,287.22	465.04	71.40	165.09
	10,000	20,904.32	49,924.80	14,777.50	547.92	665.80	298.50
PG-Ring	1	129.84	**1.49**	**0.28**	**4.98**	105.26	**0.08**
	10	262.09	**3.04**	**1.22**	**1.46**	117.94	**0.13**
	100	1,131.02	**23.18**	**5.26**	**24.55**	212.99	**0.52**
	1,000	8,804.49	180.28	**28.82**	157.19	1,136.67	4.37
	10,000	56,307.18	1,603.29	281.60	911.71	10,081.64	**23.35**